SYNGRESS®

Asterisk Hacking

Turn Your Phone System Into a Samurai Sword—for Attack or Defense

- Asterisk Live CD (SLAST) Contains All the Tools in the Book Ready to Boot!
- Understand the Threats to Asterisk: Denial-of-Service, VoIP Service Disruption, Call Hijacking and Interception, H.323-Specific Attacks, and SIP-Specific Attacks
- Complete Coverage of Interfacing Asterisk with Hardware: Security Cameras, Electronic Door Locks, and Card Readers

Ben Jackson aka Black Ratchet
Champ Clark III aka Da Beave

Johnny Long Technical Editor
Larry Chaffin Technical Editor

Visit us at

www.syngress.com

Syngress is committed to publishing high-quality books for IT Professionals and delivering those books in media and formats that fit the demands of our customers. We are also committed to extending the utility of the book you purchase via additional materials available from our Web site.

SOLUTIONS WEB SITE

To register your book, visit www.syngress.com/solutions. Once registered, you can access our solutions@syngress.com Web pages. There you may find an assortment of value-added features such as free e-books related to the topic of this book, URLs of related Web sites, FAQs from the book, corrections, and any updates from the author(s).

ULTIMATE CDs

Our Ultimate CD product line offers our readers budget-conscious compilations of some of our best-selling backlist titles in Adobe PDF form. These CDs are the perfect way to extend your reference library on key topics pertaining to your area of expertise, including Cisco Engineering, Microsoft Windows System Administration, CyberCrime Investigation, Open Source Security, and Firewall Configuration, to name a few.

DOWNLOADABLE E-BOOKS

For readers who can't wait for hard copy, we offer most of our titles in downloadable Adobe PDF form. These e-books are often available weeks before hard copies, and are priced affordably.

SYNGRESS OUTLET

Our outlet store at syngress.com features overstocked, out-of-print, or slightly hurt books at significant savings.

SITE LICENSING

Syngress has a well-established program for site licensing our e-books onto servers in corporations, educational institutions, and large organizations. Contact us at sales@syngress.com for more information.

CUSTOM PUBLISHING

Many organizations welcome the ability to combine parts of multiple Syngress books, as well as their own content, into a single volume for their own internal use. Contact us at sales@syngress.com for more information.

SYNGRESS®

SYNGRESS®

Asterisk Hacking

Toolkit and LiveCD

Benjamin Jackson
Champ Clark III
Larry Chaffin and Johnny Long Technical Editors

KEY	SERIAL NUMBER
001	HJIRTCV764
002	PO9873D5FG
003	829KM8NJH2
004	BAL923457U
005	CVPLQ6WQ23
006	VBP965T5T5
007	HJJJ863WD3E
008	2987GVTWMK
009	629MP5SDJT
010	IMWQ295T6T

PUBLISHED BY
Syngress Publishing, Inc.
Elsevier, Inc.
30 Corporate Drive
Burlington, MA 01803

Asterisk Hacking

Printed in the United States of America
1 2 3 4 5 6 7 8 9 0
ISBN: 978-1-59749-151-8

Publisher: Amorette Pedersen
Acquisitions Editor: Andrew Williams
Technical Editors: Johnny Long and Larry Chaffin
Cover Designer: Michael Kavish
Project Manager: Anne B. McGee
Page Layout and Art: Patricia Lupien
Copy Editor: Michael McGee
Indexer: Richard Carlson

For information on rights, translations, and bulk sales, contact Matt Pedersen, Commercial Sales Director and Rights, at Syngress Publishing; email m.pedersen@elsevier.com.

Co-Authors

Benjamin Jackson (Black Ratchet) is a jack of all trades computer guy from New Bedford, MA. Ben holds a BS in Computer Engineering Technology from Northeastern University and spends his days developing applications and doing database administration for the Massachusetts Cancer Registry. By night, he toys with Asterisk, develops security tools, and generally breaks things.

Ben is a co-founder of Mayhemic Labs, an independent security research team, and has lectured at various hacker and professional conferences regarding VoIP and Open Source Software. He has also contributed code to the Asterisk source tree and other open source projects. One of the last true phone phreaks, he also enjoys playing on the Public Switched Telephone Network and spends far too much time making long distance phone calls to far flung places in the world.

Champ Clark III (Da Beave) has been involved in the technology industry for 15 years. Champ is currently employed with Vistech Communications, Inc. providing network support and applications development. Champ is also employed with Softwink, Inc. which specialized in security monitoring for the financial industry. Champ is one of the founding members of "Telephreak", an Asterisk hobbyist group, and the Deathrow OpenVMS cluster. When he's not ripping out code or writing papers, he enjoys playing music and traveling.

Technical Editors

Larry Chaffin is the CEO/Chairman of Pluto Networks, a worldwide network consulting company specializing in VoIP, WLAN, and security. An accomplished author, he contributed to Syngress's *Managing Cisco Secure Networks* (ISBN: 1931836566); *Skype Me!* (ISBN: 1597490326); *Practical VoIP Security* (ISBN: 1597490601); *Configuring Check Point NGX VPN-1/FireWall-1* (ISBN: 1597490318); *Configuring Juniper Networks NetScreen and SSG Firewalls* (ISBN: 1597491187); and *Essential Computer Security: Everyone's Guide to Email, Internet, and Wireless Security* (ISBN: 1597491144). He is the author of *Building a VoIP Network with Nortel's MS5100* (ISBN: 1597490784), and he has coauthored or ghostwritten 11 other technology books on VoIP, WLAN, security, and optical technologies.

Larry has over 29 vendor certifications from companies such as Nortel, Cisco Avaya, Juniper, PMI, isc2, Microsoft, IBM, VMware, and HP. Larry has been a principal architect designing VoIP, security, WLAN, and optical networks in 22 countries for many Fortune 100 companies. He is viewed by his peers as one of the most well respected experts in the field of VoIP and security in the world. Larry has spent countless hours teaching and conducting seminars/workshops around the world in the field of voice/VoIP, security, and wireless networks. Larry is currently working on a follow-up to *Building a VoIP Network with Nortel's MCS 5100* as well as new books on Cisco VoIP networks, practical VoIP case studies, and WAN acceleration with Riverbed.

Johnny Long Who's Johnny Long? Johnny is a Christian by grace, a family guy by choice, a professional hacker by trade, a pirate by blood, a ninja in training, a security researcher and author. His home on the web is http://johnny.ihackstuff.com.

Contents

Chapter 1

What Is Asterisk and Why Do You Need It?

Solutions in this chapter:

- **What Is Asterisk?**
- **What Can Asterisk Do for Me?**
- **Who's Using Asterisk?**

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

For years, telephone networks were run by large companies spending billions of dollars to set up systems that connected to one another over wires, radios, and microwaves. Large machines, filling entire buildings, allowed people to talk to each other over great distances. As the computer revolution progressed, the machines got smaller and more efficient, but still they were almost exclusively the domain of a small sect of companies.

Enter Asterisk… Asterisk has taken the power of the open-source software movement and brought it to the land of telephony. Much like how open source has proven that users don't need to rely on commercial companies for software, Asterisk has proven that users don't need to rely on commercial telephone companies for telephone systems. Open-source software allows you to be free of vendor lock-in, save money on support, use open standards, and change the software to suit your unique problems if the need arises. Looking at the "traditional" Private Branch Exchange (PBX) market, vendor lock-in is all too common, vendors charge exorbitant fees for support, and all too often the PBX you buy is a cookie-cutter solution with little to no customization options. It is common for people to think that their PBX is a black box that handles telephone calls. In reality, it is a bunch of computing equipment running a highly specialized software package. Open-source software can replace that customized software just as easily as it can replace any other software.

Asterisk is a veritable Swiss Army knife of telephony and Voice over Internet Protocol (VoIP). Designed to be a PBX replacement, Asterisk has grown to be all that and more. It boasts the ability to store voice mail, host conference calls, handle music on hold, and talk to an array of telephone equipment. It is also scalable, able to handle everything from a small five-telephone office to a large enterprise with multiple locations.

Thanks to Asterisk and VoIP, it is possible to run a telephone company out of a basement, handling telephone calls for people within a neighborhood, a city, or a country. Doing this only a few years ago would have required buying a large building, setting up large racks of equipment, and taking out a second mortgage. But today, everyone is jumping on the Asterisk bandwagon: hobbyists, telephone companies, universities, and small businesses, just to name a few. But what exactly *is* Asterisk? And what can it do? Let's find out.

What Is Asterisk?

Asterisk is an open-source PBX that has VoIP capabilities. However, this hardly explains what Asterisk is or what it does. So let's delve a little more deeply into PBXes, VoIP, and Asterisk.

What Is a PBX?

Asterisk, first and foremost, is a Private Branch Exchange. A PBX is a piece of equipment that handles telephone switching owned by a private business, rather than a telephone company. Initially in the United States, PBXes were for medium-to-large businesses that would create a lot of telephone traffic starting from, and terminating within, the same location. Rather than having that traffic tie up the switch that handles telephones for the rest of the area, PBXes were designed to be small switches to handle this traffic. Thus, the PBX would keep the internal traffic internal, and also handle telephone calls to and from the rest of the telephone network.

In the United States, thanks in part to the Bell System breakup of 1984, and to the computer revolution shrinking PBXes from the size of a couch to the size of a briefcase, PBXes flooded the market. Hundreds of companies started making PBXes and thousands wanted them. New features started coming into their own: voice mail, interactive menus, call waiting, caller ID, three-way calling, music on hold, and so on. The telecommunications industry grew by leaps and bounds, and the PBX industry kept up. However, with every silver lining comes a cloud. With the proliferation of digital telephone systems, each vendor had a specific set of phones you could use with their PBX. *Company X's* phones would often not work with *Company Y's* PBX. Plus, as with almost every technology, all too often a vendor would come in, set up the telephones, and never be heard from again, leaving the customer to deal with the system when it didn't work.

PBXes are one of the key pieces of hardware in businesses today, ranging from small devices the size of shoeboxes that handle a few lines to the telephone network and five phones in a small office, to a large system that interconnects ten offices across a campus of buildings. However, today's PBXes, when boiled down, all do the same things as their predecessors: route and handle telephone calls, and keep unnecessary traffic off the public switched telephone network.

Asterisk is a complete PBX. It implements all the major features of most commercially available PBXes. It also implements, for free, features that often cost a lot in

a commercial installation: Conference calling, Direct Inward System Access, Call Parking, and Call Queues, just to name a few.

Out of the box, Asterisk can be configured to replicate your current PBX install. There have been numerous installs where a company's existing PBX is taken down on a Friday, an Asterisk server is installed and configured on Saturday, wired and tested on Sunday, and is handling calls on Monday. The users only notice a different voice when they grab their voice mail.

What Is VoIP?

Voice over Internet Protocol is one of the new buzzwords of the media today. While VoIP has been around in one incarnation or another since the 1970s, the market and technology has exploded over the past three years. Companies have sprouted up selling VoIP services and VoIP software, and instant messaging services are starting to include VoIP features.

But what exactly *is* VoIP? VoIP is a method to carry a two-way conversation over an Internet Protocol–based network. The person using Vonage to talk to her neighbor down the street? That's VoIP. The person in the United States using Windows Messenger to talk to his extended family in Portugal? That's VoIP. The 13-year-old playing Splinter Cell on his Xbox and talking to his teammates about how they slaughtered the other team? That's VoIP, too.

VoIP has exploded for a number of reasons—a major one being its ability to use an existing data network's excess capacity for voice calls, which allows these calls to be completed at little to no cost. A normal call that uses the standard telephone network compression coder–decoder algorithm (codec), μ-Law, will take up 64 kilobits per second of bandwidth. However, with efficient compression schemes, that can be dropped dramatically. In Table 1.1, we list certain commonly supported codecs, and how many simultaneous calls a T1 can handle when using that codec.

Table 1.1 VoIP Codec Comparison Chart

Codec	Speed	Simultaneous Calls over a T1 Link (1.5 Mbps)	Notes
μ-Law	64 Kbps	24	
G.723.1	5.3/6.3 Kbps	289/243	
G.726	16/24/32/40 Kbps	96/64/48/38	
G.729	8 Kbps	192	Requires license

Continued

Table 1.1 continued VoIP Codec Comparison Chart

Codec	Speed	Simultaneous Calls over a T1 Link (1.5 Mbps)	Notes
GSM	13 Kbps	118	
iLBC	15 Kbps	102	
LPC-10	2.5 Kbps	614	
Speex	2.15 to 44.2 Kbps	714 to 34	"Open" codec

The savings of bandwidth comes at a cost though; the more compression placed on a conversation, the more the voice quality degrades. When using LPC10 (one of the most efficient compression codecs), the conversation, while intelligible, often sounds like two whales making mating calls. If you have no other alternative, it will be sufficient, but it's not a good choice for a business environment.

The other major benefit of VoIP is the mobility. Phone calls can be sent and received wherever a data connection is available, whether it is a residential broadband connection, the office network, or a WiFi connection at a local drinking establishment. This mobility has a many benefits: a company's sales force can be scattered across the country yet have a phone in their home office that is an extension of the company's PBX. They can enjoy a voice mail box, an extension off the company's main number, and all the other features as if they all were in the same building.

It is important to make the distinction that VoIP is not exclusive to Asterisk. There is a growing market of software-based PBXes that tout VoIP as a major feature. Some traditional PBXes are starting to include VoIP features in them, and local phone companies are offering VoIP packages for customers. As a result, the advantages of VoIP have begun to catch the attention of the entire telecom industry.

The History of Asterisk

Mark Spencer, the creator of Asterisk, has created numerous popular open-source tools including GAIM, the open-source AOL Instant Messaging client that is arguably the most popular IM client for Linux, l2tpd, the L2TP tunneling protocol daemon, and the Cheops Network User Interface, a network service manager. In 1999, Mark had a problem though. He wanted to buy a PBX for his company so they could have voice mail, call other offices without paying for the telephone call, and do all the other things one expects from a PBX system. However, upon researching his options, he realized all the commercial systems cost an arm and a leg.

Undaunted, he did what every good hacker would: he set to writing a PBX suitable to his needs.

On December 5, 1999, Asterisk 0.1.0 was released. As the versions progressed, more and more features were added by developers, gathering a following of users, conventions, and everything short of groupies along the way. Asterisk's first major milestone was reached on September 23, 2004, when Mark Spencer released Asterisk 1.0 at the first Astricon, the official Asterisk user and developer's conference. Asterisk 1.0 was the first stable, open-source, VoIP-capable PBX on the market. Boasting an impressive set of features at the time, it included a complete voice conferencing system, voice mail, an impressive ability to interface into analog equipment, and the ability to talk to three different VoIP protocols reliably.

Development didn't stop there though. Asterisk continued to grow. On November 17, 2005, Asterisk 1.2 was released, which addressed over 3000 code revisions, included major improvements to the core, more VoIP protocols, and better scalability. Also, this release introduced Digium's DUNDi (Distributed Universal Number Discovery) protocol, a peer-to-peer number discovery system designed to simplify interconnecting Asterisk servers across, and in between, enterprises.

The latest release of Asterisk, Asterisk 1.4, was released December 27, 2006. This release featured major changes in the configuration process, optimized applications, simplified the global configuration, and updated the Call Detail Records for billing purposes. Also new in this version was better hardware support, an improved ability to interface with legacy equipment, and better interfacing with Cisco's SCCP VoIP protocol. Also, as with any software project, this update addressed the bugs and issues found since the 1.2 release.

Asterisk Today

Today, Asterisk is one of the most popular software-based VoIP PBXes running on multiple operating systems. Asterisk handles most common PBX features and incorporates a lot more to boot. It works with numerous VoIP protocols and supports many pieces of hardware that interface with the telephone network. Asterisk is currently at the forefront of the much talked-about "VoIP revolution" due to its low cost, open-source nature, and its vast capabilities.

The company Mark Spencer wrote his PBX for is now known as Digium, which has become the driving force behind Asterisk development. Digium sells hardware for interfacing computers into analog telephone lines and Primary Rate Interface (PRI) lines. Digium also offers Asterisk Business Edition, an Enterprise-ready version

of Asterisk, which includes commercial text-to-speech and speech recognition product capabilities, and has gone through stress testing, simulating hundreds of thousands of simultaneous phone calls. Finally, Digium offers consulting for Asterisk installations and maintenance, and trains people for its Digium Certified Asterisk Professional certification.

Notes from the Underground…

Digi-wha?

Many companies spend millions of dollars with marketing firms to create a new name for their company. When Bell Atlantic and General Telephone and Electric (GTE) merged in 2000, they thought long and hard about their new name, and when they revealed it, millions scratched their head and said "What is a Verizon?" Thankfully, not all companies have this problem.

Digium (Di-jee-um) is the company that maintains most of the Asterisk source tree, and tries to show how Asterisk can provide solutions to the general public. According to legend, Digium got its curious sounding name when one of its employees pronounced paradigm as "par-a-did-jem." This became a meme, and "par-a-did-jem" evolved into "did-jem," which then further evolved into "Digium." Just think how much money Fortune 500 companies pay advertising executives to come up with a new name when companies merge.

What Can Asterisk Do for Me?

Asterisk is so multifaceted it's hard to come up with a general catchall answer for everyone asking what Asterisk can do for them. When a friend and I tried to think up an answer that would fit this requirement, the closest thing we could come up with was "Asterisk will do everything except your dishes, and there is a module for that currently in development."

Asterisk as a Private Branch Exchange

Asterisk is, first and foremost, a PBX. Some people seem to constantly tout Asterisk's VoIP capabilities, and while that is a major feature, they seem to forget that Asterisk

doesn't need VoIP at all to be a PBX. But even without VoIP, Asterisk has many advantages over traditional hardware-based PBXes.

Advantages over Traditional PBXes

Asterisk has numerous advantages over "traditional" PBXes. These advantages can benefit both larger and smaller businesses. Let's talk about two different scenarios, with two different problems, but one common solution.

Notes from the Underground…

Is Asterisk Right for Me?

Whether they're an individual interested in VoIP or a group of business heads wondering if they should drop their expensive PBX, people frequently ask "Is Asterisk right for me?" The answer, almost always, is a resounding "YES!" Asterisk is many things to many people, and it is malleable enough to be a perfect fit for your setup, too.

Asterisk in a Large Business Environment

Suppose you are the newly hired IT Director for a medium-sized office. While getting a tour of the server room, you happen across the PBX. What you see disturbs you: a system, which handles approximately 200 people, is about the size of two mini fridges, requiring its own electrical circuit separate from the servers, and producing enough heat it has to be tucked in a corner of the server room so as not to overload the air conditioning system. It also seems to be stuck in the early 1990s: The system has abysmal voice-mail restrictions, no call waiting, and no caller ID. Being the go-getter you are, you attempt to "buy" these features from the vendor, but the quote you receive almost gives your purchase officer a heart attack. As if this wasn't enough, you also have a dedicated "PBX Administrator" who handles adding phones to the system, setting up voice-mail boxes, making backups of the PBX, and nothing else.

Asterisk is made for this kind of situation. It can easily fit within a server environment, and will cut costs instantly since you no longer have to cool and power a giant box that produces massive amounts of heat. Also, dedicated PBX administrators, while possibly still necessary for a large environment, can be easily replaced by other

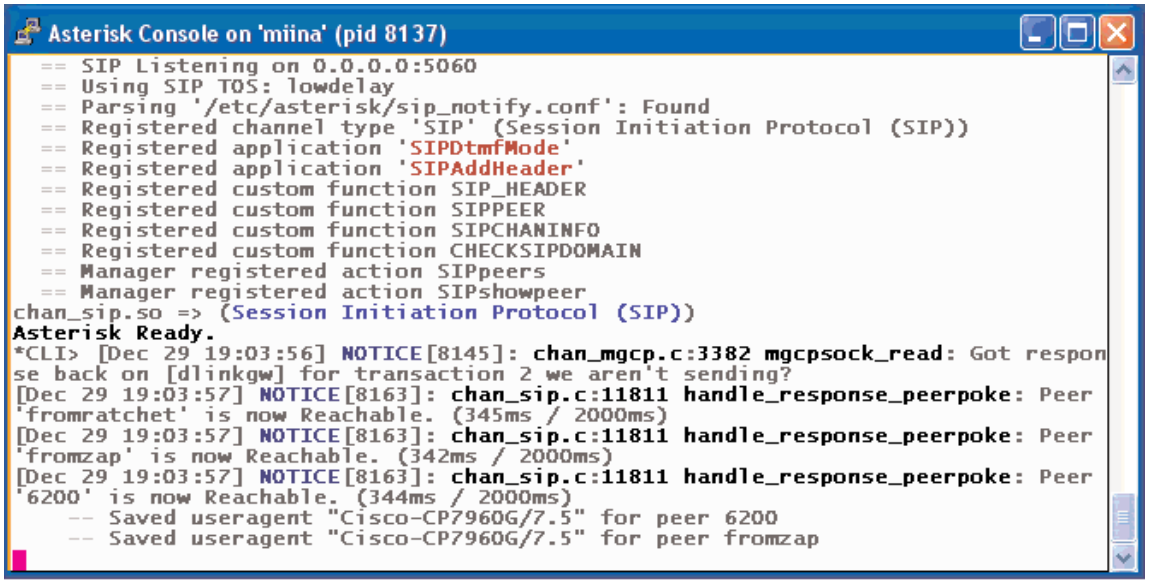
administrators, provided they know how to administrate a Linux box. A competent Linux user can be taught how to administer an Asterisk PBX easily. Finally, as stated repeatedly, Asterisk is open source, which really cuts the software upgrade market off at the knees. Plus, if Asterisk lacks a feature a company needs, there are more than a few options available to the firm: they can code it themselves, hire someone to code it for them, or use Asterisk's fairly active bounty system (available at http://www.voip-info.org).

Asterisk in a Small Business Environment

Asterisk provides advantages for small businesses as well. Suppose you are a consultant to a small company that has you come in a few hours every week to fix computer problems. This company has a small, ten-phone PBX that was installed by another vendor before you came into a picture. After a while, one of the phones—the owner's, of course—will no longer work with the voice-mail system. When you dial his extension, it rings his phone, and then drops you to the main voice-mail prompt instead of going directly to his voice-mail box. When he dials his voice mail from his phone, it prompts him for a mailbox rather than taking him directly to his. The vendor no longer returns phone calls, and the owner begs you to take a look at it. You bang your head against the wall for several hours trying to figure the system out. Besides the basic "How to use your phone" info, no documentation is available, there are no Web sites discussing the system, and diagnostic tools are non-existent. Even if you do figure out the problem, you have no idea how to correct it since you don't know how to reprogram it. In other words, you're licked.

Asterisk will fix most of the issues in this situation as well. Documentation, while admittedly spotty for some of the more obscure features, is widely available on the Internet. Asterisk debugging is very complete; it can be set up to show even the most minute of details. Also, in a typical Asterisk installation, vendor tie-in wouldn't be an issue. If the owner's phone was broken, a replacement phone could have been easily swapped in and set up to use the PBX—no vendor needed (see Figure 1.1).

Figure 1.1 Asterisk Can Be as Verbose, or as Quiet, as You Want



Features and Uses

As previously stated, Asterisk has numerous features, some common to almost all PBXes, and some only found in very high-end models. Let's highlight a few. This is by no means a complete list, but just a sampling of the many features Asterisk has to offer.

Conference Calls

Asterisk's conference calling system, called "MeetMe," is a full-featured conferencing system. All the features you would expect in a conferencing system are included, such as protecting conferences with PINs so only approved users can attend, moderating conferences to allow only certain people to speak to the group, recording conferences so you can have a record of it, and playing music before a conference begins so users don't have to wait in silence.

MeetMe is a huge feature for Asterisk, as the price of commercial conferencing services isn't cheap. Let's look at a simple example: We want to conduct an hour-long conference call with ten members of the press concerning our new Asterisk book. A certain reputable conferencing service costs 18 cents per minute per participant. So, doing the math, 13 users talking for 60 minutes at a cost of 18 cents/minute would cost us $140.40. Let's compare that with Asterisk. Using Asterisk, MeetMe, and an

average VoIP toll-free provider whose rates are 2.9 cents per minute per call, the same conference would cost us $22.62. That's a savings of $117.78!

Voice Mail

Voice mail has become critical to business in today's market. Many people have developed a reflexive tendency to check the "Message Waiting" indicator on their phone when first entering their workspace. Technically, voice mail is quite simple. It is simply audio files stored on some kind of storage medium, such as a hard drive or flash storage, on your PBX. Some vendors think a two-hour voice-mail storage card, otherwise known as a 128MB Smart Media card, should cost over $200. Asterisk, considering it's run on a PC, affords you an amazing amount of storage space for your company's voice mail. Since it's not locked into a specific storage media, you can add an extra hard drive, flash card, or network share if you have the need to expand.

Asterisk's voice mail also incorporates almost every feature one would expect from a voice-mail system: a complete voice-mail directory, forwarding, and the ability to play different outgoing messages depending on whether the user doesn't pick up their phone, is already on the phone, or is out for a long period of time. Some of the more advanced features include the ability to send the voice mail as an attachment to an e-mail address. This is useful if you are on the road and do not have a phone available to you, but do have access to e-mail. It's also very handy when you have a voice-mail account you do not monitor regularly.

Call Queues

While everyone might not know what a call queue is, almost everyone has experienced one. When dealing with some kind of customer service department, it's not uncommon to wait on hold while a disembodied voice tells you that all the representatives are currently helping other people. That is a call queue. It is used for handling large volumes of calls with a set amount of people answering the phones. When the amount of calls ("callers") exceeds the amount of people answering the phones ("answerers"), a queue forms, lining up the callers till an answerer can attend to each. When one of the answerers becomes available, the first caller in line gets routed to that answerer's phone. Call queues are essential in any kind of call center environment. Asterisk supports both queues in the traditional sense of a call center full of people, and also a virtual call center in which the call agents call in from home and sit on the phone in their house. It supports ringing all agents at once, a

round-robin system, or a completely random ring pattern. Asterisk also can assign priorities to callers when they enter a queue. For example, this is commonly done in cell phone companies. Have you ever wondered how when you visit a cell phone store and they call up customer service, they get answered in about 30 seconds? They call a separate number and are thus assigned a higher priority than if you called from your home. Another use of this is if you run a helpdesk and want to assign problems with mission-critical applications a higher priority than others. Users calling the telephone number for the mission-critical applications would thus receive a higher priority than users that call the general helpdesk number.

Asterisk as a VoIP Gateway

Asterisk's biggest and most talked about feature is its VoIP capabilities. Thanks to the expansion of Broadband into almost every company and an ever-increasing number of residences, VoIP has taken off in the past few years. Asterisk has turned out to be a tool no one really knew they needed, but realized what they were missing once they started using it.

Notes from the Underground…

PSTN Termination and PSTN Bypassing

Don't worry, PSTN termination has nothing to do with the PSTN becoming self aware and sending robots after us. PSTN termination providers are companies that allow third parties to transition their VoIP call between the Internet and the PSTN, or vice versa. These companies don't force users to invest in equipment to connect Asterisk to a phone line and are often much cheaper than what a telephone company would charge.

Of course, the cheapest phone call is the one that's free. The Internet Telephony Users Association, a non-profit organization, runs e164.org, which allows users to publish telephone numbers that can be reached directly via VoIP. This allows other VoIP users to dial a regular number and have Asterisk route it over the Internet rather than the PSTN letting the user save money without making an effort.

People have started using Asterisk to augment, and sometimes even replace, their existing telephone setup. Thanks to Asterisk, an abundance of cheap Internet-to-

PSTN-termination providers, and organizations such as e164.org, Asterisk has allowed people to choose the cheapest path to their destination when placing a phone call. Companies with multiple offices can save money on phone calls that are long distance from the originating office but local to one of the other offices by using Asterisk to route them over the Internet to the remote office and having the Asterisk server dial the remote phone line, thus saving them an expensive long-distance bill.

The Possibilities of VoIP

Looking at various trade magazines and Web sites, it is easy to get the feeling that pundits always rant and rave about VoIP, but companies and end users either have no interest in it or do have an interest but no idea what to do with it. Asterisk and VoIP provide many possibilities for both the end user sitting at home and the company looking to cut costs.

Virtual Call Centers and Offices

Before VoIP, when running a call center, the company either needed to pay for a large building to house all the employees, or pay the cost of forwarding the incoming phone calls to the employee's houses. With the advent of VoIP, a third option has emerged: using the employee's broadband connections to handle telephone calls over VoIP.

Thanks to Asterisk, it is possible to run a call center out of a back pocket. The only physical presences the call center needs are servers to handle the routing of the calls, and some way to terminate the incoming phone calls, such as a VoIP provider or PRI(s). The people answering the calls can either use their computer with a softphone and a headset, or some kind of Analog Telephone Adapter to hook up a VoIP connection to a physical phone (more on these later). Agents can then sign into the call queue without tying up their phone line or costing them money. They can also work anywhere a broadband connection is available.

This benefit isn't limited to call centers either. Would you like to save some money on your road warrior's cell phone bills? Or, would you like to have an option for your employees to work from home for a few days a week, but still have the ability to be contacted by phone like they were in their office? The same concept applies. Once a phone signs into Asterisk, it doesn't matter if it's in the office, down the street, or half a continent away, it becomes an extension on your PBX, with all the features and benefits.

Bypassing the Telephone Companies

Another way people have been using Asterisk is to set up their own "VoIP only" telephone network over the Internet. Suppose you have a group of friends you never talk to. With Asterisk, you can essentially set up your own virtual telephone company. After setting up Asterisk and then arranging the connections between your servers, you can establish a telephone network without even touching the PSTN. Plus, thanks to MeetMe, you can conduct conference calls with ease.

Also, while the media and most of the public associate "VoIP" with "phone calls over the Internet" this is only partly the truth. The "IP" in VoIP means "Internet Protocol," and Internet Protocol is Internet Protocol no matter where it is. If your company has data links between buildings, campuses, or regions, but not voice links, Asterisk can be used to send voice conversations over your data links as opposed to the phone lines, saving money and allowing your phone lines to remain free for other purposes.

One of the best hobbyist roll-your-own examples we've seen to highlight Asterisk's ability to act as an inexpensive gateway for telephones over large geographic areas is the Collector's Net at http://www.ckts.info. Founded in 2004, the Collector's Net is a group of telephony buffs who have, over time, collected old telephone switching equipment. For years, this equipment sat in basements and garages collecting dust until one owner had the bright idea of using Asterisk and VoIP to interconnect the gear over the Internet. And so Collector's Net was born. It is growing monthly and now boasts an Asterisk backbone connecting more than a dozen switches over two continents. While it may seem trivial or downright odd to some, this highlights the ability of Asterisk to provide a connection between a group of people who would have hardly spoken to each other had they not set up this network.

Being Your Own Telephone Company

Asterisk can save money, but it can make money as well. It's also simpler than you think. NuFone, one of the first PSTN termination providers that supported Asterisk's Inter-Asterisk eXchange (IAX) VoIP protocol, started as a computer and a Primary Rate Interface (PRI), sitting in the owner's apartment. It's now one of the more popular PSTN termination providers on the Internet.

However, don't start wearing your monocle and lighting cigars with $20 bills just yet. In years past, termination providers were largely flying under the radar of the various regulatory agencies. However, this golden age is rapidly coming to a close, and VoIP providers are slowly becoming more and more regulated. Today, VoIP

providers must provide 911 services, are required to contribute to the Federal government's "Universal Service Fund," must handle taps by law enforcement agencies, and are subject to all kinds of regulations.

Asterisk as a New Dimension for Your Applications

The Internet has grown by leaps and bounds over the past ten years. Most companies have mission-critical applications, applications to monitor the applications, and applications to monitor the applications that monitor the applications, ad nauseam. There are also information systems designed to provide important information to the general public. These systems all have something in common: they require the use of a computer.

Computers, while common, aren't used by everyone. People constantly talk about the "digital divide," referring to people who are unable to afford computers. Plus, sizable portions of the populations, for one reason or another, still treat the computer with apprehension.

Phones, however, are very much ubiquitous. Almost every home has a land-based telephone in it, and with pre-paid mobile phones finally showing up in the United States, mobile phones are further penetrating the market. Despite this large market, developing voice-aware applications has always been costly and time-consuming, making them less common and less functional than their Web-based counterparts.

Asterisk can be a bridge between the world of text and the world of speech. Thanks to programs like Sphinx (a program that translates speech to text), Festival (a program that translates text to speech), and Asterisk's own application interface, programs can be written by any competent programmer. Asterisk's interface is simple to learn yet extremely powerful, allowing programs for it to be written in almost any language. Asterisk can be the conduit for taking your applications out of the text that is the Internet and letting them cross over into the voice arena that is the Public Switched Telephone Network (PSTN)

A great example of how telephone-aware systems can benefit the general public is Carnegie Mellon University's "Lets Go!" bus dialog system. It has been developed to provide an interactive telephone program that allows people in Pittsburgh to check the schedule of buses that run in the city. The system has become such a success that the bus company has had its main phone number forward calls to the application during off-hours, allowing callers to access transportation schedules despite the

office being closed. Asterisk can also be used to build similar systems with the same tools used by CMU.

Who's Using Asterisk?

Asterisk really started to make a splash on the Internet in late 2003 when it became fairly stable and early adopters started to pick up on VoIP. Since most early adopters were hard-core technophiles who were looking for a program that was free or cheap, and could be easily configured to do everything from the simple and the mundane to the downright odd, Asterisk was in the right place at the right time. To say it caught on like wildfire is a bit of an understatement.

Today, Asterisk is still very active within the hobbyist's realm. Small groups are setting up Asterisk servers for both public and private use, one of them being the Collector's Net previously mentioned. There are also groups of phone phreaks—people who hack on the telephone network—who are taking the leap into the digital realm, setting up projects such as Bell's Mind (http://www.bellsmind.net) and Telephreak (http://www.telephreak.org). For phone phreaks, the ability to run a telephone system in the privacy of one's own home is just as exciting as when the first personal computers became available to computer hackers.

Not only is Asterisk actively thriving in the hobbyist scene, it is also making beachheads into the Enterprise realm. A university in Texas recently replaced their 1600-phone strong mix of Nortel PBXes and Cisco Call Manager installations with Asterisk. The reasons for this were both the cost of licensing each phone to Cisco, and security concerns due to the fact they ran on Windows 2000. A town in Connecticut recently deployed a 1500-phone Asterisk system, where each department customized it for its own needs, such as the school department's automated cancellation notification system.

Not only is Asterisk making it easy for companies to replace their existing telephone systems, it is making it easy for telephone companies to have the ability to handle VoIP. Numerous Competitive Local Exchange Carriers (CLECs) are jumping onto the VoIP bandwagon and setting it up to handle VoIP from the consumer side (or handle it internally) for either a value-added service or a cost-saving measure.

Summary

PBXes and VoIP have been around for decades: PBXes since the early part of the century, and VoIP since the 1970s. However, despite the vast market and the fact that they are used by almost every business, PBXes not only still cost thousands of dollars, but one vendor's equipment is often incompatible with another vendor's.

Asterisk, created in 1999 because Mark Spencer found commercial PBXes hideously expensive, has put the power of telephony in the hands of the masses. It can be many things to many people, and can be configured to fit into many roles in an Enterprise. From saving money on telephone calls, to making voice-enabled applications, Asterisk can be configured to fit in where it's needed.

Asterisk can augment, or entirely replace an existing telephone system, whether the user is a hobbyist with a single telephone line, or an executive running a large call center with multiple PRIs. An existing PBX installation can be swapped out with ease, and most, if not all functionality can be retained. Asterisk also has numerous advantages over traditional PBXes in the areas of cost, reliability, usability, and hardware support.

Asterisk is not only a traditional PBX, but can also handle Voice over IP telephone calls. This allows users to take advantage of the numerous advantages VoIP provides: low-cost telephone calls, the ability to communicate with remote offices using the Internet rather then the PSTN, or using existing data links instead of connecting buildings with telephone lines.

Asterisk also allows you to integrate existing applications into the world of telephony. Users can interact with existing applications over telephones, rather than their current interface—such as a Web page or a data terminal. This has advantages in both usability and flexibility.

In the current market, Asterisk is being utilized by both large and small companies. It lets small companies find a PBX that won't tie them down to a vendor and incur a hefty initial investment, while large companies see a way of leveraging their existing infrastructure that saves them money by not having to rely on the telephone company.

Solutions Fast Track

What Is Asterisk?

- ☑ Asterisk is an open-source Private Branch Exchange that replicates, for free, many expensive features found in expensive high-end PBXes.
- ☑ Created in 1999 by Mark Spencer, it was initially made because commercial PBXes were far too expensive for his company. Today, his company is the driving force behind Asterisk.
- ☑ Asterisk's current version, 1.4, boasts a load of new features over its predecessors.

What Can Asterisk Do for Me?

- ☑ Asterisk can be fit into both the large and small business environment, saving time and money in the workplace. It can also be useful to the hobbyist.
- ☑ Asterisk can replace your traditional hardware PBX and replicate most of its features. It can also bring many new features to the table to replace other telephony services you currently use.
- ☑ Thanks to the advantages provided by VoIP, Asterisk allows you to run virtual call centers and bypass the telephone company for phone calls. It also lets you be your own telephone company.
- ☑ With the ubiquity of voice communication channels, Asterisk lets you bring a whole new dimension to your current suite of applications.

Who's Using Asterisk?

- ☑ Asterisk took the market by storm by being in the right place at the right time, and by also being free.
- ☑ Hobbyists are using Asterisk to set up their own private telephone playlands, complete with voice conferences, voice-mail systems, and voice bulletin boards.

☑ Companies both large and small are using Asterisk to replace their current PBX systems and are saving themselves both time and money in the process.

Links to Sites

- **Asterisk (http://www.asterisk.org)** Here, you can download the source, keep up-to-date on Asterisk-related news, read developer weblogs, and generally get your daily dose of Asterisk scuttlebutt.
- **Digium (http://www.digium.com)** These folks are the driving force behind Asterisk. Get trained, buy hardware, and find out about developer programs.
- **Collector's Net (http://www.ckts.info)** This is an inventive group of old Bell System workers and telephone system collectors who have hooked together their antique equipment using Asterisk. Not as much Asterisk stuff here, but a cool enough group of people that warrant a mention, and it shows that Asterisk can be used to do almost anything.
- **Bell's Mind (http://www.bellsmind.net)** A project that provides information regarding various telephone systems, and a PBX for public use.
- **Telephreak (http://www.telephreak.org)** Telephreak is a free voice-mail and conferencing service run *for* phone phreaks and computer hackers *by* phone phreaks and computer hackers.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is Asterisk?

A: Asterisk is an open-source PBX. Built by Digium Incorporated and developers across the globe, it is at the forefront of VoIP usage.

Q: How much does Asterisk cost?

A: While Asterisk itself is completely free, the cost of a complete install depends greatly upon your existing installation, what you want to use Asterisk for, and what kind of hardware you are willing to invest in. As always, your mileage may vary.

Q: I currently have a PBX, what advantage is there for me to move to Asterisk?

A: Asterisk has a lot of features that your current PBX likely does not have. It also has numerous advantages over a "traditional" PBX, such as the support of open standards, not being tied down to a specific vendor, and the common advantages of being open source.

Q: Do I need to move to VoIP to use Asterisk?

A: No. Asterisk supports numerous hardware devices, allowing you to use both analog phones and analog telephone lines with the system.

Q: What companies can most benefit from Asterisk?

A: There is no right kind of company for an Asterisk setup. Safe to say, if you have a PBX already, you can, and probably should, run Asterisk.

Chapter 2

Setting Up Asterisk

Solutions in this chapter:

- **Choosing Your Hardware**
- **Installing Asterisk**
- **Starting and Using Asterisk**

Related Chapters: 3, 7

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

Setting up and installing any kind of PBX server isn't easy. Adding Asterisk to the mix does simplify some areas, but further complicates others. Asterisk is flexible, but this flexibility creates many options that can overwhelm a novice. Everything from picking out a server, picking a phone setup, to picking an install method can leave you in awe of the options available. Let's not sugarcoat it: Asterisk is hard.

Choosing hardware is a key decision and not one that can be taken lightly, because if something goes wrong with the server or the phones, productivity is lost. Making the proper decision on a server, choosing phones for the users, and selecting the network configuration can mean the difference between a happy user base and a group of angry users outside your office with pitchforks and torches.

Even choosing a method to install Asterisk is filled with options, such as Live CDs, Asterisk Linux distributions, binaries for your operating system, or compiling from scratch. And there is no "correct" option either. Each method has benefits and drawbacks, and each one suits certain situations differently than others. Making sure you choose the right method of installing can save you a lot of heartburn later.

If you're scared right now, don't be. While Asterisk isn't easy, it is nowhere near impossible. While Asterisk may have a high learning curve, once you become familiar with its intricacies, everything suddenly starts to make sense.

Choosing Your Hardware

One of the first things to do when setting up Asterisk is to figure out your hardware needs. Hardware is a bit of a catch-all term and refers to the server, the phones, and the connections between them. There is no standard ratio for Asterisk that dictates "To support *A* calls over a *B* period of time, you need a server with *X* megabytes of RAM, a processor faster than *Y*, and a hard drive bigger than *Z*" or that "If you are in a call-center environment, *X* brand phones is the best choice." To figure out what is the correct fit for your situation, research is required.

Picking the Right Server

Picking the right server is a key decision when running Asterisk. The last thing a company wants to hear is that their phone system is down. Asterisk *can* run on obsolete hardware, but you will get what you pay for. Reliable, capable equipment is the foundation for any reliable, capable PBX system.

Processor Speed

Processor speed is the most important feature when looking at a server to run Asterisk. The more processing power, the more responsive the system will be when it is placed under heavy call loads. Asterisk runs well on any modern processor, handling moderate call loads without any issue. However, this does depend on how the system is configured to handle calls.

Transcoding and Protocol Translation

Transcoding is when the server is handling a conversation that is coming in with one codec and coverts it on-the-fly to another. This happens a lot more than thought, as most VoIP telephones transmit in μ-Law, which is the standard codec for telephone conversations. If the server is using the GSM codec for outbound calls, it needs to "transcode" the conversation and convert it from μ-Law to GSM. This, by itself, is pretty simple; however, when the server starts having to transcode multiple conversations simultaneously, more processing time is required. If a performance bottleneck develops, the conversations will start to exhibit delays in the conversation, more commonly referred to as "lag."

Protocol translation is the same problem as transcoding, except instead of converting the audio codec, it needs to translate the protocol used. This is also common with VoIP providers who only offer access to their networks via specific protocols.

RAM

RAM usage on Asterisk is pretty low. Asterisk can easily fit within a 64MB footprint even on a fairly large install. Since Asterisk is modular, trimming RAM consumption is as easy as removing modules from the startup sequence. A bare bones Asterisk startup can fit within a memory footprint of fewer than 30MB.

Storage Space

Storage space is probably one of the least important choices when choosing a server for Asterisk. Hard drives keep getting larger and cheaper with each passing month, allowing even a low-end computer to have massive amounts of space. Asterisk, by itself, hardly takes up any room; however, when voice prompts for Interactive Voice Response (IVR) menus and voice mail start being added to the system, Asterisk's footprint starts growing. Hard drive size needs to be determined by the amount of users on the system and the amount of voice mail expected.

For example, a sound file encoded with μ-Law takes up about a megabyte a minute. While this may not seem like a lot at first, consider that a person can average about five voice mails a day in a busy office. If each of those messages is about a minute each, and there are 100 people in the office, that's 500MB of storage per day! When you calculate the math per year, we're talking almost 13GB! Plus, other factors exist as well. Let's say a team leader sends a five-minute group message to his or her team of ten people. That 5MB message just copied across the system into ten separate mailboxes consumes 50MB. Also, don't forget to factor in saved messages, people on extended absences, and group mailboxes that may be accessed by the public.

Asterisk, like any high-demand server application, benefits from Redundant Arrays of Independent Disks (RAIDs). RAIDs are very important in any kind of high-availability environment. They are a system in which multiple disks are grouped together in a redundant fashion, allowing the computer to write data across all the disks at once. The upside of this is that it allows for one disk to fail within the group but let the computer still function. Using a RAID allows Asterisk to continue to handle phone calls and voice mails despite one of the server's hard drives no longer functioning.

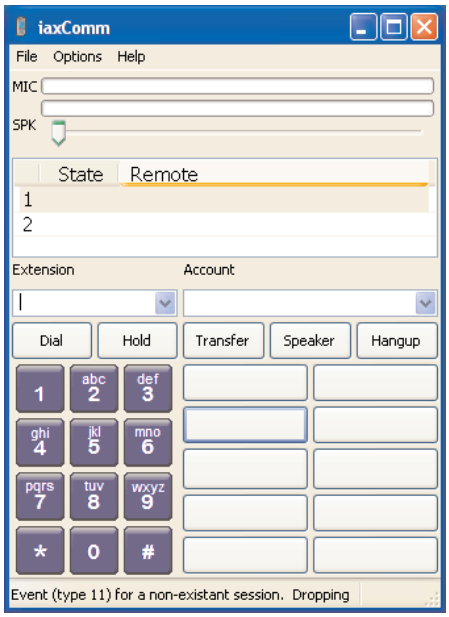
Picking the Right Phones

Phones are arguably the most important part of a PBX setup. This is how most users interface with the PBX system. Picking the proper phone is key to a successful PBX deployment. There have been instances where users were ready to give up on Asterisk solely because they hated their phones. Thankfully, changing phones is easy and these users quickly changed their opinions once new phones were installed.

Soft Phones

The easiest phone to set up with Asterisk is a soft phone. A soft phone is a computer program that emulates a phone on your PC. Soft phones are easy to set up and can be configured in a matter of minutes. They're usually very easy to use, often displaying a telephone-like interface on the screen. Soft phones utilize the computer's sound card for transmitting and receiving audio, or optionally a "USB phone," which is a phone-like device that plugs into the computer's USB port. Soft phones are inexpensive (often free) and USB phones generally cost less than $50. Figure 2.1 shows iaxComm.

Figure 2.1 iaxComm, an IAX2-Compatible Soft Phone



Soft phones have the advantage in price and ease of setup and configuration, but that's about it. It's common to see people preferring some kind of physical device rather than a program that runs on a PC. USB phones sometimes can help, but they usually aren't geared for a business environment. In addition, these users are tethered to a PC. If the PC crashes, no phone calls.

Soft phones are handy though if a user wants to make VoIP calls while on the road without wanting to lug another device with them. Simply install and configure the soft phone on the user's laptop with a headset and they're ready to go—all they need is an Internet connection. However, soft phones are not fit for most tasks common to a business environment.

Hard Phones

The alternatives to soft phones are hard phones—the phones we've used the past 125+ years: a physical device that sends and receives telephone calls. Hard phones are on the opposite side of the spectrum from soft phones: they're expensive and often harder to set up than their software counterparts. However, most users prefer a hard phone; it's what they're accustomed to.

The most common hard phones include IP phones: analog phones connected to an Analog Terminal Adapter (ATA) and analog phones connected via interface cards.

Each of these has their advantages and disadvantages, which we'll discuss in the following sections.

IP Phones

IP phones are one of the most common solutions you'll see for VoIP in a business environment. They plug in to an Ethernet connection and emulate a regular analog phone. They're made by numerous companies, including Cisco Systems, Polycom, Aastra, and Siemens, just to name a few. The price and quality of these phones run the gamut, but the general rule of "you get what you pay for" applies here. In today's market, a good IP phone will cost you at least $150 per unit, like the Cisco 7960 IP Phone shown in Figure 2.2.

Figure 2.2 A Cisco 7960 IP Phone

Analog Telephone Adapters

ATAs are the bridge between the world of analog telephones and the world of VoIP. They are small devices, usually in the form of a small plastic cube, with a power port, one or more telephone jacks, and an Ethernet port. An analog phone connected through an ATA can participate in phone calls on a VoIP network.

ATAs are cheaper then IP phones, mainly because they are slightly simpler. ATAs are often offered by the same companies that make IP phones and range in price from $50 to $100 depending on the protocols they support, the number of ports, and, of course, the number of features. Some ATAs have both a port for a phone and a port for an outside phone line, allowing a quick and easy way to interface Asterisk with both your phone and the public switched telephone network.

ATAs work with most phones, the exceptions being proprietary phones from digital PBXs and older rotary dial phones. Digital phones are nearly impossible to support due to their complexity and the differences between one manufacturer and another. Rotary phones aren't supported by most ATAs because most developers consider, somewhat correctly, that pulse dialing is an obsolete protocol. Figure 2.3 shows a D-Link analog telephone adapter controlling two older analog phones.

Figure 2.3 A D-Link Analog Telephone Adapter Controlling Two Older Analog Phones

Interface Cards

Analog phones do not always need an ATA. Asterisk supports multiple interface cards that allow analog phones to connect directly to an internal port on the server. Digium sells numerous cards supported by its Zaptel drivers. These cards support anywhere from 1 to 96 phones depending on how they are configured. There are also other cards that support anywhere from a single phone line to an entire PRI.

PRIs can be attached to a device called a "channel bank," which will split the PRI's 24 channels into 24 separate interfaces, allowing a single interface card to support up to 24 phones. Cards also aren't limited to a single PRI interface, either. And some cards out there can support four simultaneous PRIs.

Digium also sells cards that sustain up to four modular sockets that can either support telephone lines or telephones depending on the modules purchased. While these are rather pricey, they are cheaper than PRI cards and will allow you to avoid purchasing a channel bank on top of a card.

Sadly, interface cards do not support digital phones either. Another issue when considering these is that there needs to be wiring run between the phones and the cards, which can be difficult in an existing server setup. The good news is that most of these cards support pulse dialing, allowing older equipment to interface into the system.

Configuring Your Network

A network is like a car. You can use it every day and not notice it until the day it breaks down. This is even truer when the network is also the phone system's backbone. For most folks, phone service is much more important than Internet access.

When looking at it from a network management standpoint, VoIP conversations using the μ-Law codec are 8KB/s data transfers that run for the duration of the calls. While this amount of traffic is negligible if designing a network for an office of ten people, it starts to add up quickly when designing the network for a voicemail server serving 10,000 people. For example, if there are 2500 simultaneous phone calls connecting to and from the server, that would be a constant stream of 20 megaBYTES per second being transferred across the network.

When designing networks for VoIP, virtual local area networks (VLANs) are a big help. VLANs are a software feature in networking switches that allow managers to set up virtual partitions inside the network. For example, you can set up a switch to have even-numbered ports on VLAN A and odd-numbered ports on VLAN B. When

plugging networking equipment into the switch, equipment on VLAN A won't be able to connect to equipment on VLAN B, and vice versa, allowing the two VLANs to be independent of one another. VLANs help immensely for a VoIP network since they keep voice and data traffic separate from each other. The last thing you want is a giant multicast session DoS-ing your phones. By keeping the computers on separate VLANs, computer traffic will not interfere with voice traffic, allowing a user to make a large file transfer and not see any degradation of the voice quality on their phone.

Notes from the Underground…

Who's Listening to Your Phone Calls?

VLANs not only help immensely with traffic management, but also with security. Much like how attackers can sniff your existing traffic via ARP poisoning and other attacks, they can do the same with your VoIP traffic. Automated tools such as VoMIT and Cain and Abel allow attackers to sniff and record all voice traffic they intercept.

The most secure solution to this is to set up a second Ethernet network or VLAN on your network and limit the connections to the phones only. While this is not a completely foolproof solution, since attackers on the network can spoof MAC addresses, thus bypassing the restrictions, this will keep random script kiddies from recording the boss's phone calls to his mistress.

WAN links are another part of the chain. WAN links can vary from a simple DSL connection to a massive Optical Carrier connection, but they each have something in common, they are a link to the outside world. When thinking about setting up a WAN connection or making changes to your current one, you need to figure out what the current bandwidth consumption is, and how much more bandwidth will be consumed by adding VoIP to the equation. If the link's free bandwidth during lunch is under 100Kb/s, it will be able to support one μ-Law encoded VoIP call during that timeframe without running into issues. If there are usually five simultaneous telephone calls during that timeframe, that's a major issue.

While with WANs it's impossible to have a VLAN, it is possible to shape the bandwidth. Bandwidth shaping is when a device, called the bandwidth shaper, gives certain traffic priority over others. Numerous ways exist to do this, the most common being to dedicate a portion of bandwidth exclusively to VoIP, or giving pri-

ority to VoIP traffic. Each has their pluses and minuses: dedicating a portion of the bandwidth to VoIP allows you to guarantee there will always be a set amount of bandwidth for telephone calls. While this may seem desirable, this is inefficient; if there is no voice traffic but the data portion is at 100-percent utilization, the voice portion will sit idle while the data portion suffers. The alternative, giving priority to VoIP traffic, allows the WAN link to fluctuate how much bandwidth is being used for data and how much is being used for voice. This allows data to use 100 percent of the bandwidth if there is no voice traffic, but still permits voice traffic to get through if the need arises. This is accomplished by letting the bandwidth shaper dynamically allocate bandwidth for the voice traffic when a conversation starts: if a voice packet and a data packet reach the bandwidth shaper at the same time, the bandwidth shaper gives the priority to the voice packet over the data packet. This does have a downside though: in certain shaping schemes, if voice packets keep reaching the bandwidth shaper faster than it can send data packets, the data packets will take longer and longer to get through. This will result in the data connections timing out and failing.

Installing Asterisk

So, you've purchased your server, installed an operating system, and you're ready to plunge head first into Asterisk. Determining the "right way" to install Asterisk depends on your situation. If you just want to try Asterisk out and are worried about messing up an existing system, the Live CD would likely be your best route. If you are not too familiar with Linux installation, but are looking to set up a dedicated Asterisk system, you may want to look at a CD distribution of Asterisk. If you are an experienced Linux administrator and want to configure Asterisk to fit into a custom environment, you'll likely just want to compile it from scratch. Finally, if you are either a Mac OS X or Microsoft Windows user, and you just want to use your existing operating system for an Asterisk install, you'll likely just want to use the packages for your operating system.

Asterisk's ability to be customized isn't just limited to the final setup; it starts at the installation phase of the system. You can easily make it fit almost any environment.

Using an Asterisk Live CD

Live CDs are bootable CDs that contain a complete operating system. After booting, your machine will run the operating system from the CD without installing it to the hard drive. If something goes wrong, you can turn off the computer, eject the CD,

reboot, and boot back to the operating system installed on your hard drive. Although this installation method is not recommended for most production environments, it is a perfect way for a novice user to try out the features of an operating system without altering the boot machine in any way. In this section, we'll take a look at one of the more popular live Asterisk CDs: SLAST.

SLAST

SLAST (SLax ASTerisk) is an Asterisk-ready version of the Slackware-based SLAX Live CD. Maintained by the Infonomicon Computer Club, SLAST was designed to help educate people about the advantages of Asterisk and allow them to set up a simple Asterisk server in the easiest way possible.

Getting SLAST

SLAST is available at http://slast.org. The ISO image is available from their download page. The download size comes in at just a bit over 100MB, so any broadband connection should make quick work of the download. Once the ISO is downloaded, the disk image can be burned to a CD using the "image burn" feature of most popular CD recording programs.

Booting SLAST

Booting SLAST is as simple as inserting the CD into an Intel-based machine, and rebooting. Depending on how your machine is configured, you may need to press a key during startup to instruct the machine to boot from a CD. Once the CD is booted, the SLAST screen is displayed, as shown in Figure 2.4.

Once SLAST loads the system into memory, the login screen is displayed. The login screen has a quick "cheat sheet" of sorts showing file locations of Asterisk configuration files, Asterisk sounds, Asterisk modules and the SLAST documentation. The root password is also displayed. Log in with the username *root* and the password *toor*, and you will be presented with a root shell, as shown in Figure 2.5.

Figure 2.4 The SLAST Splash Screen Booting SLAST

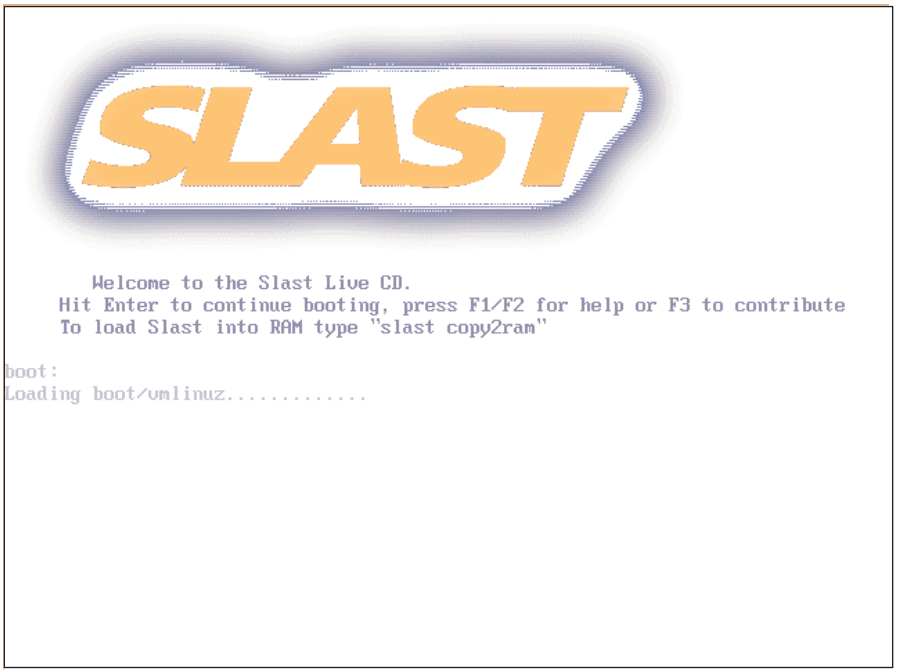
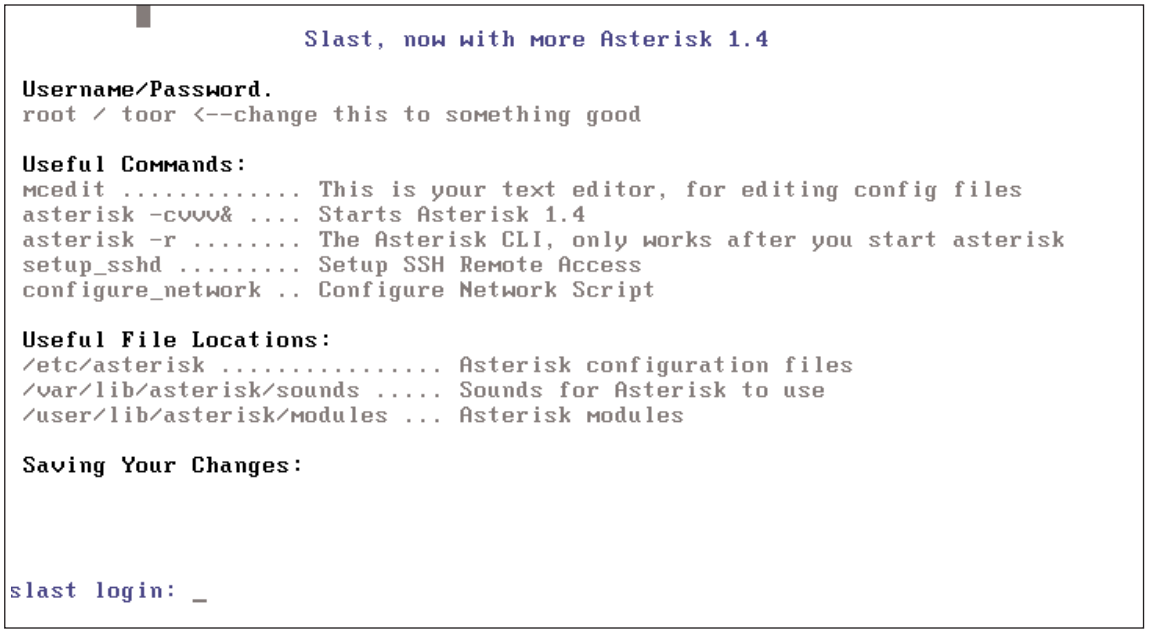


Figure 2.5 The SLAST Login Screen



Configuring the Network

While a network connection isn't specifically required for Asterisk, unless the target system has hardware to connect it directly to a phone, some kind of network connection will likely be necessary if you want to connect to something besides the local computer. SLAST, as with most live distributions, does a pretty good job at detecting any and all hardware on the target system. If everything is plugged in and turned on, SLAST should have no issues setting up the hardware. However, SLAST, like other Live CDs, may have trouble detecting networks settings. If you're running a DHCP server, Asterisk should automatically configure your settings. However, if manual intervention is required to configure these settings, you may need to rely on *ifconfig*, the InterFace Configurator.

Running *ifconfig* without any arguments will display any configured network interfaces on the system. Ethernet interfaces will be shown labeled by their abbreviations *ethX*, where X is a number starting at 0 for the first interface. Next to the name will be fields for the IP address labeled as "inet addr," the broadcast address labeled as "Bcast," the network mask labeled as "Mask," along with various statistics regarding the interface. See Figure 2.6.

Figure 2.6 Running the ifconfig Utility to See Your Configured Network Interfaces

```
slast login: root
Password: ****

root@slast:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:95:4D:4E
          inet addr:192.168.248.132  Bcast:192.168.248.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:808 (808.0 b)  TX bytes:1240 (1.2 KiB)
          Interrupt:11 Base address:0x1400

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

root@slast:~# _
```

If the Ethernet connection is not displayed when running *ifconfig* without arguments, it is either not configured, or it has not been detected on your system. To determine this, run the command *ifconfig eth0*. This will show the first Ethernet interface on the system, configured or not. If no text is displayed, SLAST has not found your Ethernet card and it will need to be manually set up. However, if text is displayed similar to the preceding figure, but missing the text regarding the IP address, the Ethernet interface is set up, just not configured with an address. SLAST provides a script to perform this configuration.

The *configure_network* script allows the system's network interface to be configured with minimal user interaction. The user can run the script by entering *configure_network* at the prompt and pressing **Enter**. The script will execute, prompting you for information regarding your desired network configuration, as shown in Figure 2.7.

Figure 2.7 Running the *configure_network* Script to Configure Your Network

```
root@slast:~# configure_network

Configure Networking on Slast
-----------------------------

What network device do you want to configure?

eth0

What is the ip address you want to use?

192.168.248.42

What is your subnet mask?

255.255.255.0

What is your gateway IP?

192.168.248.1

What is your primary DNS Server?

4.2.2.1_
```

The *configure_network* script first prompts for the name of the interface you are looking to configure. This will most likely be the first Ethernet interface, eth0. In case the system has multiple Ethernet interfaces, this could be eth1 or eth2, depending on which card was detected first and how many Ethernet interfaces are installed. After entering the desired interface, *configure_network* will prompt you for the desired IP address, followed by the network's subnet mask. These are very important

to configure correctly since entering incorrect values would at best cause the system to be unable to access the network, and at worst cause the entire network to be taken offline! The next piece of information *configure_network* needs is the network's gateway IP address. If the system is on a standalone network—that is, a network without a connection to the Internet—leave this blank. Finally, *configure_network* will prompt for the network's DNS primary and secondary DNS servers.

After entering the entire network configuration, the script will prompt you to confirm all the settings entered. If the network settings are correct, the script will apply the changes. Otherwise, it will return you to the root prompt. This script can be run later, allowing you to change any of the information, and it can be aborted at any time by pressing **Ctrl + C**.

Saving Your Changes

One major advantage of a Live CD is that they do not make any permanent changes to your system, allowing you to undo any changes simply by rebooting your computer. This, while handy if you mess something up, can become a problem in certain situations: if the computer restarts for any reason, all the configuration changes are lost. SLAST, because it is based on SLAX, has two utilities that address this problem: *configsave* and *configrestore*. These utilities allow a user to back up and restore any changes they made. One of the more interesting ways to save the changes is to do so to a USB memory stick. This way, you can easily carry around the bootable CD and any configuration changes made to it, allowing you to essentially take your Asterisk server with you in your pocket.

To save your configuration changes, use the command *configsave*, followed by the name of a file to save to. For example, to save to a USB memory stick, run the command *configsave /mnt/sda1/asteriskconfigs.mo*. SLAST will then save any changed file from the /var, /etc, /home, and /root directories.

To restore your changes, use *configrestore* with the same syntax. If you saved your configurations to a USB memory stick, as in the preceding example, you can restore them by booting SLAST, inserting the memory stick, and then running *configrestore /mnt/sda1/asteriskconfigs.mo*. This will restore the files saved in the file. Remember, after you restore your files, if you make changes, you will need to run *configsave* again.

Installing Asterisk from a CD

Four Linux distributions focus on Asterisk: PoundKey, a Linux distribution supported by Digium; Evolution PBX, a distribution made for small businesses with commercial support; Elastix, a distribution supported by a commercial company; and trixbox.

trixbox was released in 2005 as "Asterisk@Home," a simple and easy way to install Asterisk on a computer. Self contained within a bootable CD, Asterisk@Home focused on ease of use and ease of install, allowing someone with little to no Linux experience to start playing with Asterisk. In 2006, Asterisk@Home was acquired by Fonality, a California-based VoIP services firm, who renamed the new version of Asterisk@Home to "trixbox." Today, trixbox is one of the leading Asterisk Linux distributions. With over 30,000 downloads a month, it takes its place among the "heavy hitters" of Asterisk distributions.

The trixbox CD contains numerous add-ons to Asterisk: freePBX, a Web-based configuration manager; HUDLite, a cross-platform operator panel; and SugarCRM, a complete Customer Relationship Manager suite. All of these are configured to run out of the box with trixbox, allowing a complete suite of tools for managing and maintaining your Asterisk installation.

Getting trixbox

trixbox is available at www.trixbox.org. The most up-to-date version at the time of this writing is trixbox 2.0 which contains Asterisk 1.2.13. The download size clocks in at a hefty 550MB, so you may want to put on a pot of coffee before you start downloading. Like the live CD's discussed earlier, the downloaded image can be burned with the "image burn" function of any standard CD recording program.

Tools & Traps…

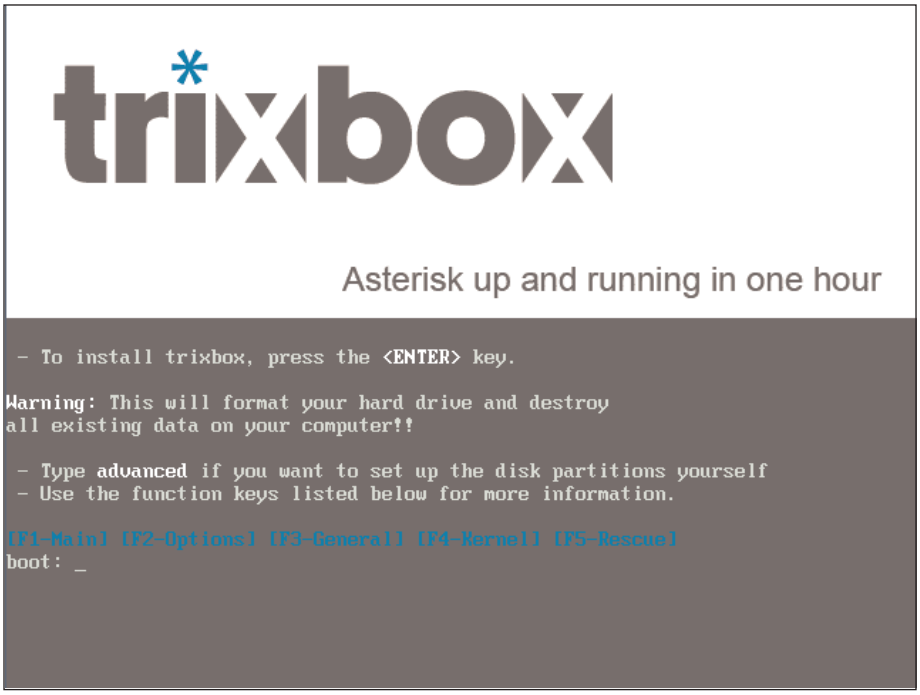
Getting Messed Up by Old Asterisk Versions

It's common to think "Hmmmm… You know, I don't NEED the latest version of Asterisk" if you're looking at installing it from a binary package or an installer CD. However, watch out. Sometimes the differences between the versions are pretty big, and while what this book covers will work in Asterisk 1.4, it may not work in earlier versions.

Booting trixbox

After burning the trixbox CD, use it to boot the machine you will be installing to. Again, as mentioned in the earlier "Booting SLAST" section, the computer may need some kind of setting changed to boot from a CD. Once the CD is booted, the trixbox boot screen is displayed, as shown in Figure 2.8.

Figure 2.8 trixbox Booting



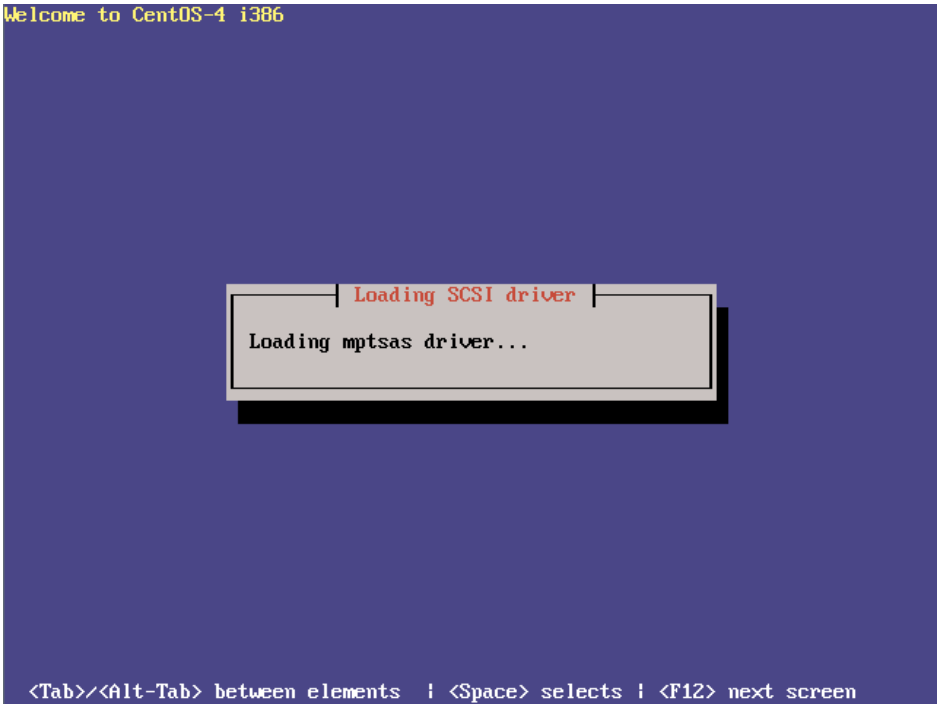
Tools & Traps…

Behold trixbox, Destroyer of Data

The trixbox CD is an Installer CD, not a Live CD. Installing trixbox onto a system will wipe out all existing data. If you are using a current system, it would be wise to make sure it has no data you want to keep, or that you have good backups of that data. The alternative is to use someone else's system, preferably someone you do not like.

After about five seconds, the CentOS installer will start loading up, as shown in Figure 2.9.

Figure 2.9 Anaconda, the CentOS Installer, Loading Drivers for SCSI Hardware



After all the system's hardware is detected, the installer will start prompting you for questions regarding keyboard layouts and time zones. Answer these as appropriate to your system. Once done with that, it will prompt you for a root password. Once enough information is gathered, the installer will start formatting your hard drive and the installation will begin, as shown in Figure 2.10.

Tools & Traps…

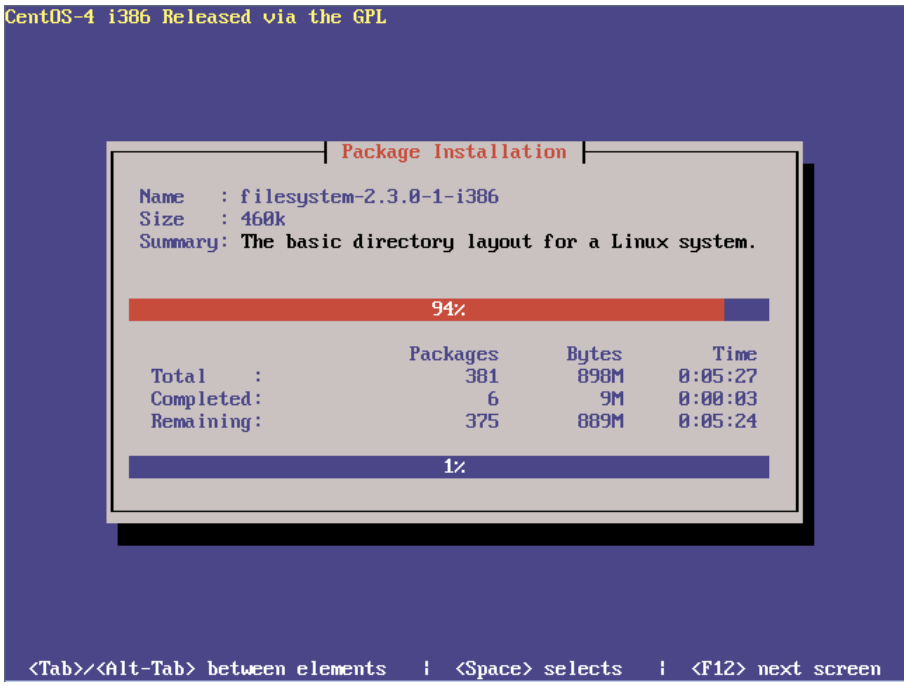
Excuse Me… Your Users Are Showing….

trixbox, allows Secure Shell (SSH) by default. This by itself is not much of a security issue, but root access is allowed from remote terminals. This means that if your trixbox system is publicly accessible on the Internet, anyone can

Continued

log in to your system if they guess your root password. This may seem unlikely, but it's common for script kiddies to scan entire networks looking for badly configured servers that allow root access and have common root passwords. So, either have an excellent root password, keep your system behind a firewall that disallows inbound port 22 traffic, or read up on how to disable root logins via SSH.

Figure 2.10 trixbox Installing CentOS Packages to the System



The trixbox installer will copy files, reboot, and begin to install specific packages on the system (see Figure 2.11).

After installation, trixbox will reboot one last time and display a login prompt. Log in with the username *root* and the password you specified in the setup process and you will be presented with a root shell. After logging in, the URL of the Web management interface will be displayed, as shown in Figure 2.12.

Figure 2.11 trixbox Installing the trixbox Packages

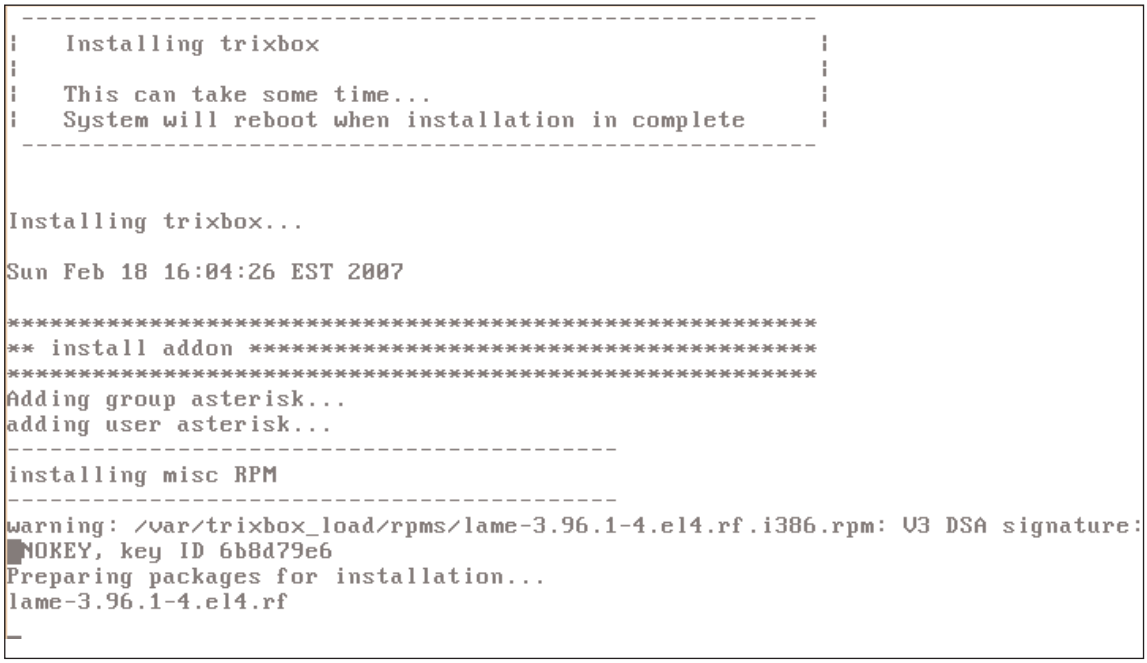
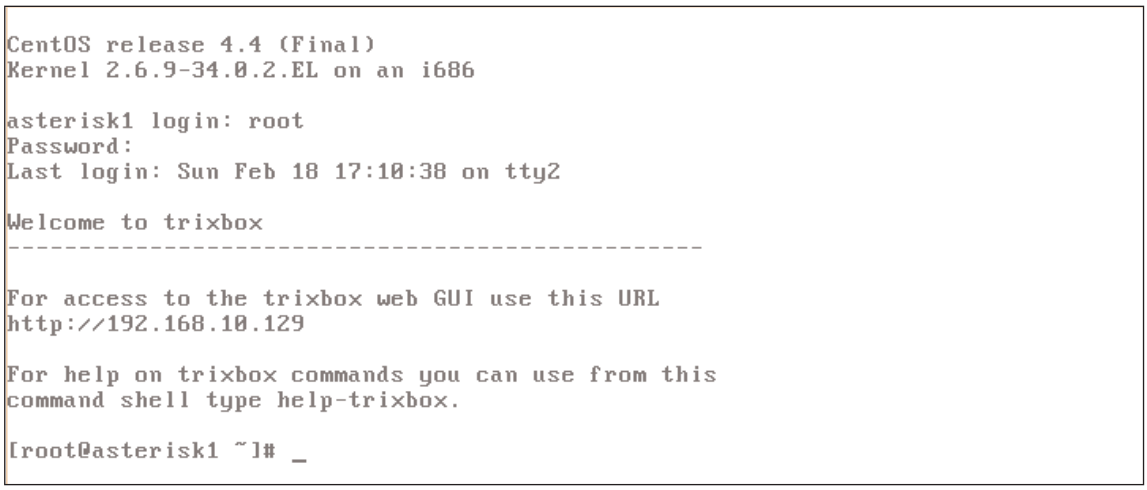


Figure 2.12 Logging In to trixbox

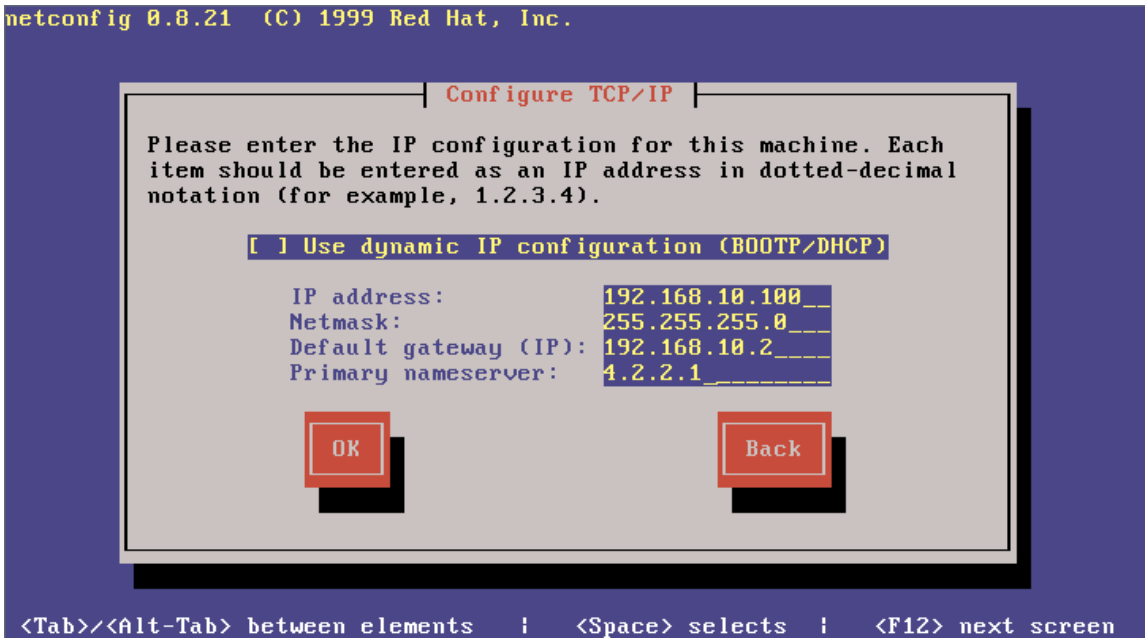


Configuring trixbox

trixbox, like SLAST, should configure its network automatically if there is a DHCP server on the network. If it didn't, or if the DHCP address is not the address you

want for the server, you can run the *netconfig* utility to manage network settings, as shown in Figure 2.13.

Figure 2.13 The Main netconfig Screen



netconfig will prompt you for the IP address, netmask, gateway, and nameserver of your network. Enter these as appropriate for the system. After confirming these settings, the utility will exit. Reboot the system, and the new network settings will take effect.

trixbox's Web Interface

One of trixbox's nicer features is a Web interface that allows you to manage the system through a Web browser. It uses PHPConfig Asterisk config editor, which allows you to edit the files directly, in addition to using freePBX, which is a standardized interface for managing certain Asterisk features.

Tools & Traps…

The Danger with Frameworks

freePBX is an amazing system for simplifying the Asterisk configuration process. However, as with any framework, you are constrained by what the framework supports. Trying to go beyond what the framework supports is often a tedious process. So, while freePBX lowers the bar for learning Asterisk, you can grow out of it quickly.

By entering the system's IP address into your Web browser, you'll be greeted with trixbox's home page. You'll see links for the system's Asterisk Recording Interface which manage the ability to record audio conversations on Asterisk, scripts to manage Asterisk's recordings, voice mail, and call monitoring recordings; the MeetMe management system, a system to manage MeetMe conferences; Flash Operator Panel, a phone operator panel for Asterisk written in Flash; and SugarCRM customer relationship management software. In the upper right, you'll see a link to switch into "Maintenance" mode. Clicking the link will prompt you for a username and password. Log in with the username **maint** and the password **password**.

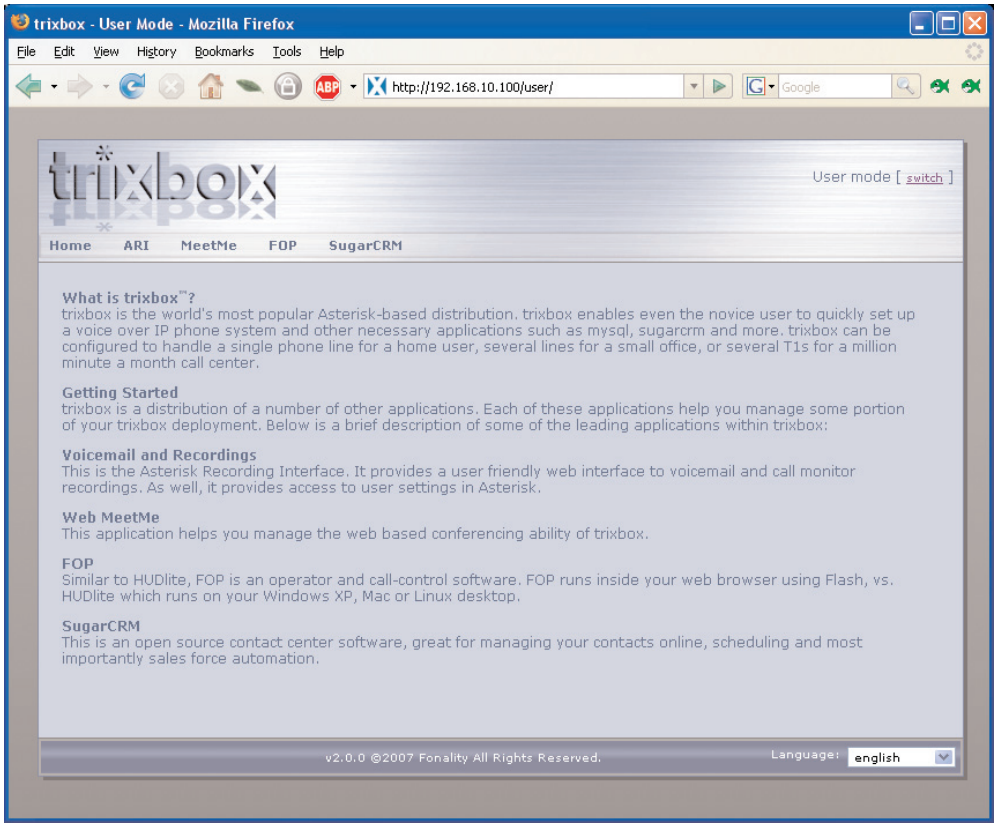
Tools & Traps…

I See What You Did There…

trixbox doesn't use an SSL-encrypted Web session when maintaining the system. This means anyone sniffing the network can see exactly what you are doing on the Web page, including any usernames and passwords you may enter.

The trixbox management system is very full featured, and a book could be written on these two systems alone, so let's just take a (very) quick tour of the two major configuration editors on the system: The PHPConfig Asterisk config editor and the freePBX system. Figure 2.14 shows the trixbox system default page.

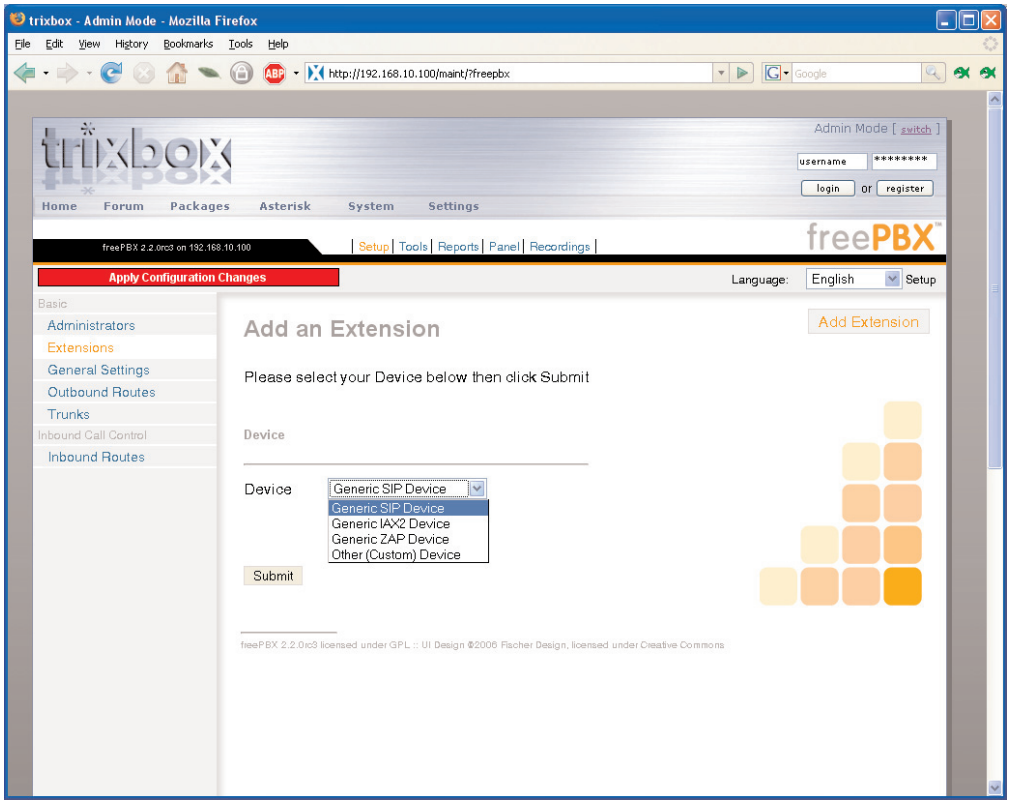
Figure 2.14 The trixbox System Default Page



freePBX

freePBX is accessed by clicking the *Asterisk* link of the main menu, and then clicking the *freePBX* link. freePBX will greet you with a welcome screen and a list of menu options on the top. From here you can access the setup options, system tools, call activity reports, Flash Operator Panel, and the Asterisk recording interface. Clicking *Setup* will take you to the setup main page. The main page has a list of options on the left, which will allow you to administer user accounts, extensions, and general Asterisk settings; configure dial plans; and set up and control inbound and outbound trunks. See Figure 2.15.

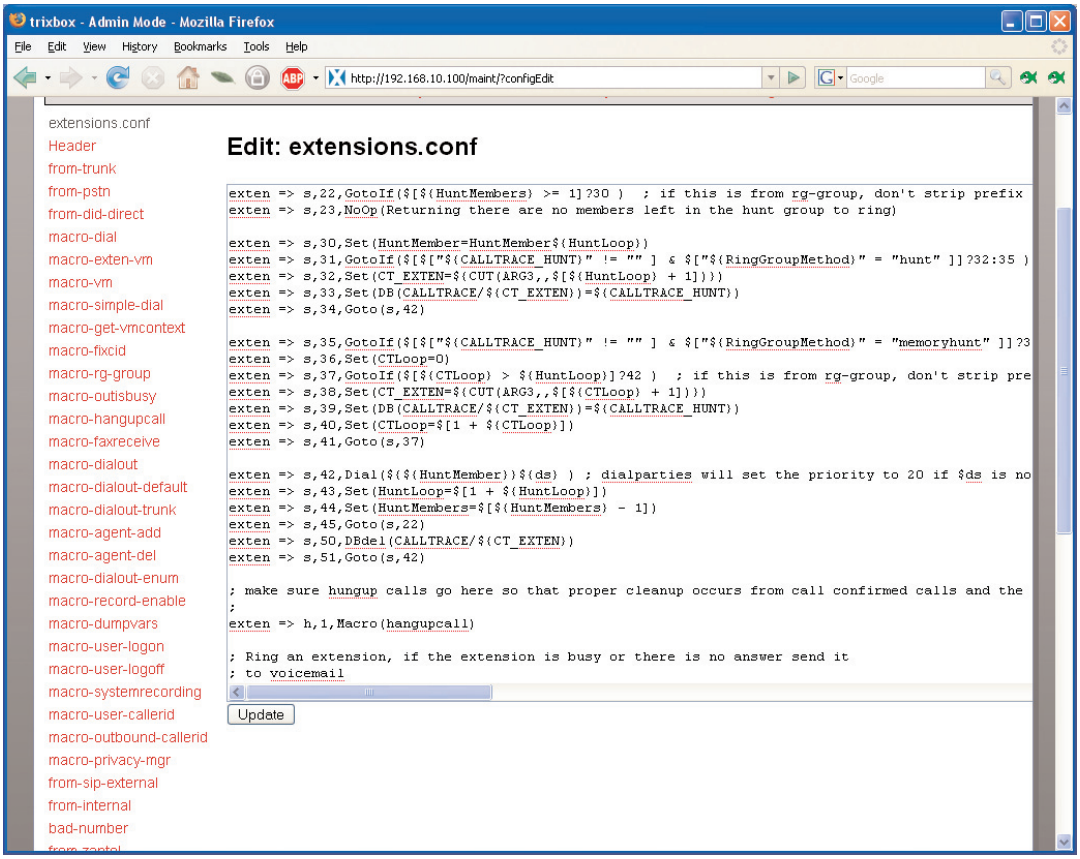
Figure 2.15 Setting Up an SIP Account in freePBX



PHPConfig

PHPConfig is a great way to edit configuration files without having to deal with a shell terminal. It allows you to edit files just like they were in a text editor, but without having to learn how to use a Linux shell. It provides the best of both worlds. PHPConfig can be accessed by clicking the Asterisk link on the maintenance home page and then clicking the Config Edit link. Afterward, PHPConfig lists all the files in the Asterisk configuration directory. Clicking the name of one of these files brings the file up in an edit window. To the left of the edit window, PHPConfig lists all the sections it reads from that file, allowing you to quickly jump to and edit the section you wish to work on. When finished editing, click the **Update** button below the edit window. PHPConfig will then write the file to disk. The changes are not immediately reflected in Asterisk though. To reload all the configs, you will need to click the **Re-Read Configs** link at the top of the page. This tells Asterisk to perform a "reload" command that will reload all the configuration files. If there are no errors, PHPConfig will then display "reset succeeded." See Figure 2.16.

Figure 2.16 Editing extensions.conf in PHPConfig



Installing Asterisk from Scratch

Before there were live CDs and distributions, there was source code. Asterisk's availability of source code is one of its biggest features, allowing anyone to "poke under the hood," see the internal workings, and rewrite portions if needed. Compiling Asterisk from its sources gives you the greatest amount of control as to what files are installed, and where they are installed. Unneeded options can be removed entirely, allowing a leaner Asterisk install. However, as always, there is a downside. Compiling anything from source is intimidating if you aren't used to doing it. However, it's terribly once you figure it out.

The Four Horsemen

When compiling Asterisk from source, there are four major pieces to the puzzle: LibPRI, Zaptel, Asterisk-Addons, and Asterisk.

Asterisk is, you guessed it, the PBX itself. This package contains the code for compiling the PBX and all its modules. You aren't going to get far compiling Asterisk without this package.

LibPRI is a library for handling the PRI signaling standard. The PRI standard was created by the Bell System back in the 1970s and is now an ITU standard. LibPRI is a C implementation of the standard. This package may be required depending on the hardware installed on the system.

Asterisk-Addons is a package that contains certain optional "bells and whistles," such as an MP3 player so Asterisk can handle sound files encoded in MP3, and modules for logging calls to a MySQL database. While these modules are completely optional, they are good to have, especially the MP3 player, and the resources they take up are minimal. Installing them is recommended.

Zaptel is the package that contains the driver and libraries for Asterisk to talk to Zapata telephony hardware, which are the telephone interface cards discussed earlier. This is a handy package to install, even if there is no Zaptel hardware on the system, since the conferencing software requires it for timing purposes.

Asterisk Dependencies

Before you start compiling Asterisk, you must make sure you have all the requirements satisfied. First off is the compiler. If you don't have a compiler like GNU C Compiler (gcc) installed, you aren't going to get very far compiling the source code. Next, make sure you have the libraries required to compile, otherwise you will likely have some kind of odd error at compile time. Asterisk has three dependencies: ncurses (www.gnu.org/software/ncurses/), a library for text-based "graphical" displays; OpenSSL (www.openssl.org/), an open-source library of the TLS and SSL protocols; and zlib (www.zlib.net/), a data compression library.

Asterisk requires both the library itself and the associated include files. These are included automatically if you compile from source. However, if you install the libraries from a binary repository, you will need to include the development packages as well. For instance, you would need to get both *zlib* and *zlib-devel*.

Getting the Code

Links to all of the Asterisk code are available at http://www.asterisk.org. Clicking the **Downloads** tab will take you to a page with links to grab all the necessary files. The links to get Asterisk provide options for downloading either Asterisk 1.2 or Asterisk 1.4 directly, or visiting the source archive. Grabbing Asterisk directly only downloads the Asterisk package, so you'll want to download the LibPRI, Zaptel, and Asterisk-Addons separately. The latest versions of each package should end in *-current*. Since there are multiple source archives, it is best to put all of them in a common subdirectory wherever the system's source code directory is located (for example: /usr/local/src/asterisk/). See Figure 2.17.

Figure 2.17 Getting the Source Archives via *wget*

Gentlemen, Start Your Compilers!

Compiling is simpler than one might think. Often, all that's required is three commands: *./configure*, *make*, and *make install*. Once you have these three commands memorized, you'll do fine.

Compiling LibPRI

The first step is to compile LibPRI. This is required if you have a PRI interface hooked into the system, but optional if you do not. First, expand the archive.

```
tar xvzf libpri-1.4-current.tar.gz
```

This will expand the source archive into a directory. At the time of this writing, it is `libpri-1.4.0/`. After the file is done expanding, change to the LibPRI directory.

```
cd libpri-1.4.0/
```

LibPRI doesn't have a configuration command yet, so the only two steps are to compile it via the *make* command, wait until it finishes, and then run *make install*.

It is important to run the *make install* command as a root user, otherwise the library will not be installed correctly due to permission errors. Once everything is done, you can exit the LibPRI directory.

```
cd ../
```

Compiling Zaptel

Compiling Zaptel more or less follows the same steps that compiling LibPRI did. However, there are a few changes. First though, expand the archive.

```
tar xvzf zaptel-1.4-current.tar.gz
```

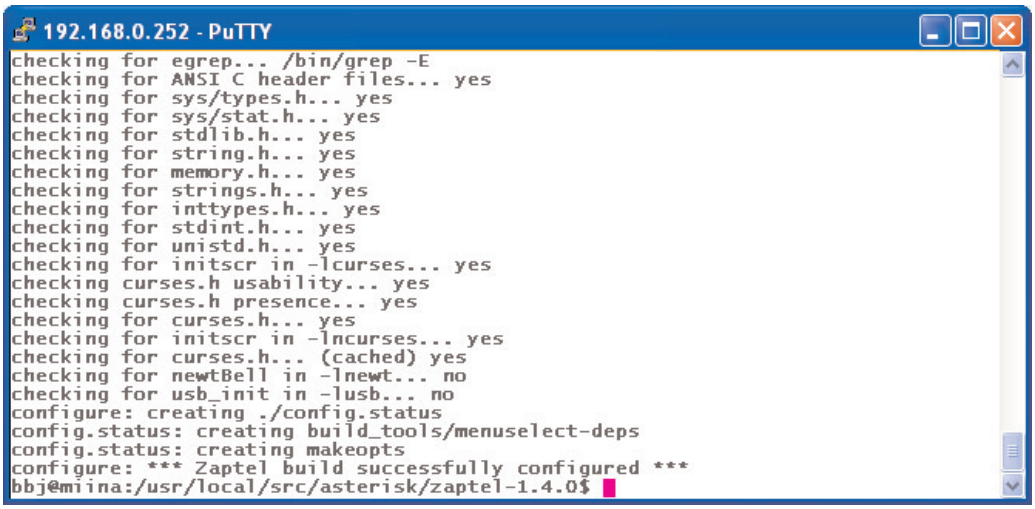
Next, enter the Zaptel directory:

```
cd zaptel-1.4.0/
```

This is where things change from LibPRI. Zaptel is a bit more complicated than LibPRI, so it includes a configuration script. (See Figure 2.18.) You can run this by executing

```
./configure
```

Figure 2.18 The Zaptel Configure Script

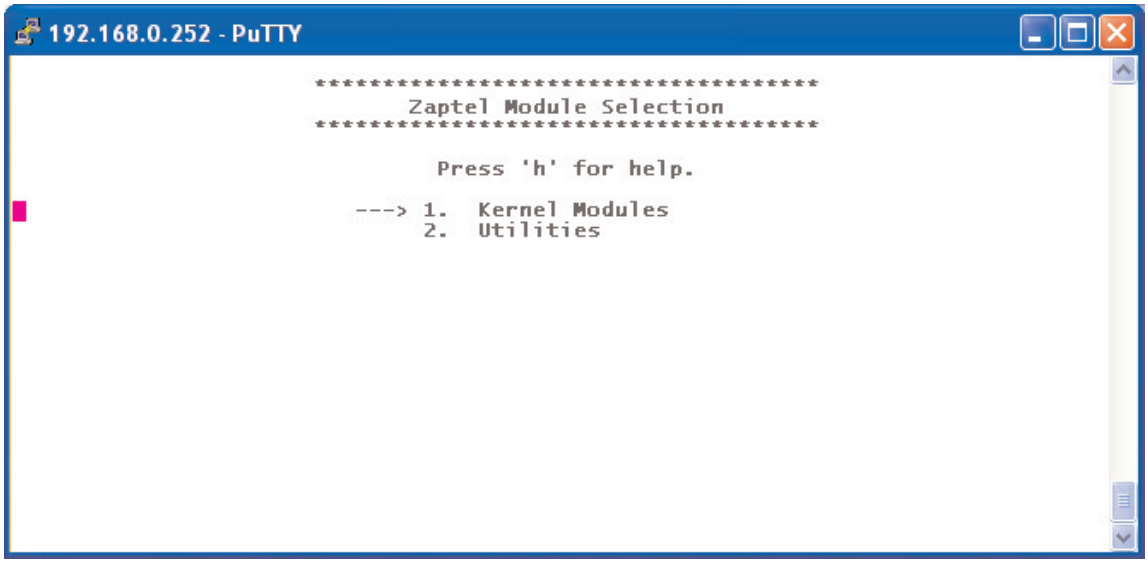


The configure script will make sure all the dependencies are fulfilled and that Zaptel knows where to look for all the libraries. Once the configure script is done, the next step is to run the following command:

```
make menuselect
```

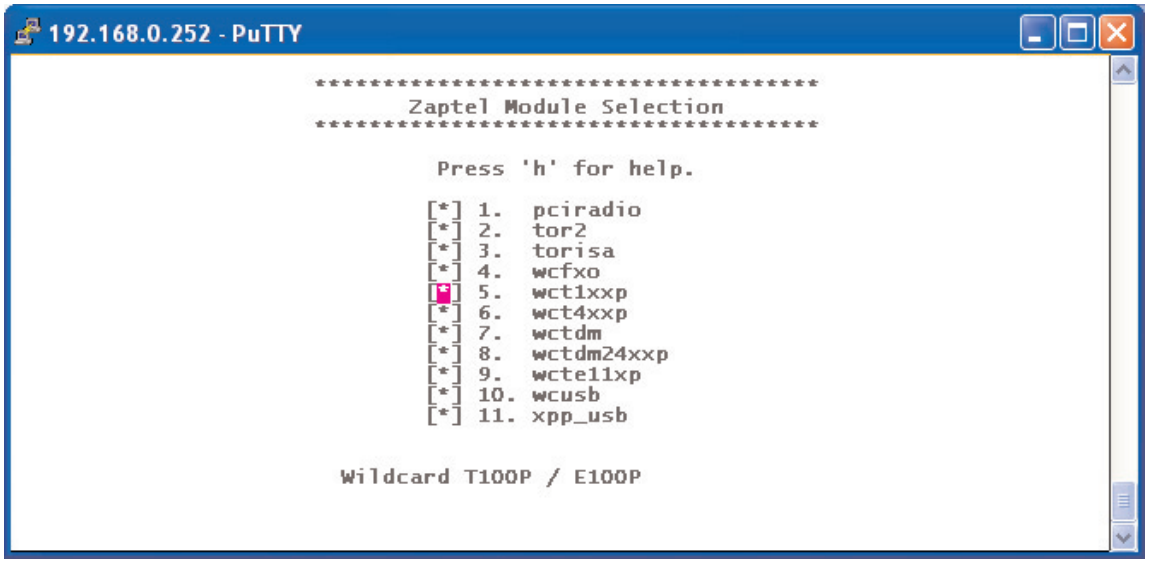
This will compile and execute the menuselect utility. menuselect is a new feature in Asterisk 1.4 that allows you to choose which modules to compile and install, permitting you to "trim the fat" of any software not required in your particular situation. For example, if you do not have a Digium TDM400, you can deselect the *wctdm* module during *menuselect* and that module will not be compiled or installed. See Figure 2.19.

Figure 2.19 The Initial Zaptel menuselect Menu



You can navigate through menuselect with the arrow keys—up and down scroll through the menu, left exits to the previous menu. Pressing **Enter** or the **Spacebar** will select/deselect a module or enter a menu. **F8** will select all the modules, and **F7** will deselect all the modules. To save and quit, press **x**, and to quit without saving, press **q**. If you forget any of the keys, press **h** and the help screen will be displayed, as shown in Figure 2.20.

Figure 2.20 The Zaptel Kernel Module Menu



Menuselect lists a description at the bottom of the screen that explains which module supports which card. You can safely deselect any cards your system does not have installed. If a dependency is broken, menuselect will inform you of this and allow you to correct the configuration.

Once you are done trimming modules from the menu, exit and save. This will bring you back to the shell. Next, compile the Zaptel modules. This is done in one of two ways. If the system is running a 2.4.X kernel, simply run:

```
make
```

However, if the system is running a 2.6.X kernel, run:

```
make linux26
```

After the modules are done compiling, regardless of the system kernel version, run the installation command as a root user:

```
make install
```

And so the Zaptel modules will install. Finally, once everything is done compiling, move back up to the asterisk subdirectory:

```
cd ../
```

Compiling Asterisk

Believe it or not, Asterisk is just as easy to compile as LibPRI and Zaptel. Despite the menuselect system being more complex and the compile taking a bit longer, compiling the code more or less follows the same process as Zaptel. First, expand the archive:

```
tar xvzf asterisk-1.4-current.tar.gz
```

Next, enter the Asterisk directory:

```
cd asterisk-1.4.0/
```

Asterisk has a configure script, same as Zaptel. Run it by issuing the same command:

```
./configure
```

Next, compile and execute the menuselect utility:

```
make menuselect
```

The Asterisk menuselect is fairly more involved than the Zaptel one because the amount of options available for Zaptel pale in comparison to those for Asterisk. You can poke around and see if there are things you want to skip, but remember to be careful about choosing what modules to include. As the old saying goes "It is better to have it and not need it, then need it and not have it."

Once you are done with the menuselect process, start compiling Asterisk:

```
make
```

Compile time varies from system to system. Once completed, the next step is to install Asterisk onto the system.

```
make install
```

Sample programs, demos, and configuration references can then be (optionally) installed.

```
make samples
```

Finally, move back up into the source subdirectory.

```
cd ../
```

Compiling Asterisk-Addons

Same steps, different package. First, expand the archive:

```
tar xvzf asterisk-addons-1.4-current.tar.gz
```

Next, enter the Asterisk directory:

```
cd asterisk-addons-1.4.0/
```

Run the configure script:

```
./configure
```

Next, compile and execute the menuselect utility:

```
make menuselect
```

Once you done with the menuselect process, start the compile.

```
make
```

And, finally, install:

```
make install
```

Installing Asterisk with Binaries

Another option available for Linux users is to install Asterisk via an installer package. Installer packages are files that install software packages onto a Linux distribution. Installer packages vary from distribution to distribution: For example, a Debian's DPKG format will not install on a Fedora system, nor will Fedora's RPM format install correctly on a Debian system.

Asterisk installer packages exist in various forms for the various distributions of Linux, Windows, and Mac OS X. While these packages are maintained by third parties, and are sometimes not completely up-to-date, these provide an almost completely painless way to install Asterisk.

Installing Asterisk on Windows

AsteriskWin32 is a version of Asterisk compiled for Windows. Created by Patrick Deurel, it is currently the only real option for running Asterisk on Windows. However, AsteriskWin32 suffers from the same issues as Asterisk on Mac OS X, namely, the inability to keep up with Asterisk development. While the current version of Asterisk at the time of this writing is currently at Version 1.4.0,

AsteriskWin32 is at 1.0.10, being two major revisions behind. However, it has the advantage of being the only game in town, so it can choose its own pace.

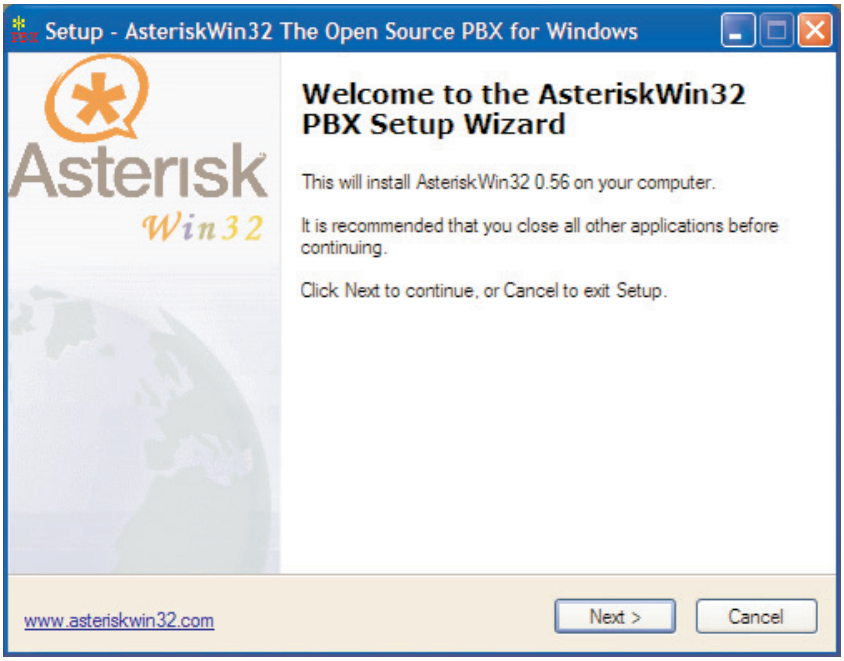
Getting AsteriskWin32

The installer package is available for download in the download section of http://www.asteriskwin32.com/. The latest version is 0.56 which is based on Asterisk 1.0.10.

Installing AsteriskWin32

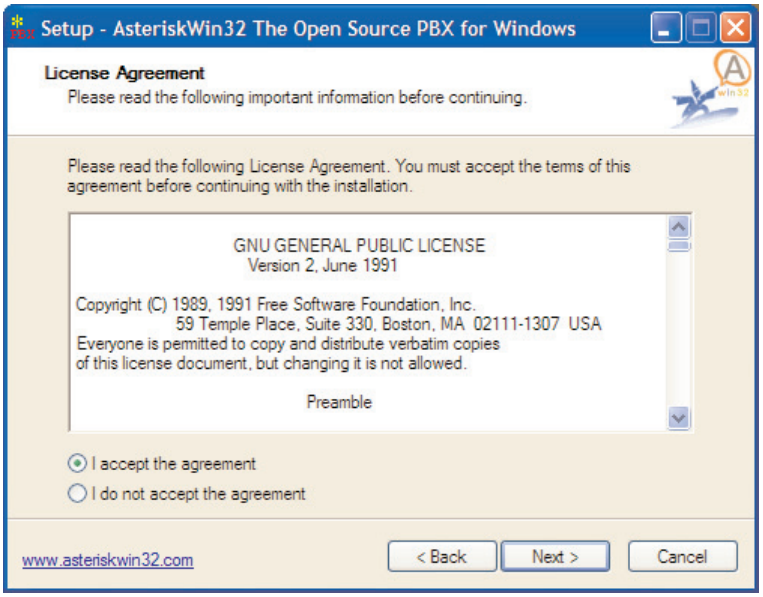
After downloading the installer package, locate the downloaded file and execute it. Click **Next**, as shown in Figure 2.21.

Figure 2.21 Welcome Window to AsteriskWin32 Setup



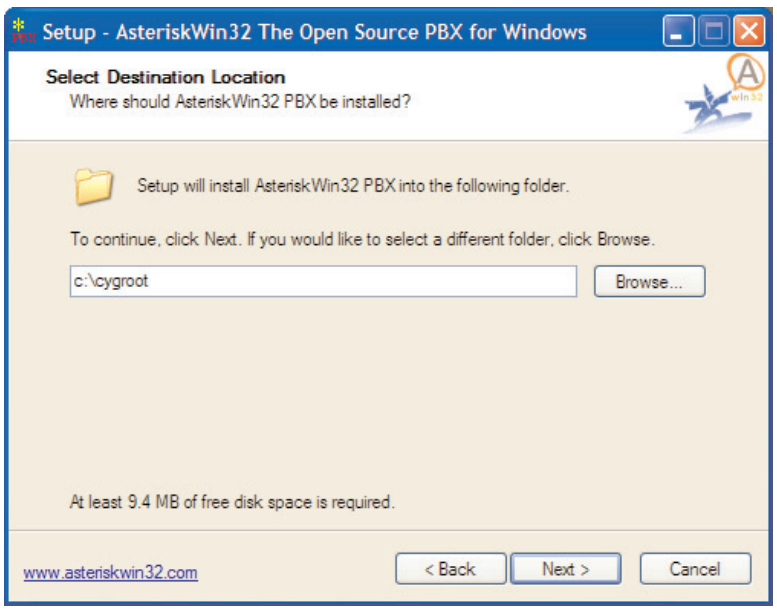
Scroll through the license agreement (Figure 2.22), read it carefully (You always read the license agreements carefully, right?) and click **Next**. After an "Information" screen that further disclaims the author from any issues his program may cause, the installer prompts you to choose a directory for it to install its files to.

Figure 2.22 License Agreement



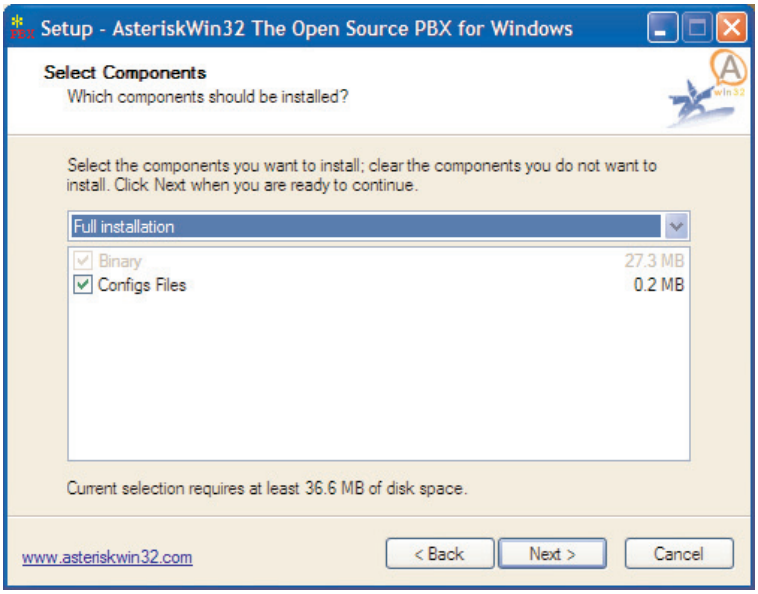
Since this version of Asterisk is compiled with Cygwin (a Windows port of many popular Linux commands), the main install directory is c:\cygroot. Asterisk will be installed as a subdirectory within this directory. See Figure 2.23.

Figure 2.23 Selecting Destination Location



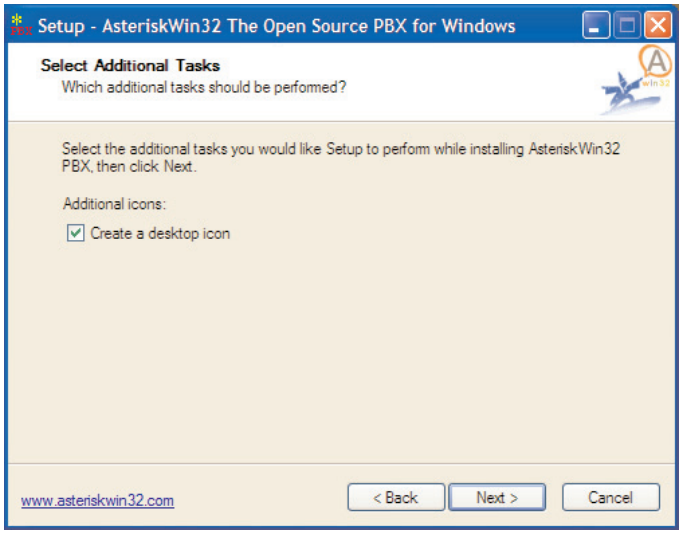
Unless the system has a working Asterisk configuration installed on it already, it is best to keep both options selected, as shown in Figure 2.24. The sample configuration files guarantee that Asterisk will find everything it needs to start itself up correctly.

Figure 2.24 Additional Tasks Selection



Next, the installer will prompt you as to whether to create a shortcut to the PBX console on your desktop, as shown in Figure 2.25.

Figure 2.25 Components Installation Selection



Choosing this option is purely personal preference. The installer will create a group under **Start | Programs** that will have all the necessary shortcuts. Click **Next**. AsteriskWin32 will start to copy files over. Finally, Asterisk will be installed (Figures 2.26 and 2.27). Pat yourself on the back. Wasn't that easy?

Figure 2.26 Installation of AsteriskWin32

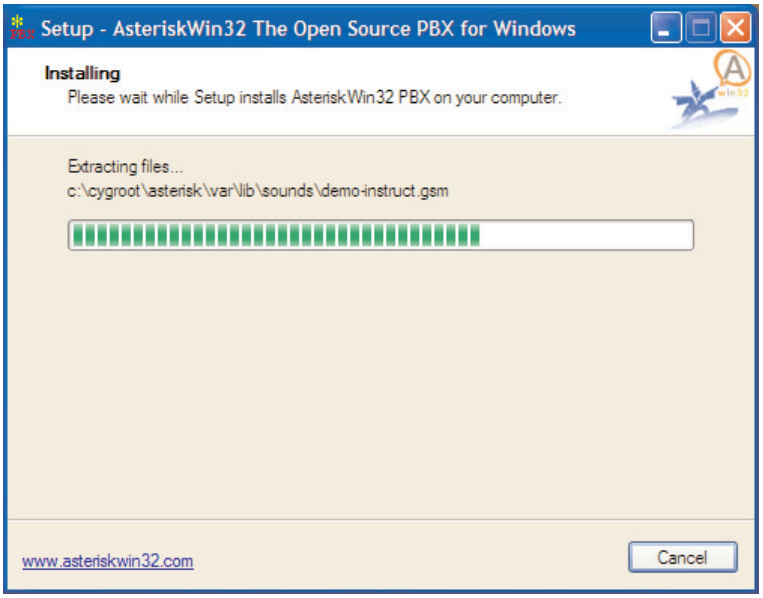
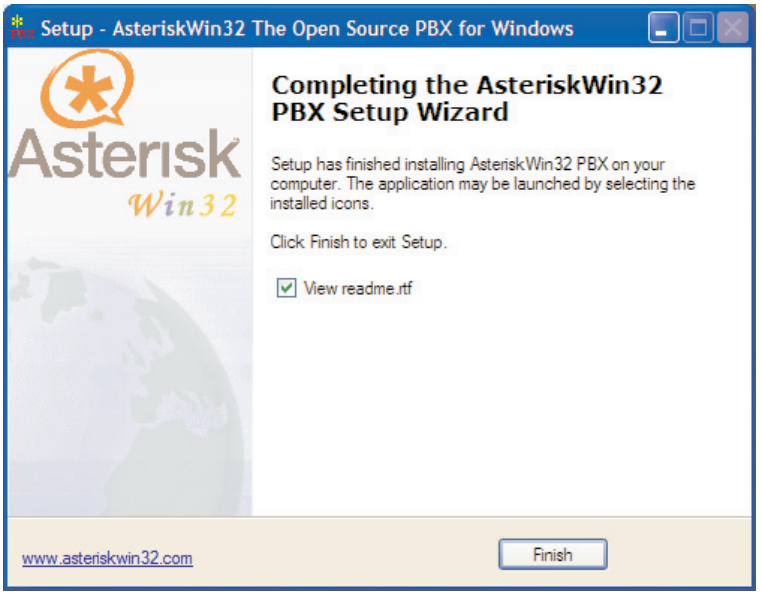


Figure 2.27 AsteriskWin32 Setup completion



Starting AsteriskWin32

AsteriskWin32 has three different "consoles": The PBX Manager & Console, the AsteriskWin32 Console, and AsteriskWin32 GUI. All of these serve the same purpose: to start, run, and manage the Asterisk server. However, each of these has slightly different abilities and caveats.

The AsteriskWin32 Console can be started by choosing **Start | Programs | AsteriskWin32 | AsteriskWin32 Console**. This is the standard Asterisk console that is part of every Asterisk install. You'll be met with the same exact console if you start up Asterisk in Linux. When executed, the Asterisk process starts up and never goes into the background, leaving the console up on the screen. From here, you can interact with Asterisk just as you would anything else. However, when that console is closed, Asterisk does not continue running. Because it never put itself in the background, it will exit when the console closes.

Another option is AsteriskWin32's GUI. This is a GUI frontend to the Asterisk CLI. While it behaves similarly to the CLI console, it has the advantage of being able to minimize itself to the system tray, keeping itself running while not having to be up on your screen. However, just like the CLI console, if the window is closed, the server will stop running.

Finally, AsteriskWin32 has its own PBX manager, which is designed to automate the starting and stopping of the Asterisk process. This is available under **Start | Programs | AsteriskWin32 | PBX Manager & Console**. When the console starts, it will try to connect to Asterisk. If Asterisk is running on the system, it will connect and display "Connected to Asterisk" and start displaying system messages in the main window. However, if Asterisk isn't running, it will display "Unable to connect to remote Asterisk" in the main window. To start Asterisk, select **PBX Tools | Start** and the console will start the Asterisk GUI minimized to your system tray. After the server is started, it will connect to it.

Key differences exist between the PBX managers and the consoles. The biggest difference is that when the manager is closed, the server process continues to run separately, be it in the form of the GUI or the CLI console. There are also some rudimentary options for controlling voice-mail boxes, loaded modules, call parking, and the call manager system. While these do simplify the process, and let you avoid editing the configuration files directly, they only hit on the basic options and do not let you configure the advanced capabilities.

Starting and Using Asterisk

Congratulations, you are the proud owner of a full-fledged Asterisk installation. Feel free to pass out cigars in the office. If you're under 18, make sure they're candy cigars. After Asterisk is installed, the next step is to start it. Thankfully, if you installed the sample configuration files, Asterisk should run out of the box without any additional changes.

Starting Asterisk

Starting asterisk is easy, just run *asterisk -vvvc*, which will execute the server. These options tell the server not to run in the background and to run at a verbosity level of three, which means all the important messages will be displayed and enough less important ones so as to not overwhelm and that the user will see all diagnostic messages. While many messages will quickly scroll by on the screen, most of these are simple initialization messages that can be ignored. If any fatal errors occur, Asterisk will stop and exit so the message remains on the screen. Asterisk will display "Asterisk Ready" when it has successfully run, as shown in Figure 2.28.

Figure 2.28 Congratulations! You're Running Asterisk!

```
    -- Added extension '600' priority 1 to demo
    -- Added extension '600' priority 2 to demo
    -- Added extension '600' priority 3 to demo
    -- Added extension '600' priority 4 to demo
    -- Added extension '8500' priority 1 to demo
    -- Added extension '8500' priority 2 to demo
    -- Registered extension context 'default'
    -- Including context 'demo' in context 'default'
pbx_config.so => (Text Extension Configuration)
  == Parsing '/etc/asterisk/dundi.conf': Found
  == Using TOS bits 0
  == DUNDi Ready and Listening on 0.0.0.0 port 4520
  == Registered custom function DUNDILOOKUP
pbx_dundi.so => (Distributed Universal Number Discovery (DUNDi))
pbx_loopback.so => (Loopback Switch)
pbx_realtime.so => (Realtime Switch)
pbx_spool.so => (Outgoing Spool Support)
  == Registered application 'DeadAGI'
  == Registered application 'EAGI'
  == Registered application 'AGI'
res_agi.so => (Asterisk Gateway Interface (AGI))
res_clioriginate.so => (Call origination from the CLI)
res_convert.so => (File format conversion CLI command)
Asterisk Ready.
*CLI> _
```

The other way to run Asterisk is to start the daemon by running the *asterisk* command without any arguments at the command prompt. This will start the server in the background. Starting in the background as opposed to the foreground has advantages and disadvantages. While the server won't tie up a terminal or exit when the terminal is closed, it will not display any diagnostic messages to the terminal during startup either. Running Asterisk in the background is the most common way to run Asterisk since normally an Asterisk process would be running at all times. One would want to run Asterisk in the foreground if diagnosis information is needed.

To connect to an already running Asterisk process, run the command *asterisk -vvvr*. This will duplicate the verbosity settings to the *above asterisk -vvvc* command, except it will not start the server process, only attempt to connect to an existing one.

Restarting and Stopping Asterisk

Every beginning has an ending. Asterisk can be stopped and restarted many ways, from the immediate and abrupt stop, to the slow and graceful shutdown. While stopping and restarting is usually not required in the normal course of operation, occasionally it is required.

The ways to stop and restart Asterisk are syntactically similar. You can issue the *stop* or *restart* command to Asterisk in three ways. When issuing the *restart* or *stop* command, you can tell Asterisk to do it *now*, *gracefully*, or *when convenient*. These control how the server will go about shutting down.

now is the proverbial "neck snap" when shutting down or restarting. The server process is shut down immediately, without any concerns for activity. Any active calls are terminated and all active threads are killed. This is not normally the way to shut down the server in a production environment. However, if the server needs to be quickly downed, this is the command to issue.

gracefully is a much cleaner way to shut down or restart. After the command is issued, Asterisk stops answering all new calls. However, unlike *now*, Asterisk does not terminate calls currently in process. While this is much better in a production environment, this can also be undesirable since it leaves calls unanswered.

Stopping or restarting *when convenient* solves this problem. After issuing this command, Asterisk continues functioning normally, the server restarts or stops when there are no active calls within the system. While this is the best when talking in terms of lost productivity, if the system constantly has active calls on it, the system will never stop or restart.

Updating Configuration Changes

Configuration changes are one of those day-to-day changes Asterisk faces. Users are added, voicemail boxes are deleted, extensions change. Every time you edit one of the configuration files, the changes aren't immediately reflected by the system. Restarting Asterisk allows these changes to be loaded, but on a high-traffic system, this will either stop phone calls, or possibly wait a long time. *reload* fixes that. Rather than shut down the Asterisk process and restart it, *reload* reloads all the configuration files on-the-fly without interrupting system activity.

Checklist

- Make sure voice and data networks are separated either physically or by VLANs. VLANs allow you to control both reliability and security. If the voice and data networks are not separated, it is possible for an attacker to monitor all telephone calls on the network.
- Make sure that trixbox is isolated from the public Internet or that root logons are disabled from remote hosts via SSH.
- Ensure that precautions are taken when entering passwords for trixbox's Web management software since these passwords will go over the wire in plain text.

Summary

Setting up Asterisk is a tedious process. Servers need to be designed to handle the expected call load. Figuring this out requires figuring out if the calls must be transcoded or have protocol translation, along with storage space for the voice prompts and voice mail. In addition to the server, networks also must be redesigned in order to provide reliability and security for the phone conversations.

Installing Asterisk can be done one of many ways. Live CDs are the easiest way to try Asterisk, just boot a CD and the system is running Asterisk. Installation CDs allow you to install Asterisk onto a clean system and set up a working system. Compiling Asterisk permits you to have maximum flexibility as to how Asterisk is set up and installed. Binaries can allow you to set up a system quickly and easily, but that system may be a few versions behind. How you set up Asterisk depends on your situation.

Starting and using Asterisk is mostly done through the command-line interface. The CLI allows you to start and stop Asterisk, along with reloading the configurations. Different options on the *shutdown* and *restart* commands let you control exactly when and how the system will shut down or restart.

Asterisk isn't an easy system to learn, but once you get the hang of it, it's a breeze to work with.

Solutions Fast Track

Choosing Your Hardware

- ☑ Choosing a reliable server for a PBX is important, because if the server goes down, the telephones go down.
- ☑ Choosing the proper RAM and processing speed will allow a server to handle multiple calls without overtaxing the processor, including situations where transcoding and protocol translation are required.
- ☑ Two types of phones are in use today: soft phones, which are software-based telephones; and hard phones, which are physical hardware devices or interfaces that emulate an analog phone system.
- ☑ VLANs are important for both security and network management reasons

- ☑ Different types of bandwidth management have both their pros and cons.

Installing Asterisk

- ☑ There are numerous ways to install Asterisk, Live CDs, Asterisk distributions, binaries, and compiling from scratch.
- ☑ Live CDs, such as SLAST, are great if you want a system where you can try out Asterisk without fear of screwing something up.
- ☑ Asterisk Linux distributions, such as trixbox, provide a simple and easy way to install Asterisk on a new system. trixbox also comes with numerous bells and whistles such as CRM software and a Web-based configuration editor.
- ☑ Compiling from scratch permits you to take the most control over the installation of Asterisk, allowing you to determine what modules are compiled and installed.
- ☑ Binaries allow you to set up Asterisk easily and quickly, but you are at the mercy of the package maintainer.

Starting and Using Asterisk

- ☑ Asterisk has both a debug and a remote console, allowing you to run it in the foreground when needed and keep it in the background when it is running.
- ☑ You can start and stop an Asterisk server in three ways: *now*, *gracefully*, and *when convenient*. Each method controls when Asterisk will restart.
- ☑ Reloading Asterisk lets you reread configuration files without restarting the system.

Links to Sites

- http://slast.org – The SLAST home page.
- www.infonomicon.com – The Infonomicon Computer Club, maintainers of SLAST.
- www.trixbox.org – The trixbox home page.

- www.centos.org – CentOS, the Linux distribution trixbox is based upon.
- www.gnu.org/software/ncurses/ – The NCurses home page, a dependency of Asterisk.
- www.openssl.org/ – The OpenSSL project, a dependency of Asterisk.
- www.zlib.net/ – The ZLib compression library, a dependency of Asterisk.
- www.asteriskwin32.com/ – The AsteriskWin32 home page.
- www.imgburn.com/ – A free ISO burner for Microsoft Windows.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is the best way for me to install Asterisk?

A: There is no "best way" to install Asterisk. It depends heavily on your situation. Different methods are better for different situations. If you want to just test the waters, however, perhaps use a Live CD on your personal workstation. If you want to task an existing server to store voice mail for your company, you might want to consider compiling Asterisk from scratch.

Q: How much should I spend on phones?

A: Phones follow the "you get what you pay for" rule. If you're cheap when it comes to phones, you will get cheap phones. A good VoIP phone should cost about $150.

Q: I have Windows. How can I burn an ISO?

A: The Windows XP CD burning system does not support burning ISOs to disk. However, there is a freeware utility that will burn ISOs called ImgBurn, which is available at http://www.imgburn.com/.

Q: How can I make my computer boot from a CD?

A: This depends greatly on your computer. Certain BIOSes, in order to get the computer to boot from a CD, may need a special key pressed during startup, or a setting may need to be configured within the BIOS itself.

Q: Is there a disadvantage to running Asterisk 1.0 versus Asterisk 1.4?

A: Yes! Asterisk 1.4 has many major bug fixes and feature additions. Plus, since this book is based on Asterisk 1.4, certain descriptions in the book may not work on Asterisk 1.0.

Chapter 3

Configuring Asterisk

Solutions in this chapter:

- **Figuring Out the Files**
- **Configuring Your Dial Plan**
- **Configuring Your Connections**
- **Configuring Voice Mail**
- **Provisioning Users**
- **Configuring Music on Hold, Queues, and Conferences**

Related Chapters: 1, 2

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

Installing Asterisk is only half the battle. The other half is configuring it. Asterisk configuration can be just as difficult as installing the program, so don't think you're in for an easy ride. Configuring Asterisk depends heavily on how exactly you want your PBX to function and what features you want available to users.

Configuring Asterisk can be somewhat of an adventure. Asterisk, like many Unix utilities, has many small configuration files all interconnected to one another. This has its pros and cons: While it adds a level of complexity to the system by requiring you to remember what feature is in which specific file, it allows you to make a mistake in one file and not have the entire proverbial house of cards come crashing down.

The plus about configuring Asterisk is that once you get the hang of it, you can easily start flying through configuration files and tackle larger and more complex problems. Asterisk's configuration files have a certain way of doing things and once you figure it out, picking up the advanced stuff is easy.

Figuring Out the Files

If you enter into your Asterisk configuration directory, /etc/asterisk, you'll see 62 files by default. If you're taking over a previous installation administered by someone else, you may see more than that. Looking at the file names, you'll see they have cryptic labels like rtp.conf, or file names that seem to be the same thing, like *asterisk.adsi* and *adsi.conf*. When trying to configure your system, finding the right file to edit can be like unearthing the proverbial needle in a haystack. (See Table3.1 for information on what each file controls.)

Table 3.1 Asterisk Configuration Files

Filename	Role
adsi.conf	Controls Asterisk Analog Display Services Interface settings
adtranvofr.conf	Contains settings related to Voice over Frame Relay and AdTran equipment
agents.conf	Contains settings for call agents that work call queues
alarmreceiver.conf	Contains settings for the Alarm Receiver application
alsa.conf	Contains settings for the CLI sound system if using ALSA sound drivers

Continued

Table 3.1 continued Asterisk Configuration Files

Filename	Role
amd.conf	Contains settings for answering machine detection on outbound calls
asterisk.adsi	Asterisk Analog Display Services Interface script
cdr.conf	Contains settings for Call Detail Records (CDRs)
cdr_custom.conf	Contains settings for custom Call Detail Record mappings
cdr_manager.conf	Contains settings for sending CDRs to the Asterisk Management Interface
cdr_odbc.conf	Contains settings for storing your CDRs into a database connected via ODBC
cdr_pgsql.conf	Contains settings for storing your CDRs into a PostgreSQL SQL database
cdr_tds.conf	Contains settings for storing your CDRs into a FreeTDS database
CODECs.conf	Contains CODEC settings
dnsmgr.conf	Contains settings about Domain Name System (DNS) lookups done by Asterisk
dundi.conf	Controls Distributed Universal Number Discovery connections and settings
enum.conf	Controls Telephone Number Mapping/E164 connections and settings
extconfig.conf	Contains mappings for external database connections for configuration settings
extensions.ael	Contains the dial plan settings, written in Asterisk Extension Language
extensions.conf	Contains the dial plan settings
Features.conf	Contains settings for call parking
festival.conf	Contains settings for the connection between Asterisk and the Festival TTS Engine
followme.conf	Contains settings for the FollowMe application
func_odbc.conf	Contains settings for template-based SQL functions accessed via ODBC
gtalk.conf	Controls Google Talk connections and settings
h323.conf	Controls H323 Protocol connections and settings

Continued

Table 3.1 continued Asterisk Configuration Files

Filename	Role
http.conf	Contains settings for Asterisk's integrated HTTP server
iax.conf	Controls Inter Asterisk eXchange Protocol Connections and Settings
iaxprov.conf	Contains settings for IAXy provisioning
indications.conf	Contains settings for the system's Ring, Busy, Reorder, and Special Information tones
jabber.conf	Controls Jabber Protocol connections and settings
logger.conf	Contains settings about where and what to log
manager.conf	Contains settings for the Asterisk Management Interface
meetme.conf	Contains settings for the MeetMe conferencing system
mgcp.conf	Controls Media Gateway Control Protocol connections and settings
misdn.conf	Controls Integrated Serial Digital Networks (ISDNs) connections and settings
modem.conf	Controls ISDN modem settings
modules.conf	Controls which applications and modules are loaded when the server is started
musiconhold.conf	Contains Music on Hold settings
muted.conf	Contains settings for the Mute Daemon
osp.conf	Controls settings and connections for the Open Settlement Protocol
oss.conf	Contains settings for the CLI sound system if using OSS sound drivers
phone.conf	Contains settings for Linux Telephony devices
privacy.conf	Contains settings for the PrivacyManager application
queues.conf	Contains settings for call queues
res_odbc.conf	Contains settings for external database connections for configuration settings
res_snmp.conf	Contains Simple Network Management Protocol settings for the SNMP application
rpt.conf	Controls settings of the app_rpt application, which enables radio systems to be linked via VoIP
rtp.conf	Contains Real-time Transport Protocol settings

Continued

Table 3.1 continued Asterisk Configuration Files

Filename	Role
say.conf	Contains string settings for the various say_* applications
sip.conf	Controls Session Initiation Protocol (SIP) connections and settings
sip_notify.conf	Contains settings for SIP's NOTIFY command
skinny.conf	Controls Skinny Client Control Protocol connections and settings
sla.conf	Controls Shared Line Appearance connections and settings
smdi.conf	Contains settings for the Simplified Message Desk Interface
telcordia-1.adsi	Default Telcordia Analog Display Services Interface script
udptl.conf	Contains settings for UDPTL, one of the transports for Faxing over IP Networks
users.conf	A file that controls a combination of settings, allowing for easier user maintenance
voicemail.conf	Contains voice mail settings and mailbox details
vpb.conf	Contains settings for VoiceTronix hardware
zapata.conf	Controls settings for Zapata hardware

To say Asterisk has a lot of settings would be an understatement of mammoth proportions. While this is a plus when you want to tweak Asterisk to fit your needs exactly, it is a bit overwhelming. However, don't fret. Certain configuration files can be ignored if you don't have certain hardware, and other files can be ignored if you do not need to enable certain features of Asterisk.

Configuring Your Dial Plan

The dial plan is the logic behind how phone calls are routed through your Asterisk installation. Asterisk runs every incoming call, every outgoing call, and every call in between extensions through the dial plan logic in order to determine where it should go and whether or not it should be completed. The dial plan is contained in *extensions.conf*, and therefore it can be said that *extensions.conf* is easily the most important configuration file in Asterisk. Removing *extensions.conf* is similar to removing a traffic

light at a complicated intersection; cars will want to enter and cars will want to leave, but there will be no way to direct them.

extensions.conf is a bit more complicated than a typical configuration file. On top of the usual sections and settings, there is a logical flow similar to a program. Essentially, *extensions.conf* is one giant script. The sooner you keep this in mind, the easier it will be for you to write a good dial plan.

Contexts, Extensions, and Variables! Oh My!

extensions.conf can be broken down into three major parts: contexts, extensions, and variables. Each has their own unique and important function and needs to work together for a good dial plan to function.

Contexts

To put it simply, contexts are the fences that keep your extensions from getting tangled up in a big mess. A context is a simple way for grouping extension commands based on what the user has dialed. To begin a context, put the name of a context by itself in square brackets. Each context then contains a list of commands. In *extensions.conf* there are two special contexts called *[general]* and *[globals]* in which certain settings can be set.

general has a few special settings that define how *extensions.conf* behaves. First off is the *static* setting. This, can be set to either *yes* or *no*, but for some reason, only yes has been implemented. This will eventually control Asterisk from rewriting the *extensions.conf* every time an extension is added or deleted. The next setting is *writeprotect*. This can also be set to either *yes* or *no*, and this controls the ability of someone at the CLI to rewrite your dial plan via the *save dialplan* command. This may seem handy, but doing so will delete all comments in the file.

Each extension follows a similar syntax. *exten => EXTENSION,PRIORITY,COMMAND(ARGS)*. *exten =>* precedes every extension. This is a directive that tells Asterisk to define an extension, as opposed to a context. The next three parts of an extension are *EXTENSION*, *PRIORITY, and COMMAND()*. Let's cover these three portions.

Extensions

Extensions can be broken down into three types: a constant extension, a wildcard extension, and a special extension. A constant extension is an extension that when coded to a literal constant is the dial plan. A wildcard extension is a context that uses

wildcards to match multiple possibilities for the extension. Wildcards can be either internal Asterisk wildcards or RegEx-like patterns (see Table 3.2).

Table 3.2 Extension Wildcards Used in Asterisk

Wildcard	Patterns Matched
[0126-9]	Any digit within the pattern. (In this case: 0,1,2,6,7,8, and 9).
X	Any number 0 through 9. The equivalent of [0-9].
Z	Numbers between 1 through 9. The equivalent of [1-9].
N	Numbers between 2 through 9. The equivalent of [2-9]. This scheme is used most commonly in Area Code and Prefix assignments.
.	Any number, one or more times.

So with Wildcard extensions, it is simple to reroute numerous extensions with one line of code. Let's say a department in your building, the ever-important widget department, have moved to another division and wanted to leave a message at their old extensions informing callers that they had moved. They previously occupied Extensions 300 through 329 on your PBX. Rather than rewrite 30 lines; you can add a single extension of

```
exten => 3[0-2]X,1,Playback(WidgetDeptHasMoved)
```

This will have any caller dialing the department's former extensions greeted by a message informing them of the move. Playback is a command that plays back a sound file stored on the system; we'll cover it and its counterparts later.

In addition to wildcard and literal extensions, there are also special extensions that correspond to special events in the dial plan (see Table 3.3).

Table 3.3 Special Extensions Used in Asterisk

Extension	Name	Description
S	Start	Used when a caller is put in a context before dialing a number.
I	Invalid	Used when a caller dials an extension not defined in the current context.
H	Hangup	Used when a caller hangs up.
T	Time Out	Used when a caller does not respond within the response timeout period

Continued

Table 3.3 continued Special Extensions Used in Asterisk

Extension	Name	Description
T	Absolute Time Out	Used when a caller does not respond within the about timeout period
O	Operator	

Extensions do not necessarily need to be numbers either. They can be made with any type of text. While extensions like "fuzzybunnydept" cannot be dialed by a caller if included in your context, it can be used internally by your dial plan. We'll see how this can come in handy later in the chapter.

Priorities

PRIORITY controls the flow in which commands are executed. For each extension, this is either controlled by an increasing number or a special *n* syntax. The *n* syntax tells Asterisk to execute the extension one line after the other:

```
[incomingcall]
exten => s,1,Answer()
exten => s,n,Playback(mainmenu)
exten => s,n,Hangup()
```

In this example, any call being routed to the "incomingcall" extension in Asterisk would have its call answered, a menu would then play, and then the call would be terminated. After Asterisk finishes executing one line, the next line would be executed. Numbering the steps provides greater flexibility with the dial plan since it is possible to control the flow logically rather than line by line. For example, the extension shown earlier could be rewritten with a numbered sequence

```
[incomingcall]
exten => s,2,Playback(mainmenu)
exten => s,1,Answer()
exten => s,3,Hangup()
```

Asterisk still answers, plays the menu, and hangs up because it executes by line number rather than by the order in which the lines appear. It executes step 1, followed by steps 2, and then 3. These steps could be scattered throughout the context and intertwined with hundreds of extensions. As long as they are numbered correctly, Asterisk will execute them in order for that context.

Dial Plan Commands

The commands are the heart of any dial plan. They are what actually cause Asterisk to answer the call, ring the phone, transfer the call, play the menu, and do numerous other things. See Table 3.4 for a look at some of the more common ones.

Table 3.4 Common Commands in Asterisk

Command	Description
Dial(CHANNEL)	Dials a channel
Answer()	Answers a ringing channel
Playback(FILE)	Plays a sound file in the foreground
Background(FILE)	Plays a sound file in the background, while waiting for the user to input an extension
Hangup()	Hangs up the call
SayDigits(NUMBER)	Says a number, digit by digit

Notes from the Underground…

Channels vs. Extensions

It's easy to get confused when people start tossing around terms like "extensions" and "channels" when the two words seem interchangeable. Sometimes, people do use them as if they are identical, but don't be one of these people. Channels and extensions are two separate and completely different things. Extensions are the physical numbers assigned to a device, while channels, on the other hand, are the connections to the devices themselves. For example, you can have a phone at your desk set up to ring on three separate extensions; however, each of these extensions will ring the same channel—namely, your phone.

Variables

Variables in extensions.conf are nothing special. They act like variables in any other language. Variables are set via the *Set()* command and are read via the variable name encased in *${}*:

```
[example]
exten => s,1,Set(TEST=1)
exten => s,2,NoOp(${TEST})
```

Variables are common in simple dial-plan applications and Asterisk uses certain variables for internal functions, but their use is somewhat uncommon in regular dial plan usage.

Tying It All Together

All of these pieces of dial plans make little to no sense when thinking about them in the abstract, so you may be scratching your head right now. Let's take a look at how all of these would be used in an everyday environment, by looking at a simple *extensions.conf*:

Example 3.1 A Very Simple *extensions.conf*

```
 [default]
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)
exten => s,4,Hangup()

exten => 100,1,Dial(SIP/10)
exten => 200,1,Dial(SIP/20)
```

When a call enters the *[default]* context, it is answered by Asterisk. Asterisk then starts playing the mainmenu sound file while waiting for the caller to enter digits. At this point, the caller can either enter 100 and be connected to the channel SIP/10 or 20 and be connected to the channel SIP/20. If the menu finishes playing and the user has not entered any digits, the call will be hung up on.

Using Special Extensions

Now, hanging up on your caller if they wait to listen to the whole menu seems kind of rude, doesn't it? So let's take the file we had before and use some special extensions to have the menu replay if the user hasn't entered an extension and inform them if the extension they entered is invalid.

Example 3.2 A Very Simple *extensions.conf* with Special Extensions

```
 [default]
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

exten => t,1,Goto(s,2)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,2)

exten => 100,1,Dial(SIP/10)
exten => 200,1,Dial(SIP/20)
```

That's much nicer. Now the behavior of the dial plan is the same, up until the main menu ends. At that point, the menu repeats. Also, now if the caller dials an incorrect extension, the dial plan will play a menu that informs them the extension they entered is not valid.

Creating Submenus

Normally, most small to medium-sized companies only require a single menu, but let's say your boss wants to have a support menu that allows customers to direct their questions to the appropriate support group. We can accomplish this by creating a second context that contains the appropriate menu and extensions. Let's build on the previous example again and add a second menu that allows callers to be connected to the Blivet, Widget, or Frob support lines.

Example 3.3 Creating Submenus in *extensions.conf*

```
 [default]
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

exten => t,1,Goto(s,2)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,2)
```

```
exten => 3,1,Goto(s,1,supportment)
exten => 100,1,Dial(SIP/10)
exten => 200,1,Dial(SIP/20)

[supportmenu]
exten => s,1,Background(conglomocorp-supportmenu)

exten => 1,1,Dial(SIP/blivetsupportline)
exten => 2,1,Dial(SIP/widgetsupportline)
exten => 3,1,Dial(SIP/frobsupportline)
exten => #,1, Goto(s,2,default)

exten => t,1,Goto(s,1)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,1)
```

In this example, we've added a third option to the main menu. If a caller dials 3, they are connected to the *[supportmenu]* context with a *Goto()* statement. *Goto()* can be called many different ways. You can jump between priorities in the same extension by just specifying *Goto(priority)* or you can jump between extensions in the same context by specifying *Goto(priority,extension)*. Lastly, you can switch contexts by specifying *Goto(context, extension, priority)*.

Tools & Traps…

Watch Your Spaces!

Goto() is a bit finicky with its syntax and whitespace. For example: *Goto(supportmenu,s,1)* will behave differently than *Goto(supportmenu, s, 1)*. In the first example, *Goto* will behave as expected and jump to the "s" extension, priority 1. However, in the second example, *Goto* will jump to the "s" extension, priority 1. Note how there is a space that precedes the "s". This can be a source of frustration if you don't know to look for it.

Including Other Contexts within the Current One

It's important to note that when creating another context, the settings and extensions from one context do not propagate to another. Setting up these extensions over and over again can be tedious and will lead to a duplication of code and effort. Thankfully, Asterisk permits other contexts to be joined together via the *include* => directive. This allows other contexts to be *include*-ed into the current context and act as one giant context.

Let's go back to our example. The *t* and *i* context are duplicated in both the *[default]* and *[supportmenu]* contexts. With a couple of small changes, we can make a separate context with just the *t* and *i* extensions and *include* => them into both contexts.

Example 3.4 Using *includes* in *extensions.conf*

```
 [default]
include => specialextensions
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

exten => 3,1,Goto(supportmenu,s,1)
exten => 100,1,Dial(SIP/10)
exten => 200,1,Dial(SIP/20)

[supportmenu]
include => specialextensions
exten => s,1,Background(conglomocorp-supportmenu)

exten => 1,1,Dial(SIP/blivetsupportline)
exten => 2,1,Dial(SIP/widgetsupportline)
exten => 3,1,Dial(SIP/frobsupportline)
exten => #,1,Goto(s,2)

[specialextensions]
exten => t,1,Goto(s,1)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,1)
```

Okay, pop quiz time. Did you notice the difference between this example and the previous one? Don't worry if you didn't, it's pretty subtle. Because we are including the same *t* and *i* context between two files, the same code will be executed between both. Namely, they will be going to step 1 of the *s* extension in both contexts. Previously in the *[default]* context, the *t* and *i* extension went to step 2 of the *s* extension, bypassing the *Answer()* command. What does this change? Not a single thing. Technically, you're adding an extra step every time a caller times out or enters an invalid extension, which may affect performance if this happens repeatedly in a very high-traffic environment, but, in the grand scheme of things this extra step will not be perceptible. *Answer()* only answers the call if the call is in an unanswered state. It ignores being called if the call is already in answered.

Writing Macros

include-ing (other contexts within the current one is a handy way to save lines of code and duplication of code. Another easy way to increase efficiency and decrease code duplication is through Asterisk's macro abilities. Macros can be described as special contexts that accept arguments. They allow for more flexibility than contexts, and allow common tasks to be automated and not repeated.

In our previous examples, if someone dialed an extension, it rang a channel. It would continue ringing the channel until someone picked up, or the call terminated. What happens if we want to have that extension drop to voice mail playing the user's "I'm not here" message after 20 seconds of ringing, or playing the user's "I'm currently on the phone" message if the phone line is busy?

Example 3.5 Creating Voice Mail Support for Existing Extensions without the Use of Macros

```
 [default]
include => specialextensions
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

exten => 3,1,Goto(supportmenu,s,1)
exten => 100,1,Dial(SIP/10,20)
exten => 100,2,Goto(s-100-${DIALSTATUS},1)
exten => s-100-NOANSWER,1,Voicemail(u100)
exten => s-100-NOANSWER,2,Hangup()
```

```
exten => s-100-BUSY,1,Voicemail(b100)
exten => s-100-BUSY,2,Hangup()
exten => _s-.,1,Goto(s-100-NOANSWER,1)
exten => 200,1,Dial(SIP/20)
exten => 200,2,Goto(s-200-${DIALSTATUS},1)
exten => s-200-NOANSWER,1,Voicemail(u200)
exten => s-200-NOANSWER,2,Hangup()
exten => s-200-BUSY,1,Voicemail(b200)
exten => s-200-BUSY,2,Hangup()
exten => _s-.,1,Goto(s-200-NOANSWER,1)

[supportmenu]
include => specialextensions
exten => s,1,Background(conglomocorp-supportmenu)

exten => 1,1,Dial(SIP/blivetsupportline)
exten => 2,1,Dial(SIP/widgetsupportline)
exten => 3,1,Dial(SIP/frobsupportline)
exten => #,1,Goto(s,2)

[specialextensions]
exten => t,1,Goto(s,1)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,1)
```

Yikes. That got complicated quickly. Can you imagine having to set that up for multiple extensions? A single typo in the various extensions could suddenly have people's voice mails intended for one person wind up in someone else's voice-mail box. Plus, the various extensions would get out of hand very quickly; your *extensions.conf* could start topping over thousands of lines of code. Let's insert a Macro to tame this beast. The macro, *macro-stdexten*, is included in Asterisk by default for this exact reason.

Example 3.6 Creating Voice Mail Support for Existing Extensions with the Use of Macros

```
 [default]
include => specialextensions
exten => s,1,Answer()
```

```
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

exten => 3,1,Goto(supportmenu,s,1)
exten => 100,1,Macro(stdexten,10,SIP/10)
exten => 200,1,Macro(stdexten,20,SIP/20)

[supportmenu]
include => specialextensions
exten => s,1,Background(conglomocorp-supportmenu)

exten => 1,1,Dial(SIP/blivetsupportline)
exten => 2,1,Dial(SIP/widgetsupportline)
exten => 3,1,Dial(SIP/frobsupportline)
exten => #,1,Goto(s,2)

[specialextensions]
exten => t,1,Goto(s,1)

exten => i,1,Playback(sorry-thats-not-valid)
exten => i,2,Goto(s,1)

[macro-stdexten]
exten => s,1,Dial(${ARG2},20)
exten => s,2,Goto(s-${DIALSTATUS},1)
exten => s-NOANSWER,1,Voicemail(u${ARG1})
exten => s-NOANSWER,2,Hangup()
exten => s-BUSY,1,Voicemail(b${ARG1})
exten => s-BUSY,2,Hangup()
exten => _s-.,1,Goto(s-NOANSWER,1)
```

Using the macro allowed us to write a single piece of code that would duplicate the function of the code in the previous example. It's also modular, allowing for the easy addition of extra extensions and extra voice-mail boxes. The *stdexten* macro takes two arguments: The first being the channel to ring, and the second being the voice-mail box to send the call to if the channel is busy or does not answer. The macro rings the channel for 20 seconds and then sends it to voice mail telling voice mail to use the unavailable message. If the channel is busy, it immediately sends the caller to voice mail, telling voice mail to use the busy message if the user has one. If there is

some other condition on the call, like if the phone cannot be found on the network, the macro sends it to voice mail with the unavailable message.

The *Macro()* command takes at least one argument, the macro name. You can also pass multiple arguments to the macro by calling the *Macro()* command with additional arguments. In our example, macro- *stdexten* takes two arguments: the channel to ring, and the voice-mail box to call. Upon calling the macro, the macro is executed like a normal context, with the exception of extra variables *${ARGX}*, where X is 1 through the number of variables you passed to the macro.

This takes care of incoming calls, but what about phones on the inside dialing out? Setting these up is as simple as setting up another context. Each time you set up a connection, you need to specify which context calls coming from that connection will go into. Setting up a context in which calls can use your outside line and then assigning all internal phones into that context will allow the phones to send calls via the outside lines. Continuing our example, let's set up a context for internal calls:

```
[internal]
exten => _1617NXXXXXX,1,Dial(Zap/1/${EXTEN})
exten => _1310454XXXX,1,Dial(IAX2@/mass:Sk5S@cali.conglomocorp.com/${EXTEN})
exten => _1NXXNXXXXXX,1,Dial(IAX2/conglomocorplogin@IAXProvider/${EXTEN})
exten => _011X.,1,Dial(SIP/SIPProvider/${EXTEN})
exten => 100,1,Macro(stdexten,10,SIP/10)
exten => 200,1,Macro(stdexten,20,SIP/20)
```

Let's go over what each line accomplishes. Each one shows a different way of composing a dial command. The first line tells Asterisk that if a user dials a telephone number in the 617 area code, it will match the _1617NXXXXXX wildcard and the phone call will be sent out via the fist Zaptel device. The next line matches anything within the 310-454 prefix and will connect to a server called "cali.conglomocorp.com" with the username "mass" and the password "Sk5S" and send the phone call through them. This is an explicit connection created in *extensions.conf*. If a user dials a U.S. telephone number that isn't in 617 or 310-454, it will match the _1NXXNXXXXXXX wildcard, and will be sent via the IAXProvider connection, which would be created in *iax.conf*. Finally, if a user dials an international number beginning with 011, it will match the "_011X." wildcard and be sent via the SIPProvider connection, which would be created in *sip.conf*. Also, the user can dial either of the two extensions on the system and be connected to them directly. These extensions would already be connected in *sip.conf*.

It is important to note that if we placed the _1NXXNXXXXXXX wildcard above the _1617NXXXXXX wildcard or the _1310454XXXX wildcard, anything below the _1NXXNXXXXXX wildcard would never be used since the _1NXXNXXXXXXX wildcard would match everything. Asterisk reads lines from the top down and will match the first line it sees. Remembering this can save you a lot of headaches, and depending on your setup, possibly some money.

Configuring *extensions.ael*

The alternative to *extensions.conf* is *extensions.ael*. *extensions.ael* is *extensions.conf* written in a scripting language called Asterisk Extensions Language (AEL). AEL is language maintained by Digium solely for writing dial plans in Asterisk. While it is functionally equivalent to *extensions.conf*, AEL is syntactically much more powerful and allows for greater flexibility in simple scripting and logical operations. If you're familiar with scripting in other languages, AEL can often be easier to pick up than the regular *extensions.conf* syntax.

extensions.ael can be used as a replacement for *extensions.conf* or have both used side by side. *extensions.ael* is not in widespread use in today's installations. However, due to its greater functionality, it would not be surprising to see *extensions.conf* depreciated in future versions of Asterisk in favor of *extensions.ael*.

Using AEL to Write Your Extensions

Everything that can be written in extensions.conf can be rewritten in extensions.ael. Let's take our simple example from Example 3.1 and rewrite it into AEL.

Example 3.7 Rewriting Example 3.1 into AEL

```
context default {
       s => {
              Answer();
              Background(thank-you-for-calling-conglomocorp);
              Background(conglomocorp-mainmenu);
              Hangup();
       };

100 => Dial(SIP/10);
200 => Dial(SIP/20);
};
```

Execution-wise this does the same exact thing Example 3.1 did. Asterisk answers the call, starts playing the mainmenu sound file while waiting for the caller to enter digits. The caller can then either enter 100 and be connected to the channel SIP/10 or 200 and be connected to the channel SIP/20. The caller is then hung up on when the menu stops playing.

Notice how, despite being mixed up a bit, there are still contexts, extensions, and variables. In this case, however, the *exten => EXTENSION,PRIORITY,COMMAND(ARGS)* syntax is completely scrapped. In *extensions.ael*, the *exten =>* is removed, along with any use of priorities. *extension.ael* follows more of a line-by-line execution pattern the way *extensions.conf* executes when the *n* priority is used. While this simplifies things so you don't have to worry about making sure every extension has the right priority, it provides a lack of flexibility in execution order and *Goto()* statements. Let's see what happens when we rewrite the code in Example 3.2.

Example 3.8 Rewriting Example 3.2 into AEL

```
context default {
       s => {
              Answer();
              restart:
              Background(thank-you-for-calling-conglomocorp);
              Background(conglomocorp-mainmenu);
              Hangup();
       };

       100 => Dial(SIP/10);
       200 => Dial(SIP/20);

       t => { goto s|restart;}
       i => {
              Playback(sorry-thats-not-valid);
              goto s|restart;
       }
};
```

Because we can't specify the exact step to jump into in the *s* context, we need to create a label in the *s* extension to tell the *Goto()* statement where to enter. The *restart:* label in the *s* context is the where the *t* and *i* extensions jump to when they are

done executing. This label needs to be explicitly specified within the *s* context because there are no steps numbered within the context.

Macros also function much in the same way they do in *extensions.conf*. They are set up as if contexts, but have extra variables that can be passed to them. In AEL, variables passed to the macro are not referred to as *${ARG1}* through *${ARGX}*. In AEL you can assign them local variables names, which cuts down on the confusion factor when trying to remember which values are assigned to a certain variable. Another difference in AEL is that the *Macro()* command is not used when calling a macro. Instead, the macro's name has an ampersand added in front of it. Let's add the *std-exten* macro to our AEL example to see how it fits in.

Example 3.9 A Macro in AEL

```
context default {
       s => {
              Answer();
              restart:
              Background(thank-you-for-calling-conglomocorp);
              Background(conglomocorp-mainmenu);
              Hangup();
       };

       100 => &std-exten("10","SIP/10");
       200 => &std-exten("20","SIP/20");

       t => { goto s|restart;}
       i => {
              Playback(sorry-thats-not-valid);
              goto s|restart;
       }
};
macro std-exten(vmb,channel) {
       Dial(${channel},20);
       switch(${DIALSTATUS) {
       case BUSY:
              Voicemail(b${vmb});
              break;
       case NOANSWER:
              Voicemail(u${vmb});
       };
       catch a {
              VoiceMailMain(${vmb});
```

```
                return;
        };
};
```

AEL is a very powerful language that allows for a much cleaner dial plan. It is still in heavy development, and may change in future Asterisk revisions, so it may not be quite ready for production yet. However, it is a very good idea to learn the mechanics of it because Asterisk may move toward it in the future.

Configuring Your Connections

Connections are what make Asterisk useful. If there are no connections to Asterisk, you wouldn't be able to connect a phone or use a link to the outside, which really limits the things you can do with it. Asterisk, when first installed, actually has a connection to a demonstration server hosted by Digium. This connection shows how calls can be transferred via VoIP to a completely different server as easily as dialing a number, and gives you a taste of what can be accomplished. This connection, however, is a nice demonstration, but doesn't really have any use besides showing off what can be done with Asterisk. If you want to actually accomplish tasks, you will need to set up your own connections with the outside world.

Connections, Connections, Connections!

Numerous files control the various protocols for Asterisk. Some protocols are commonly used in today's VoIP setups, while some are quite vestigial and are likely not to be used unless you have specialty hardware. Let's take a look at the various protocols supported by Asterisk (see Table 3.5).

Table 3.5 VoIP Protocols Supported by Asterisk

Protocol	Name	Notes
SIP	Session Initiation Protocol	Most common VoIP protocol. Used in numerous devices.
IAX	Inter Asterisk eXchange Protocol	Used primarily in connections between Asterisk servers.
SCCP	Skinny Client Control Protocol	Used in Cisco devices.

Continued

Table 3.5 VoIP Protocols Supported by Asterisk

Protocol	Name	Notes
MGCP	Media Gateway Control Protocol	Used in some VoIP devices, notably D-Link.
H323	H.323 Protocol	Used in some older VoIP devices.

Each protocol is controlled by a different file. Multiple connections can be set up in a single file, or the files can be broken down and linked via include statements. What you opt to do is a choice of personal preference. Each file has certain specific configuration options that are used only for the protocol the file governs, and they also have options that are common across all files. Let's go over some of the conventions:

Configuration File Conventions

All Asterisk configuration files have certain conventions that run throughout them. We went through some of them when we were talking about *extensions.conf*. However, some differences exist in the terminology and layout when comparing *extensions.conf* to another file.

Much like how *extensions.conf* is broken down into contexts, most configuration files are broken down into sections. Context and sections have the same syntax—namely, that the headers are surrounded by brackets, as shown in the following example.

Example 3.10 *extensions.conf* Context Compared to an *iax.conf* Section

```
 [default]
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)

[my_iax_server]
type=peer
auth=md5
notransfer=yes
host=10.0.23.232
disallow=all
allow=ulaw
```

Each configuration file often has a *[general]* section as well, which functions more or less the same way as the *[general]* section in *extensions.conf*: settings in that section are applied to each section unless they are overridden within the specific section.

Configuration File Common Options

Each protocol has its own specific options, but they share a number of options common across files. Let's go over a few common tasks and the options that control them that you'll likely run into when editing configuration files.

Users, Peers, and Friends

Asterisk uses some peculiar classifications for its VoIP connections. They are classified by the *type=* setting, which is either set to *user*, *friend*, or *peer*. These are often accompanied by little to no explanation, which is a shame because they're actually quite simple.

A *user* is a connection that will be used to make telephone calls to the local server; a *peer* is a connection that will be used to make telephone calls from the local server; and a *friend* is a connection that will be used to make telephone calls both to, and from, the local server.

These classifications are most commonly used in IAX2 and SIP connections. However, using them in SIP connections is actually starting to become redundant due to how SIP connections are normally set up. We will cover that later in the chapter.

Allowing and Disallowing Codecs

Asterisk supports numerous codecs for audio. Codecs can save bandwidth and allow for more simultaneous phone calls on a data link. For a big list of the codecs Asterisk supports, refer to the table in Chapter 1.

Codecs are configured via the allow and disallow directives. Disallow can be used to explicitly deny use of specific codecs, or it can be used in conjunction with allow to grant the use of only specific codecs. Confused yet? Let's look at a common situation:

Say your shiny new Asterisk server has a connection to your telephone provider via the IAX2 protocol. However, whenever a phone call is made through the provider, the GSM codec is used, rather than the ulaw codec that is used when you

call between extensions in the office. This needs to be fixed. So opening up the *iax.conf* configuration file you add the following line to the section controlling the connection:

```
disallow=gsm
```

Then issue a *reload* command to Asterisk. Problem solved, right? Not necessarily. While yes, this will disallow use of the GSM codec, the behavior that results might not be the one expected. The added line tells Asterisk not to use GSM; however, it still has the option of picking from all the other codecs it supports. The correct way to ensure ulaw is used as the codec would be to add the following lines to *iax.conf*.

```
disallow=all
allow=ulaw
```

Now, if you're scratching your head at the *disallow=all* statement, don't worry. While, yes, that directive essentially tells Asterisk to disallow every codec from being used, it is followed by the *allow=ulaw* statement, which tells Asterisk that ulaw is okay to use. Essentially, those two lines are the same as typing out disallow statements for every codec Asterisk supports except the one you want to use. When receiving a phone call, Asterisk will check each allow and disallow statement to see which codecs it can and cannot use. It will first see the *disallow=all* statement, stopping the use of all codecs, but then it will allow the ulaw codec once it reads the *allow=ulaw* statement.

This can be expanded to work with multiple codecs as well. If you wanted to allow both ulaw and alaw, ulaw's European equivalent, the same steps would be followed, except this time there would be two *allow* lines, allowing both ulaw and alaw.

Including External Files

Asterisk's configurations files support the inclusion of other files into the "current" one. This can be important when setting up a large installation and wishing to spread the configuration over many files rather than maintain a large single file.

Including other files is accomplished through the *#include* statement. For example, if you wanted to split three departments in your *extensions.conf* between three files, just add the following lines to extensions.conf:

```
#include </etc/asterisk/extensions/department1.conf>
#include </etc/asterisk/extensions/department2.conf>
#include </etc/asterisk/extensions/department3.conf>
```

You can then add extension contexts to *department1.conf*, *department2.conf*, or *department3.conf* as if they were *extensons.conf* themselves. Asterisk will read these at runtime and interpret them the same as if they were all joined together in *extensions.conf*.

It is recommended you store your included files somewhere other than the root Asterisk configuration directory. That way it will be unlikely there will be a naming conflict between an existing configuration file and a file you create.

Configuring SIP Connections

SIP is the most common VoIP protocol in use today. It is an official Internet standard and is supported by almost every VoIP device and service on the market. SIP is a very complex and involved protocol and has its fair share of shortcomings, but often is the only game in town when dealing with devices or VoIP providers. Let's look at how to set up connections, too, from a server.

SIP connections are configured in the *sip.conf* file in the system's configuration directory, usually /etc/asterisk.

General SIP Settings

General SIP settings are contained within the *[general]* section.

SIP, Firewalls, and Network Address Translation

SIP was created before Network Address Translation (NAT) use was widespread. Therefore, it never really took into account the possibility of one of the sides of the conversation not having a publicly routable IP address. Today, it is very common to see a residential broadband connection without a cheap router doing NAT for the connection. This is related to another problem with SIP and firewalls: the two do not get along, period.

The reason for these problems is because SIP phone calls rely on two different protocols: SIP for the setup and takedown of the connection, and Realtime Transport Protocol (RTP) for the voice stream. When SIP receives a notification for an incoming phone call from a remote server, it sets up an RTP listener on a port and waits for the RTP stream. This is all fine and dandy, unless you have a firewall that blocks incoming connections. If you do, the phone calls will set up, but the audio path will not be carrying audio.

NAT suffers from the same problem, but with different issues. When the call is set up, if one side of the connection tells the other to connect to a nonpublic IP

address, the connecting side will not know where to connect to send the RTP stream, and so the audio path isn't set up correctly. There have been attempts to address this issue, notably in RFC3581 – "An Extension to the Session Initiation Protocol (SIP) for Symmetric Response Routing," but with all the existing hardware currently in use, not all devices support the newer features,

Thankfully, despite the protocol not really addressing these issues, solutions can be found for these problems—not necessarily good solutions, but solutions none the less. To address the firewall issue, you need to open up the firewall to allow connections from external sources to the Asterisk server on a massive amount of ports. This is a bit of an issue if the server is accepting connections from all over the Internet since there is no way to lock the access down to specific address blocks. A way to limit the amount of ports you need to open up is to edit *rtp.conf* in the Asterisk configuration directory:

Example 3.11 A Typical *rtp.conf*

```
;
; RTP Configuration
;
[general]
;
; RTP start and RTP end configure start and end addresses
;
rtpstart=10000
rtpend=20000
```

The two settings *rtpstart* and *rtpend* are the ports that RTP will try to use when it sets up a connection with another server. Adjusting these variables will give you control over which ports you need to open up in your firewall settings.

To address the NAT issue, there are kludges built into Asterisk to work around the problem. In *sip.conf*, there are three settings: the *externip* setting, the *localnet* setting, and the *nat* setting. The *nat* setting determines whether or not the server is behind a NAT. This can be set to four different settings: *yes*, *no*, *never*, and *route*. The *yes* setting is the straightforward setting. It informs Asterisk that we are behind a NAT and it should assume so whenever it sends SIP messages. The *no* setting is a bit more complicated than "No, the server is not behind a NAT." The *no* setting tells Asterisk it should use RFC3581 to determine whether or not there is a NAT between the local server and the remote server. The next setting, *route*, is a bit of a kludge to help NAT

work with certain phones that do not completely support RFC3581; you likely will never use this, and hopefully this behavior will be moved to another setting in future versions. Finally, there is *never*, which informs Asterisk to never think the server is behind a NAT.

Now, *localnet* and *externip* are settings used when Asterisk is using NAT functionality—namely, when *nat* is set to something else other than *never*. They give the system information regarding what is behind the NAT and what isn't, along with what IP the NAT is using for an external IP. For example, let's say we have a server at our office on a 196.168.42.0/24 network that is NATed behind a gateway with an external IP address of 118.23.45.76. This is how we would make our NAT settings:

```
[general]
nat=yes
externip=118.23.45.76
localnet=192.168.42.0/24
```

If you have extra networks behind the NAT with you, but that are on separate IP segments, you can add additional *localnet* statements to list those networks as well.

Connecting to an SIP Server

Most VoIP service providers support SIP over IAX, so connecting to an SIP server is a common task when setting up a new provider. Thankfully, it's fairly simple. In this example, we'll assume there are preexisting settings in the *[general]* section pertaining to whether or not the server has a NAT address and what codecs the server will be using. These are normally set up in the *[general]* settings since they don't vary between connections.

Registering Your Connection

Most providers do not have your account tied to a specific IP address since it's becoming less and less common to have static IP addresses in most situations and it's less of a hassle for you to come to them. So how do we let the provider know where to route the incoming calls? We register with them. Registering is a way of checking in with a remote server, letting them know where to route calls and that the local server is still alive. A typical register line in *sip.conf* would look like this:

```
register => mgaribaldi:peekaboo@voip.defuniactelephone.com/3115552368
```

In which, after a *reload*, we would be registering the phone number "311-555-2368" with the server *voip.defuniactelephone.com* using the username *mgaribaldi* and the

password *peekaboo*. Once we registered with the remote server, it would know to send any phone calls for 311-555-2368 to our local server. Please note that all of these would be assigned by the provider. If we tried to register with another phone number, the server would, at best, not send us any phone calls, or at worst, likely reject our registration.

All register statements need to go under the *[general]* section. If you are registering to multiple providers, all that must be done is just have multiple register statements. Registration depends on your provider. If you have a static IP address that your provider automatically sends phone calls to, registration is unnecessary. However, this is highly uncommon.

Tools & Traps…

Passwords, Plaintext, and Privacy

This seems like as good a time as any to mention it, but when storing your passwords in your configuration files, you're storing them in plaintext. Also, these configuration files are world-readable by default. Put these together and you're stuck in a bit of a security nightmare. Asterisk doesn't have any security on its configuration files by default, so before you add any sensitive information, you may want to make sure the file permissions are locked down enough that the only nonprivileged user that can read them is the user Asterisk is running under.

Setting Up Outbound Settings

Registering lets the remote server know where we are. Thus, it will start sending telephone calls to us. By default, Asterisk will use the settings specified in the *[general]* section of *sip.conf*. This will work fine, unless we want to apply special settings to phone calls coming from a specific connection. We can also provide connection-specific options, such as usernames and password, so we do not have to specify the username and password in the dial string.

Using our DeFuniac Telephone example, let's create a section that will route incoming telephone calls from them to their own special context in your dial plan and allow the *Dial()* command to omit the username and password.

```
[defuniactelephone]
type=peer
secret=peekaboo
username=mgaribaldi
host=voip.defuniactelephone.com
fromuser=mgaribaldi
fromdomain=voip.defuniactelephone.com
context=incoming_defuniac
```

After you add this, issue a *reload* command. What this specifically does is create an account on the system for the connection. This account will match any phone calls coming into the server *voip.defuniactelephone.com* with the username *mgaribaldi* and the password *peekaboo*, and route those phone calls into the context *incoming_defuniac* in your dial plan.

This account also allows us to use the Dial application without specifying a username and password like this:

```
exten => _1NXXNXXXXXX,1,Dial(SIP/defuniactelephone/${EXTEN})
```

This saves a bit of typing and allows us to quickly adjust usernames and passwords should they ever change.

Setting Up an SIP Server

Setting the server up to accept a SIP client is pretty easy. In fact, it has much in common with connecting to an SIP server. The only real difference is that you don't need to register, and the account type is set to *friend* rather than *peer*.

Let's jump in head first and set up an account in our *sip.conf*:

```
[sipclient]
type=friend
context=internal
username=sipclient
secret=password
mailbox=201
host=dynamic
callerid="SIP Client" <3115552368>
dtmf=inband
```

What this does is set up an account for a channel called "sipclient" that is identified via the username "sipclient" and the ultra-secure password "password". We specify it is a dynamic host, which means the client can connect from anywhere so it

will be registering with us. The client will sit in the internal context where the appropriate dial strings should be. Also, we assign the voice-mail box 201 to the client so they can be notified about waiting messages. We also specify that outbound calls from the client will have the caller ID string *SIP CLIENT <3115552368>*.

Notes from the Underground…

DTMF and SIP

SIP has three settings for DTMF: *inband*, *info*, and *rfc2833*. SIP, because of the separate connections used for the audio and signaling path, has trouble relaying information about DTMF. *inband* sends the DTMF over the audio path like a regular telephone call would. This is the simplest way to do things; however, certain codecs mangle the audio enough that the called party cannot pick the DTMF signal up. *info* and *rfc2833* send signals across the stream so the called party can translate them back into DTMF, but these are not supported by some providers.

That's it. After a *reload*, the system is now ready to accept an SIP client connection. Point an SIP phone to the server with the correct username and password and you will be ready to dial away.

Configuring IAX2 Connections

IAX2 (Inter-Asterisk eXchange version 2) is the protocol designed to connect Asterisk servers between each other. Designed by Digium as an alternative to SIP, it is not an official standard, but is instead an open protocol with a freely available protocol library. It is well supported in Asterisk, and is starting to make inroads into other devices and programs. It is less common to find soft phones and devices that support IAX2, but it is not as surprising at it once was.

Everything in IAX2 is controlled by the file *iax.conf* in your asterisk configuration directory. This is set up similarly to *sip.conf.*

Connecting to an IAX2 Server

Connecting to an IAX2 server is a lot like connecting to an SIP server. A lot of the options are the same and the methodology is identical. So let's take a look.

Registering Your Connection

Registering is not just a SIP-only thing. The same problems affect IAX2 as well. Thankfully, the same command applies:

```
register => mgaribaldi:peekaboo@voip.defuniactelephone.com
```

The main difference between the SIP register command and the IAX2 register command is that there is no phone number appended to the end of the IAX2 version. This is because IAX2 is designed to be a trunking protocol (a protocol that can carry numerous telephone lines at once), as opposed to SIP, which is designed more to carry one telephone line at one time.

Setting Up Outbound Settings

Much like in SIP, we can specify the outbound settings in *iax.conf* to allow the connection to have special settings and connect to a different context other than the one specified in the *[general]* section. Let's set up this provider:

```
[defuniactelephone]
type=peer
secret=peekaboo
username=mgaribaldi
host=voip.defuniactelephone.com
context=incoming_defuniac
```

As you can see, the settings are very similar to the SIP version. The only difference is that some of the SIP-specific directives have been trimmed out. This will accomplish the same thing its SIP counterpart did: incoming calls will be routed to the *incoming_defuniac* context, which will allow us to use a shortened *Dial()* string:

```
exten => _1NXXNXXXXXX,1,Dial(IAX2/defuniactelephone/${EXTEN})
```

Setting Up an IAX2 Server

Much like how connecting to an IAX2 server is similar to connecting to an SIP server, becoming an IAX2 server is a lot like becoming an SIP server.

```
[iaxclient]
type=friend
username=iaxclient
secret=password
host=dynamic
callerid="SIP Client" <3115552368>
context=internal
```

This sets up an IAX client with a username of *iaxclient* and a password of *password*. Again, the host is dynamic, so the client will have to register with the server and the client will be assigned to the "internal" context. While in this example the client has an assigned caller ID string, IAX2 can support sending its own Caller ID string. This can be handy if there are multiple lines coming across a connection, or if you just want to give the client an ability to send its own Caller ID string. This ability does have some security ramifications, but we'll talk more about that later in the book.

Configuring Zapata Connections

Zapata telephony devices are what the majority of Asterisk systems employ if they want a physical connection to the outside world. They come in single line models all the way up to quadruple T1 models that have 96 channels.

Setting Up a Wireline Connection

Wired telephone connections are what most of us are used to when we think of a telephone: pieces of copper wire molded into an RJ-11 jack that we plug into our telephone. However, the physics behind the connections are a tad more complicated.

There are two basic types of signaling telephones with wired connections. FXO signaling is used by a telephony device to receive signals from the telephone network, while FXS signaling is used by a telephone switch to send signals to a telephony device. This means that the type of card you should have depends on what you want to accomplish.

Configuring a Zapata Card

This assumes you have a Zapata card installed and the drivers compiled and loaded. If you don't have the drivers compiled, flip back to Chapter 2 and follow the instructions there. In this example, we are going to assume you have installed a four-port Zapata card with two FXO modules installed in slots 1 and 2, and two FXS modules installed in ports 3 and 4.

The first step is to open up the Zaptel configuration that is independent of Asterisk. This is located in */etc/zaptel.conf*. This is a very well-documented file with lots of examples, so if you don't have the card in this example, you should be able to follow along and configure your own setup.

There are no sections in here, so you'll be able to toss directives wherever you want. It's common to put them with the commented out examples so you'll know

where to look if you need to make changes. The first step is to tell the modules which signaling methods to use:

```
fxsks=1-2
fxoks=3-4
```

This instructs modules 1 and 2 to use the FXS KewlStart protocol and modules 3 and 4 to use the FXO KewlStart protocol. KewlStart is a newer method of telephone signaling that is used by a majority of telephone equipment today. Other protocols are available as well, such as Ground Start and Loop Start, but unless you have very old equipment, KewlStart is the way to go.

Now, I'm sure some of you are feeling rather smug that you've picked up a typo in the book. I just said that modules 1 and 2 are FXO modules but we told them to use FXS signaling, and vice versa for modules 3 and 4. Nope. They are supposed to be that way. We are specifying what signaling the modules should be receiving, which for FXO modules connected to the PSTN is FXS from the switch. For FXS modules driving telephones, they should receive FXO signaling from the phone. This is rather confusing at first, but makes sense when you think about it.

If you aren't in the United States, you may want to scroll down to the *loadzone* options and comment out the *loadzone = us* line and uncomment the line appropriate to your country. This will allow proper ring and busy tones to be sent to the devices connected.

Now that we are done with that, exit out of the file and load the appropriate module for your card. In this example, we would run:

```
modprobe wctdm
```

This will load the module into the kernel and configure the hardware modules on the card. The next step is to open up *zapata.conf* in the Asterisk configuration directory. Unfortunately, *zapata.conf* is a bit arcane even by Asterisk's standards. The file duplicates a lot of information we already entered into *zaptel.conf*. This may seem silly, but the files serve two separate purposes: *zaptel.conf* sets up the modules, while *zapata.conf* tells Asterisk how to talk to them. Here's how we would create *zapata.conf* in our example:

```
[channels]
usecallerid=yes
echocancel=yes
echocancelwhenbridged=no
echotraining=800
```

```
signalling=fxs_ks
group=0
context=fromzap
channel=1-2

signalling=fxo_ks
group=1
context=internal
channel=3-4
```

It's important to know that Asterisk reads *zapata.conf* from top to bottom. Options that are set are applied to all channels below it unless unset at a later point. In this option, we set up the cards to use echo cancellation with a moderate setting (*800*). We then configure channels 1 and 2 for PSTN operation and put them in the "fromzap" context. After that, we configure channels 3 and 4 for telephones and put them in the "internal" context.

From here, we'll open up *extensions.conf* and add the specific contexts we need:

```
[internal]
exten => _1NXXNXXXXXX,1,Dial(Zap/G0/${EXTEN})
[fromzap]
exten => s,1,Dial(Zap/3&Zap/4)
```

This will accomplish two things. The two telephones we have connected to channels 3 and 4 will be able to dial U.S. telephone numbers, which will be dialed out on the first available FXO channel, either 1 or 2. The "G" in the *Zap/G0* refers to group 0, of which channels 1 and 2 are members. If a phone call comes in on either channel 1 or 2, the server will then ring both channels 3 and 4 until someone picks one of the telephones up or the call terminates.

At this point, we need to start or restart Asterisk. Zapata configuration changes do not get read with a *reload* command, so the entire system must be restarted. Once the system is restarted, the Zapata modules should be functioning as expected and ready to receive and dial telephone calls.

Configuring Voice Mail

Voice mail has played a key role in business over the past 20 years. The case can be made that it is more important than e-mail for some people. Voice-mail settings are listed within *voicemail.conf* in the Asterisk configuration directory.

Configuring Voice-Mail Settings

There are a lot of bits to configure in voice mail, such as time zone settings, voice mail to e-mail settings, and options on how to pronounce time, among others. Unless you want to get fancy, most of the defaults should work fine. A common option that may need to be adjusted is the *maxmsg* option which limits the number of messages a user can have in their mailbox. Another option that may need to be adjusted is the *tz* option that controls what time zone the messages will be based in. This is commonly used if the server's time zone is different than the time zone the company is based in. The *tz* option, by default, can only be set to options specified in the *[zonemessages]* section, which by default is set to the following:

```
[zonemessages]
eastern=America/New_York|'vm-received' Q 'digits/at' IMp
central=America/Chicago|'vm-received' Q 'digits/at' IMp
central24=America/Chicago|'vm-received' q 'digits/at' H N 'hours'
military=Zulu|'vm-received' q 'digits/at' H N 'hours' 'phonetic/z_p'
european=Europe/Copenhagen|'vm-received' a d b 'digits/at' HM
```

The syntax for this is

```
ZONENAME=TIMEZONE|DATESTRING
```

where *ZONENAME* is the name you want to give the setting, *TIMEZONE* is the Linux time-zone name you want the system to use for the setting, and *DATESTRING* is a string of Unix date variables and sound files. Not the most elegant solution, but it is very customizable. Let's say we wanted to add a Pacific time zone, we would just add the following line:

```
pacific=America/Los_Angeles|'vm-received' Q 'digits/at' IMp
```

which would make a *pacific* zone based on the America/Los_Angeles time zone and would play the standard voice-mail envelope string.

Configuring Mailboxes

Mailboxes are in the *[default]* section. A typical run-of-the-mill mailbox for Joe would look like this:

```
867 => 5309,Steve Example,steve@example.net
```

This sets up mailbox 867 for Steve Example, with a password of 5309. Any messages left in the mailbox would be attached to an e-mail sent to steve@example.net,

allowing him to listen to the message without calling the server. This setup is suitable for most users; however, there are other options as well. Asterisk has the ability to send a second message without the attachment that is more suitable for text messages or mobile phone e-mail as well:

```
867 => 5309,Steve Example,steve@example.net,3115552368@defuniactelephone.com
```

This is handy since it allows the user to receive a notification on their mobile device about a voicemail message without having to download a possibly large audio file over a slow mobile data link.

You can also specify per-user settings on the mailbox line as well. Let's say Steve doesn't have a cell phone, and has dial up so he doesn't want to attach the voice-mail messages to the e-mail messages, but still wants to receive a notification. This is done with the *attach* option:

```
867 => 5309,Steve Example,steve@example.net,,|attach=no
```

Also, notice the blank "pager e-mail" field since Steve doesn't need a notification to a cell phone he doesn't have. You can also attach multiple options separated by the pipe character. Let's say Steve is in a separate time zone from the company and wants to have his mailbox say the time in the Central time zone. We would then adjust the mailbox like this:

```
867 => 5309,Steve Example,steve@example.net,,|attach=no|tz=central
```

Options can be tacked on as needed until each mailbox is configured as you, or the user, want.

Leaving and Retrieving Messages

All of the voice-mail functions are contained in two applications: *Voicemail()*, which handles the portions of a user leaving a message on the system; and *VoicemailMain()*, which handles the users of the PBX to access their voice mail. We briefly touched upon *VoiceMail()* earlier when we were talking about dial plans, but let's take a slightly more in-depth look now:

```
[default]
exten => s,1,Answer()
exten => s,2,Background(thank-you-for-calling-conglomocorp)
exten => s,3,Background(conglomocorp-mainmenu)
exten => 100,1,Voicemail(u100)
exten => 200,1,Voicemail(b200)
```

```
exten => 300,1,VoicemailMain()
exten => 400,1,VoicemailMain(${CALLERID(num)})
```

This example has four different voice-mail extensions that do four different things. Extension 100 sends you to voice mail to leave a message for mailbox 100. The *u* preceding the mailbox number tells Asterisk to use that mailbox's "unavailable" greeting. Extension 200 does the same thing, except this time the *b* preceding the mailbox number tells Asterisk to use that mailbox's "busy" greeting. Besides the greetings, both of these do the same thing: they take a message for the mailbox they are given.

Extension 300 sends you to the voicemail system as if you are a user of the system. In this case, the system will prompt you for a mailbox number and password and if you give it valid credentials, it will let you listen to messages for that mailbox. Extension 400 does the same thing, except it attempts to find a mailbox corresponding to the caller's caller ID number. If it does, it will prompt just for the password. If it does not, it will behave as if there was no number given to it.

Moving around the voice-mail system is just like navigating a regular voice-mail system. The default keys are "1" to play messages, "6" to skip to the next message, "4" to go to the previous message, and "7" to delete the current message. There are also options to forward messages to other users and save the messages into different folders. The keys are not customizable unless you want to recode the mail application.

Provisioning Users

Configuring IAX2 and SIP connections, as well as dial plans in an abstract sense, gives you a good sense of how their respective configuration files work, but really doesn't give you a sense in how all the configuration files tie together in a typical Asterisk installation. When provisioning a user, all the configuration files seem less separate and more like pieces that function as part of a whole. Let's walk through a typical user provision and see how everything fits together.

Let's say you are the new administrator of a medium-sized business's Asterisk PBX system. Your boss walks in and tells you that a new employee, Joe Random PBXUser, is starting next week and you need to have everything ready to go on Monday.

Decision Time

The first step is to figure out what the new user is going to use for a phone. Is he going to use a new phone or an existing one? Has the phone already been provisioned? In this example, we are going to assume the user needs a new phone and that, thankfully, you have one right at your desk just waiting to be configured.

Next, you need to check what extension the new user should get. This depends on how the existing extensions are configured. In this example, you've consulted your chart and extension 221 is open, so the user will get that one. Now, let's get to work.

Configuring Phone Connections

The phone you have is SIP, so let's add the following to *sip.conf*:

```
[jrpbxuser]
type=friend
context=internal
username=jrpbxuser
secret=s3kr1tp@ss
mailbox=221
qualify=yes
host=dynamic
callerid="Joe Random PBX User" <3115550221>
dtmf=inband
```

It's important to note we already assigned Joe a voice-mail box, but we haven't set it up yet. We'll do that later. Next, issue a *reload* command to the Asterisk CLI and configure the phone to use these settings. If the phone syncs up to the server correctly, you're ready to head on over to the next step. If it doesn't, double-check all your settings and make sure your phone is finding the server.

Configuring Extensions

Next, you need to find out if this user is going to be part of any extensions that ring multiple phones, call queues, or any other special extensions. In this example, Joe is just going to get a normal extension and not be part of anything else. So, we need to edit the dial plan and add the following line to any contexts that have internal extensions written in them:

```
exten => 221,1,Dial(SIP/jrpbxuser)
```

This will assign extension 221 to ring Joe's phone. What contexts you need to put this in will depend heavily on your installation. Under normal situations, you would need to give access from the default context so callers can dial extensions directly and the context in which internal phones can dial each other.

Configuring Voice Mail

Your boss informed us that Joe has a private e-mail account on his mobile phone and wants to receive voice-mail notifications on both his regular e-mail and his mobile phone. We picked voice-mail box 221 for him earlier, so let's go ahead and set that up

```
221 => 90210,Joe Random PBXUser,jrpbxuser@example.net,jp@joescellphone.com
```

This setup will now send a notification to Joe's e-mail, along with a mail to his cell phone when someone sends him a voice mail.

Finishing Up

Once this is all done, issue one final reload command to Asterisk to see if there are any problems you may have missed. If there are no complaints, make a few phone calls from Joe's phone to ensure everything behaves as it should. If it does, you're all set!

Joe is now ready to head into work Monday and have a phone on his desk. Go out and celebrate a job well done with a couple of chocolate chip cookies and a large glass of milk.

Configuring Music on Hold, Queues, and Conferences

The three most common "specialty" features used in Asterisk are Call Queues, Conference Calls, and Music on Hold. These are common features found when calling a medium- to large-sized business, and businesses often pay an arm and a leg to get support for them in their PBX. Asterisk supports them by default, So let's go over how to configure them.

Configuring Music on Hold

Music on hold is regarded by some as both a blessing and a curse. While it is useful to provide feedback to callers that their call is still connected and to give them

something to listen to, music on hold is often lampooned by the public as an annoyance. Whether or not to use it is up to you, but let's walk through configuring it anyway.

Music on hold is a breeze to configure. The *musiconhold.conf* comes with a music on hold class ready for files, so often all you need to do is put some ulaw encoded files of your favorite songs in the *moh/* subdirectory of your Asterisk sounds directory, usually */var/lib/asterisk/*. Once this is done, issue an *asterisk reload* command to the CLI and you should be ready to go. If you put a caller on hold, they should enjoy the sweet sounds of whatever files you added to the *moh/* directory.

Music on Hold Classes

Music on hold can be assigned to separate "classes," and each class can be assigned to a different directory and given different audio clips to play. This is handy if you want to have an audio clip for the support department that tells callers to check the support Web site, but you don't want to have that clip anywhere else. Simply create two classes of music on hold. You can do this by opening up *musiconhold.conf*. You should see something that looks like the following:

```
[default]
mode=files
directory=/var/lib/asterisk/moh
```

This is the default music class. Each call put on hold will be here unless you specify another class. Let's say you want to add another class for the support department. Just add:

```
[support]
mode=files
directory=/var/lib/asterisk/moh/support
```

Then create the directory and add ulaw encoded files to */var/lib/asterisk/moh/support*. Once this is done, you will need to edit the support context and assign a new music on hold class to it. You can do this via the *SetMusicOnHold()* command. Using the *supportmenu* context from Example 3.3, we would set the class like this:

```
[supportmenu]
exten => s,1,SetMusicOnHold(support)
exten => s,2,Background(conglomocorp-supportmenu)

exten => 1,1,Dial(SIP/blivetsupportline)
exten => 2,1,Dial(SIP/widgetsupportline)
```

```
exten => 3,1,Dial(SIP/frobsupportline)
exten => #,1,Goto(s,2)
```

This now assigns the caller to the support class until another command assigns it to somewhere else.

Music on Hold and MP3s

Since a lot of people already have their entire collection of music already in MP3 format, a common request is to set up music on hold to play MP3 files. While it is possible, music on hold and MP3s can be difficult to work with. However, they can be supported by using MPG123. To configure your *musiconhold.conf* to support MP3s, you will need to change the *mode*= to custom and specify the exact syntax of the MP3 player command:

```
[RiverBottomGang]
mode=custom
directory=/var/lib/asterisk/moh/RiverbottomNightmareBandMP3s
application=/usr/bin/mpg123 -q -r 8000 -f 8192 -b 2048 --mono -s
```

This example would create a new class called *RiverBottomGang*, which would then use MPG123 to play all the songs in */var/lib/asterisk/moh/RiverbottomNightmareBandMP3s*. This is somewhat less reliable than using ulaw encoded files because of the conversions involved. Sometimes, if your files are not encoded in a way that is just right, your music on hold will sound like it is playing a twice the speed.

Configuring Call Queues

Call queues are important in any end-user support environment. The way call queues work is explained in Chapter 1, but let's quickly review them here: In a call queue, all callers form a virtual line wait to be answered by a person answering a phone. When an "answerer" hangs up, the system takes the next person out of the queue and rings the answerer's phone. This allows for a small group of people to efficiently answer a larger group of calls without the callers receiving busy signals.

Setting Up a Call Queue

Call queues are managed by *queues.conf*. A typical call queue configuration would look like this:

```
[supportqueue]
musicclass=support
```

```
strategy=ringall
timeout=10
wrapuptime=30
periodic-announce = conglomocorp-your-call-is-important
periodic-announce-frequency=60
member=>SIP/10
member=>SIP/20
```

Let's go over the options. Starting off each queue section is the queue's name written in brackets. The next line defines the queue's music on hold class—which, here, is the support class we defined in the last section. The *strategy* line defines the ringing strategy—in this case, *ringall*: ring all the phones until someone picks up. The system can be configured to use a roundrobin system that will ring the phones one by one starting from the first, or do a roundrobin with memory called *rrmemory* in which the system will start with the next phone after the phone it rang last. The *timeout* line specifies how long, in seconds, a phone should ring until the system determines that no one is there. The *wrapuptime* line specifies how long, also in seconds, after a call is completed that the system should wait before trying to ring that phone again. The *periodic-announce* and the *periodic-announce-frequency* specify a sound file the system should play for callers instead of the music on hold music and how long it should wait after playing a file until playing it again. Finally, each member line adds a member to the pool of phones that have people answering the queue.

Setting up the queue in the dial plan is easy. Let's take our support queue and instead of having the users ring individual channels, let's just put them into the support queue.

```
[supportmenu]
exten => s,1,SetMusicOnHold(support)
exten => s,2,Playback(conglomocorp-welcome-to-support-queue)
exten => s,3,Queue(supportqueue)
```

After that, just create a recording for the welcome message, issue a *reload* command to the Asterisk CLI and the queue should be up and running. Any customers entering the *supportmenu* context should have the recording you just created played back to them and then they should enter the queue.

Getting Fancy with Call Queues and Agents

"Agents" in Asterisk are people who call into the system from a nonlocal phone and take calls from call queues. This allows people to call from home and interact with a

call queue as if they are in the call center. With agents, you can even eliminate a physical call center and rely solely on agents calling in remotely. Let's take our support queue and add a few agents into it.

Setting Up Agents

The first step in setting up a queue with agents is to set up the agents themselves. This is controlled by *agents.conf* in the Asterisk configuration directory. In this file, you can control the sounds the agents hear when they log on and off, whether or not you want to record the conversations they have with callers, and what music on hold class the agents should be assigned. The part in which you would control agents is at the bottom of the file at the end of the *[agents]* section. Each agent will be configured by an *agent* line. The agent line syntax is

```
agent => AgentNumber,Password,AgentName
```

So, let's add a couple of agents for our queue:

```
agent => 1001,867,Joe Random Agent
agent => 1002,5309,James Random Agent
```

Now that we've added a couple of agents, let's edit our support queue to support these agents. Adding agents is just like adding any other members to a queue:

```
[supportqueue]
musicclass=support
strategy=ringall
timeout=10
wrapuptime=30
periodic-announce = conglomocorp-your-call-is-important
periodic-announce-frequency=60
member=>SIP/10
member=>SIP/20
member=>Agent/1001
member=>Agent/1002
```

With this done, now comes the tricky part: The agents need a place where they can log in to the system to accept phone calls. Normally, you would want this to be a separate number from your main line so that regular customers won't get prompted to log in. However, that is up to you. Let's set up a separate context for the agents to log in:

```
[agentlogin]
exten => s,1,Playback(conglomocorp-this-is-private)
exten => s,2,Background(conglomocorp-please-login)
exten => XXXX,1,AgentLogin(${EXTEN})
```

Next, create sound files for the "This is a private system" and "Please log in" sound clips listed above and point a telephone number to that context. Next, issue a *reload* command to the Asterisk CLI. From here, you should be able call the number you set up, or enter the context another way you may have set up, and enter the agent's ID. Next, the system will prompt you for a password. Enter the agent's password and you should start to hear the music on hold for the system. Congratulations! The next time a caller enters the queue, you'll hear a beep and be connected to him or her!

Configuring MeetMe

Second to VoIP, conferencing was one of Asterisk's killer apps. Using commercial conference call systems can easily add up very quickly given they will charge you per minute per user. With Asterisk's conference calling system, MeetMe, you can carry out the same calls for pennies on the dollar.

It's All about Timing

MeetMe has one significant drawback. It requires a timing device. MeetMe uses the timing devices to ensure that the conversation won't go horribly out of sync with each other. Currently, timing devices are only supported under Linux. There are two officially supported ways of using a timing device: either using the timing device on a Zaptel device or using a Zaptel-like device called ztdummy.

If you already have a Zaptel card in use, you're all set. Asterisk and MeetMe will automatically recognize this as a source for timing and use it. If you don't have a Zaptel timing device, you need to install the Zapata telephony drivers. If you haven't got the Zaptel drivers, go back to Chapter 2 and follow the instructions there. Once you get everything compiled, you will need to make sure the *ztdummy* driver is loaded into the kernel. Run the following as root:

```
modprobe ztdummy
```

After this completes, restart Asterisk. Asterisk must be fully restarted, not reloaded in this instance. When Asterisk restarts and *ztdummy* is loaded, MeetMe should load without a hitch.

Setting Up a Conference

The first step in setting up a conference is opening up *meetme.conf* and adding a conference room. Conference rooms are numbered, but these are only used when connecting to conferences from *extensions.conf*. Users should never have to interact with them.

In *meetme.conf*, the conference rooms are listed under the [rooms] section. The syntax for rooms are

```
conf => RoomNumber,UserPIN,AdminPIN
```

In a conference room, both the User PIN and Admin PIN are optional. Let's set up a simple, un-PINed conference room.

```
conf => 1234
```

Now, let's edit our *extensions.conf*. We are putting this in a separate context for the same reason we put the agent login in a separate context: we don't want regular users to stumble into the conference by accident.

```
[conference]
exten => s,1,MeetMe(1234)
```

Now, just set up a way to access this context and issue a *reload* command to the Asterisk CLI. You should be all set. When entering the context, you should hear two beeps and silence. Then, when someone else calls in, you should both hear the same two beeps and subsequently be connected to each other. This process repeats for each person who connects. Pat yourself on the back.

Checklist

- Since all passwords for connections are stored plaintext within files, ensure that all configuration files are readable only by the user that Asterisk is running under.
- If you are behind a firewall and need to use SIP, make sure there are no services left running on exposed ports of the server.

Summary

Asterisk has a lot of configuration files. Rather than assign all settings into one master configuration file; Asterisk opts to have many smaller files. This is advantageous since, depending on what hardware and features you are using, there are some files you may never touch. Another advantage is that a syntax error in one file may not necessarily bring down the entire system.

The dial plan is the keystone for the entire Asterisk system. Every phone call handled by Asterisk goes through the dial plan for routing information. Dial plans consist of three major parts: contexts, extensions, and variables. Contexts are groups of extensions that function together. Extensions are groups of commands that tell Asterisk what to do. Variables are simply used to store data. A special kind of context is a macro, which allows you to write small functions for common tasks in order to save code.

Two dial plan types are available: the common *extensions.conf*, and the newer more powerful *extensions.ael*. AEL stands for Asterisk Extension Language, which is a programming language developed by Digium for writing extensions. AEL is more powerful than the regular *extensions.conf* syntax, but is still very new and not fully mature.

Connections are the lifeblood of Asterisk. Without them you wouldn't be able to accomplish much since you wouldn't be able to talk to anyone. Asterisk supports numerous VoIP protocols and many models of hardware. The two most commonly used protocols in Asterisk are SIP and IAX2. SIP stands for Session Initiation Protocol and is the most common VoIP protocol currently supported. IAX2 stands for Inter Asterisk eXchange version 2 and is a protocol designed by Digium to interconnect Asterisk servers. SIP, while widely support, has a share of issues with firewalls, NAT, and DTMF. IAX2 doesn't suffer from those issues; however, support for the protocol is much smaller.

Voice mail can be configured in many different ways to support users across the globe. Voice-mail messages can be sent via e-mail and Asterisk supports sending notifications of new messages via pages and cell phones. Asterisk has two different voicemail applications, *Voicemail* which is used for sending voice mail to system users; and *VoicemailMain*, which is utilized by system users to pick up their voice mail.

Music on hold, call queues, and conference calls are often big features to buy in commercial PBXs, but Asterisk supports them out of the box. Music on Hold can be set up to support multiple audio tracks and assign each group to a different class, allowing you to assign different classes to callers depending on what context they are

in. Call queues can be set up to be answered by local users or agents who call in remotely. Conference calls are run by the MeetMe application and require a timing source such as a Zaptel card or an emulation of one. Once you get the timing source configured, multiple conference rooms can be set up on the system with feature such as PINed access.

Asterisk has a lot of options to configure, but by giving you a lot of options, Asterisk allows you to tailor a solution that will fit your needs exactly.

Solutions Fast Track

Figuring Out the Files

- ☑ Asterisk has over 60 configuration files, often with very cryptic names.
- ☑ Asterisk configuration files are small and short in an effort to reduce complexity.
- ☑ Some configuration files can be ignored depending on what features you are using.

Configuring Your Dial Plan

- ☑ Every call that goes through Asterisk goes through the dial plan.
- ☑ Every dial plan consists of three major parts: contexts, extensions, and variables
- ☑ *Extensions* and *channels* are two completely separate terms. Don't use them interchangeably.
- ☑ Macros are an easy way of eliminating code duplication, allowing you to create small functions to automate simple tasks.

Configuring Your Connections

- ☑ Asterisk supports multiple VoIP protocols and numerous hardware connections.
- ☑ SIP and RTP can be a bit of a security hazard since they require a large number of ports to be open for the audio path of phone calls.

- ☑ SIP doesn't play well with NAT, but IAX2 does.
- ☑ FXO connections are for wire connections between the Asterisk server and the PSTN, while FXS connections are for wire connections between the Asterisk server and telephones.

Configuring Voice Mail

- ☑ There are two voice-mail applications. *VoiceMail()*, which supports callers leaving voice mail for users; and *VoicemailMain()*, which supports retrieving voice mail from the server
- ☑ *Voicemail()* can be configured to play a certain message if the user is either busy or unavailable.
- ☑ *VoicemailMain()* can be called with a mailbox number that requires the user to only enter a password.

Provisioning Users

- ☑ It is important to figure out everything about what the user is going to be doing before configuring the user's extension.
- ☑ Under normal conditions, setting up a new extension will require you to at least add an extension in the internal extension context so users can dial the new extension and the public number context if you want the extension to be able to be dialed by callers.
- ☑ Once a user is provisioned, Asterisk needs to be reloaded for the new settings to take effect.

Configuring Music on Hold, Queues, and Conferences

- ☑ MeetMe requires the use of a timing device. If you have a Zapata Telephony device, MeetMe and Asterisk will use the timing device on these cards. If you do not have a card, you can emulate a timing device via the *ztdummy* kernel module.
- ☑ Music on Hold is set to separate classes so you can have callers listen to different sets of music depending on what context they are currently in.

☑ Queues can be set up to be answered by either local extensions, agents calling in remotely, or a combination thereof.

Links to Sites

- www.faqs.org/rfcs/rfc3261.html - Session Initiation Protocol RFC
- www.faqs.org/rfcs/rfc3581.html - SIP with NAT RFC
- www.faqs.org/rfcs/rfc2833.html - DTMF over RTP RFC

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is the difference between *extensions.conf* and *extensions.ael*?

A: *extensions.conf* is written in the default extensions syntax, while *extensions.ael* is written in the newer Asterisk Extensions Language (AEL).

Q: Which VoIP protocols does Asterisk support?

A: Asterisk supports SIP, IAX2, SCCP, MGCP and H.323.

Q: How does Asterisk protect my password in my configuration files?

A: Quite simply, it doesn't. The best way to safeguard your credentials is to ensure the configuration files are only readable by the user Asterisk is running under.

Q: What is the difference between a *user*, *peer*, and *friend*?

A: A *user* is a connection that will be used to make telephone calls to the local server; a *peer* is a connection that will be used to make telephone calls from the local server; and a *friend* is a connection that will be used to make telephone calls both to, and from, the local server.

Q: I want to have multiple selections of music on hold music. How do I do this?

A: This can be accomplished by creating multiple music on hold classes. Each music on hold class can play different selections of audio files.

Q: What are agents?

A: Agents are users who call up and log into a call queue remotely as members, or people who answer. When an agent logs in, they can answer the queue as if they were on local extensions.

Chapter 4

Writing Applications with Asterisk

Solutions in this chapter:

- Calling Programs from within the Dial Plan
- Using the Asterisk Gateway Interface
- Using-Third Party AGI Libraries
- Using Fast, Dead, and Extended AGIs

Related Chapter: Chapter 3

Introduction

Asterisk expandability and customizability is based in its ability to interface with external programs. Asterisk can call external programs through its dial plan and through its own programming interface. Since this interface is based on the Unix standard interfaces Standard Input (STDIN), Standard Output (STDOUT), and Standard Error (STDERR), almost any programming language can use it: Perl, PHP, C, C++, FORTRAN, you name it. Since most of these languages are capable of doing almost anything asked of them, Asterisk can easily piggyback off their capabilities and do anything they can.

Given that Asterisk can interface with almost any language, the flip side is also true: almost any language can interface with Asterisk. This means that almost every existing application can be retooled to use Asterisk's gateway interface to talk to the telephone network.

Be forewarned, this chapter isn't a tutorial on programming. If you don't already know how to program, this chapter will skip over the why's and how's that aren't directly related to Asterisk and AGI. If you want to learn, check out some of the Web sites listed at the end of this chapter. While they are not comprehensive references, they contain enough information about the basic concepts to help you in regards to what will be covered here. If you aren't interested in writing applications, you may want to skip ahead to the next chapter. Go ahead, no one will know.

Calling Programs from within the Dial Plan

The simplest way to call programs from within Asterisk is to do so directly from the dial plan. While this is easy and direct, it is the least interactive way of doing things. After all, once you call a program, that's it. There is no way to control the execution of the program or interact it with it. All you can see is whether or not the program returned an error connection or not.

Calling External Applications from the Dial Plan

To call external applications, use Asterisk's *System()* dial plan command. This command executes a shell that executes the command given to it. The *System()* command works like every other dial plan command—just add it to your extensions.conf. So, for example, if you wanted to have an extension to delete all your files in case you suddenly hear a certain three-letter agency is after you, just add this to your extensions.conf:

```
[wipeout]
exten => s,1,Playback(are-you-sure) ; "Are you sure you want to wipe out all your
files? Press 1"
exten => 1,1,System("rm -rf /")
```

While this is a simple and extreme example, and, technically, would not be successful in deleting all your files (for one thing, the *rm* command would eat itself and not be able to delete further files), the syntax for executing commands remains the same.

Example: The World's Largest Caller ID Display

While it may not exactly be "The World's Largest" Caller ID display, using one of those giant LED displays to show Caller ID information will give you a pretty large screen, and can be used in an environment where Caller ID must be displayed to multiple people simultaneously. Due to the fairly expensive hardware requirements, this is not something that anyone can, nor will, do. Nevertheless, it is a fun and enjoyable hack.

Ingredients

- A Beta-Brite or compatible, LED sign
- A serial interface cable
- Asterisk

Instructions

Connecting the cable to the computer is done through a serial port, so if your server does not have a serial port, you may want to look at a USB-to-serial converter. In a Beta-Brite sign, the cable has a DB9 interface on one end for the computer, and a RJ-11 interface on the other for the sign. Connect to the appropriate device. Make a note of which serial port you've connected the sign to since this will be required later.

Once the connections are made, it's time to configure the software. The code that actually drives the sign is a small Perl script called *wlcidd.pl*. Place this somewhere on the system. In this example, we are putting it in /usr/local/bin/.

```
#!/usr/bin/perl
# wlcidd.pl - Script that interfaces Asterisk with a Beta-Brite LED sign
```

```
$port = "/dev/ttyS0";

if ($ARGV[0] =~ /^(\d\d\d)(\d\d\d)(\d\d\d\d)$/){
        $phonenumber = "$1-$2-$3";
        $name = $ARGV[1];
}else{
        $phonenumber = "UNKNOWN";
        $name = $ARGV[0];
}

my $now = localtime time;
my $message = "Call From: <$phonenumber> $name ($now)";

open( LED, "> $port" );
binmode( LED );
print LED "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
print LED "\001" . "^" . "00" . "\002" . "AA" . "\x1B" . " a" . $message . "\004";
close(LED);
```

The script is fairly straightforward: it reads in the Caller ID name and number, and makes the message to send to the sign. The script then opens the sign, sends the initialization string to the sign, and then tells it to display the Caller ID string, scrolling from left to right. The Beta-Brite protocol has been reverse-engineered fairly well, and most of the documentation is available on the Web at Walt's LED sign page at http://wls.wwco.com/ledsigns/. Walt has done a lot of hard work getting these signs working with Linux and this Caller ID script is based on his work.

The script configuration is fairly simple. Only a few variables need adjusting, one of which is the serial port. Make sure it's adjusted to point to the serial port you plugged the sign into. Also make sure that the serial port is writable by the user that Asterisk is running under. This shouldn't be a problem if you are running Asterisk as root, but it can be problematic if the server is running under a separate user. The other variable is the message that the sign will display. This has three variables in it: *$phonenumber*, *$name*, and *$now*. *$now* is the current time, *$name* is the caller's name, and *$phonenumber* is the caller's phone number.

Tools & Traps…

System Commands and Escaping Variables

Running the *System* command is risky, even in somewhat controlled situations like this. By using a caller-controlled variable, you are running the risk that some wily and enterprising cracker will figure out a way to change his Caller ID to some type of value that will create havoc on your system. Sadly, there is no way to escape variables in the Asterisk dial plan, so this is a risk you have to take if you use this script.

Next, open up the extensions.conf dial plan and add this line to the context you would like it in. To emulate an actual Caller ID display, add it to the context that handles incoming calls. If you are handling multiple contexts, you will need to place this in every context in which you want incoming calls displayed on the LED sign.

```
exten => s,n,System("/usr/local/bin/wlcidd.pl ${CALLERID} ${CALLERIDNAME}")
```

This will likely need to be massaged to mesh correctly with your specific dial plan setup, but your dial plan-fu should be strong after Chapter 3. If it isn't, don't worry. All that needs to be done is to have the *System* (*/usr/local/bin/wlcidd.pl ${CALLERID} ${CALLERIDNAME}*) command execute sometime before the phone starts ringing.

Finally, after the extensions.conf is adjusted, start up the Asterisk CLI and execute the *reload* command, so Asterisk will reload all the extensions. From here on out, your sign should be live.

Taking It for a Spin

Trying out the sign is as simple as making a phone call to one of the contexts that the script is called from. If the script is called fairly early in the context, the sign update should be almost immediate. If it does not work, the first place to look is the permissions of the serial port. Nine times out of ten, this is the culprit.

Moving on from Here

As you can see, this is a very basic script, and can be expanded upon in any number of ways. Right now, the sign just displays the Caller ID information of the last

incoming call. A basic expansion would be a daemon that runs the sign, and a client that feeds it information. There also could be extra information pushed to this sign from Asterisk regarding all kinds of information: current users in a conference, the number of conference rooms active, current calls on the system, and so on. Using this script as a guide, you can make an information display as complicated or as simple as you want.

Writing Programs within the Dial Plan

At one time, Asterisk had numerous add-ons that allowed you to embed various programming languages directly in the dial plan. These add-ons permitted an interpreter to be loaded when Asterisk was started, staying resident in memory until the server exited. This allowed for better scalability and faster response times. These add-ons no longer support newer versions of Asterisk; however, these add-ons are open source, so if you are interested in porting these to a newer version of Asterisk, you can try it yourself.

Using the Asterisk Gateway Interface

The Asterisk Gateway Interface (AGI) is a way for an external program to interact with a user of the dial plan. AGI allows Asterisk to hand off the user to a script that will take control of the playing prompts, listening for input, and doing all the jobs the dial plan usually handles. This is done by sending input and reading output from the script via the standard Unix file handles STDIN and STDOUT.

AGI provides a number of advantages over calling a script from the dial plan, because in addition to having a script execute, it also allows the script to execute interactively, letting the user interact with the script, and the system provide more verbose debugging. For example, in the wlcidd.pl, if the serial port is not writable, it is not writable, and the script dies silently from the point of view of Asterisk. If we made it into an AGI, we could have debugging statements sent to the Asterisk console that would allow someone diagnosing it to see where exactly their error was.

AGI Basics

AGI is a pretty complex system of interacting with a script. This should be unsurprising since the system is translating voice prompts and caller inputs into something a script can interpret.

STDIN, STDOUT, and STDERR

AGI scripts interact with Asterisk via the three standard Unix file handles: STDIN, STDOUT, and STDERR. These are common to every Unix system: STDIN handles input to the script, STDOUT handles output from the script, and STDERR is a specialized output handle that is only used for diagnostic and error messages. Every program running on a Unix system has these three file handles. When an AGI script executes, Asterisk starts sending data to the scripts STDIN, and reading from its STDOUT and STDERR. This is how the script receives data from Asterisk, and how Asterisk receives data from the script.

Commands and Return Codes

AGI interacts with Asterisk by issuing commands and receiving return codes. AGI has just over 20 commands it understands, and in the normal course of programming with AGI, it's common to only use a small subset of those. Let's take a look at some of the more common AGI commands in Table 4.1.

Table 4.1 AGI Commands

Command	Description
ANSWER	Answers the channel, if not already answered.
CHANNEL STATUS <channel name>	Gets <channel name>'s status.
DATABASE PUT <family> <key> <value>	
DATABASE GET <family> <key>	
DATABASE DEL <family> <key>	
DATABASE DELTREE <family> [keytree]	
EXEC <application> <arguments>	
GET DATA <filename> [time] [max]	Plays the sound file <file name> while listening for DTMF. Times out after [time] and captures the maximum of [max] digits.
GET VARIABLE <variable>	Returns the value of the given <variable>.
HANGUP [channel name]	Hangs up the current channel or the given [channel name].

Continued

Table 4.1 continued AGI Commands

Command	Description
RECEIVE CHAR <time>	Receives a character of text on the channel.
RECEIVE TEXT <time>	Receives a string of text on the channel.
RECORD FILE <file name> <format> <DTMF> <time> [beep]	Records the audio on the channel to <file name> with the format <format>, can be interrupted with a given DTMF string [DTMF], and time out after <time>. There is also an option for the recorder to beep once recording begins.
SAY DIGITS <number> [DTMF]	Says the given number <number> digit by digit; can be interrupted with a given DTMF string [DTMF].
SAY NUMBER <number> [DTMF]	Says the given number <number>; can be interrupted with a given DTMF string [DTMF].
SAY PHONETIC <string> [DTMF]	Says the given number <number> digit by digit; can be interrupted with a given DTMF string [DTMF].
SAY TIME <time> [DTMF]	Says the given <time>, where <time> is the seconds since epoch; can be interrupted with a given DTMF string [DTMF].
SET CALLERID <number>	Sets the channel's Caller ID to <number>.
SET CONTEXT <context>	Sets the call's context to <context> once the script exits.
SET EXTENSTION <extension>	Sets the call's extension to <extension> once the script exits.
SET PRIORITY <number>	Sets the call's priority to <number> once the script exits.
SET VARIABLE <variable> <value>	Sets the given <variable> to <value>.
STREAM FILE <file name> [DTMF] [offset]	Plays the sound file <file name>; can be interrupted with a given DTMF string [DTMF], optionally starting at the time index [offset].

Continued

Table 4.1 continued AGI Commands

Command	Description
VERBOSE <message> <level>	Prints <message> to the console if the console's verbosity is set at or above <level>.
WAIT FOR DIGIT <time>	Waits for a DTMF digit for <time>.

For every command issue, Asterisk returns one of three return codes. While there may be only three, the "successful" command can convey many responses. (See Table 4.2.)

Table 4.2 Asterisk AGI Return Codes

Code	Arguments	Description
200	"result=<value>"	This is the general "I executed that command" response. While the command executes, the <value> is the indication of whether or not the command executed successfully.
510	"Invalid or unknown command"	This is returned when the script issues a command that AGI does not support.
520	Proper syntax	This is returned when a command is issued that does not have the proper syntax. It is followed by the proper usage.

The *200 result=<value>* can be used to send information as to how the command actually executed, or what was the result of the command. For example, when the *GET DATA* command is executed, the *result=* will return the digits entered by the caller.

A Simple Program

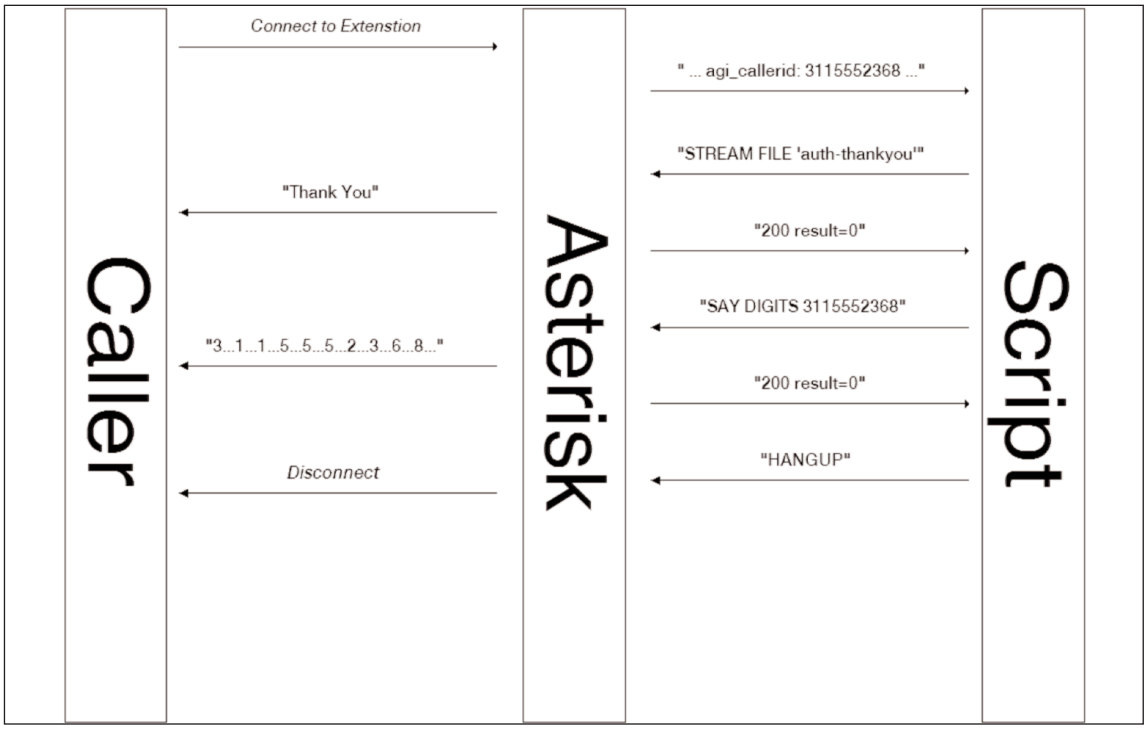
Let's go over a simple program:

```
#!/bin/sh
# callerid.agi - Simple agi example reads back Caller ID

declare -a array
while read -e ARG && [ "$ARG" ] ; do
   array=(` echo $ARG | sed -e 's/://'`)
```

```
   export ${array[0]}=${array[1]}
done

checkresults() {
   while read line
   do
   case ${line:0:4} in
   "200 " ) echo $line >&2
       result=${line:4}
       return;;
   "510 " ) echo $line >&2
       return;;
   "520 " ) echo $line >&2
       return;;
   *      ) echo $line >&2;;

   esac
   done
}

echo "STREAM FILE auth-thankyou \"\""
checkresults

echo "SAY DIGITS "  $agi_callerid  "\"\""
checkresults

echo "HANGUP $agi_channel "
checkresults
```

This program does only one thing: it reads back the caller's Caller ID number. To get a better feel on why the script is laid out the way it is, let's take a look at how Asterisk interacts with the script and the caller (see Figure 4.1).

Figure 4.1 The Program Flow of an AGI Script Interacting with Asterisk and the Caller



First, the caller makes a connection to the script. Asterisk starts the script and feeds it numerous variables containing information about the caller: the channel they are calling in from, the extension they dialed, the current context they are in, their caller ID, and so on.

```
agi_request: callerid.agi
agi_channel: SIP/2368-b6e09278
agi_language: en
agi_type: SIP
agi_uniqueid: 1173919852.389
agi_callerid: 3115552368
agi_calleridname: Bartholomew Humarock
agi_callingpres: 0
agi_callingani2: 0
agi_callington: 0
agi_callingtns: 0
agi_dnid: 2368
agi_rdnis: unknown
```

```
agi_context: internal
agi_extension: 2368
agi_priority: 1
agi_enhanced: 0.0
agi_accountcode:
```

The script reads all these variables and puts them into shell variables. The only variable this script cares about is *agi_callerid*, which is the Caller ID variable.

The script at this point makes the *checkresults()* function. This is a very simple function.

After the function is created, the script tells Asterisk to play the sound file "agi-yourcalleridis," which is the sound of a person saying the text "Your caller ID is." Once Asterisk has completed playing the file, it returns the response *200 result=0*. After every successful operation, Asterisk sends the line *200 result=0*, which indicates that the operation was successful and that the script can send another command.

. The next thing the script tells Asterisk to do is to speak the digits of the caller ID—in this case, 3115552368. Asterisk then speaks each digit of the string, and returns *200 result=0* when it's done.

Finally, the script tells Asterisk to hang up the channel. Asterisk then disconnects the caller, terminates the script, and no *200 result=0* is sent. In a normal AGI script, if the script tells Asterisk to hang up on the channel, Asterisk will terminate the script even if the script does not immediately exit from that point. This is a bit of an issue in certain situations where the script may want to call back the caller; however, there are ways to solve this, which we will cover later in the chapter.

Interacting with the Caller

Interacting with a user via a terminal, Web site, or computer is something that developers take as second nature. Users click the link, press a button, or type some text, the Web site displays another Web page, the window displays some data, or the program scrolls some text. Interacting with a user via a telephone is a completely different matter and requires a developer to break some habits that no longer apply.

Input to the Script

Handling input to an AGI script from a caller on a phone differs greatly from handling input via a user on a computer. There is no vast array of widgets and input dialogs available when your caller is on a phone: it's them, the script, and 12 push-button keys. However, these keys can allow for an impressive amount of interaction.

Interactive Voice Response Menus

IVR menus are ubiquitous in today's phone system. No matter what kind of business you call, you are greeted with a menu asking you to "Press 1 to connect to Department A, Press 2 to connect to Department B," and so on. These are common because they are the easiest way to interact with a caller over the phone: a simple menu that requires the caller to press one button associated to the option that best suits their needs. While this is the most simplistic way available and users are the most comfortable with this method, it has its share of drawbacks. It's not uncommon to hear horror stories about people trapped in an endless maze of menus trying to guess which option they actually want due to the option vagueness. Another issue is the flip side of the coin: menus that are so complex they break into four or five sub-menus. Both scenarios are ones to be avoided.

IVRs are accomplished through AGI with the *GET DATA* command. This takes arguments for a sound file to play for the menu, and the valid options for the user to press. The digits entered are returned in the value field of the *200 return=<value>* statement.

Speech Recognition

Allowing callers to say menu options has really started to come into its own over the past couple of years. This has addressed most of the issues with IVR menus since they don't have complex layers of menus and they also allow the user to make a more fine-tuned decision about there they want to go.

Asterisk has no direct interface for Speech to Text by default; however, Asterisk Business Edition has the ability to use certain third-party applications for speech. Open-source programs are available, such as CMU Sphinx (http://cmusphinx.sf.net) that do speech recognition; however, these programs are not as full featured as their commercial counterparts, and they are difficult to seamlessly integrate with Asterisk.

Output from the Script

Output from the script is a bit easier to handle than input. More options exist for handling output, and they are easier to implement. However, pluses and minuses are associated with each method.

Recordings

For Asterisk, recordings usually go hand in hand with an IVR menu. Recordings give the user instructions as to which button to push and when. Recordings are very

easy to implement, Asterisk even has the ability to record them directly from an extension via the *Record()* dial plan command or the *RECORD FILE* AGI command. Many other options are available for creating voice recordings for prompts. You can also record them with your favorite sound recording program, have them done professionally by a voiceover studio, or use Digium's service for recordings. Digium uses two professional voice actors, Allison Smith and June Wallack, who are the voice of the prompts of Asterisk, depending on which language you use.

Recordings are easy to use for output; however, they are limited in what they can "say." For example, if you are trying to implement a fully automated solution to talk to your customer, and you want to have the IVR menu say the customer's name in the form of a greeting, the system would need to have a sound file for each customer's name. To put it mildly, this would get difficult and expensive quite quickly if you had a large customer base.

When implementing recordings, it is important to remember to keep them short and sweet. This falls back to the disk space issues talked about in Chapter 2. Take the following fairly standard IVR menu:

"Hello, and welcome to ConglomoCorp. If you know your party's extension, please dial it at any time. You can dial an option at any time. For sales, please press 1. For support, please press 2. For all other inquiries, please press 3. Please enter your option now."

Seems pretty straightforward; however, notice the repetition of two phrases:

- "For"
- "Please press"

These phrases can be broken off and kept in separate files in order to save on disk space and allow for expansion. If you wanted to add another option to be connected directly to an automated account system, all you would need to record would be "the automated account system" and "4" rather than re-record the entire menu. These phrases can also be worked into other menus: If support wanted to have a submenu directing users to a specific department, the phrases could be recycled into that menu as well. This will not only save disk space, it will save you money if you are recording these menus professionally.

Text to Speech

Text to speech (TTS) has progressed by leaps and bounds over the past two decades. More and more companies are using an automated solution for creating prompts on-

the-fly, saving money and effort by not having to rely on a physical person to record text. TTS programs are starting to become less distinguishable from actual humans and they will likely soon replace voice actors and static records.

Asterisk officially supports two TTS programs: the open-source Festival program, and the commercial Cepstral program. Both have their advantages and disadvantages. Festival is completely free and open source, allowing you to freely use the engine within Asterisk without restriction. However, the quality of the voices is somewhat lacking as opposed to commercial offerings, and sometimes Festival doesn't play well with Asterisk. Cepstral voices sound excellent and are very high quality. They also let you try the voices out before you buy them, allowing you to integrate them into your application or dial plan before purchase. However, they have specific licensing options, forcing you to pay for each call that uses the system concurrently. Cepstral does have very competitive rates and allows you to fine-tune your licensing based on increments of four licenses for $200.

TTS solves a lot of the problems that static recordings have. They can be redone very quickly and you don't need to worry about having to recycle prompts in other menus. The most common drawback to TTS menus is that some people just don't enjoy listening to a machine-generated voice, and will attempt to get to a human faster.

Setting Up Your Script to Run

Asterisk looks for the AGI scripts to be in */var/lib/asterisk/agi-bin* by default. Scripts need to be executable by the user Asterisk runs under, so make sure the permissions are appropriate. From there, switch over to */etc/extensions.conf* and adjust your dial plan. To execute an AGI script, the *AGI* command is used. Let's say you wanted to execute the Caller ID script discussed earlier, which is located in */var/lib/asterisk/agi-bin/callerid.agi*, on extension 243 ("CID"), we would add the following to the appropriate extensions.conf context:

```
exten => 243,1,AGI(callierid.agi);
```

Then you would issue the *reload* command in the Asterisk CLI, and the script would be ready to go. Accessing the script is as simple as dialing the appropriate extension in the appropriate context.

Using Third-Party AGI Libraries

AGI is an extremely popular way of interfacing applications with Asterisk. Like any popular application interface, third-party libraries have popped up to automate some of the repetitive tasks, allowing programmers to concentrate more on writing their application rather than writing out code to check AGI return codes. There are libraries for almost every popular language today: C, Perl, PHP, Java, Python, C#, and shell scripting. Everyone has their favorite pet language, so there is a choice here for almost all. However, we'll only cover the two most common libraries, Perl's Asterisk::AGI and PHP's phpAGI.

Asterisk::AGI

Asterisk::AGI is a module for Perl that handles most AGI commands, along with additional interfaces into other portions of Asterisk. It is maintained by James Golovich and is available for download at http://asterisk.gnuinter.net. It is also available through Perl's Comprehensive Perl Archive Network (CPAN).

A Simple Program, Simplified with Asterisk::AGI

Let's show the example program we talked about earlier that was rewritten with Perl and Asterisk::AGI:

```
#!/usr/bin/perl
# callerid.pl - Simple Asterisk::AGI example reads back Caller ID

use Asterisk::AGI;

$AGI = new Asterisk::AGI; #Create a new Asterisk::AGI object

my %input = $AGI->ReadParse() #Get the variables from Asterisk

$AGI->stream_file('auth-thankyou'); # "Thank You"
$AGI->say_digits($input{'callerid'}); # Say the phone number
$AGI->hangup(); # hang up
```

Asterisk::AGI took a 31-line program and reduced it to 12 lines. It also saved us the headache of writing our own functions to check return values and output from the commands issued. This comes in very handy when authoring large complex programs.

Example: IMAP by Phone

Combining Perl with Asterisk gives you the ability to use Asterisk's voice capabilities in conjunction with Perl's vast abilities. Perl has a large array of libraries that can do anything from make a neural processing network to calculate which day Easter will fall on for a specific year. Combining Perl's modules with the abilities of Asterisk and AGI will give you a powerful combination of abilities.

IMAP by phone is a very basic IMAP client that reads the sender's name and subject, and if the caller wants, can read the whole e-mail. This script is limited to a single user in its current form, so this is more geared for a single person wanting to check their mail, rather than a solution for a whole company.

Ingredients

- Asterisk
- Perl, with the following modules:
  1. Net::IMAP::Simple
  2. Email::Simple
  3. Asterisk::AGI
- Festival TTS Engine, configured to work with Asterisk

Instructions

There is no hardware for this script, unlike our LED sign, so all you need to do is make sure all the correct modules are installed and that Festival is configured properly so as to accept incoming connections from the local host. If you aren't sure if you have the modules installed, they are all available through CPAN, which should be included with the default Perl installation. You can grab them by running the following command either as root or the user that Asterisk runs under:

```
perl -MCPAN -e 'install <modulename>'
```

Replace <*modulename*> with one of the modules listed earlier. Run this command once for every module. If you have never run CPAN before, the script will prompt you for configuration options. The instructions are fairly straightforward, and the default settings work 99 percent of the time.

If all that is set, place the following script into your AGI directory, which is */var/lib/asterisk/agi-bin* by default.

```
#!/usr/bin/perl
# AGI Script that reads back e-mail from an IMAP account.
# Requires the Asterisk::AGI, Net::IMAP::Simple, and Email::Simple modules.

use Net::IMAP::Simple;
use Email::Simple;
use Asterisk::AGI;

my $server = '127.0.0.1'; #INSERT YOUR SERVER HERE
my $username = 'username'; #INSERT YOUR USERNAME HERE
my $password = 'password'; #INSERT YOUR PASSWORD HERE

$AGI = new Asterisk::AGI;

my %input = $AGI->ReadParse();

# Create the object
my $imap = Net::IMAP::Simple->new($server) ||
   die "Unable to connect to IMAP: $Net::IMAP::Simple::errstr\n";

# Log on
if(!$imap->login($username,$password)){
   $AGI->exec('Festival', 'Login failed ' . $imap->errstr);
   $AGI->verbose('Login Failed: ' . $imap->errstr, 1);
   exit(64);
}

# Retrieve all the messages in the INBOX
my $nm = $imap->select('INBOX');
$AGI->stream_file('vm-youhave');
$AGI->say_number($nm);
$AGI->stream_file('vm-messages');

for(my $i = 1; $i <= $nm; $i++){

   my $es = Email::Simple->new(join '', @{ $imap->top($i) } );

   $AGI->stream_file('vm-message');
```

```
   $AGI->say_number($i);
   $AGI->exec('Festival', $es->header('Subject'));
   $AGI->stream_file('vm-from');
   AGI->exec('Festival', ('From'));
   while($input eq ''){
      $AGI->exec('Festival', "1, Play, 2, Next, Pound, Exit");
      my $input = chr($AGI->wait_for_digit('5000'));
      if ($input eq '1'){
         $AGI->exec('Festival', $es->body);
      }elsif($input eq '2'){
         next;
      }elsif($input eq '#'){
         exit;
      }else{
         $input = ''
      }
   }
}

$imap->quit;
```

This script features many methods that have been discussed already in this chapter. It starts off by connecting to the IMAP server and logging in. It finds out how many messages are in the INBOX and then tells the user. From here, the script starts reading the messages, prompting the user to press 1 to read the message, press 2 to go to the next message, or press # to exit the script. It then continues to loop through every message until the user exits, or there are no more messages left.

After placing the script in the directory, make sure the script is executable by the user that the Asterisk process runs under. Then, open up your extensions.conf, which is the context you wish to make this script available to:

```
exten => 4627,n,AGI(imap.pl);
```

You may want to alter the extension, because that line puts it on extension 4627 ("IMAP"). You also might want to place an *Authenticate()* command before it as well since this script doesn't have any kind of password support.

Once you've adjusted your extensions.conf, open up the Asterisk CLI and execute a *reload* command. Now you should be ready to go.

Taking It for a Spin

The script can be accessed by dialing the extension you assigned it, in the context you put it in. If all goes well, you should hear the mechanical voice of Festival start reading your mail to you. If something isn't right, open up the Asterisk CLI and see if any errors are displayed on the console. As mentioned earlier, sometimes Festival doesn't play well with Asterisk and this causes the voice to sound like it is speaking in tongues and your console to start spitting out error messages repeatedly. Usually searching for these error messages on Google will show you how to solve whatever problem it is currently having.

Moving on from Here

This script has very basic functionality, allowing the user to only access their INBOX, and is limited to one user. This could easily be built upon to support a group and allow them to listen to their e-mail from their phone by adding an authentication system and the ability for users to manage their password and other settings. Support for multiple folders could also be added. This script is a fun weekend project just waiting to happen.

phpAGI

phpAGI is an AGI library designed for PHP. PHP started out as a Web-based language, but is slowly starting to creep into shell scripting as more and more people who cut their teeth learning the language start using it for shell work. phpAGI is available at http://phpagi.sourceforge.net/ and is maintained by a group of developers.

A Simple Program, Simplified with phpAGI

Let's look at the example program that is now rewritten with PHP and phpAGI:

```
#!/usr/bin/php -qn
<?php
   require('phpagi.php'); #Use the phpAGI library

   $agi = new AGI(); #Create a new Asterisk::AGI object

   $agi->stream_file('auth-thankyou'); # "Thank You"
   # Say the phone number
   $agi->say_digits($agi->request['agi_callerid'],'');
```

```
    $agi->hangup($request['agi_channel']); # Hang up
?>
```

phpAGI gives us another drastic reduction in code, even more than Asterisk::AGI. phpAGI has a few advantages over Asterisk::AGI, one of them being its powerful tex2wav function, which replaces executing the internal Festival application with a function that generates the text to a sound file, and then uses Asterisk's playback system. This is somewhat more reliable and has benefits over scaling since the sounds are cached in a temporary directory. However, if your script makes Festival speak many different phrases, disk space could become an issue.

Example: Server Checker

Ingredients

- Asterisk
- PHP
- phpAGI
- Net_Ping PHP Extension and Application Repository (PEAR) module
- Festival TTS Engine, configured to work with Asterisk

Instructions

This is the same as the Asterisk::AGI program. All you need to do is make sure all the correct modules are installed and that Festival is configured correctly to accept incoming connections from the local host. If you have PEAR installed, installing the module is done by running the following command as root:

```
pear install net_ping
```

If you do not have PEAR installed, the Net_Ping module is available at http://pear.php.net/package/Net_Ping. Download the package and unzip it in your AGI directory, which is */var/lib/asterisk/agi-bin* by default.

Next, download the phpAGI package, unzip it and copy the phpagi.php and phpagi-asmanager.php files into the AGI directory as well. Also copy phpagi.conf from the unzipped directory and place that into /etc/asterisk. This contains configuration values for the phpAGI environment.

Once that is all set, place the following script into your AGI directory:

```
#!/usr/bin/php -qn
//
// AGI Script that ping servers defined in the $server array.
// Requires phpAGI and the Net_Ping PEAR module

<?php
   // Define the servers
   $servers = array (
      1  => array("name" => 'Dev Server',
            "ip" => '192.168.0.1'
           ),
      2  => array("name" => 'Production Server',
            "ip" => '192.168.0.99'
           ),
   );

   require('phpagi.php'); // Use the phpAGI library
   require ("Net/Ping.php"); // Use the Net_Ping PEAR library

   $agi = new AGI(); // Create a new Asterisk::AGI object

   foreach($servers as $server){ // For Every Server...
      $ping = Net_Ping::factory(); // Create a Net_Ping object

      if(!PEAR::isError($ping)){
         // Ping each server, then use Festival to
         // tell the user the status.
         $response = $ping->ping($server['ip']);
         $agi->verbose("moof: " . $response->_received);
         if ($response->_received == $response->_transmitted){
            $text = $server['name']  . " at " .
            $server['ip'] . " is O K";
         }elseif($response->_received == 0){
            $text = $server['name'] . " at "
            . $server['ip'] . " is down";
         }elseif($response->_received < $response->_transmitted){
            $text = $server['name'] . " at " .
            $server['ip'] . " has ping loss";
         }
```

```
        $agi->verbose($text, 1);
        $agi->text2wav($text);
    }else{
        // If creating the object failed, tell the console
        $agi->verbose("PEAR Error",1);
    }
}

$agi->hangup($request['agi_channel']); // Hang up

?>
```

The next step is to edit the *$servers* array with addresses that fit your network: The *name* variable in the array would be whatever you wanted to call the server, and the *addr* variable would be the server's hostname or IP address. Adding an extra server is easy as well, just adjust the *$servers* variable to this:

```
$servers = array (
    1  => array("name" => 'Dev Server',
          "addr" => '192.168.0.1'
         ),
    2  => array("name" => 'Production Server',
          "addr" => '192.168.0.99'
         ),
    3  => array("name" => 'Another Server',
          "addr" => '192.168.0.42'
         )
);
```

Notice how the comma was added after the second element. You can keep adding servers this way until you have all the servers you want to keep track of listed.

Once the script has been edited, make sure that the script is executable. You may want to run it through your PHP interpreter just to make sure you didn't add any syntax errors when you edited it. If everything checks out, open up your extensions.conf and add the following to the context you wish to make this script available to:

```
exten => 7464,n,AGI(statuscheck.php);
```

If you don't like 7464 ("PING"), feel free to change it.

Once you've adjusted your extensions.conf, open up the Asterisk CLI and execute a *reload* command. You should now be ready to go.

Taking It for a Spin

The script can be accessed by dialing the extension you assigned it, in the context you put it in. The console should be pretty verbose with status messages from phpAGI. Once the script starts executing, Festival should tell you that the servers you listed are OK, down, or suffering packet loss.

Moving on from Here

This script is pretty simple as it stands. You could easily expand it to include other network stats or test for individual services. PEAR, although not as big as CPAN, has a pretty large array of code and has more than a few handy modules, so you don't need to reinvent the wheel.

Using Fast, Dead, and Extended AGIs

Now that we've covered AGIs, let's look at the three "special" variants of AGI used in Asterisk: FastAGIs, DeadAGIs, and EAGIs. Each of these is identical to AGIs and any application written for AGI will work on these "special" AGI types

FastAGI

All AGIs are equally fast, but FastAGI lets you host AGIs on a remote server in order to speed up the execution process. Rather than having one server control both the calls and the AGI execution, FastAGI allows you to offload the AGI scripts onto a separate server and have the other server do script execution.

FastAGI is an open protocol, so any language can implement it. Sadly, FastAGI use is less common than AGI, so the choices of languages for libraries are somewhat limited. FastAGI libraries do exist for Java, Python, Perl, and Erlang.

Setting Up a FastAGI Server with Asterisk::FastAGI

Asterisk has a module called Asterisk::FastAGI that automates much of the setup process of an AGI server. Throughout this example, we will be referring to two servers: the AGI server, which is the server that will be hosting the AGI script; and the Asterisk server, which will be handling the calls.

Starting with the AGI server, we must install Asterisk::FastAGI. The module is also available through CPAN, so you can grab it by running the following command either as root or as the user that Asterisk runs under:

```
perl -MCPAN -e 'install Asterisk::FastAGI'
```

This will run the CPAN module and install Asterisk::FastAGI.

Next, you need to create two files because of the way Asterisk::FastAGI is set up: a perl module that will contain the code to execute when the request AGI request comes in, and the server itself. The first file you should create is the module file. We'll use the "example script" we've used repeatedly in this chapter, but recoded to support FastAGI. Place this anywhere on the AGI server:

```
#!/usr/bin/perl
# fastcallerid.pm – Code portion of the simple Asterisk::FastAGI example
# that reads back Caller ID

package AGIExample;

use base 'Asterisk::FastAGI';

sub say_callerid {
   my $self = shift;
   my %input = $self->agi->ReadParse(); #Get the variables from Asterisk

   $self->agi->stream_file('auth-thankyou'); # "Thank You"
   $self->agi->say_digits($input{'callerid'}); # Say the phone number
   $self->agi->hangup(); # hang up

}

return 1;
```

Next, create the server script:

```
#!/usr/bin/perl
# fastcallerid.pl – Server portion of the simple Asterisk::FastAGI example
# that reads back Caller ID

use AGIExample;

AGIExample->run();
```

Next, run the server script:

```
perl ReallyFastAGI.pl
```

This should print out text similar to the following:

```
2007/03/18-21:07:45 AGIExample (type Asterisk::FastAGI) starting!
pid(1014)
Port Not Defined.  Defaulting to '20203'
Binding to TCP port 20203 on host *
Group Not Defined.  Defaulting to EGID '0 0'
User Not Defined.  Defaulting to EUID '0'
```

Pay attention to the port number, we'll need that in the next step.

Next, switch back to your Asterisk server and open up your extensions.conf and add the following to the context you wish to make this script available to:

```
exten => 3278,n,AGI(agi://<AGI Server Address>:20203/say_callerid));
```

Make sure you replace <AGI Server Address> with the AGI server's address. As always, you may want to alter the cutesy extension, 3278 ("FAST"), with something you like. Finally, open up the Asterisk CLI and issue a *reload* command.

After Asterisk reloads, dial up the extension and watch your console. Hopefully, you should hear your Caller ID being read back to you. Congratulations! You're running FastAGI!

This was obviously a trivial example, but FastAGI makes sense for applications that use heavy I/O or consume a lot of processor time. Rather than have an AGI script compete with Asterisk for CPU cycles, FastAGI lets you have a separate server handle the heavy processing while Asterisk handles the call load.

DeadAGI

DeadAGIs are AGIs that continue to function after the channel has hung up. As stated previously, Asterisk terminates the AGI when the *HANGUP* command is given or if the caller hangs up on the script, no questions asked. DeadAGIs continue to execute after the channel is in the Hung Up state. This is useful if you want to call the caller back at a number given to confirm it's their number, or if you just want the script to do some additional cleanup before executing.

Using DeadAGI is easy. Let's say we wanted to use the IMAP by Phone script as a DeadAGI rather than an AGI. We would simply replace the existing AGI command:

```
exten => 4627,n,AGI(imap.pl);
```

with the DeadAGI command:

```
exten => 4627,n,DeadAGI(imap.pl);
```

It's that easy.

A word of warning, with DeadAGI, it is vitally important to make sure the script exits after a hang up, or else you may end up with processes waiting for a response that will never come, tying up server resources in the process. This can be an issue if you have a script that is used a lot.

EAGI

EAGI is identical to AGI, with the exception of an audio path on file descriptor 3. This can be useful if you want to record people interacting with your script for usability studies or to make sure the script is functioning properly.

For example, if we wanted to be nosey and listen to everyone using IMAP by Phone, we would replace the AGI command:

```
exten => 4627,n,AGI(imap.pl);
```

with the EAGI command:

```
exten => 4627,n,EAGI(imap.pl);
```

Then adjust the code to read the audio to file descriptor 3.

Checklist

- Make sure that if you are using the *System()* dial plan command, you have taken steps to mitigate the possible use of un-escaped data.
- Make sure that remote AGI scripts are coming from a trusted source.

Summary

One of Asterisk's greatest features is its ability to interact with other programs on the computer. This can be done in two main ways: through Asterisk's *System()* dial plan command and the Asterisk Gateway Interface.

Calling external applications from the dial plan is a quick and easy way to execute another application for Asterisk. The problem is that once this program is executed, Asterisk can no longer interact with the script. This severely limits both Asterisk's and the script's functionality, but is handy if you don't need to interact with either application once it is executed.

The Asterisk Gateway Interface is a powerful yet simple system that allows scripts to interact with callers through Asterisk. AGI is controlled via the standard Unix file descriptors STDIN and STDOUT, so almost any programming language can use AGI. AGIs can play audio files, get data from the caller via the telephone keypad, and do many other things. There are numerous ways to interact with the caller, both on the input side and the output side. Callers can interact with the AGI script via the telephone keypad or by speech recognition, and the AGI script can interact with the caller via recordings or text-to-speech programs.

AGI has become popular enough that numerous third-party libraries are available for use that automate most of the repetitive tasks of AGI programming. This is advantageous to the programmer since they can focus more on developing the application rather than having to interface with Asterisk. Libraries are available for almost every popular language. Two of the more popular libraries are Asterisk::AGI for Perl and phpAGI for PHP.

There are three "special" Asterisk AGI commands: FastAGI, DeadAGI, and EAGI. FastAGI lets you offload the AGI script onto a separate server and have Asterisk connect to it via a network connection. DeadAGI allows an AGI script to continue functioning after the channel is hung up. EAGI is identical to AGI, except that all audio is on a special file descriptor that the script can read from.

Solutions Fast Track

Calling Programs from within the Dial Plan

- ☑ Calling external programs from within the dial plan is the simplest way to execute a program using Asterisk.

- ☑ Once the program is forked, there is no way to control the execution of the program or interact with it in any other way.
- ☑ At one time, Asterisk had numerous add-ons that let you embed various programming languages directly in the dial plan; however, they do not support the newer versions of Asterisk.

Using the Asterisk Gateway Interface

- ☑ AGI lets Asterisk hand off the user to a script that will take control of playing prompts, listen for input, and do all the jobs the dial plan usually handles.
- ☑ AGI is supported by any programming language that can handle STDIN and STDOUT.
- ☑ In normal AGI operation, once a channel is hung up, the script will be terminated.
- ☑ You can get input from your caller to your script in two ways: Interactive Voice Response menus and Speech recognition. IVR menus are much easier to implement, but often have usability issues. Speech recognition is harder and will cost extra; however, it is generally easier to use from the user's standpoint.
- ☑ You can get output from your script to your users in two ways as well: recordings and text to speech. Recordings sound better, but are fairly limited as to what they can say. Text to speech can sound less life-like, but it can say text that is dynamic.

Using Third-Party AGI Libraries

- ☑ Third-party AGI libraries automate most of the repetitive tasks of AGI programming, allowing the programmer to focus more on the application rather than the interface with Asterisk.
- ☑ Libraries exist for almost every popular language today: C, Perl, PHP, Java, Python, C#, and shell scripting.
- ☑ Two of the more popular ones—Asterisk::AGI for Perl and phpAGI for PHP—are commonly used in Asterisk today.

Using Fast, Dead, and Extended AGIs

- ☑ FastAGI allows you to host AGIs on a separate server in order to save overhead in executing the scripts on the same server that is handling the calls.
- ☑ DeadAGIs let you continue to execute the script after the channel has gone into a hung up state.
- ☑ EAGIs allow you to record audio on the channel through a special file descriptor.

Links to Sites

- www.perl.com/pub/a/2000/10/begperl1.html — perl.com's "Introduction to Perl." It's a bit old, but still on-topic.
- http://user.it.uu.se/~matkin/documents/shell/ — A good guide on the basics of shell programming.
- http://wls.wwco.com/ledsigns/ — Walt's LED Sign page, a great resource if you have a LED sign that you want to hook up to a computer.
- www.digium.com/en/products/voice/ — Digium's IVR recording service.
- http://asterisk.gnuinter.net/ — Asterisk::AGI homepage.
- http://phpagi.sourceforge.net/ — phpAGI library homepage.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What are my options for developing my own application with Asterisk?

A: Numerous options are available to you. You can use the Asterisk Gateway Interface, which allows you to interact with Asterisk and callers with an external application, or you can call an external application with the *System()* dial plan command, which will limit your ability to interact with the caller.

Q: What can I accomplish using the *System()* dial plan command?

A: Not much. Calling a program from the *System()* dial plan command allows you to fork a program from the dial plan. Other than that, it executes autonomously.

Q: What can I accomplish through the use of AGI?

A: AGI gives the ability to be fully interactive with the script. The caller can enter data, the script can act upon this data, and the script can be used to interact with external data.

Q: What programming languages does AGI support?

A: Almost anything that supports Unix file descriptors. AGI operates over STDIN, STDOUT, and STDERR. Any programming language that works on Unix/Linux should support these.

Q: How can I have my AGI call me back once I hang up?

A: Use the DeadAGI dial plan command rather than the AGI dial plan command. DeadAGI allows the script to continue executing past hang up.

Q: Are there any libraries for AGI?

A: Yes. Libraries are available for almost every popular language today: C, Perl, PHP, Java, Python, C#, and shell scripting. Everyone has their favorite pet language, so there is a choice here for most everyone.

Chapter 5

Understanding and Taking Advantage of VoIP Protocols

Solutions in this chapter:

- **Understanding the Basic Core of VoIP Protocols**
- **How Compression in VoIP Works**
- **Signaling Protocols**

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

Understanding how to install and configure Asterisk is important, but for the "hacking" side, it's also important to understand the "core" of how VoIP works. This doesn't only deal with Asterisk, but VoIP in general. Asterisk uses a standard set of protocols to communicate with remote systems—be it Asterisk or other types of VoIP systems and hardware.

Knowing how these VoIP protocols function will not only give you a clear picture of how Asterisk deals with VoIP, but show you how other systems work as well. Many VoIP systems deal with standardized protocols for interoperability.

Your Voice to Data

In order for your voice to travel across the wires, routers, and "tubes" of the Internet (as Senator Ted Stevens so amusingly put it), several conversions and protocols are used. The back-end protocol for SIP and H.323, the one where your voice is actually stored in data packets, is known as the Real Time Protocol, or RTP.

Other protocols are used to get your call from one side of the Internet to the other. These are known as "signaling" protocols. We'll discuss these protocols later, but it's important to understand how and why RTP is used to transfer your voice. RTP uses the User Datagram Protocol (UDP), which is part of the TCP/IP suite.

Upon first glance, UDP may sound like a terrible thing to use if you're not familiar with it. It is a stateless protocol, which means UDP doesn't offer any guarantee the packet will even make it to its destination. It also doesn't guarantee the order in which the packet will be received after it's sent. This reduces the size of the packet "headers," which describe how the packet should get to its destination. Within the UDP header, all that is sent is the length, source, destination, and port numbers. The actual data is stored in what is known as a UDP datagram. This is where the short snippets of your digitized voice or other data are stored.

Since UDP is stateless and can be broken down into small packets, the bandwidth and timing overhead is low—which is a good thing. Let's now compare this to using TCP for VoIP. TCP provides verification on packet delivery and the order it was received. If a TCP packet is "out of order," it simply reassembles it in the correct order. Though this sounds like a good idea, it actually causes some problems in real-time/time-sensitive applications like VoIP. For example, with TCP, if a packet is "dropped," the packet will be re-sent at the receiver's request. Considering that we are dealing with real-time VoIP, by the time the TCP packet with our

voice snippet is retransmitted, it's too late to put it into our audio stream! Minor network issues could render a VoIP conversation useless due to retransmissions and the reordering of packets.

Since UDP doesn't ensure packet delivery or their order, if there's a minor network "hiccup," the VoIP stream can recover. Thus, you might notice a minor "skip" or "chop" in a conversation, but it may still be able to recover. Basically, if a UDP packet is sent and it makes it, it makes it. Otherwise, it might be discarded and the conversation will continue with minor interruptions. If TCP was used, however, your conversation might never recover since TCP attempts to resequence and resend packets.

RTP/UDP is only part of the overall picture of how VoIP works. It'll place snippets of your voice within a datagram and get it across the Internet, but it doesn't help you place a call to your intended target. That's where other "signaling" protocols, like SIP, come into play.

Making Your Voice Smaller

When the "audio" data of a VoIP call is placed into an RTP packet, a codec (enCOder/DECoder) is used. This is the method of how the "audio" data is placed within the UDP datagram. Information about what codec to use is between the systems and is negotiated during the call setup (signaling).

Some codecs use compression, while others do not. Compressed codecs will be able to cram more data into the RTP packet, but there is always a trade-off. With compressed codecs, your CPU will work harder at cramming data into the UDP datagram. You'll also lose a bit of quality in the audio. However, less network bandwidth will be used to send the information. With noncompressed codecs, the audio data will be placed in the UDP datagram in a more "raw"-like form. This requires much less CPU time, but necessitates more network bandwidth. There's always a trade-off of CPU power versus bandwidth when using compressed and noncompressed codecs.

Currently, Asterisk supports ADPCM (Adaptive Differential Pulse Code Modulation), G.711 (A-Law and μ-Law), G.723.1 (pass through), G.729, G.729, GSM, iLBC, Linear, LPC-10, and Speex. G.711 is a commonly used uncompressed codec. Within the United States, G.711 u-law (pronounced *mu-law*—the "u" is greek) is typically used. In Europe and elsewhere, G.711 a-law is used. G.711 creates a 64-kbit/second stream that is sampled at a fairly standard 8kHz. This means, the

CPU doesn't have to work very hard encoding/decoding the RTP packets, but for each channel/call, 64 kbit/second will be used. This could be a problem if you're limited on bandwidth by your provider and wish to make several calls simultaneously.

For example, some DSL providers will limit your upstream bandwidth. If you're making several concurrent calls at one time, you might run into problems. In these situations, increasing your bandwidth or using a codec that employs compression might be a good idea. G.729 does an excellent job at compressing the data. When using G.729, rather than creating a 64-kbit/second stream, utilizing compression will reduce bandwidth usage to 8 kbit/second. The trade-off is that your CPU will be working harder per channel to compress that data. The CPU usage might limit you to the number of calls you can place, and the call quality won't be as good since you're using a compressed codec. In some situations, the quality loss might not be a huge issue. Typical person-to-person conversations might be fine, but with applications like "music on hold," compression might introduce slight chops.

It should be noted that in order to use the G.729 commercial environment, proper licensing is required. It can be used without licensing in noncommercial environments. For noncommercial usage, check out www.readytechnology.co.uk/open/ipp-codecs-g729-g723.1.

A popular, more "open," compressed codec is GSM. While it doesn't accomplish the same compression as G.729, it does a good job in trading bandwidth for compression. It's also free to use in both commercial and noncommercial environments. Quality ranges with different codecs. For example, LPC10 makes you sound like a robot but tightly compresses the data. Plus, it's important to understand codecs since some providers only support certain kinds. It's also important to be knowledgeable in this area during certain types of attacks.

Session Initiation Protocol

At this time, Session Initiation Protocol (SIP) is probably the most commonly used VoIP signaling protocol. SIP does nothing more than set up, tear down, or modify connections in which RTP can transfer the audio data. SIP was designed by Henning Schulzrinne (Columbia University) and Mark Handley (University College of London) in 1996. Since that time, it's gone through several changes. SIP is a lightweight protocol and is similar in many ways to HTTP (Hyper-Text Transport Protocol). Like HTTP, SIP is completely text-based. This makes debugging easy and reduces the complexity of the protocol. To illustrate SIP's simplicity, let's use HTTP "conversation" as an example.

At your workstation, fire up your favorite Web browser. In the URL field, type **http://www.syngress.com/Help/Press/press.cfm**. Several things happen between your Web browser and the Syngress Web server. First off, your local machine does a DNS (Domain Name Service) lookup of www.syngress.com. This will return an IP address of the Syngress Web server. With this IP address, your browser and computer know how to "contact" the Syngress Web server. The browser then makes a connection on TCP port 80 to the Syngress Web server. Once the connection is made, your Web browser will send a request to "GET" the "/Help/Press/press.cfm" file. The Syngress Web server will respond with a "200 OK" and dump the HTML (Hyper-Text Markup Language) to your Web browser and it'll be displayed. However, let's assume for a moment that the "press.cfm" doesn't exist. In that case, the Syngress Web server will send to your browser a "404 Not Found" Message. Or, let's assume that Syngress decided to move the "press.cfm" to another location. In that case, your Web browser might receive a "301 Moved Permanently" message from Syngress's Web server, and then redirect you to the new location of that file.

The 200, 404, and 301 are known as "status codes" in the HTTP world. Granted, the HTTP example is a very basic breakdown, but this is exactly how SIP works. When you call someone via SIP, the commands sent are known as "SIP Methods." These SIP methods are similar to your browser sending the *GET* command to a remote Web server. Typically, these SIP methods are sent on TCP port 5060. See Table 5.1.

Table 5.1 SIP Methods

INVITE	Invite a person to a call.
ACK	Acknowledgment. These are used in conjunction with INVITE messages.
BYE	Terminates a request
CANCEL	Requests information about the remote server. For example, "what codecs do you support?"
OPTIONS	This "registers" you to the remote server. This is typically used if your connection is DHCP or dynamic. It's a method for the remote system to "follow you" as your IP address changes or you move from location to location.

Continued

Table 5.1 continued SIP Methods

REGISTER	This "registers" you to the remote server. This is typically used if your connection is DHCP or dynamic. It's a method for the remote system to "follow you" as your IP address changes or you move from location to location.
INFO	This gives information about the current call. For example, when "out-of-band" DTMF is used, the INFO method is used to transmit what keys where pressed. It can also be used to transmit other information (Images, for example).

As stated before, response codes are similar and extend the form of HTTP/1.1 response codes used by Web servers. A basic rundown of response codes is shown in Table 5.2.

Table 5.2 Response Codes

Code	Definition
100	Trying
180	Ringing
181	Call is being forwarded
182	Queued
183	Session in progress
200	OK
202	Accepted: Used for referrals
300	Multiple choices
301	Moved permanently
302	Moved temporarily
305	Use proxy
380	Alternate service
400	Bad request
401	Unauthorized: Used only by registrars. Proxies should use Proxy authorization 407.
402	Payment required (reserved for future use)
403	Forbidden
404	Not found (User not found)
405	Method not allowed

Continued

Table 5.2 continued Response Codes

Code	Definition
406	Not acceptable
407	Proxy authentication required
408	Request timeout (could not find the user in time)
410	Gone (the user existed once, but is not available here any more)
413	Request entity too large
414	Request-URI too long
415	Unsupported media type
416	Unsupported URI scheme
420	Bad extension (Bad SIP protocol extension used. Not understood by the server.)
421	Extension required
423	Interval too brief
480	Temporarily unavailable
481	Call/transaction does not exist
482	Loop detected
483	Too many hops
484	Address incomplete
485	Ambiguous
486	Busy here
487	Request terminated
488	Not acceptable here
491	Request pending
493	Undecipherable (could not decrypt S/MIME body part)
500	Server internal error
501	Not implemented (The SIP request method is not implemented here.)
502	Bad gateway
503	Service unavailable
504	Server timeout
505	Version not supported (The server does not support this version of the SIP protocol.)

Continued

Table 5.2 continued Response Codes

Code	Definition
513	Message too large
600	Busy everywhere
603	Decline
604	Does not exist anywhere
606	Not acceptable

Intra-Asterisk eXchange (IAX2)

Inter-Asterisk eXchange (IAX) is a peer-to-peer protocol developed by the lead Asterisk developer, Mark Spencer. Today, when people refer to IAX (pronounced *eeks*), they most likely mean IAX2, which is version 2 of the IAX protocol. The original IAX protocol has since been depreciated for IAX2. As the name implies, IAX2 is another means to transfer voice and other data from Asterisk to Asterisk. The protocol has gained some popularity, and now devices outside of Asterisk's software support the IAX2 protocol.

The idea behind IAX2 was simple: build from the ground up a protocol that is full featured and simple. Unlike SIP, IAX2 uses one UDP port for both signaling and media transfer. The default UDP port is 4569 and is used for both the destination port and the source port as well. This means signaling for call setup, tear down, and modification, along with the UDP datagrams, are all sent over the same port using a single protocol. It's sort of like two protocols combined into one! This also means that IAX2 has its own built-in means of transferring voice data, so RTP is not used.

When IAX was being designed, there where many problems with SIP in NAT (Network Address Translation) environments. With SIP, you had signaling happening on one port (typically TCP port 5060) and RTP being sent over any number of UDP ports. This confused NAT devices and firewalls, and SIP proxies had to be developed. Since all communications to and from the VoIP server or devices happen over one port, using one protocol for both signaling and voice data, IAX2 could easily work in just about any environment without confusing firewalls or NAT-enabled routers.

This alone is pretty nifty stuff, but it doesn't stop there! IAX2 also employs various ways to reduce the amount of bandwidth needed in order to operate. It uses a different approach when signaling for call setup, tear down, or modification. Unlike

SIP's easy-to-understand almost HTTP-like commands (methods) and responses, IAX2 uses a "binary" approach. Whereas SIP sends almost standard "text" type commands and response, IAX2 opted to use smaller binary "codes." This reduces the size of signaling.

To further reduce bandwidth usage, "trucking" was introduced into the protocol. When "trunking" is enabled (in the iax.conf, "trucking=yes"), multiple calls can be combined into single packets. What does this mean? Let's assume an office has four calls going on at one time. For each call, VoIP packets are sent across the network with the "header" information. Within this header is information about the source, destination, timing, and so on. With trunking, *one packet* can be used to transfer header information about all the concurrent calls. Since you don't need to send four packets with header information about the four calls, you're knocking down the transmission of header data from 4 to 1. This might not sound like much, but in VoIP networks that tend to have a large amount of concurrent calls, trunking can add up to big bandwidth savings.

IAX2 also supports built-in support for encryption. It uses an AES (Advanced Encryption Standard) 128-bit block cipher. The protocol is built upon a "shared secret" type of setup. That is, before any calls can be encrypted, the "shared secret" must be stored on each Asterisk server. IAX2's AES 128-bit encryption works on a call-by-call basis and only the data portion of the message is encrypted.

Getting in the Thick of IAX2

As mentioned before, IAX2 doesn't use RTP packets like SIP. Both the signaling and audio or video data is transferred via UDP packets on the default port 4569. In the iax.conf file, the port can be altered by changing the "bindport=4569" option; however, you'll probably never need to change this. In order to accomplish both signaling and stuffing packets with the audio data of a call, IAX2 uses two different "frame" types. Both frame types are UDP, but used for different purposes.

"Full Frames" are used for "reliable" information transfer. This means that when a full frame is sent, it expects an ACK (acknowledgment) back from the target. This is useful for things like call setup, tear down, and registration. For example, when a call is made with IAX2, a full frame requesting a "NEW" call is sent to the remote Asterisk server. The remote Asterisk server then sends an ACK, which tells the sending system the command was received.

With Wireshark, full frame/call setup looks like the following:

```
4.270389 10.220.0.50 -> 10.220.0.1 IAX2 IAX, source call# 2, timestamp 17ms NEW
4.320787 10.220.0.1 ->  10.220.0.50 IAX2 IAX, source call# 1, timestamp 17ms ACK
4.321155 10.220.0.1 ->  10.220.0.50 IAX2 IAX, source call# 1, timestamp 4ms ACCEPT
4.321864 10.220.0.50 -> 10.220.0.1 IAX2 IAX, source call# 2, timestamp 4ms ACK
```

Full frames are also used for sending other information such as caller ID, billing information, codec preferences, and other data. Basically, anything that requires an ACK after a command is sent will use full frames. The other frame type is known as a "Mini Frame." Unlike the Full Frame, the Mini Frame requires no acknowledgment. This is an unreliable means of data transport, and like RTP, either it gets there or it doesn't. Mini Frames are not used for control or signaling data, but are actually the UDP datagram that contains the audio packets of the call. Overall, it works similar to RTP, in that it is a low overhead UDP packet. A Mini Frame only contains an F bit to specify whether it's a Full or Mini Frame (F bit set to 0 == Mini Frame), the source call number, time stamp, and the actual data. The time stamps are used to reorder the packets in the correct order since they might be received out of order.

Capturing the VoIP Data

Now that you understand what's going on "behind the scenes" with VoIP, this information can be used to assist with debugging and capturing information.

Using Wireshark

Wireshark is a "free" piece of software that is used to help debug network issues. It's sometimes referred to as a "packet sniffer," but actually does much more than simple packet sniffing. It can be used to debug network issues, analyze network traffic, and assist with protocol development. It's a powerful piece of software that can be used in many different ways. Wireshark is released under the GNU General Public License.

In some ways, Wireshark is similar to the tcpdump program shipped with many different Unix-type operating systems. tcpdump is also used for protocol analysis, debugging, and sniffing the network. However, tcpdump gives only a text front-end display to your network traffic. Wireshark comes with not only the text front-end, but a GUI as well. The GUI layout can assist in sorting through different types of packet data and refining the way you look at that data going through your network.

While tcpdump is a powerful utility, Wireshark is a bit more refined on picking up "types" of traffic. For example, if a SIP-based VoIP call is made and analyzed with

tcpdump, it'll simply show up as UDP traffic. Wireshark can see the same traffic and "understand" that it's SIP RTP/UDP traffic. This makes it a bit more powerful in seeing what the traffic is being used for.

Both tcpdump and Wireshark use the "pcap" library for capturing data. pcap is a standardized way to "capture" data off a network so it can be used between multiple applications. PCAP (libpcap) is a system library that allows developers not to worry about how to get the network packet information off the "wire," and allows them to make simple function calls to grab it.

We'll be using pcap files. These are essentially snapshots of the network traffic. They include all the data we'll need to reassemble what was going on in the network at the time. The nice thing with pcap dump files is that you can take a snapshot of what the network was doing at the time, and transfer it back to your local machine for later analysis. This is what we'll be focusing on. The reason is, while Wireshark might have a nice GUI for capturing traffic, this doesn't help you use it with remote systems.

Unfortunately, not all pcap files are the same. While Wireshark can read tcpdump-based pcap network files, characteristics of that traffic might be lost. For example, if you create a pcap file of SIP RTP traffic with tcpdump and then transfer that "dump" back to your computer for further analysis, tcpdump will have saved that traffic as standard UDP traffic. If created with Wireshark, the pcap files a "note" that the traffic is indeed UDP traffic, but that it's being used for VoIP (SIP/RTP).

As of this writing, Wireshark can only understand SIP-based traffic using the G.711 codec (both ulaw and alaw). The audio traffic of a VoIP call can be captured in two different ways. In order to capture it, you must be in the middle of the VoIP traffic, unless using arp poisoning. You can only capture data on your LAN (or WAN) if you are somehow in line with the flow of the VoIP traffic. For these examples, we'll be using the command-line interface of Wireshark to capture the traffic. The reason for this is that in some situations you might not have access to a GUI on a remote system. In cases like this, the text-only interface of Wireshark is ideal. You'll be able to fire up Wireshark (via the command *tethereal* or *twireshark*) and store all the data into a pcap file which you can then download to your local system for analysis.

To start off, let's create an example pcap file. In order to capture the traffic, log in to the system you wish to use that's in line with the VoIP connection. You'll need "root" access to the system, because we'll be "sniffing" the wire. We'll need more than normal user access to the machine to put the network interface in promiscuous mode. Only "root" has that ability.

Once the network device is in promiscuous mode, we can capture all the network traffic we want. Running Wireshark as "root" will automatically do this for us. To begin capturing, type

```
# tethereal -i {interface} -w {output file}
```

So, for example, you might type

```
# tethereal -i eth0 -w cisco-voip-traffic.pcap
```

Unfortunately, this won't only capture the VoIP traffic but everything else that might pass through the *eth0* interface. This could include ARP requests, HTTP, FTP, and whatever else might be on the network. Fortunately, the tethereal program with Wireshark works on the same concept as tcpdump. You can set up "filters" to grab only the traffic you want. So, if we know our VoIP phone has an IP address of 192.168.0.5, we can limit what we grab by doing the following:

```
# tetheral -i eth0 -w cisco-voip-traffic.cap host 192.168.0.5
```

Once fired up, you should then see *Capturing on eth0*. As packets are received, a counter is displayed with the number of packets recorded. You can further reduce the traffic by using tcpdump type filters. Depending on the amount of calls, we might need to let this run for a while.

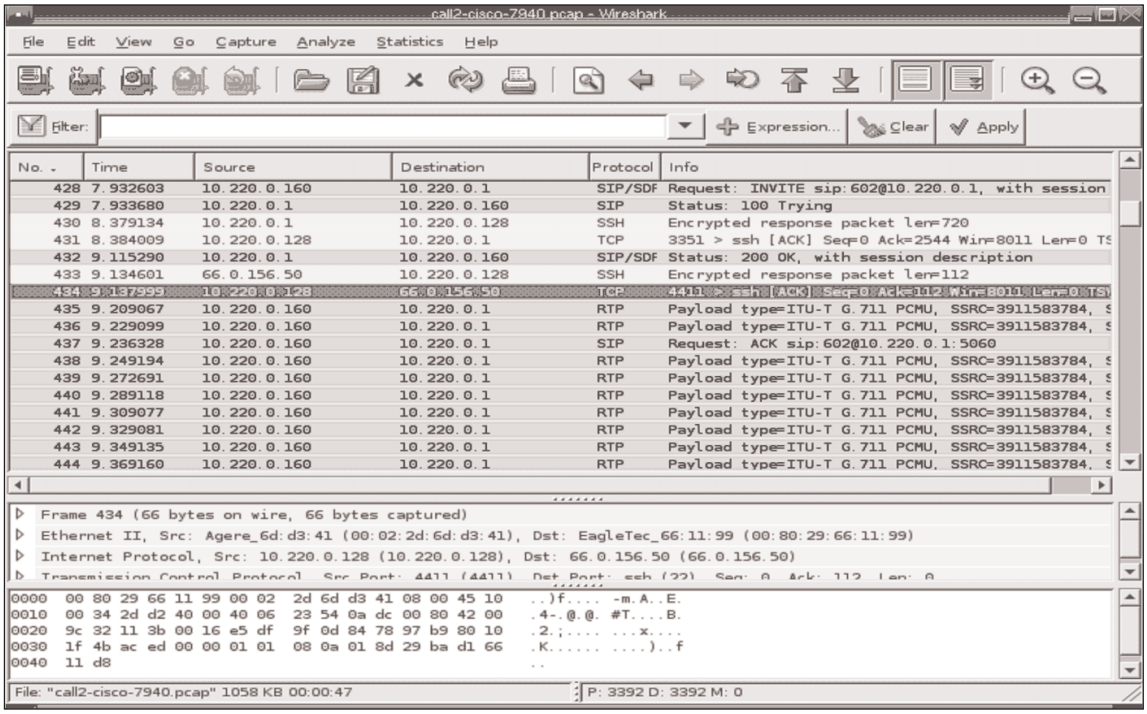
Extracting the VoIP Data with Wireshark (Method # 1)

Once you've captured the data, you'll need to get it to a workstation so you can do further analysis on it. This may require you transferring it from the target system where you created the pcap file to your local workstation. Once you have the data in hand, start Wireshark and load the pcap. To do this, type

```
$ wireshark {pcap file name}
```

You'll no longer need to be "root" since you won't be messing with any network interfaces and will simply be reading from a file. Once started and past the Wireshark splash screen, you'll be greeted with a screen similar to Figure 5.1

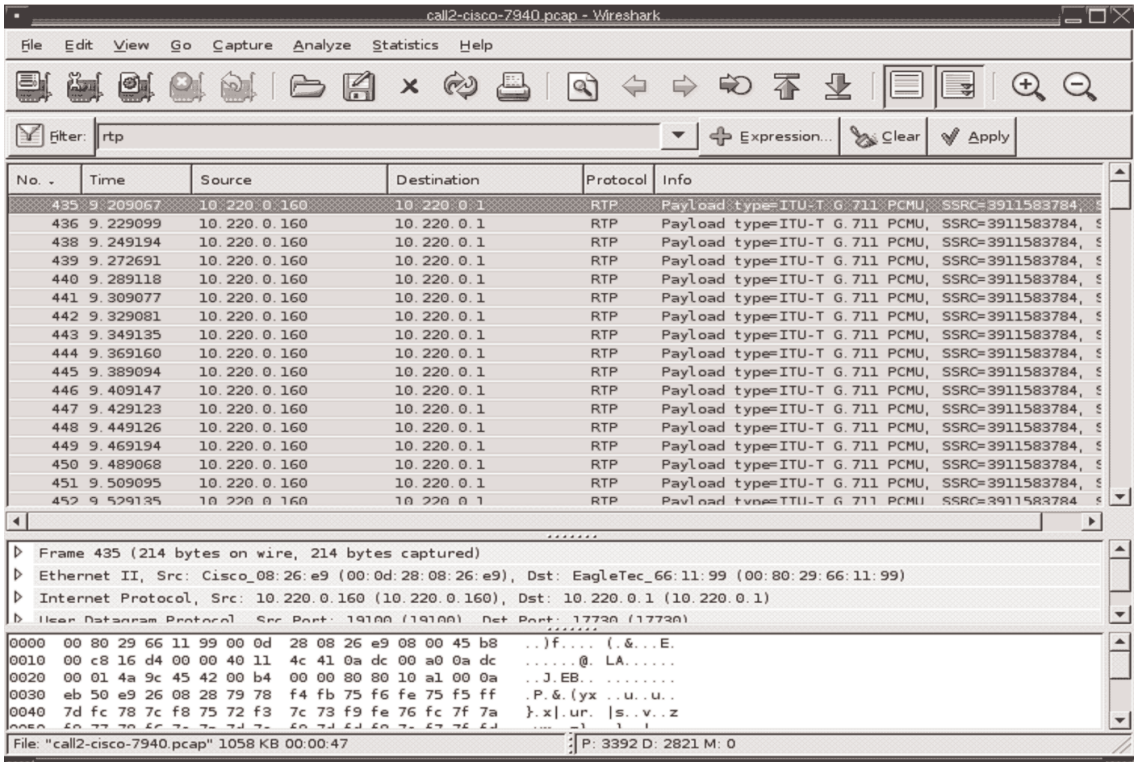
Figure 5.1 pcap Wireshark



If you look closely at the example screen, you'll notice things like "SSH" traffic. We now need to filter out all the unwanted traffic, since we're only interested in UDP/RTP/VoIP traffic. So, the first thing we need to do is "Filter" the traffic. Note the "Filter" option at the top left-hand corner. This allows you to enter the criteria used to filter the packet dump. For example, you could enter "tcp" in this field and it'll only show you the TCP packets. In this case, we'll filter by **RTP**, as shown in Figure 5.2.

After entering **RTP** and clicking the **Apply** button, Wireshark removes all other TCP/IP packet types and only leaves you with RTP (UDP) type packets. In this case, the Source of 10.220.0.160 is my Cisco 7940 IP phone using the SIP. The Destination is my in in-house Asterisk server. Also notice the Info field. This tells us the payload type of the RTP packet. In this case, it's G.711.

Figure 5.2 Filter by RTP

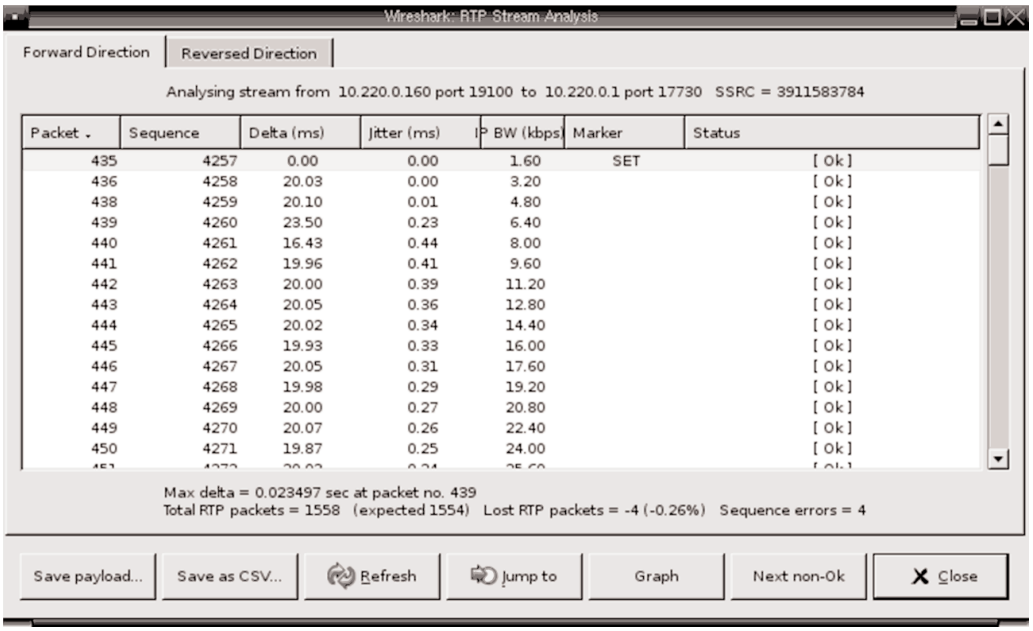


Now that we're only looking at RTP-type packets, this might be a good time to browse what's left in our filtered packet dump. Wireshark will also record what phone pad buttons (DTMF) were pressed during the VoIP session. This can lead to information like discovering what the voicemail passwords and other pass codes are that the target might be using.

To get the audio of the VoIP conversation, we can now use Wireshark's "RTP Stream Analysis." To do this, select **Statistics | RTP | Stream Analysis**.

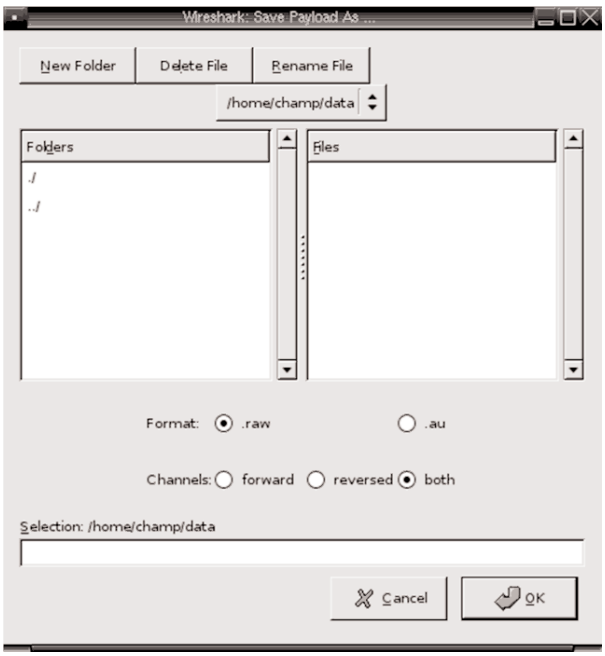
Afterward, you should see a screen similar to Figure 5.3.

Figure 5.3 Wireshark RTP Stream Analysis



From here, it's as simple as selecting **Save Payload**. You should then be greeted with a menu that looks similar to Figure 5.4.

Figure 5.4 Wireshark Save Payload



Before saving, look at the "Format" radio box. I typically change this from ."raw" (the default) to ."au." The reason is because it's an older audio format for Unix that was produced by Sun Microsystems. Conversion from the .au format to other formats is trivial and well supported. Under the "Channels" field, you'll probably want to leave this set to "both." With "both" enabled, you'll save the call as it was recorded with both sides of the conversation. The "forward" and "reversed" allows you to save particular channels of the conversation. This might be useful in certain situations, but most of the time you'll probably want the full conversation recorded to the .au file as it happened.

Once your .au file is recorded, conversion to other formats is trivial; using sound utilities like "sox" (http://sox.sourceforge.net/) is trivial. At the Unix command line with "sox" installed, you'd type: **sox {input}.au {output}.wav**.

Extracting the VoIP Data with Wireshark (Method # 2)

As of Wireshark version 0.99.5, VoIP support has improved a bit and will probably get even better. Wireshark versions before 0.99.5 do not contain this method of extracting and playing the VoIP packet dump contents.

To get started, we once again load Wireshark with our pcap file:

```
$ wireshark {pcap file name}
```

After the Wireshark splash screen, you'll be greeted with a screen similar to that from Figure 5.1. This time, the menu options we'll select are **Statistics | VoIP Call**.

Unlike before, we won't need to filter by "RTP" packet. Wireshark will go through the packet dump and pull out the VoIP-related packets we need. You should see a screen similar to Figure 5.5.

In this example, the packet dump contains only one VoIP call. Like the previous example, this packet dump is from my Cisco 7940 VoIP phone using SIP (10.220.0.160) to my Asterisk server (10.220.0.1). If multiple calls were present, this screen would show each call. Since there is only one call, we'll select that one. Once chosen, the Player button should become available. Upon selecting the Player button, you should see something similar to Figure 5.6.

Figure 5.5 VoIP Calls Packet

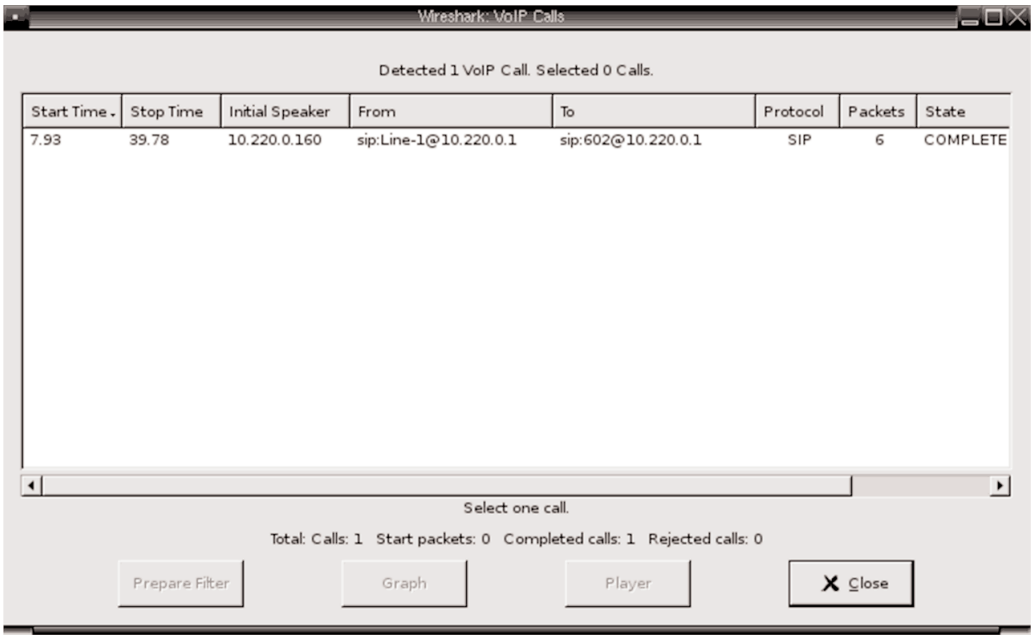
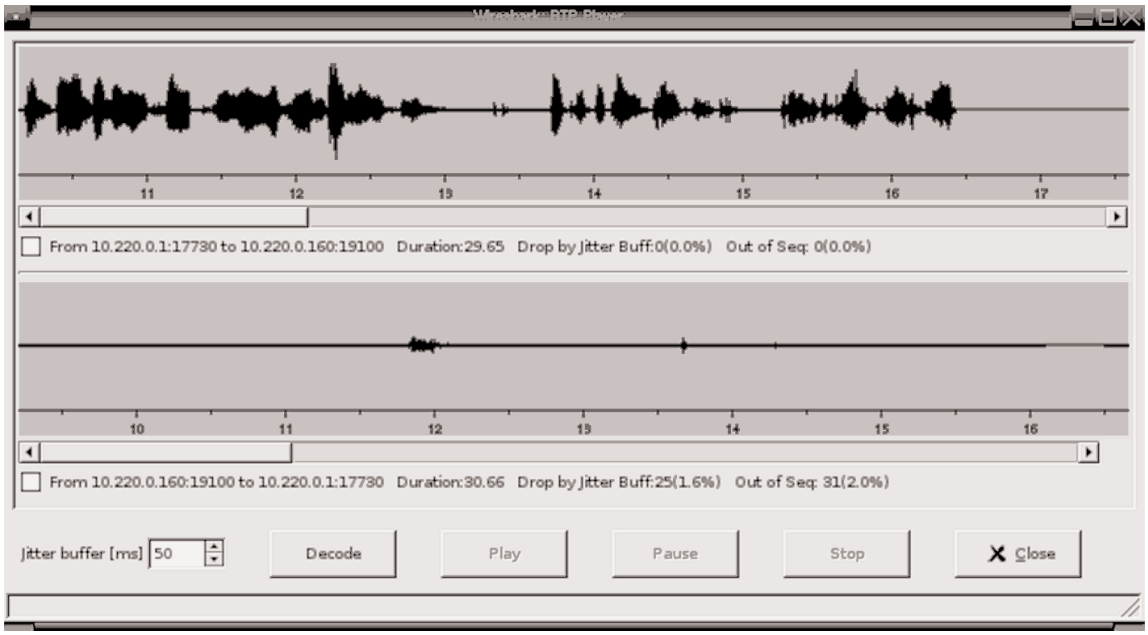


Figure 5.6 Wireshark RTP Player

Select the Decode button. The Wireshark RTP player should then appear, looking something like the one in Figure 5.7.

Figure 5.7 Decoded Wireshark RTP Player

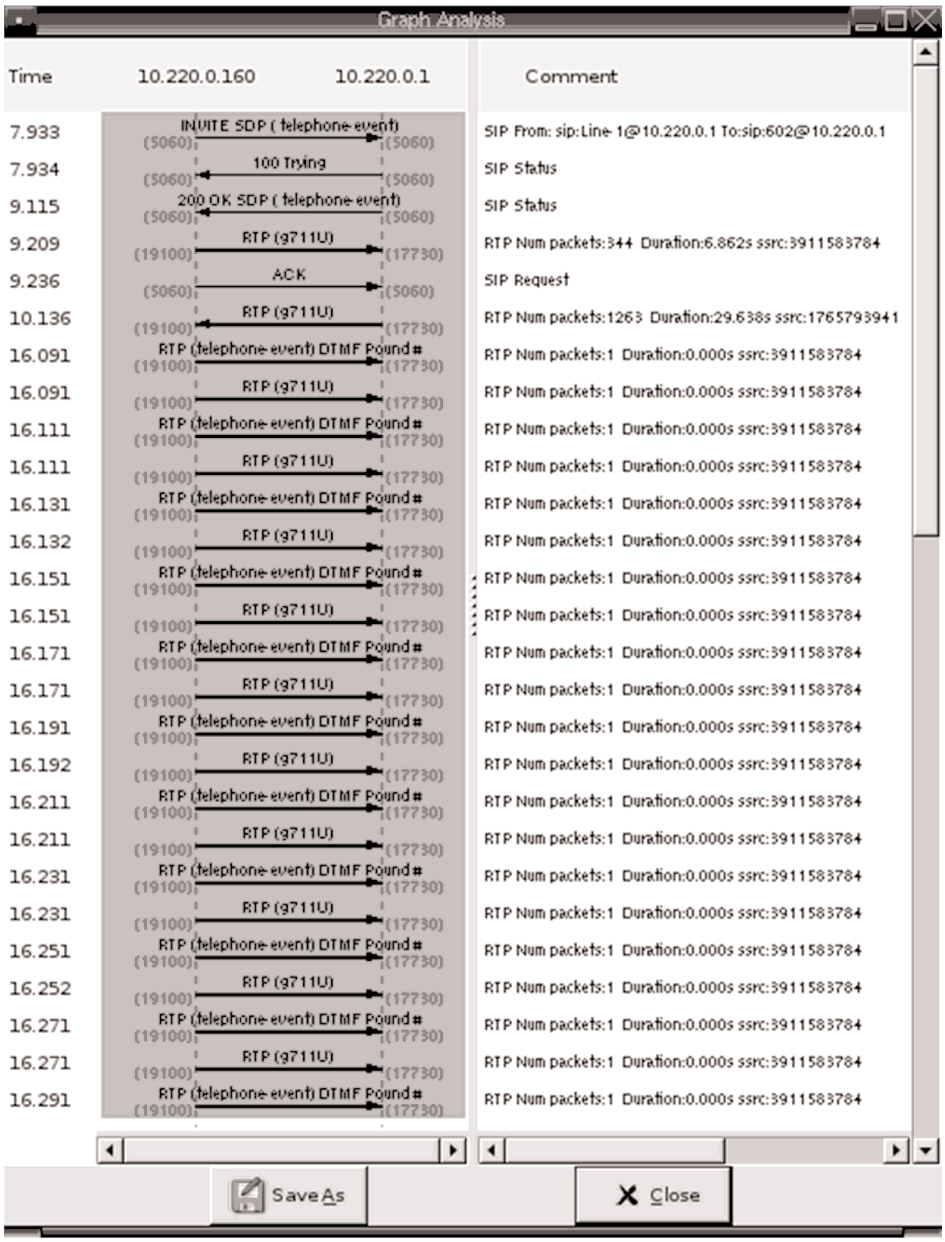


From here, select the stream to listen to and then click Play. It's as easy as that. The only disadvantage at this time is that you can't save the audio out to a file. That'll probably change as Wireshark supports more VoIP options.

This method also has a nice "Graph" feature, which breaks down the call into a nice, simple format. To use this, we perform the same steps to get to the Player button, but rather than selecting Player, we click Graph. Clicking the Graph button should generate a screen similar to that in Figure 5.8.

This breaks down the VoIP communication data. Note that timestamp 16.111 shows that the DTMF of "#" was sent. This type of information can be useful in determining what DTMF events happened. This can lead to revealing pass codes, voice-mail passwords, and other information.

Figure 5.8 VoIP Graph Analysis

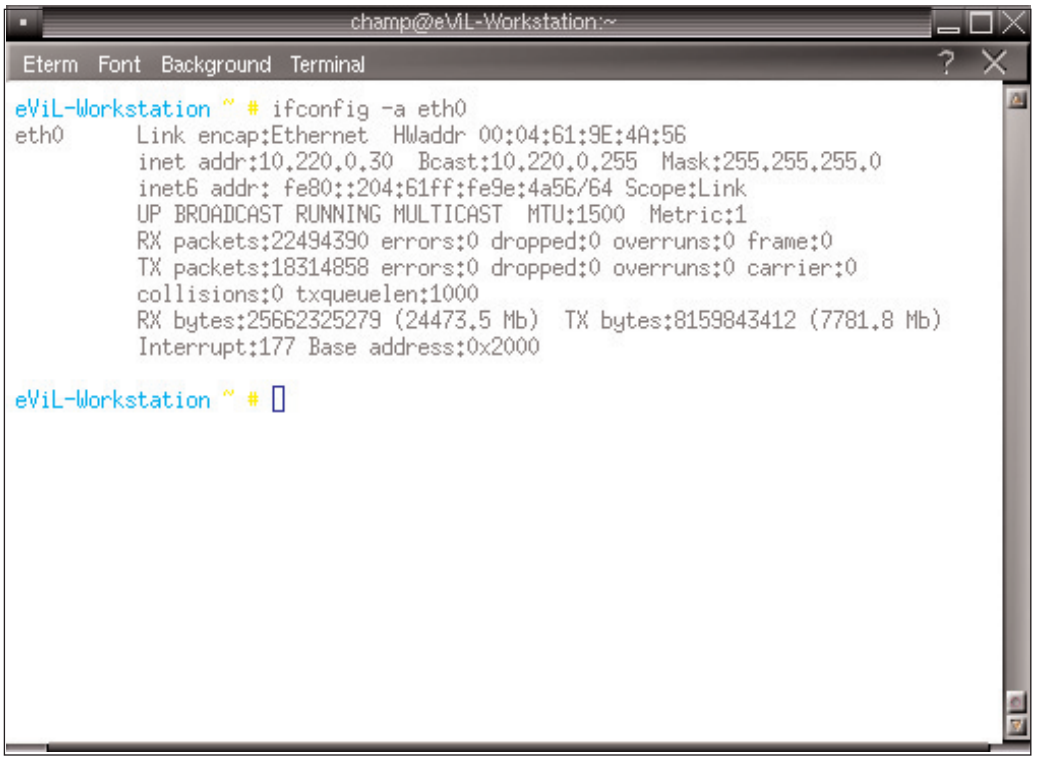


Getting VoIP Data by ARP Poisoning

ARP (Address Resolution Protocol) is used to located equipment within a LAN by the hardware MAC (Media Access Control). A MAC address is a preassigned to the

network hardware. It uses a 48-bit address space, so there is plenty of room to grow. The 48-bit address space is expressed as 12 hexadecimal digits. The first six are assigned to the manufacture of the network device. For example, on my home Linux workstation, the Ethernet card MAC address is 00:04:61:9E:4A:56. Obtaining your MAC address depends on what operating system you're running. Under Linux, an *ifconfig -a* will display the various information, including your MAC address. On BSD-flavored systems, a *netstat -in* will usually do it. The output from my workstation is shown in Figure 5.9.

Figure 5.9 The MAC Address of the Author's Workstation

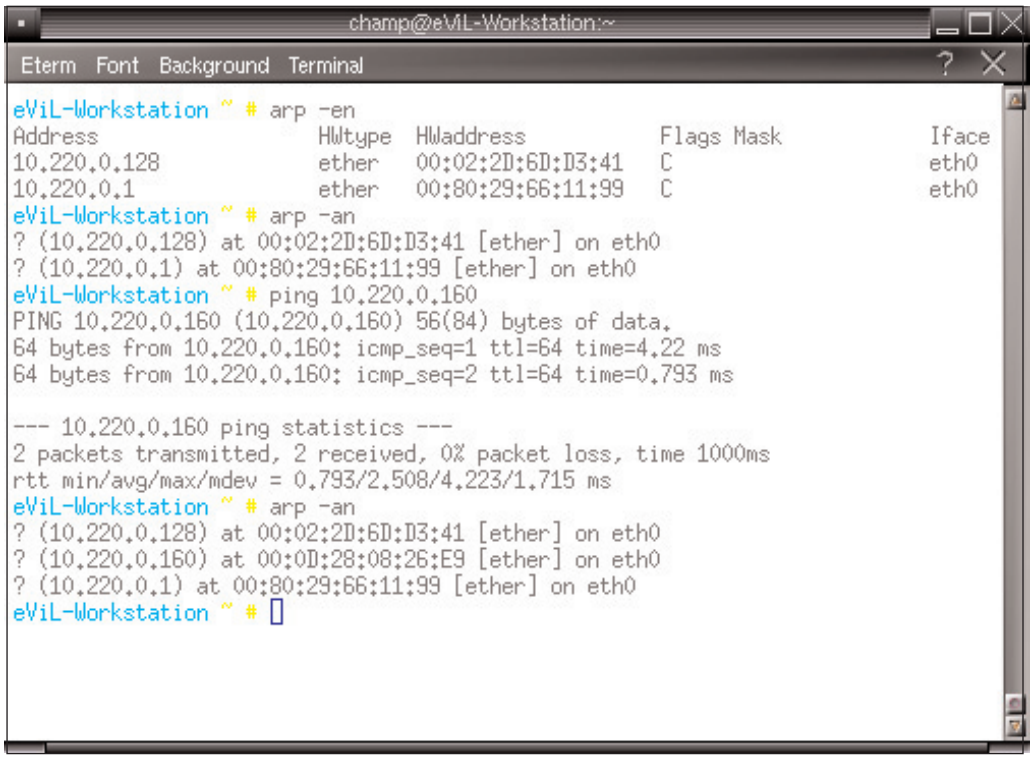


As you can see, the *HWaddr* field contains my MAC address. As stated earlier, the first six digits reveal the vendor of the hardware. So how do you determine the vendor, you ask? Well, it just so happens that the IEEE (Institute of Electrical and Electronics Engineers) maintains a list of vendors that is freely available at http://standards.ieee.org/regauth/oui/oui.txt. It is a flat ASCII text file of all vendors and their related MAC prefixes. So, looking up my MAC address in that list, we see that the 00:04:61 prefix belongs to:

```
0-04-61    (hex)                          EPOX Computer Co., Ltd.
000461      (base 16)            EPOX Computer Co., Ltd.
                                 11F, #346, Chung San Rd.
                                 Sec. 2, Chung Ho City, Taipei Hsien 235
                                 TAIWAN TAIWAN R.O.C.
                                 TAIWAN, REPUBLIC OF CHINA
```

This is the company that made my network card. While this is all interesting, you might wonder how it ties in to ARP address poisoning. Well, MAC addresses are unique among all networking hardware. With TCP/IP, the MAC address is directly associated with a TCP/IP network address. Without the association, TCP/IP packets have no way of determining how to get data from one network device to another. All computers on the network keep a listing of which MAC addresses are associated with which TCP/IP addresses. This is known as the systems ARP cache (or ARP table). To display your ARP cache, use *arp -an* in Linux, or *arp -en* in BSD-type systems. Both typically work under Linux, as shown in Figure 5.10.

Figure 5.10 Display of ARP Cache in Linux



Notice that when I entered *arp -en* and *arp -an*, there were only two entries. Did you see what happened when I sent a *ping* request to my Cisco phone (10.220.0.160) and re-ran the *arp -an* command? It added the Cisco IP phone's MAC address into the ARP cache. To obtain this, my local workstation sent out what's known as an "ARP request." The ARP request is a network broadcast, meaning the request was sent networkwide. This is done because we don't know "where" 10.220.0.160 is. When an ARP request is sent, a packet is sent out saying "Who has 10.220.0.160?" networkwide. When 10.220.0.160 receives the ARP request, it replies "That's me. My MAC address is 00:0D:28:08:26:E9."

The following is a Wireshark dump of an ARP request and reply:

```
04:04:12.380388 arp who-has 10.220.0.160 tell 10.220.0.30
04:04:12.382889 arp reply 10.220.0.160 is-at 00:0d:28:08:26:e9
```

As you can see, this is literally what is happening! Now that both sides have their TCP/IP network addresses associated with the MAC, they can start transferring data. If 10.220.0.30 (my workstation) needs to talk to 10.220.0.160 (my Cisco IP phone), my workstation knows to send the data to the 00:0D:28:08:26:E9 MAC address, which is 10.220.0.160.

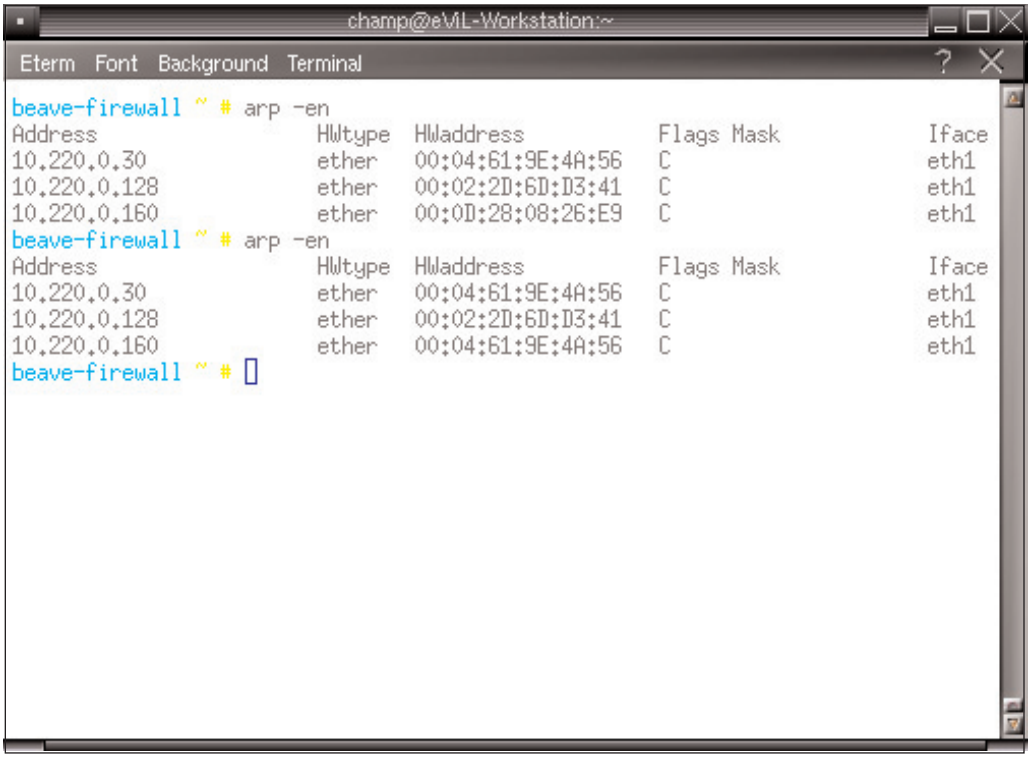
The underlying flaw with ARP is that in many cases it's very "trusting" and was never built with security in mind. The basic principle of ARP poisoning is to send an ARP reply to a target that never requested it. In most situations, the target will blindly update its ARP cache. Using my Cisco IP phone and Linux workstation as an example, I can send a spoofed ARP reply to a target with the Cisco IP phone's TCP/IP network address, but *with my workstation's MAC address.*

For this simple example, I'll use the arping2 utility (www.habets.pp.se/synscan/programs.php?prog=arping). This utility works much like the normal *ping* command but sends ARP requests and ARP replies. My target for this simple example will be my default route, which happens to be another Linux machine (10.220.0.1). The command I'll issue from my workstation (10.220.0.30) is

```
# arping2 -S 10.220.0.160 10.220.0.1
```

This *-S* option tells arping2 to spoof the address. So my Linux workstation will send an ARP request to 10.220.0.1 informing it that 10.220.0.160 is my workstation's MAC address. Figure 5.11 shows a screenshot from my Linux gateway.

Figure 5.11 Display from Author's Linux Gateway



If you look closely at the first time I issue the *arp -en* command, the MAC address is that of the Cisco IP phone (00:0D:28:08:26:E9). This is before the arping2 spoof command was issued. The second time *arp -en* is run is after I've spoofed with arping2. You might have noticed that the *Hwaddress* has changed to my Linux workstation (00:04:61:9E:4A:56). Until the ARP tables get updated, whenever my Linux gateway attempts to communicate with the Cisco phone, it'll actually be sending packets to my workstation.

This basic example is not very useful other than in causing a very basic temporary DoS (Denial of Service). While I'll be receiving packets on behalf of the Cisco IP phone, I won't be able to respond. This is where the Man-in-the-Middle attack comes in.

Man in the Middle

A Man-in-the-Middle (MITM) attack is exactly what it sounds like. The idea is to implement some attack by putting your computer directly in the flow of traffic. This can be done in several ways, but we'll keep focused on ARP poisoning. To accom-

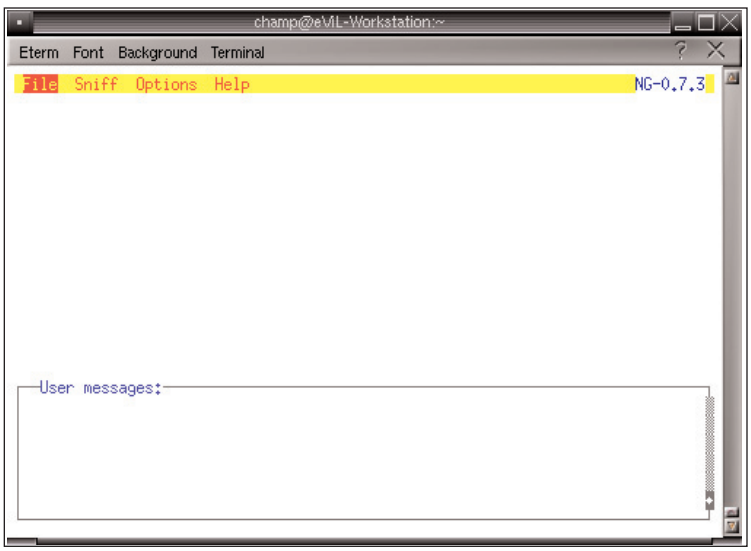
plish a MITM and capture all the VoIP traffic, we'll be ARP poisoning two hosts. The Cisco IP phone (10.220.0.160) and my gateway's ARP cache (10.220.0.1). I'll be doing the actual poisoning from my workstation (10.220.0.30), which is connected via a network switch and is not "in line" with the flow of VoIP traffic. Considering I have a network switch, I normally shouldn't see the actual flow of traffic between my Cisco phone and my gateway. Basically, my workstation should be "out of the loop." With a couple of nifty tools, we can change that.

Using Ettercap to ARP Poison

Ettercap is available at http://ettercap.sourceforge.net/. It primarily functions as a network sniffer (eavesdropper) and a MITM front end. It's a fairly simple utility that helps assist in grabbing traffic you shouldn't be seeing. Ettercap comes with a nice GTK (X Windows) interface, but we won't be focusing on that. We'll be looking more at the command line and ncurses interfaces. One nice thing about ettercap is that the curses interface is similar to the GUI, so moving from curses to GUI shouldn't be a hard transition.

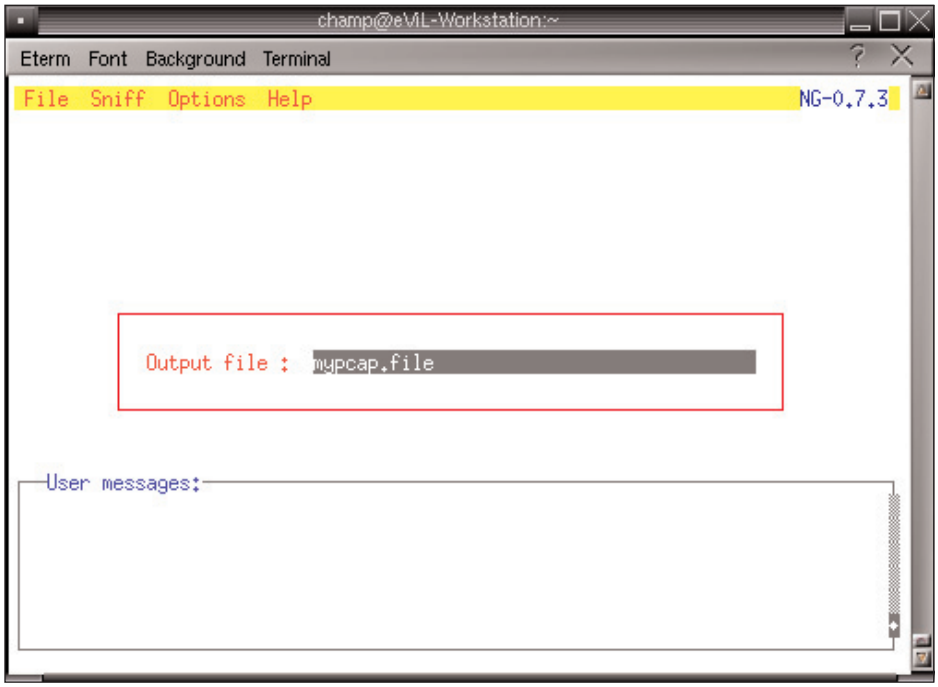
I also don't want to focus on the GUI because many times your target might not be within your LAN. It's much easier to use the command line or curses menu when the network you're testing is remote. To kick things off, we'll look at the ncurses front end. In order to use Ettercap for sniffing and ARP poisoning purposes, you'll need to have "root" access. To start it up, type ***ettercap –curses***, and you should see something like Figure 5.12.

Figure 5.12 Ettercap Sniffing Startup



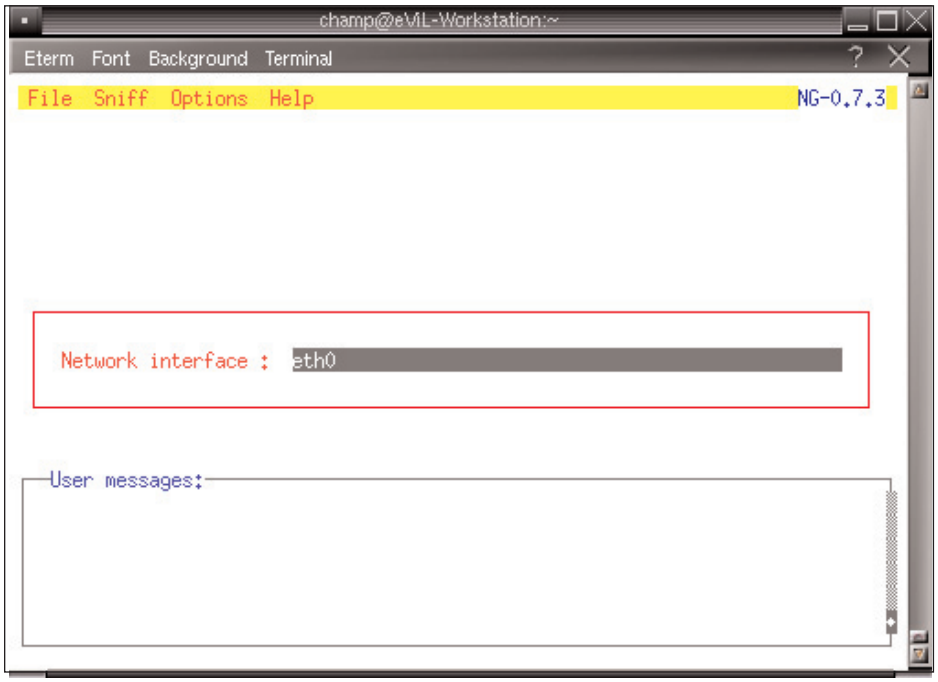
You'll want to store the data you've captured while sniffing, so you'll need to build a PCAP file you can later analyze. To do this, press **Shift + F**. Notice that the curses menu options are almost always the **Shift** key and the first letter of the menu option. To get more information about Ettercap's menu function, see the Help (**Shift + H**) options shown in Figure 5.13.

Figure 5.13 Help Option for Ettercap's Menu Function



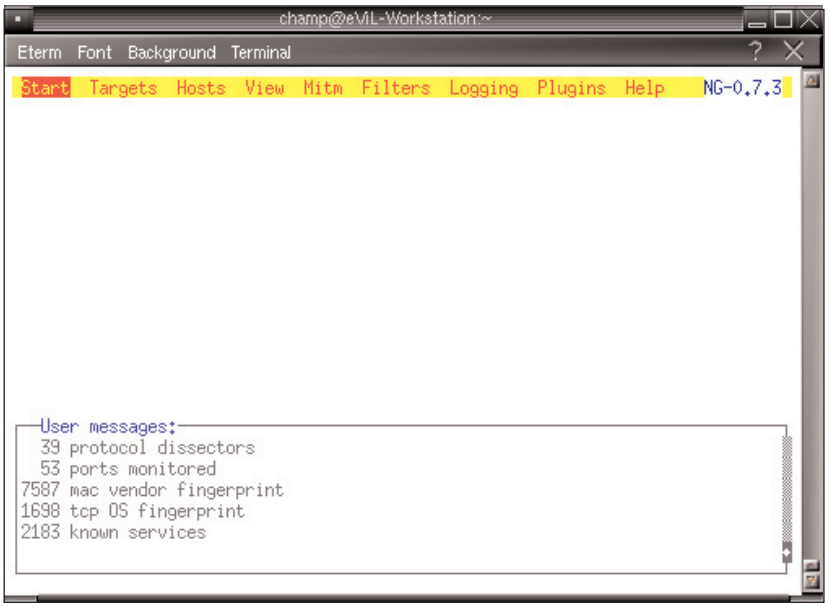
Type in the filename you wish to store the PCAP file as and press **Enter**. You'll now want to start sniffing the network. To do this, press **Shift + S** for the Sniff menu option, shown in Figure 5.14.

Figure 5.14 The Sniff Menu Option



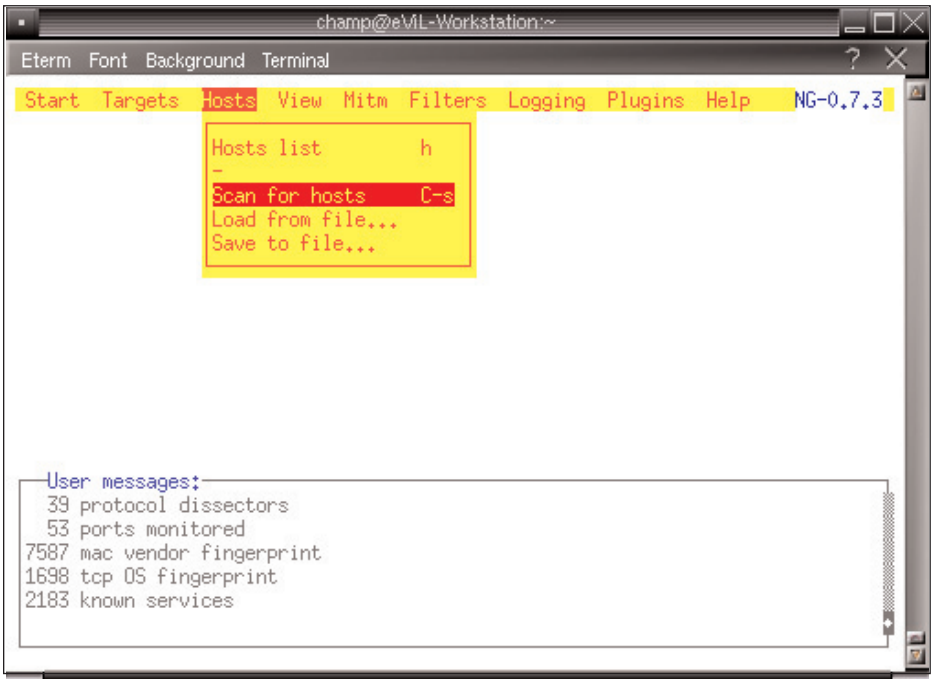
It'll now ask you which Ethernet device to use. Enter the device and press **Enter**. The screen should change and look something like Figure 5.15.

Figure 5.15 Ethernet Device Selection



Press **Shift + H** to select the hosts in your network. The easiest way to populate this list is to choose **Scan for hosts**. So, select this option, as shown in Figure 5.16.

Figure 5.16 Selecting Network Hosts



The way Ettercap scans for local network hosts is that it examines your network setup. In my case, I use a 10.220.0.0 network, with a netmask of 255.255.255.0. So, Ettercap sends out ARP requests for all hosts. In my case, 10.220.0.1 to 10.220.0.255. Ettercap stores all these responses in a "host list." My host list looks like Figure 5.17.

If you press the spacebar, it'll give you a little help, as shown in Figure 5.18.

Figure 5.17 Host List Displayed

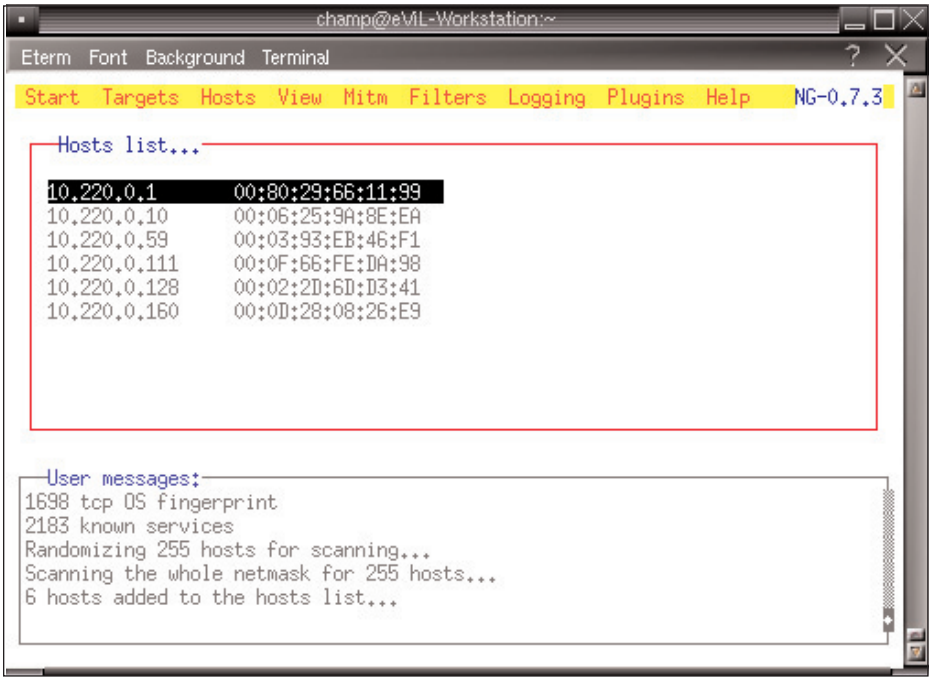
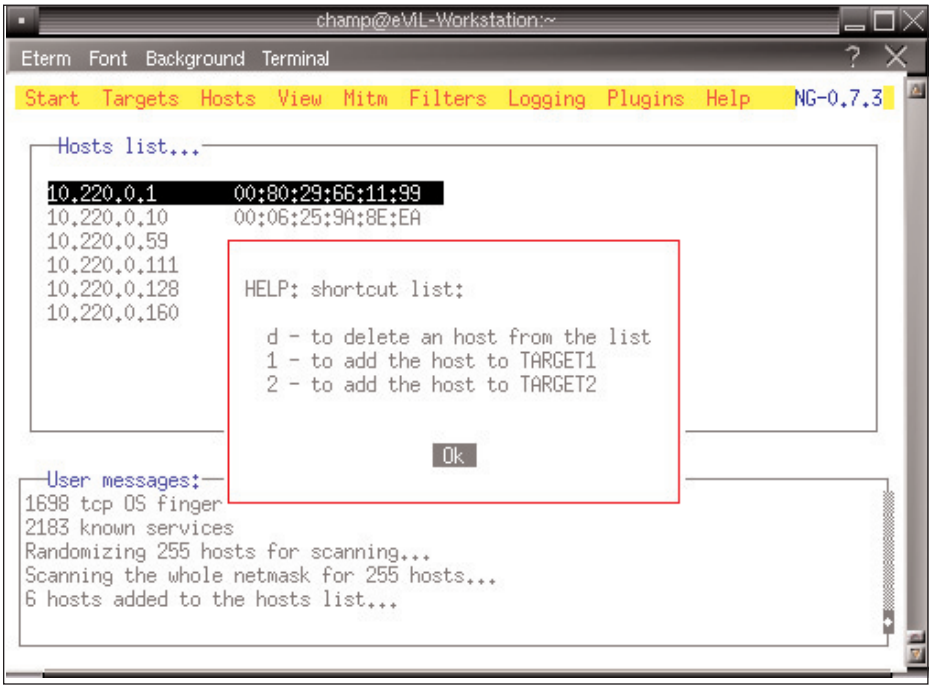
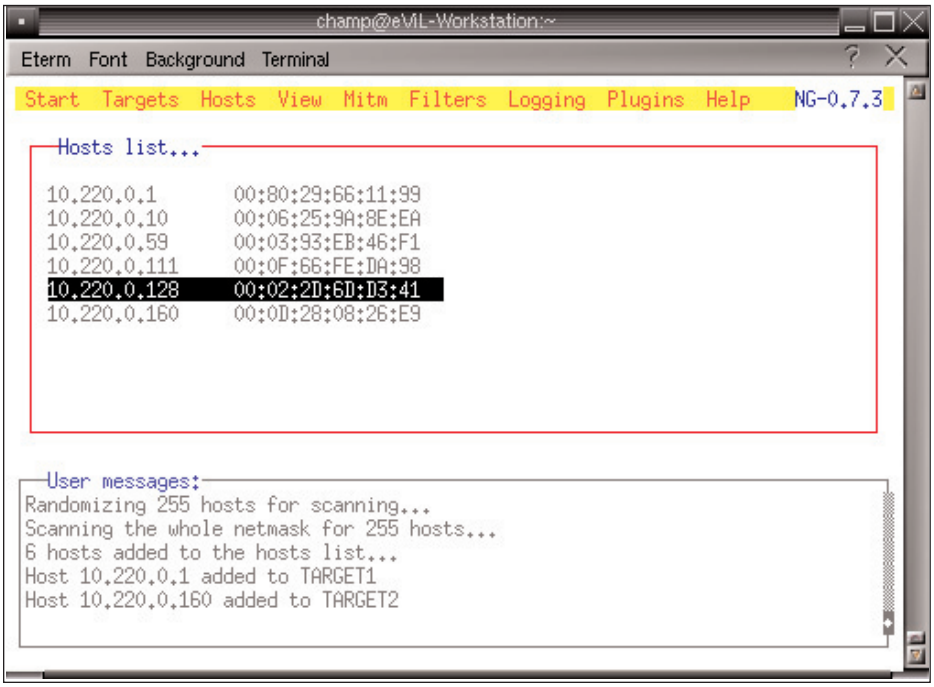


Figure 5.18 Help Shortcut List



You can press enter to exit the Help screen. Once you exit the Help screen, you can use your up and down arrow keys to "mark" your target. To mark a target, use the **1** or **2** keys. In this example, I'm going to select 10.220.0.1 (my gateway) as Target 1, by pressing the numeric 1. I'll then add 10.220.0.160 (my Cisco IP phone) to the second target list by pressing the numeric 2, as shown in Figure 5.19

Figure 5.19 Target Selection



Note that when I select a target, in the User Messages section at the bottom of the screen it confirms my targets. Now that our targets are selected, you can double-check your target setup by pressing **Shift + T**, as shown in Figures 5.20A and 5.20B.

Figure 5.20A Target Setup Check

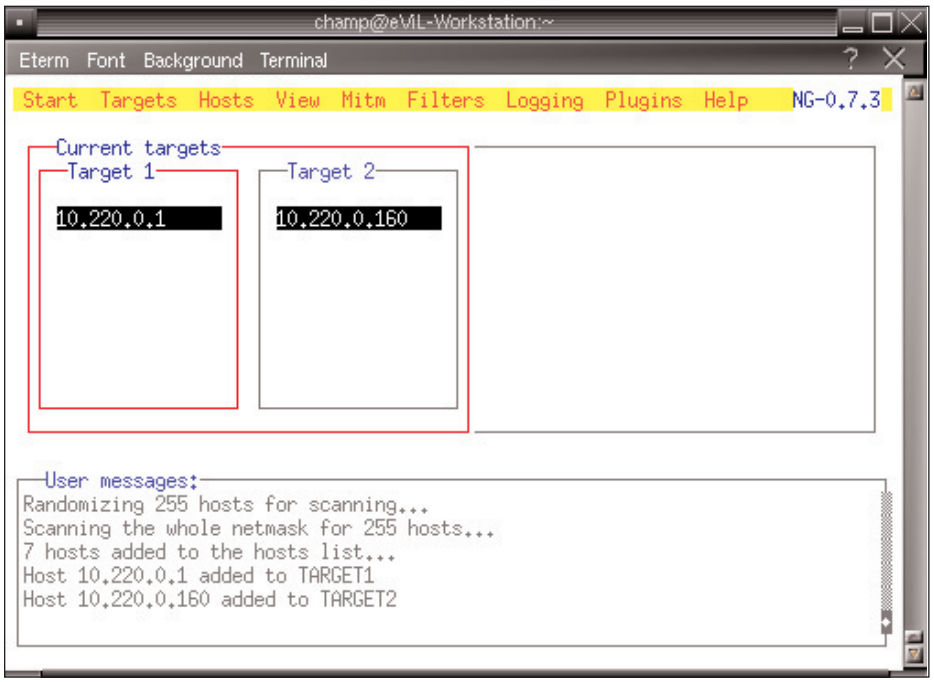
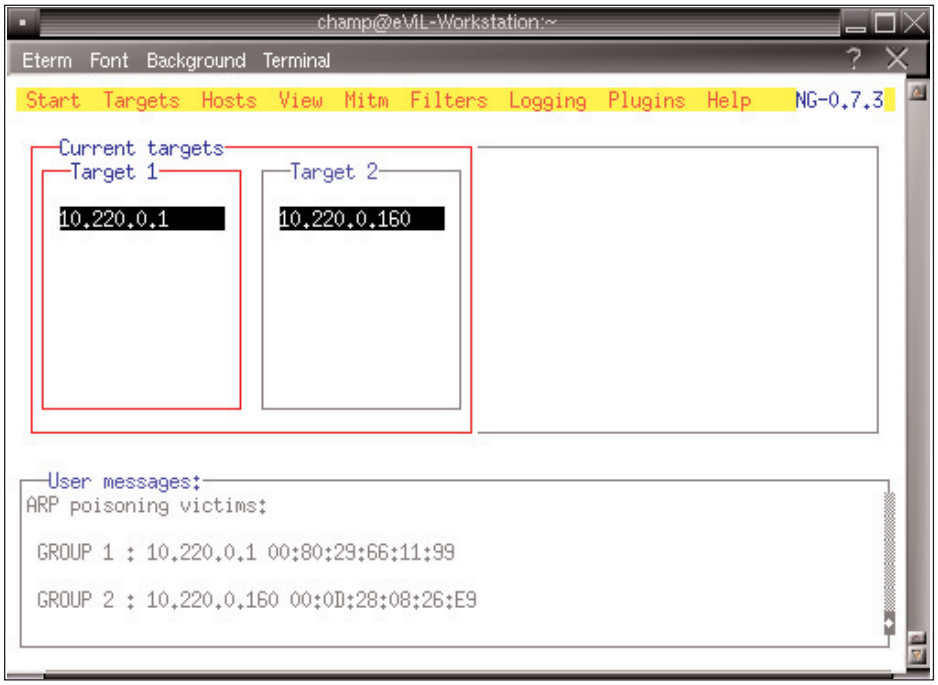
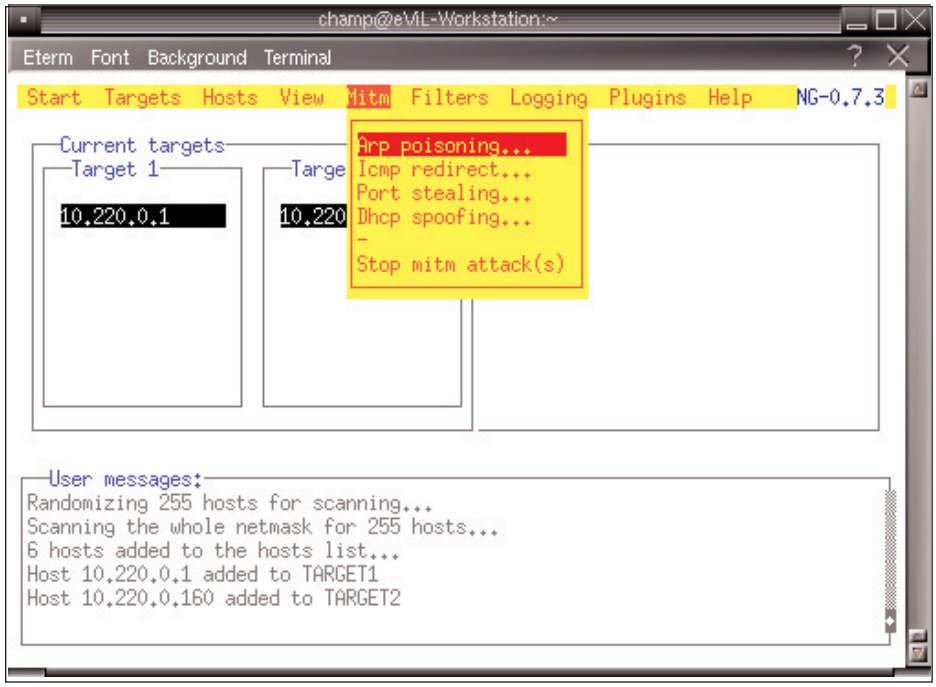


Figure 5.20B Target Setup Check



Now we're ready to set up the MITM attack. To do this, press **Shift + M** and select **ARP poisoning**, as shown in Figure 5.21.

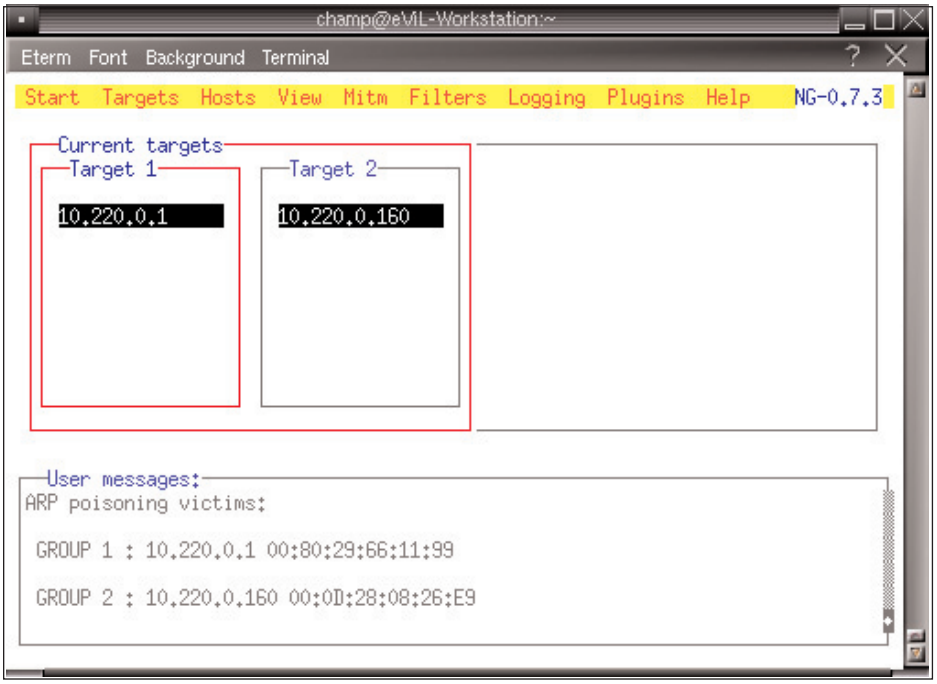
Figure 5.21 MITM Attack Setup



Once selected, it will prompt you for "Parameters." We want to do a full session sniffing MITM attack, so enter **remote** in this field. Now press **Enter**. You should see something like Figure 5.22.

Again, note the bottom of the screen. We are now ARP poisoning our targets and sniffing the traffic! Once you've let it run and feel that you've gotten the data you want, you can stop the MITM attack by pressing **Shift + M** (Stop MITM attack). This will re-ARP the targets back to what they originally were before the attack. You can then press **Shift + S** and select **Exit**. You should now have a PCAP file to analyze.

Figure 5.22 Parameters for Full Session Sniffing MITM Attack



As you can see, the ncurses Ettercap interface is quite nice and powerful, but we can accomplish the exact same thing much easier! How could we possibly make it simpler? We can do all of the preceding in one simple command line. As "root," type

```
# ettercap -w my.pcap --text --mitm arp:remote /10.220.0.1/ /10.220.0.160/
```

That's all there is to it! The —*text* tells Ettercap we want to remain in a "text" mode.

We don't want anything fancy, just your basic good ol' text. The —*mitm* should be pretty obvious by now. The *arp:remote* option tells Ettercap we want to ARP poison the remote targets and we'd like to "sniff" the traffic. Once you capture the traffic, you can load it into something like Wireshark or Vomit and extract the SIP or H.323-based traffic.

Summary

Understanding how VoIP protocols function is important. This knowledge will help you debugging problems, assist in generating attacks in a security audit and help protect you against attacks targeting you Asterisk system. Like any other network protocols, there is no "magic" involved but a set of guidelines. These guidelines are covered in various RFC's (Request for Comments) and describe, in detail, how a protocol functions. Developers follow and use these RFC's to assist in development to help build applications. There are multiple RFCs covering various VoIP protocols. These describe how signaling works, how audio and video data is transferred and various other features. Reading and understanding these RFC's can help you unlock the "magic" of how VoIP works.

As shown in the chapter, two major functions with IAX2 and SIP is signaling and passing the audio/video data. Signaling handles the call build up, tear down and modification of the call. The two protocols handle passing the audio data and signaling differently. While SIP is a signaling protocol in itself and uses RTP to pass the audio/video data, IAX2 chose to build both into one protocol.

If you understand how the protocols work, building attacks becomes easier. For example, *fuzzing* or looking for flaws at the SIP level (typically TCP port 5060). If you know the SIP methods supported on a particular piece of SIP hardware, you can *probe* the target with bogus or invalid requests and see how it responds.

In conjunction with other hacking techniques, like ARP poisoning, you can perform man in the middle attacks. These types of attacks will not only let you grab the audio of a conversation, but other data as well. For example, authentication used between devices during the call build up or the DTMF used to authenticate with other devices. For example, voice mail.

Solutions Fast Track

Understanding the Core of VoIP Protocols

- ☑ VoIP data is transferred using small UDP packets.
- ☑ UDP is not time sensitive, which is good for VoIP.
- ☑ With SIP, these UDP packets are known as RTP packets. IAX2 uses a built in method known as mini-frames.

How Compression in VoIP Works

- ☑ Compression can further reduce the bandwidth needed for VoIP by compressing the UDP/VoIP packets.
- ☑ Compression uses more CPU time and less bandwidth. No compression uses more bandwidth but less CPU time.
- ☑ Compression codecs come in open and closed standards. For example, GSM and Speex is open, while G.729 requires licensing to use in corporate environments.

Signaling Protocols

- ☑ SIP is a signaling protocol used to setup/tear down/modification calls.
- ☑ SIP uses RTP (Real Time Protocol) packets for voice data. These are small UDP packets.
- ☑ The SIP protocol is similar to HTTP. This makes debugging easier, but requires a little bit more bandwidth.
- ☑ IAX2 has signaling and audio transfer built into one protocol. Unlike SIP, IAX2 does signaling via binary commands, which uses less bandwidth. VoIP audio is sent by mini-frames (small UDP packets).

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Since SIP is similar to the HTTP protocol, could similar methods be used to attack SIP and find weaknesses.

A: Yes. Fuzzing and probing equipment at the SIP level (typically port 5060) could possibly reveal programming flaws. The basic idea would be to build bogus SIP methods and see how the hardware responds. SIP responses to bogus or invalid methods could also help reveal flaws

Q: Could attacks, like brute forcing passwords, reveal password?

A: A good administrator would notice this, but it is possible. For example, brute forcing via the SIP REGISTER method would be trivial. Brute forcing is possible, but slow and might get noticed.

Q: Wouldn't encryption help prevent easy dropping?

A: Of course. However, many organizations don't bother to implement encryption on the LAN between the Asterisk server and the phone equipment. It is not always that the equipment cannot handle protocols like SRTP (Secure RTP); it is just rarely thought of. Between remote/satellite endpoints, using IPSec, OpenVPN, SRTP or IAX2's built in encryption is advised. Whatever type of VPN you chose to use, it'll need to be UDP based as TCP VPNs can wreak a VoIP network.

Q: Can't VLANs prevent ARP spoofing?

A: If properly setup, yes. The VoIP equipment should be setup on its own VLAN, away from the typical users. The idea is that the "users" VLAN won't be able to ARP poison the "voip" VLAN.

Chapter 6

Asterisk Hardware Ninjutsu

Solutions in this chapter:

- **Serial**
- **Motion**
- **Modems**
- **Legalities and Tips**

- ☑ **Summary**
- ☑ **Solutions Fast Track**
- ☑ **Frequently Asked Questions**

Introduction

With Asterisk and the flexibility it offers, you can do some truly amazing things. With a "stock" configuration, only using what Asterisk has built in, you can build systems that do some really nifty stuff. If you throw the power of AGIs (Asterisk Gateway Interfaces) into the mix, you can write customized applications that might be difficult to accomplish with other VoIP systems.

Most AGI examples are typically written to take advantage of external resources that Asterisk itself might not have direct access to, or know how to deal with. For example, AGIs have been written to look up ISBNs (book numbers), ANACs (Automatic Number Announcement Circuits) that look up a telephone number information from external sources, text-based games, and IDSs (Intrusion Detection Systems) for monitoring.

We can take the power of AGIs a bit further to interface Asterisk with actual hardware. For example, security cameras, electronic door locks, and card readers to name a few. Creativity is the key.

If you can interface with the hardware externally and interact with it, odds are you can come up with some means to write an AGI to pass that information back.

Serial

To start off, we'll touch on serial communications—yes, that old communications method you used with a modem to connect to the Internet. Even though it's old, traditional serial is used to communicate with room monitoring equipment, magnetic card readers, robotics, environmental control systems, and various other things. It's used where high-speed bandwidth isn't important, but getting data and passing commands is.

These examples are only meant to stir your mind so you come up with creative ways to integrate hardware with Asterisk. While the code does function, the idea is to plant a seed on things you might be able to do with hardware and Asterisk.

Serial "One-Way" AGI

For the first example, we'll be using "one-way" communication via a serial port to the Asterisk server. "One way" means that we don't have to send commands to the device attached via serial. It'll send the information over the serial port automatically. For the generic example code, we use a magnetic stripe reader like the ones that read

your credit card. The idea behind this simple code is that the user must "swipe" a card before they are allowed to place a call. If the card information matches, the call is placed. If it does not, the user is notified and the call is dropped. Before we jump into the code, we must place the AGI in line with outbound calls. That is, before the call is completed, it must run through our routine first. To our extensions.conf, we'd add something like:

```
[ serial-code-1 – extensions.conf ]

exten => _9.,1,agi,serial-code-1.agi
exten => _9.,2, Dial(.....)
```

This is a simple example, and depending on your environment and how you make outbound calls through your Asterisk server, you'll need to modify this. The idea is that, if the number starts with a 9, it'll go through this part of the extensions.conf. If it does, before Asterisk gets to step 2 and dials out, it'll have to pass the *serial-code-1.agi* tests first.

```
[ serial-code-1.agi perl routine ]

#!/usr/bin/perl -T
#
##############################################################################
# serial-code-1.agi                                                          #
#                                                                            #
# By Champ Clark - June 2007                                                 #
# Description: This is a simple routine that'll take data from a serial
# port and respond to it. The example is something like a magstripe
# reader (credit card type). This only deals with one-way communication
# from the device to the AGI. We don't have to send commands to the
# device, so we'll simply listen and parse the data we get and act
# accordingly.                                                               #
##############################################################################

use strict;

use Asterisk::AGI;                      # Makes working with Asterisk AGI
                                        # a bit easier
use Device::SerialPort;                 # Used to connect/communicate with
                                        # the serial device.
```

```
# Following is the string we'll be searching for from the serial port. For
# this simple example, we'll hard-code in a fake driver's license
# to search for. The idea is that before anyone can make an outbound
# call, they must first swipe their licenses through the magstripe
# reader. Of course, this is just an example and could be used for
# anything.

my $searchfor =      "C000111223330";     # My fake driver's license number to
                                          # search for.

my $device    =      "/dev/ttyS1";        # Serial device used.

my $welcomefile = "welcome-serial"; #      This file is played at the
                                          # beginning of the call. It
                                          # explains that some form of
                                          # authentication is needed.

my $grantedfile =     "granted-serial";    # If authentication succeeds, we
                                          # play this and continue through
                                          # the extensions.conf.

my $deniedfile  =     "denied-serial";     # If the authentication fails,
                                          # we'll play this.

my $timeoutfile =  "timeout-serial";       # If we don't see any action on the
                                          # serial port for $timeout seconds,
                                          # we play this file and hang up.

my $errorfile   =     "error-serial";       # This is only played in the event
                                          # of a serial error.


my $serial    =      Device::SerialPort->new ($device) ||
                     die "Can't open serial port $device: $!";

# These are the settings for the serial port. You'll probably want to alter
# these to match whatever type of equipment you're using.

$serial->baudrate(9600) ||
die "Can't set baud rate";
```

```
$serial->parity("none") ||
die "Can't set parity";

$serial->databits(8) ||
die "Can't set data bits";

$serial->stopbits(1) ||
die "Can't set stop bits";

$serial->handshake("none") ||
die "Can't set handshaking";

$serial->write_settings ||
die "Can't write the terminal settings";

# After being prompted to "swipe their card," or do whatever you're trying
# to accomplish, we give the user 30 seconds to do so.
# If they don't, we play the $timeoutfile.

my $timeout="30";

# Various other variables are used to pull this together to make it work.

my $string;                          # From the serial port, concatenated.
my $serialin;                        # What we receive from the serial port.
my $i;                               # Counter (keeps track of seconds passed)
my $AGI = new Asterisk::AGI;

$serial->error_msg(1);               # Use built-in error messages from
$serial->user_msg(1);                # Device::SerialPort.

# Play the welcome file and inform the user that we'll need a serial-based
# authentication method (as in the example magstripe reader). Something
# like "Swipe your card after the tone..."

$AGI->exec('Background', $welcomefile );

# Enter the serial "terminal" loop. We now start watching the
# serial port and parsing the data we get.
```

```
while($i < $timeout )
       {

       # We sleep for a second so we don't hammer the CPU monitoring the
       # serial port. We also use it to increment $i, which keeps track
       # of how long we've been in the loop (for $timeout). To increase
       # polling, you might want to consider using Time::HiRes. I've
       # not run into any problems.

       sleep(1); $i++;

       # Do we have data?

       if (($serialin = $serial->input) ne "" )
          {
         # Append it to $string so we can search it.
         $string = $string . $serialin;


# Now, search for the magic string ( $seachfor ) that will let us continue.
         if ( $string =~ /$searchfor/i)
               {
               $AGI->exec('Background', $grantedfile );
                exit 0;
               }

         # If we receive an enter/carriage return, we'll assume the unit
         # has sent all the data. If that's the case, and we've not
         # matched anything in the above, we'll play $deniedfile and
         # hang up.

         if ( $string =~ /\cJ/ || $string =~ /\cM/ )
              {
               $string = "";
                $AGI->exec("Background", $deniedfile );
                $AGI->hangup();
                exit 0;
               }
        }
```

```
    # If there is some sort of serial error, we'll play this file to let
    # the user know that something isn't set up correctly on our side.

    if ( $serial->reset_error)
       {
       $AGI->exec("Background", $errorfile );
       $AGI->hangup();
       exit 0;
       }
}

# If the user doesn't respond to our request within $timeout, we
# tell them and hang up.

$AGI->exec("Background", $timeoutfile );
$AGI->hangup();
exit 0;
```

Before this routine will function, you'll need to record a few prompt and response audio files.

welcome-serial	This tells the user that they'll need to "swipe" their card before the call is placed. Use something like "Please swipe your card after the tone (tone)."
granted-serial	Lets the user know that the card was read and accepted. For example, "Thank you. Your call is being placed."
denied-serial	Lets the user know the card was declined for the call. For example, "I'm sorry. Your card was not accepted." The call will automatically terminate.
timeout-serial	Informs the user that they didn't swipe their card within the allotted amount of time (via *$timeout*). For example, "I'm sorry. This session has timed out due to inactivity."
error-serial	Lets the user know that there has been some sort of communication error with the serial device. The call is not placed. For example, "There has been an error communicating with the card reader." The call is automatically hung up.

In the example, we are looking for a hard-coded string (*$searchfor*). You could easily make this routine search a file or database for "good" responses.

Dual Serial Communications

Unlike the first example, which relies on simple serial input from a remote device, this code "probes" (sends a command) to a serial device and parses the output for information we want. In the example code, we'll use an "environmental control" system. We want to know what the "temperature" is in a particular room. If the temperature goes above a certain level, we'll have Asterisk call us with a warning.

The interesting idea behind this AGI is that it works in a circular method that requires no addition to the extensions.conf. If the routine is called with a command-line option, it will probe the serial port. If nothing is wrong, it will simply exit. If something *is* wrong, it will create a call file that loops back to itself (without a command-line option) and notifies the administrators.

```
#########################################################################
# serial-code-2.agi                                                     #
#
# By Champ Clark - June 2007                                            #
# Description: This is a simple routine that serves two roles. If
# called with a command-line option (any option), it will send a command to
# a serial device to dump/parse the information. In this example,
# it will send the command "show environment" to the serial device.
# What it looks for is the "Temperature" of the room. If it's under a set
# amount, nothing happens. If it's over the amount, it creates a call
# file (which loops back to serial-code-2.agi).
#
# That's where the second side of this routine kicks in. If _not_ called
# with a command-line argument, it acts as an AGI. This simply lets
# the administrator know the temperature is over a certain amount.      #
#########################################################################

use Asterisk::AGI;                # Simply means to pass Asterisk AGI
                                  # commands.
use Device::SerialPort;           # Access to the serial port.


# We check to see if the routine was called with a command-line argument.
# If it was (and it really doesn't matter what the argument was),
# we can safely assume we just need to check the serial port and
# parse the output (via cron). If the routine was called without
# a command-line argument, then the routine acts like a
```

```
# traditional perl AGI.

if ( $#ARGV eq "-1" ) { &agi();  exit 0; }


my $device      =       "/dev/ttyS1";      # Serial device to check

my $timeout    =      "10";                # Timeout waiting of the
                                           # serial device to respond.
                                           # If it doesn't respond
                                           # within this amount of
                                           # seconds, we'll assume
                                           # something is broken.

# This is the command we'll send to the serial device to get information
# about what's going on.

my $serialcommand = "show environment\r\n ";

my $searchfor = "Temperature";      # The particular item from the
                                    # $serialcommand output we're
                                    # interested in.
my $hightemp = "80";                # If the temp. is higher than this value,
                                    # we want to be notified!

my $overtemp="overtemp-serial";     # This is the audio file that's played
                                    # when the temperature gets out of range
                                    # (or whatever # you're looking for).

# This file is played if the serial device doesn't respond correctly or as
# predicted. The idea is that it might not be working properly, and so the
# system warns you.

my $timeoutfile="timeout-serial";


my $alarmfile="alarm.$$";                          # .$$ == PID
my $alarmdir="/var/spool/asterisk/outgoing";       # where to drop the call

# This tells Asterisk how to make the outbound call. You'll want to
```

```
# modify this for your environment.

my $channel="IAX2/myusername\@myprovider/18505551212";

my $callerid="911-911-0000";              # How to spoof the Caller
                                          # ID. Will only work over
                                          # VoIP networks that allow
                                          # you to spoof it.

# These should be pretty obvious...

my $maxretries="999";
my $retrytime="60";
my $waittime="30";

# This is how we'll communicate with the serial device in question. You
# will probably need to modify this to fit the device you're communicating
# with.

my $serial      =       Device::SerialPort->new ($device) ||
                        die "Can't open serial port $device: $!";

$serial->baudrate(9600) ||
die "Can't set baud rate";

$serial->parity("none") ||
die "Can't set parity";

$serial->databits(8) ||
die "Can't set data bits";

$serial->stopbits(1) ||
die "Can't set stop bits";

$serial->handshake("none") ||
die "Can't set handshaking";

$serial->write_settings ||
die "Can't write the terminal settings";
```

```
my $i;                          # Keeps track of the timer (in case of
                                # serial failure).
my $stringin;                   # Concatenation of all data received on the
                                # serial port. Used to search for our
                                # string.

$serial->error_msg(1);              # Use built-in error messages from
$serial->user_msg(1);               # Device::SerialPort.

my $AGI = new Asterisk::AGI;

# Here we send a command (via the $serialcommand variable) to
# our device. After sending the command, we'll parse out what we need.

$serial->write( $serialcommand );

# We now enter the "terminal loop." The command has been sent,
# and we are looking for the data we are interested in. If we
# send the command but don't receive a response within $timeout seconds,
# we can assume the device isn't working and let the administrator know.

while ( $i < $timeout )
       {

       # We sleep for a second so we don't hammer the CPU monitoring the
       # serial port. We also use it to increment $i, which keeps track
       # of how long we've been in the loop (for $timeout). To increase
       # polling, you might want to consider using Time::HiRes. I've
       # not run into any problems.

       sleep(1); $i++;

       # Did we get any data from the serial port?

       if (($serialin = $serial->input) ne "" )
       {

       # We'll probably get multiple lines of data from our $serialcommand.
       # Every time we receive an "end of line" (carriage return or Enter)
       # we "clear" out the string variable and "new string" array.
```

```
        if ( $serialin =~ /\cJ/ || $serialin =~ /\cM/ )
                {
                $string = "";              # Clear the concatenated string.
                 @newstring="";            # Clear our array used by "split."
                 }

        # If the preceding is not true, the routine concatenates $serialin
        # to $string. Once $string + $serialin is concatenated, we look
        # for the ":" delimiter. This means the serial port will return
        # something like "Temperature: 75". We want the "Temperature"
        # value and will strip out the rest.

         $string = $string . $serialin;
         @newstring=split /$searchfor:/, $string;

        # In this example, we check to see if the devices return a higher
        # temperature than what we expect. If so, we build a call file
        # to "alert" the administrator that the A/C might not be working!
        #
        if ($newstring[1] > $hightemp )
               {
                 if (!open (ALARM, "> $alarmdir/$alarmfile"))
                    {
                    die "Can't write $alarmdir/$alarmfile!\n";
                    }

                 print ALARM "Channel:  $channel\n";
                 print ALARM "Callerid: Temp. Alert <$callerid>\n";
                 print ALARM "MaxRetries: $maxretries\n";
                 print ALARM "RetryTime: $retrytime\n";
                 print ALARM "WaitTime: $waittime\n";
                 print ALARM "Application: AGI\n";
                 print ALARM "Data: serial-code-2.agi\n";
                 print ALARM "Set: tempfile=$overtemp\n";
                 close(ALARM);
               }
        }
}
```

```
# If for some reason communications with the serial device fails, we'll
# also let the administrator know.

if (!open (ALARM, "> $alarmdir/$alarmfile"))
   {
   die "Can't write $alarmdir/$alarmfile!\n";
   }

   print ALARM "Channel:  $channel\n";
   print ALARM "Callerid: Temp. Alert <$callerid>\n";
   print ALARM "MaxRetries: $maxretries\n";
   print ALARM "RetryTime: $retrytime\n";
   print ALARM "WaitTime: $waittime\n";
   print ALARM "Application: AGI\n";
   print ALARM "Data: serial-probe.agi\n";
   print ALARM "Set: tempfile=$timeoutfile";
   close(ALARM);

exit 0;

# end of routine.

# This subroutine acts as an AGI if the routine is called without a
# command-line argument.

sub agi
{

my $AGI = new Asterisk::AGI;
my %AGI;

#  If this subroutine is called, obviously something has gone seriously
#  wrong. The call (via a call file) has already been placed, this just
#  lets the administrator know "what" went wrong.

$AGI->answer();                    # Pick up! We need to tell the user
                                   # something!
$AGI->exec('Wait', '1');           # Give me a warm fuzzy...

# We grab the audio file we want to play from the "tempfile" variable in
```

```
# the call file and play it.

$tempfile=$AGI->get_variable('tempfile');
$AGI->exec('Background', $tempfile );
$AGI->hangup();

exit 0;
}
```

You will need to record a couple of prompt/audio files. They include the following:

overtemp-serial	This is the file that's played if the temperature (in our example) is over the *$hightemp*.
timeout-serial	This file is played if the serial device doesn't respond in *$timeout*. The idea is that the device might not be functioning.

Since the routine is operating as an AGI, it'll need to be copied to your AGI directory. This is typically done by using /var/lib/asterisk/agi-bin. This way, Asterisk will have access to the routine. To start monitoring the hardware, you'll want to create a cron job that would "test" every ten minutes or so. That cron entry would look something like this:

```
*/10  * * * * /var/lib/asterisk/agi-bin/serial-code-2.agi test 2>&1 > /dev/null
```

Motion

Motion is open-source software that uses video camera equipment to record "motion" in a room. It's primarily used for security purposes, and has many features. For example, you can take snapshots of an area every few seconds or create time-lapse movies.

Of course, to use Motion you'll need the proper hardware. "All weather cameras" and video capture cards have come down in price over the years. I like to keep things simple, so I use multiport BT848 (chipset) capture cards for my home security system. It's a generic chip set that works well with Linux. My particular card comes with four onboard built-in ports, but it can support up to eight ports with an external adapter. This means I can run up to eight cameras at a time. Considering I use this to monitor my home (front yard, back yard, inside my office, and so on), I'm

not worried that the cameras and capture card chipset won't produce high-definition quality. I simply want a means to record events and watch my cameras over the Internet.

If you have spare camera equipment around, you might want to look into how well it's supported under Linux. Some USB cameras require proprietary drivers to work while others do not. The first step is to get the camera up and working under Linux, and then configure it to work with Motion.

To obtain Motion, simply go to http://motion.sourceforge.net. Once you've downloaded it, installation is typically at *./configure && make && make install*. Some Linux-based distributions have motion packages you might want to look into.

The motion.conf file can be quite daunting, but don't let it scare you. It'll probably take a bit of tweaking to get your configuration up and running and that will largely depend on the type of hardware you use. If you're using more than one camera, it's better to get one camera online first before trying to configure the rest of them. Motion uses a "threaded" system in monitoring multiple cameras, so you'll actually have multiple configuration files per camera.

[tail end of a default motion.conf file]

```
############################################################
# Thread config files - one for each camera.
# However, if there's only one camera, you only need this config file.
# If you have more than one camera, you MUST define one thread
# config file for each camera in addition to this config file.
############################################################

# Remember: If you have more than one camera, you must have one
# thread file for each camera. Thus, two cameras require three files:
# This motion.conf file AND thread1.conf and thread2.conf.
# Only put the options that are unique to each camera in the
# thread config files.
; thread /usr/local/etc/thread1.conf
; thread /usr/local/etc/thread2.conf
; thread /usr/local/etc/thread3.conf
; thread /usr/local/etc/thread4.conf
```

The option we'll be focusing on is a pre-thread configuration file, so once you have a working configuration:

```
# Command to be executed when a motion frame is detected (default: none)
; on_motion_detected value
```

The idea is that when motion is detected, we can have Motion (the program) run a routine. When I leave town for an extended period of time, I want to know if motion is detected within my home. I'm not as concerned about outside because false positives would drive me crazy. For example, cats or dogs that just happen to roam through my yard, I'm not interested in.

If an event happens inside the home and I know nobody is there, then I certainly want to know! So, with the cameras that are internal, we'll use the *on_motion_detect* option to run a routine that'll call my cell phone and alert me to something or someone in my house. We can do this on a per-thread configuration file basis. So, for cameras that are outside, we won't add the *on_motion_detect* option.

The Idea behind the Code

The idea behind this code is simple, but does two different jobs. One is to create the outgoing call file to let you know when "motion" has been detected. The other is to be an AGI so that once the call is made, Asterisk can "tell you" which camera saw the motion. Since this routine handles all the necessary functions, you can simply copy it to your Asterisk AGI directory (usually /var/lib/asterisk/agi-bin) and go! No modifications are needed to Asterisk configuration files (for example, extensions.conf).

When Motion "sees motion," it will call the routine via the *on_motion_detect* command. Within the Motion configuration files for each camera we wish to monitor, we'll pass the command to alert us if something is detected. It will look something like this:

```
on_motion_detect /var/lib/asterisk/agi-bin/alarm.agi 1
```

The number "1" is passed as a command-line argument. In this example, this represents *camera 1*. Since we are passing a command-line argument, the routine is programmed to know that this is coming from Motion. When called as an AGI from Asterisk, no command-line argument is passed. Let's run through the entire routine to pull everything together.

Motion is monitoring *camera 1*, which we'll say is your home office. Motion detects "motion" in the room and starts recording the action, firing off the /var/lib/asterisk/agi-bin/alarm.agi file with the command-line option of "1," which signifies the camera that was triggered. The alarm.agi creates a call file in the Asterisk

outgoing call directory (typically, /var/spool/asterisk/outgoing). The contents of this file will look something like this:

```
Channel:  IAX2/myusername@myprovider/18505551212
Callerid: Security Camera <911-911-0001>
MaxRetries: 999
RetryTime: 60
WaitTime: 30
Application: AGI
Data: alarm.agi
Set: camera=1
```

The *Channel:* is an option in the outgoing call file that gives the method of "how" to make the outgoing call. In this example, I'm using a provider that supports IAX2. You could easily change this to use a Zap device or SIP. Note the *Applications: AGI* and *Data: alarm.agi*. When alarm.agi builds the call file, it creates it in such a way that it loops back on itself. The *Set: camera=1* passes the camera that recorded the event. We attempt to spoof the Caller ID to *911-911-0001*. This just shows that an emergency has occurred on camera 1 (0001 in the Caller ID field). It will only work if your provider allows you to modify your Caller ID (CID). On the PSTN, the *Security Camera* portion will be dropped completely, even if the number is spoofed. It'll work fine over VoIP networks, but the PSTN does a lookup of the number and fills in the Name section of the Caller ID field. On the PSTN, that's out of your control.

Once the call file is built and saved in the Asterisk "outgoing" directory, Asterisk will almost immediately grab this file and follow the instructions in it. Asterisk calls via the method in the Channel: field, and then waits for the call to supervise. *Supervision* is a term used to signify that something or someone has "picked up" the call.

Upon supervision by you answering your phone, Asterisk executes the AGI alarm.agi. Since Asterisk is calling the routine this time without command-line arguments, the routine is programmed to act as an AGI. Upon you answering, the AGI side of the alarm.agi kicks in and feeds Asterisk commands like the following:

```
ANSWER
EXEC  Wait 1
GET VARIABLE camera
EXEC Background camera-1
HANGUP
```

As you can see, it's pretty simple! Answer the call, and wait one second. Get the contents of the variable *camera*. Remember that variable? It holds the numeric value of what camera was triggered. Once that variable is obtained, we issue a "Background" (audio playback) of the file "camera{camera variable}". In this case, that'll be camera1.

This means you'll want to record a couple of audio files to use with this routine. In this example, we said that camera1 was our home office. So we'd want to pre-record some audio files that represent our cameras. In this case, we might have an audio file that says, "Warning! There appears to be motion in the home office." The following is the standalone routine we're using. Of course, you could take this simple routine and modify it to do a multitude of things.

```
[alarm.agi]

#!/usr/bin/perl -Tw
#
##############################################################################
#  alarm.agi                                                                 #
#  By Da Beave (Champ Clark) - June 2007                                     #
#  Description: This routine actually serves two purposes. It acts as the
#  routine that creates the "call files" and that the AGI routine Asterisk
#  uses. When this routine is called by Motion, a command-line argument
#  is given to specify which camera saw the motion. If there is no
#  command-line argument, then the routine services as an Asterisk AGI
#                                                                           #
##############################################################################

use strict;
use Asterisk::AGI;                 # Makes working with Asterisk AGI a
                                   # little bit easier.

# This is the name of the "sound" file to call. For example, if Motion
# sees motion on camera #1, it'll send to this routine: /var/lib/asterisk/
# agi-bin/alarm.agi 1. So, the file (in the /var/lib/asterisk/sounds)
# "camera1" is called to inform that motion was caught on "camera1".
# This is the prefix of the file (that is, camera1, camera2, and so on).

my $camerafile="camera";

# We check to see if there is a command-line argument. If not, we assume
```

```
# the routine needs to act like an Asterisk AGI. If it _does_ have a
# command-line argument, then we assume Motion has called the
# routine and given the camera information via argv....

if ( $#ARGV eq "-1" ) { &agi();  exit 0; }

# $channel contains the information about how the call is to be placed.
# In this example, we'll be using IAX2. However, you could use Zap, SIP,
# or other methods Asterisk supports. Replace with your method of
# dialing/phone number.

my $channel="IAX2/myusername\@myprovider/18505551212";

# We spoof the Caller ID. This will only work if you're VoIP provider
# allows you to modify the Caller ID information. With my VoIP carrier,
# I have to supply a full ten-digit phone number. YMMV (you might be able
# to get away with something shorter). So, when Motion calls me, it will
# send "911-911-000" as the NPA/NXX. The last digit is the camera that has
# reported motion.

my $callerid="911-911-000";

# These should be fairly obvious...

my $maxretries="999";
my $retrytime="60";
my $waittime="30";

# To keep outgoing calls unique, we build call files based on their PID.

my $alarmfile="alarm.$$";                          # .$$ == PID
my $alarmdir="/var/spool/asterisk/outgoing";       # where to drop the call
                                                   # file.

my $tmpfile;
my $setinfo;
my $camera;
```

```
#############################################################################
# This is where the actual call file is built. Remember, with Asterisk,
# any call files that show up in the outgoing queue (usually /var/spool/
# asterisk/outgoing) are used automatically.
#############################################################################

# Open the outgoing queue file and feed it the commands.

if (!open (ALARM, "> $alarmdir/$alarmfile"))
        {
        die "Can't write $alarmdir/$alarmfile!\n";
        }

print ALARM "Channel:  $channel\n";
print ALARM "Callerid: Security Camera <$callerid$ARGV[0]>\n";
print ALARM "MaxRetries: $maxretries\n";
print ALARM "RetryTime: $retrytime\n";
print ALARM "WaitTime: $waittime\n";
print ALARM "Application: AGI\n";
print ALARM "Data: alarm.agi\n";
print ALARM "Set: camera=$ARGV[0]\n";
close(ALARM);

#############################################################################
# AGI section: If no command-line arguments get passed, we can
# assume it's not Motion calling the routine (because Motion passes
# the camera on the command line). Asterisk calls alarm.agi without
# any command-line arguments, so we act as an AGI.
#############################################################################

sub agi
{

my $AGI = new Asterisk::AGI;
my %AGI;

# This pulls in our Asterisk variables. For example, $input{camera},
# which we are using to pass the camera number.

my %input = $AGI->ReadParse();
```

```
# Okay - now we do our song and dance for the user we called!

$AGI->answer();                          # Pick up! We need to tell the user
                                         # something!
$AGI->exec('Wait', '1');                 # Give me a warm fuzzy...

$camera=$AGI->get_variable('camera');    # Get the "camera" variable.
$tmpfile="$camerafile$camera";
$AGI->exec('Background', $tmpfile );
$AGI->hangup();
exit 0;
}
```

Modems

Traditional analog modems present a problem with VoIP. First off, you're probably asking "Why the heck would you even attempt to hook up a traditional modem via VoIP?" One practical reason is because many systems still use traditional analog modems for communications—for example, point-of-sales equipment, TiVo, and credit card equipment. Before attaching any devices like these to a VoIP network, security should be considered. Equipment of this type might transmit sensitive information. It's less than practical to play with the PSTN network via VoIP network, dial into old style BBS systems, use older networks that still require a dialup connection, or "scan" for modems and telephone equipment. Scanning for modems and telephone equipment is known as *war dialing*. The term comes from the 1984 film *War Games*, but the term and technique is actually older than the movie, and is sometimes referred to as *demon dialing*. The term war dialing, though, is the one that sort of stuck in the phreaking and hacking community. In the film, our hero (played by Matthew Broderick) dials every telephone number within an exchange searching for interesting telephone and computer equipment owned by a fictional company named Protovision, Inc. In real life, the idea of war dialing is the same as in the movie and can be useful during security audits. During a security audit, you're dialing numbers within a particular block around your target searching for things like modems, fax machines, environmental control systems, PBXs, and other equipment connected to the PSTN.

You'll need prior permission, and checking with your local laws is advised before war dialing!

So, why would you want this behind VoIP when you could hook up a modem on the traditional PSTN? With VoIP, you are able to mask "where" you are calling from. Unlike the PSTN, our ANI information, which cannot be easily spoofed, won't be passed. We can "spoof" things like our Caller ID. It makes it harder to track down where the calls are coming from.

For whatever reason you'll be using an analog modem with VoIP, several things must be considered. First off, you won't be able to make very high-speed connections. The top speed you'll be able to accomplish is about 4800 baud. This is due to how the modem MOdulates and DEModulates (hence the term *modem*) the signal and network latency. At very low speeds, like 300 baud, a simple means of encoding the data is used, known as frequency-shift keying (FSK). The originator of the call transmits at 1070Hz or 1270Hz. The answering side transmits at 2025Hz or 2225Hz. This is well within the range and type of encoding we can do over VoIP. A speed of 1200 baud is also achievable and stable. At that speed, a simple encoding scheme is used, known as Phase-Shift Keying (PSK). Once you step into higher speeds like 14.4k, 28.8k, 33.6k, and above, you get into very time-sensitive encoding techniques, like quadrature amplitude modulation (QAM), which don't respond well in a VoIP world.

To keep things stable, I generally keep my rates locked at 1200 baud. Not blinding fast, but it's good enough to detect and look at remote systems. You might be wondering, "Wait a minute! How come things like Fax over VoIP can handle such higher baud rates?!" Good question!

As VoIP became more and more popular, the ITU (International Telecommunications Union) created a protocol known as T.38, which is sometimes referred to as FoIP (Fax over IP). Asterisk and many VoIP adapters now support T.38. When you plug in your fax machine to a VoIP adapter, it may very well auto-detect and support the fax under T.38. What T.38 does is it takes the fax signal and converts it to more data-network-friendly SIP/SDP TCP/UDP packets that get transmitted over the Internet. Since the fax signal doesn't actually have to traverse the Internet, greater speeds can be achieved. If your adapter or provider does not support T.38 and the analog fax signal has to transverse the Internet, then you'll run into similar issues as you would with analog modem.

This might make you wonder why there isn't a Modem over IP protocol. Well… in truth, the ITU *has* created such a standard, known as V.150.1 (also known as

V.MOIP) in 2003. It operates much like T.38 in that it takes the analog signal and converts it to a UDP/TCP packet that can traverse the Internet easily. Unfortunately, even fewer VoIP providers and equipment support V.150.1. This might change as VoIP becomes more and more popular and people want to connect equipment that would traditionally connect to the PSTN. Until that time, though, we are stuck doing it the hard way.

It order to test Modem over IP, you'll obviously need an analog modem. You'll also need some sort of VoIP telephone adapter like the Linksys PAP2 or Cisco ATAs. These devices are normally used to connect normal telephones to a VoIP network. They typically have one or two RJ11 jacks on them to plug in your traditional telephone. They also have an RJ45 network jack that will connect to your LAN. Rather than plugging in a traditional telephone into the RJ11 jack, we'll use this port to attach our analog modem.

Configuration of the VoIP adapter largely depends on the hardware itself. Configuration on the Asterisk side is typically pretty straightforward and simple. I use a dual line Linksys PAP2, which employs SIP. Since it is a dual line (two RJ11s) configuration, I have a [linksys1] and [linksys2] section in my Asterisk sip.conf file. The following shows what mine looks like:

```
[linksys1]
type=friend                         # Accept inbound/outbound
username=linksys1
secret=mysecret
disallow=all
;allow=gsm                          # This will NOT work for a modem.
allow=ulaw                          # Works much better with a modem.
context=internal
host=dynamic
```

In order for Modem over IP to work, you must consider two important factors. First: the better your network connection, the better your modem connections will be. The lower the latency, the better. Second: you must *not* use any sort of compressed codec! Compressed codecs, like GSM or G.729, will alter the analog signal/encoding, which will cause connections to completely fail. You'll want to use the G.711 (u-law) codec. If you can accomplish these two requirements, you'll be much better off. If you are configuring point-of-sales equipment or some sort of consumer electronics (TiVo, and so on), you'll probably want to test a bit and play with baud rate settings to see what you can achieve.

Fun with Dialing

If the modem you wish to use is attached to a computer and is not PoS/consumer electronics gear, you can start up some terminal software and go! Under Linux, and other Unix-like operating systems, multiple terminal software programs can be used. Minicom is probably one of the more well known and useful terminal software programs around. It comes with most distributions, but if your system doesn't have it, the source can be downloaded from http://alioth.debian.org/projects/minicom/. If Minicom doesn't suit your tastes, check out Seyon for X Windows, which can be obtained from ftp://sunsite.unc.edu/pub/Linux/apps/serialcomm/dialout/ (look for the latest Seyon release). No matter which software you use, knowledge of the Hayes AT command set is a plus. Hayes AT commands instruct the modem in "what to do." For example, ATDT means *Dial Tone*. You'll probably want to read over your modem's manual to get a list of supported AT commands.

Okay, so your modem is hooked up. Now what? I like to use VoIP networks to dial in to remote countries and play with things that might not be accessible in the United States. For example, in France there is a public X.25 (packet-switched) network known as Transpac that I like to tinker with. I also use VoIP with my modem to call Russian BBSs and an X.25 network known as ROSNET. There's a lot of nifty stuff out there that's not connected to the Internet and this gives me a cheap, sometimes even free, way to call foreign countries.

War Dialing

Another useful feature for a modem connection via VoIP is security audits. Rogue modems and various telephone equipment are still a security problem in the corporate world. When hired to do a security audit for an organization, I'll suggest a "scan" of the telephone numbers around the company to search for such rogue equipment. It's not uncommon for a company to not even be aware it has equipment connected to the PSTN. The usefulness of scanning via VoIP is that I can mask where I'm coming from. That is, I can spoof my Caller ID and I know my ANI information on the PSTN will be incorrect—meaning I can hide better. One trick I do is to spoof my telephone number as the number from a known fax machine. This way, during my war dialing, if someone tries to call me back, they'll dial a fax machine. From there, they'll probably think the fax machine just misdialed their telephone number and forget about it.

Of course, spoofing Caller ID can be useful in other ways for security audits—such as with social engineering. Social engineering is nothing more than presenting yourself as someone you're not and requesting information you shouldn't have, or requesting someone do something they shouldn't—for example, spoofing the Caller ID of an Internet Service Provider (ISP) and requesting changes be done to the network (change proxies so you can monitor communications) or requesting a password. This is getting off the topic of war dialing, but it's still useful.

I don't particularly want to spoof every time I make a call through my Asterisk system, so I set up a prefix I can dial before the telephone number. In my case, if I want to call 850-555-1212 and I want to use caller ID spoofing, I'll dial 5-1-850-555-1212. The initial "5" directs Asterisk to make the outbound call using a VoIP provider with Caller ID spoofing enabled. My extensions.conf for this looks something like:

```
; Caller ID spoofing via my VoIP provider.
;

exten => _5.,1,Set,CALLERID(number)=904-555-7777
exten => _5.,2,monitor,wav|${EXTEN:1}
exten => _5.,3,Dial(IAX2/myusername@myprovider/${EXTEN:1})
```

You might be wondering why we don't do a *Set,CALLERID(name)*. There isn't really much point. Once the call hits the PSTN, the number is looked up at the telephone company database and the Name field is populated. This means, once the call hits the traditional PSTN, you can't modify the Name field anyways. One interesting thing you *can* do, if you're trying to figure out who owns a phone number is spoof the call as that phone number to yourself. Once the call reaches the PSTN and calls you, the telephone company will look up the spoofed number in its database and display the name of who owns it. This is known as backspoofing and isn't completely related to war dialing, but can be useful in identifying who owns particular numbers. The Monitor option lets you record the audio of the call, so you can listen later and see if anything was found that the war dialer might have missed. It's advised you check your local laws regarding recording telephone calls. If you don't wish to do this, the option can be removed.

With our adapter set up and Asterisk configured, we are ready to war dial! Now we just need the software to send the commands to our modem and then we can start dialing. Several programs are available, some commercial and some open source, that'll take over the dialing and analysis of what you find. One of the most popular is

the MS-DOS–based ToneLoc. While an excellent war dialer, it requires the extra overhead of running a DOS emulator. Phonesweep is another option, but runs under Microsoft Windows and is commercial. For Linux, and Unix in general, I use the open-source (GPL) program iWar (Intelligent Wardialer). It was developed by Da Beave from the network security company Softwink, Inc. Many of its features compete with commercial products.

Some of the features iWar supports are random/sequential dialing, key stroke marking and logging, IAX2 VoIP support (which acts as an IAX2 VoIP client), Tone location (the same method ToneLoc uses), blacklist support, a nice "curses" console interface, auto-detection of remote system type, and much more. It will log the information to a standard ASCII file, over the Web via a CGI, MySQL, or PostgreSQL database. You probably noticed the IAX2 VoIP support. We'll touch more on this later.

To obtain iWar, go to www.softwink.com/iwar. You can download the "stable" version, but they suggest you check out the CVS (Conversion Version System). This is a development version that typically has more features. To download via CVS, you'll need the CVS client loaded on your machine. Many distributions have CVS preloaded or provide a package to install it. If your system doesn't have it, check out www.nongnu.org/cvs/ for more information about CVS.

To download iWar via CVS, type

```
$ CVSROOT=:pserver:anonymous@cvs.telephreak.org:/root; export CVSROOT
$ cvs login
```

When prompted for a password, simply press Enter (no password is required). This will log you in to the development CVS system. To download the source code, type

```
$ cvs -z9 co -A iwar    # -z9 is optional (for compression)
```

If you're using the CVS version of iWar, it's suggested you join the iWar mailing list. It's a low-volume mailing list (one or two e-mails per week) that contains information about updates, bug fixes, and new features.

After downloading the software, installation uses the typical ./configure && make && make install. For MySQL or PostgreSQL support, you'll need those libraries preloaded on your system before compiling iWar. If you wish to compile iWar with IAX2 support, you'll need to install IAXClient. You can locate and read about IAXClient at http://iaxclient.sourceforge.net/. This library allows iWar to become a full featured IAX2 VoIP client and war dialer. For proper installation of IAXClient, refer to their mailing list and Web page.

Of course, iWar will compile without MySQL, PostgreSQL, or IAXClient support and will work fine for our purposes with a standard analog modem. Once compiled, we are ready to fire it at our target!

To give you an idea of the options with everything built in (MySQL, PostgreSQL, and IAXClient), the following is the output of *iwar –help*.

```
iWar [Intelligent Wardialer] Version 0.08-CVS-05-24-2007 - By Da Beave
(beave@softwink.com)

[ iwar -help output]

usage: iwar [parameters] --range [dial range]

-h, --help           :  Prints this screen
-E, --examples       :  Examples of how to use iWar
-s, --speed          :  Speed/Baud rate
                        [Serial default: 1200] [IAX2 mode disabled]
-S, --stopbit        :  Stop bits [Serial Default: 1] [IAX2 mode disables]
-p, --parity         :  Parity (None/Even/Odd)
                        [Serial default 'N'one] [IAX2 mode disabled]
-d, --databits       :  Data bits [Serial default: 8] [IAX2 mode disabled]
-t, --device         :  TTY to use (modem)
                        [Serial default /dev/ttyS0] [IAX2 mode disabled]
-c, --xonxoff        :  Use software handshaking (XON/XOFF)
                        [Serial default is hardware flow control]
                        [IAX2 mode disabled]
-f, --logfile        :  Output log file [Default: iwar.log]
-e, --predial        :  Pre-dial string/NPA to scan [Optional]
-g, --postdial       :  Post-dial string [Optional]
-a, --tonedetect     :  Tone Location (Toneloc W; method)
                        [Serial default: disabled] [IAX2 mode disabled]
-n, --npa            :  NPA (Area Code - ie 212)
-N, --nxx            :  NXX (Exchange - ie - 555)
-A, --nonpa          :  Log NPA, but don't dial it (Useful for local calls)
-r, --range          :  Range to scan (ie - 5551212-5551313)
-x, --sequential     :  Sequential dialing [Default: Random]
-F, --fulllog        :  Full logging (BUSY, NO CARRIER, Timeouts, Skipped, etc)
-b, --nobannercheck  :  Disable banners check
                        [Serial Default: enabled] [IAX2 mode disabled]
-o, --norecording    :  Disable recording banner data
```

```
                          [Serial default: enabled] [IAX2 mode disabled].
-L, --loadfile       :  Load numbers to dial from file.
-l, --statefile      :  Load 'saved state' file (previously dialed numbers)
-H, --httplog        :  Log data via HTTP to a web server
-w, --httpdebug      :  Log HTTP traffic for CGI debugging
-C, --config         :  Configuration file to use [Default: iwar.conf]
-m, --mysql          :  Log to MySQL database [Optional]
-P, --postgresql     :  Log to PostgreSQL database [Optional]
-I, --iax2           :  Enabled VoIP/IAX2 for dialing without debugging
                        (See iwar.conf)
-i, --iax2withdebug  :  Enabled VoIP/IAX2 for dialing with debugging
                        (--iax2withdebug <filename>)
```

iWar also comes with a configuration file to set up things like your serial port, baud rate, and various logging options. The default iwar.conf is suited to work with most hardware, but it's advised to tweak it to your hardware.

```
[ default iwar.conf ]
##############################################################################
                                                                          ##
# iWar configuration file. Please see http://www.softwink.com/iwar for   ##
# more information.                                                        ##
##############################################################################

##############################################################################
# Traditional serial port information                                      #
##############################################################################

#
# Serial port information (prt, speed, data bits, parity). Command--
# line options override this, so you can use multiple modems.
#
port /dev/ttyS0
speed 1200
databits 8
parity N

#
# Modem INIT string. This can vary for modem manufacturers. Check your
# modem's manual for the best settings. Below is a very _basic_ init
# string. The main objective toward making things work better is DTR
```

```
# hangups and dial speed. Here's what is set in this string.
#
# E1   =  Echo on
# L3   =  Modem speaker on high
# M1   =  Modem speaker on until carrier detect
# Q0   =  Result codes sent
# &C1  =  Modem controls carrier detect
# &D2  =  DTE controls DTR (for DTR hangup!)
# S11  =  50 millisecond DTMF tones. On the PSTN in my area, 45ms DTMF
#         works fine, and might work for you. It's set to 50ms to be safe.
#         My ATAs can handle 40ms DTMF, which is as fast as my modem
#         can go. If you're having dial problems, slow down this
#         setting by increasing it. For faster dialing, decrease this.
# S06  =  How long to "wait" for a dial tone. Modems normally set this
#         to two seconds or so. This is terrible if you're trying to
#         detect tones! This is for Toneloc-type tone location
#         (ATDT5551212W;). You may need to adjust this.
# S07  =  Wait 255 seconds for something to happen. We set it high
#          because we want iWar to decide when to hang up. See
#         "serial_timeout."
#
# Extra things to add to the init string:
#
# +FCLASS=1  = Want to scan for fax machines (And only fax - however,
#              the Zylex modems might do data/fax)
#
# X4         = All modems support this. If you add "X4" to the init
#              string, your modem will detect "NO DIALTONE"  and "BUSY".
# X6 or X7   = Certain modems (USR Couriers, for example) can
#               detect remote call progression. X7 is good because
#               it leaves everything on (RINGING, BUSY, NO CARRIER)
#               except "VOICE." "VOICE" is sometimes triggered by
#               interesting tones. X6 leaves everything on.
#               This is good when you're doing carrier detection!
# X0         = Set the modem to blind dialing. This is good if you're into
#              "hand scanning." The modem doesn't attempt to detect
#              anything like BUSY or VOICE (it will still detect carriers).
#              You can then use the manual keys to mark numbers.

init ATE1L3M1Q0&C1&D2S11=50S07=255
```

```
# If your modem is not capable of doing DTR hangups, then leave this
# enabled. This hangs up the modem by sending the old "+++ATH" to the
# modem on connections. If your _positive_ your modem is using DTR drops
# to hang up, you can save scan time by disabling this.
# If you enable this and DTR drops don't work, your line will NOT hang up
# on carrier detection!


plushangup 1
plushangupsleep 4

# "This only applies to modems that support remote call progression
# (for example, "RINGING"). Modems that can do this are the USR
# Couriers, Multitech and mccorma modems. If your modem doesn't
# support this, ignore it.

# remote_ring 5

# If remote ring is enabled and functional, then this is the max time
# we'll allow between rings with no result code (BUSY, CONNECT,
# VOICE). For example, if we receive two RINGING result codes,
# but for 30 seconds see nothing else, then something picked up on the
# remote side that the modem didn't register. It might be worth going back
# and checking.

# ring_timeout 20

# This is for modems that reliably detect remote "tones." This changes
# the dial string from the standard ATDT5551212 to ATDT5551212w;
# (See about the ATS06=255 - wait for dial tone). When the modem
# dials, it "waits" for another tone. If the iWar receives an "OK,"
# then we know the end supplied some sort of tone. Most modems can't
# do this. Leave this commented out if your modem doesn't support it.

# tone_detect 1

# Banner file. Banners are used to attempt to figure out what the remote
# system is.
```

```
banner_file /usr/local/etc/banners.txt

# Blacklist file. This file contains phone numbers that should
# never be called (for example, 911).

blacklistfile /usr/local/etc/iwar-blacklist.txt

# Serial connection timeout (in seconds). This is used to detect when
# the modem doesn't return a result code. In that event, we'll
# hang the modem up. See the ATS07 (S07) at the top of this config file.

serial_timeout 60

# When connected (carrier detected), this is the amount of time to
# wait (in seconds) for iWar to search for a  "login banner." If
# no data is received and/or there is no banner, we hang up
# when this amount of time is reached.

banner_timeout 20

     # On the off chance that we keep receiving data, and the banner_timeout is
# never reached, this is the amount of data we will receive before giving
# up (hang up). Data sent from the remote end reset the banner_timeout -
# without this safe guard, the system may never hang up because it keeps
# receiving data! Value is in bytes.

banner_maxcount 2048

# After connecting, this is how long we wait (in seconds) to send a
# return. Some systems won't reveal their banners until they
# receive several \r\r's. Value is in seconds.

banner_send_cr 10

# This is the number a carriage returns to send once banner_send_cr
# is reached.

banner_cr 3

# After connecting, wait this long until picking up and trying to
```

```
# redial out. Measured in seconds.

connect_re-dial 5

# How long to wait before redialing after a BUSY, NO CARRIER, or other type
# of event, in seconds. On PSTN environments, you need to wait a few
# seconds before dialing the next number (or it'll register as a
# "flash"). On VoIP hardware-based scans, you can probably lower
# this to decrease scan time. This does not affect IAX2 dialing.

redial 3

# DTR re-init. Some modems (USR Couriers), when DTR is dropped, have
# to re-init the modem (I assume the USR treats DTR drops like ATZ).
# This will re-init after DTR drops.

# dtrinit 1

# Amount of time to drop DTR. Some modems require longer DTR drops to
# hang up. Value is in seconds. (If possible, 0 is best!)

dtrsec 2

# You can log all your information into a MySQL database. These are the
# credentials to use
#

###########################################################################
## MySQL Authentication                                                  ##
###########################################################################

mysql_username iwar
mysql_password iwar
mysql_host 127.0.0.1
mysql_database iwar

###########################################################################
## PostgreSQL Authentiation                                              ##
###########################################################################
```

```
postgres_username iwar
postgres_password iwar
postgres_host 127.0.0.1
postress_database iwar

###############################################################################
## HTTP Logging                                                              ##
###############################################################################
#
# The following is an example URL that is based from iWar to a Web server
# during HTTP logging.
#
# http://www.example.com/cgi-bin/iWar-HTTP.cgi?NPA=850&NXX=555&Suffix=1225&
# Revision=0&NumberType=2&Description=Looks%20good%21%21&Username=myname&
# Password=mypassword
#
# If your CGI requires authentication (see earlier), then set these.
# Otherwise, just leave these values alone (the remote site will ignore
# the values)

http_username iwar
http_password iwar

# The web server you are logging to:

http_log_host www.example.com

# HTTP port the Web server is running on.

http_port 80

# The path of the application on the remote Web server doing the logging.
# For more information, see the example iWar-HTTP.cgi.

http_log_path /cgi-bin/iWar-HTTP.cgi

# The combination of http_log_host + http_log_path logging URL would
# look something like this:
#
# http://www.example.com/cgi-bin/iWar-HTTP.cgi
```

```
#
# The example "GET" string (at the top of "HTTP Logging") is automatically
# tacked to the end of this! Ta-da!

#########################################################################
## Following are IAX2 values that have no affect when serial scanning ##
#########################################################################

# IAX2 username of your VoIP provider/Asterisk server

iax2_username iwar

# IAX2 password of your VoIP provider/Asterisk server. This is _not_
# required if your Asterisk server doesn't use password authentication.

iax2_password iwar

# IAX2 provider/Asterisk server. Can be an IP address or host name.

iax2_host     192.168.0.1

# 99.9% of the time, it's not necessary  to "register" with your provider
# to make outbound calls! It's highly unlikely you need to enable this!
# In the event you have a strange provider that "requires" you to
# register before making outbound calls, enable this.

#iax2_register 1

# If you're using iWar directly with a IAX2 provider, then set this
# to your liking. If you're routing calls via an Asterisk server, you can
# callerid spoof there. With Asterisk, this will have no
# affect.

iax2_callerid_number 5551212

# iax2_millisleep is the amount of time to sleep (in milliseconds) between
# IAX2 call tear down and start up. Probably best to leave this alone.

iax2_millisleep 1000
```

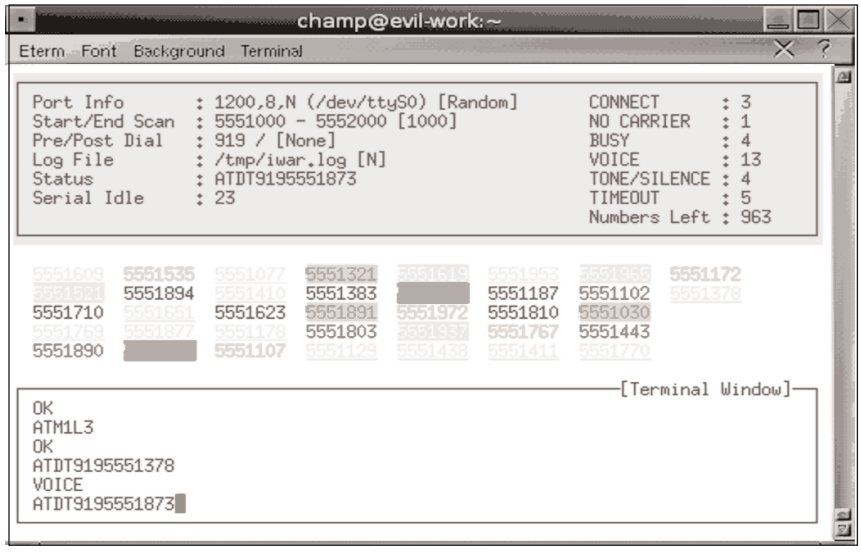
As you can see, many options must be set and tweaked depending on your hardware. With Asterisk configured to spoof the caller ID and make the outbound call using G.711, your VoIP adapter set to communicate properly via G.711 to your Asterisk server, your modem set up through the VoIP adapter, and iWar configured to properly use your hardware, we are ready to launch the dialer! To do this, type

```
$ iwar -predial 5 -npa 904 -nxx 555 -range 1000-1100
```

The —*predial* option tells iWar to dial a "5" before the rest of the number. We do this to let our Asterisk server know we want to spoof Caller ID and to go out our VoIP provider. The —*npa* option (Numbering Plan Area) is another way to say what area code we wish to dial. The —*nxx* is the exchange within the NPA that we'll be dialing, and the —*range* lets iWar know we want to dial all numbers between 1000 and 1100. When iWar starts, it'll send your modem a command that looks like *ATDT51904555XXXX*. The *XXXX* will be a random number between and including 1000 to 1100. The output of iWar will be stored in a flat ASCII text file named iwar.log. By default, as iWar dials, it will record information about interesting numbers it has found and attempt to identify remote modem carriers it runs into. If you wanted to log the information into a MySQL database, you'd have to configure the iwar.conf with your MySQL authentication. Then, to start iWar with MySQL logging enabled, you'd simply add the —*mysql* flag.

iWar is highly configurable. Once started, your screen, after a bit of dialing, should look something like Figure 6.1.

Figure 6.1 The iWar Startup Screen



The top part of the screen gives you basic information like what serial port is being used, where the log file is being stored and statistics on what it has found (on the far top right). At the bottom is a "terminal window." This allows you to watch iWar interact with the modem. In the middle of the screen, with all the pretty colors, are the numbers that have been dialed. Those colors represent what iWar has found. By looking at those colors and the number highlighted, you can tell what numbers were busy, where modem carriers were found or numbers that gave no response. The color breakdown for iWar is shown in the following table.

Green / A_STANDOUT	Manually marked by the user
Yellow / A_BOLD	BUSY
Green / A_BLINK	CONNECT (modem found)!
Blue / A_UNDERLINE	VOICE
White / A_DIM	NO ANSWER
Magenta / A_NORMAL	Already scanned (loaded from a file)
Cyan / A_REVERSE	Blacklisted number (not called)
Red / A_NORMAL	Number skipped by the user (spacebar)
Blue / A_STANDOUT	Possible interesting number!
Cyan / A_UNDERLINE	Paused, then marked (IAX2 mode only)

The idea iWar uses behind the color coding is that, at a "glance," you can get an idea of what has been located.

iWar with VoIP

Up to now, we've talked about using iWar with physical hardware (a modem, a VoIP adapter, and Asterisk). iWar does contain some VoIP (IAX2) functionality. According to the projects Web page, it's the "first war dialer with VoIP functionality."

We used iWar via good old-fashioned serial because the VoIP detection engine is still under development. That is, in VoIP mode, iWar won't be able to detect modems, fax machines, and other equipment. It simply operates as a VoIP client. With a headset, you can let iWar do the dialing and even chat with people you call through it. According to the iWar mailing list, the addition of SIP and a detection engine is in the works. "Proof of concept" code has been chatted about on the mailing list for some time, but hasn't been included. While it is interesting to let iWar do your dialing and act as a VoIP client, you manually have to identify interesting numbers. Until the detection engine matures, the more practical way to war dial is to

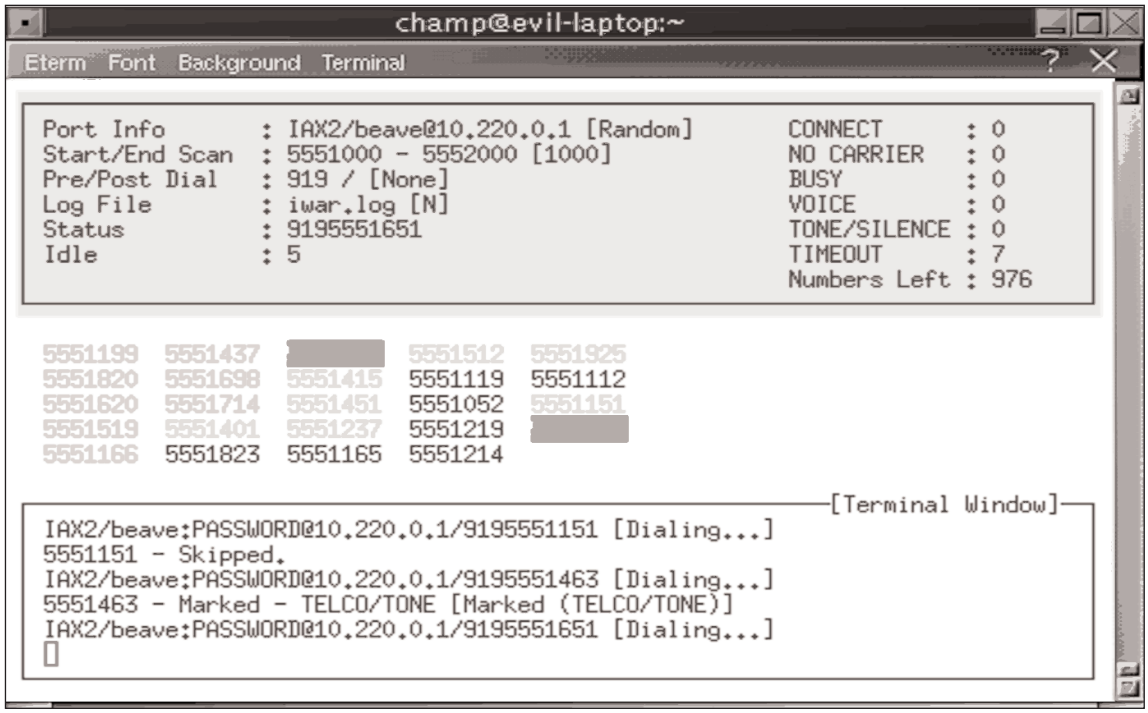
use a traditional modem. The detection engine should be added and released within the next couple of revisions of the code.

If you do wish to bypass the hardware way of scanning and have compiled iWar with IAX2 functionality, you can start iWar in IAX2 mode by passing the *—iax2* flag. For example:

```
$ iwar -npa 904 -nxx 555 -range 1000-1000 -iax2
```

Once started, the iWar curses screen will change a little bit since we are not using a traditional analog modem. It should look something like Figure 6.2.

Figure 6.2 The iWar Curses Screen



The color coding works the same as using iWar with a serial/analog modem, but the terminal window now shows the VoIP interaction with your provider or Asterisk. Future versions of iWar promise to be able to detect over VoIP the same things that a traditional modem can, and more.

All Modems Are Not Alike

Most people believe that all modems are created equally. This isn't the case. Some modems serve their basic function: to connect to another modem. Other modems are "smarter," and the smarter the modem, the better the results of your war dial. Most off-the-shelf modems will connect to other modems, but only detect things like BUSY, NO DIALTONE, and other trivial items, while smarter modems can detect remote RINGING and VOICE. Smarter modems will also speed up your scanning

If you're interested in scanning for fax machines and modem carriers, you'll probably have to make two sweeps: one to search for faxes, the other to search for modem carriers. Not many modems can do both within a single sweep.

If you are doing a serious security audit by war dialing, test the capabilities of your modem before throwing it into a war-dialing challenge.

The author of *iWar* (Da Beave) suggests the U.S. Robotics Courier (V.Everything) for the best results. You can typically find these types of modems on eBay for around $10 to $25. You can also use iWar with multiple modems to speed up your scanning.

Legalities and Tips

As stressed earlier in this chapter get *prior permission* before doing a war-dialing attack. Also, check your state's laws about war dialing. It might be that war dialing, even with prior permission, isn't legal within your state.

If you have prior consent to target a company and scan for rogue telephony-related devices and there are no legalities in your area regarding war dialing, it doesn't mean you can fire off a 10,000 number scan. Many VoIP providers have a clause against this in their terms of service. You'll want to scan "slow and low"—that is, instead of dialing 10,000 numbers at once, dial 50 numbers and wait for a while.

Timing is also an issue. Know the area you are dialing. For example, if the target is a state government agency, dialing in the evening will probably be better. Also, many state agencies have entire exchanges dedicated to them. This way, by dialing during the evening, you won't upset people at work. If it's a business that's located in a business district, the same applies. However, if the business is located in a suburban neighborhood, you'll probably want to war dial during the afternoon. The idea is that most people won't be at home, because they'll be at work. If you dialed the same exchange/area during the day, you'd likely upset many people.

War dialing is about looking for interesting things, not annoying people.

What You Can Find

There are literally thousands of different types of devices connected to the telephone network. RoguePCs with PC Anywhere installed, Xyplex terminal servers, OpenVMS clusters, SCO Unix machines, Linux machines, telco test equipment... The list goes on and on. Some require a type of authentication, while other hardware will let you in simply because you dialed the right number—that is, right into a network that might be guarded with thousands of dialers of firewalls and Intrusion Prevention Systems (IDSs).

Summary

Interfacing Asterisk with hardware can take some creativity. In these simple examples, we're using good old-fashioned serial communications. Serial is used quite a bit, but it's only one means to connect to hardware. The hardware you might want to connect to and write an interface for Asterisk might be connected by USB or something you probe over a TCP/IP network. The core ideas are still the same. Connect to the hardware, send a command if needed, and format the output so it can be used with Asterisk. Based on the information supplied by the device, an action can be taken, if needed. The AGI and functionality it will carry out is up to you. These examples use perl (Practical Extraction and Report Language) since it is a common and well-documented language. As the name implies, we are using it to "extract" information from the remote devices. perl also has some modules that assist in working with Asterisk (Asterisk::AGI), but just about any language can be used.

Solutions Fast Track

Serial

- ☑ Serial communications are simple and well documented. Many devices use serial to interface with hardware.
- ☑ One-way communications is data that is fed to us. Examples of this are things like magnetic card readers.
- ☑ Two-way communications require that a command be sent to the device beforewe can get a response. Examples of this are some environmental control systems and robotics.
- ☑ Serial is used as the basic example, but the same ideas apply with other communications protocols such as IR (Infrared). USB and TCP (for example, telnet) controlled equipment.

Motion

- ☑ Motion is a very powerful tool used to monitor camera(s) and detect events you might be concerned about. For example, if someone breaks into your home.

- ☑ One AGI/routine is used in conjunction with Motion and can be used to notify you if an event occurs.
- ☑ The routine will build the necessary "call files" and alert you by telephone if something was detected.

Modems

- ☑ Traditional analog modems are still used in point-of-sales equipment, TiVos, and other equipment that connect to the PSTN.
- ☑ Connections with a modem can be accomplished, but typically at lower speeds.
- ☑ You'll need to use noncompressed codecs, and the better the network connection, the better your modem connections will be.
- ☑ There are TCP/IP protocols for Fax over IP (T.38) and modems (ITU V.150.1). Unfortunately, ITU V.150.1 (also known as V.MOIP) isn't well supported.
- ☑ Using VoIP during security audits, you can mask where you are coming from. The data traditionally passed over the PSTN isn't passed over VoIP networks.
- ☑ VoIP can also be used to look up phone number information. This is known as backspoofing.
- ☑ When using a traditional modem and VoIP for scanning/war dialing, realize that not all modems are created equal. Some are better than others.

Legalities and Tips

- ☑ Before doing a security audit via VoIP and war dialing, check your local and state laws!
- ☑ Always get prior permission from the target before starting a security audit via war dialing.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Using VoIP with point-of-sales modem equipment is sort of dangerous isn't it?

A: It can be. Before hooking up anything, you should first see what sort of data is being sent. Odds are, it's something you wouldn't want leaked out. Proper security measures should be in place before attaching such equipment to any VoIP network (encryption, VLANs, and so on).

Q: Seriously, what *can* you find via war dialing?

A: Many people and companies would be surprised. Often, the organization being targeted doesn't even know they have hardware that is connected to the PSTN. I've seen routers, dial-up servers, SCO machines, OpenVMS servers, rogue PC Anywhere installs, Linux machines, and much, much more. Some require authentications, while others simply let you into the network, bypassing thousands of dollars of network monitoring equipment.

Q: In the example with Motion, you use it in an environment where motion should not be detected. I'd like to use Motion outside to detect if people come to my front door, walk down my driveway, and so on. Can this be done?

A: Yes. With motion you can create "mask" files that will ignore motion from certain areas of the image. For instance, you can create a mask to ignore a tree in your front yard when the wind blows, but alert you when motion is detected on a walkway.

Chapter 7

Threats to VoIP Communications Systems

Solutions in this chapter:

- **Denial-of-Service or VoIP Service Disruption**
- **Call Hijacking and Interception**
- **H.323-Specific Attacks**
- **SIP-Specific Attacks**

Introduction

Converging voice and data on the same wire, regardless of the protocols used, ups the ante for network security engineers and managers. One consequence of this convergence is that in the event of a major network attack, the organization's entire telecommunications infrastructure can be at risk. Securing the whole VoIP infrastructure requires planning, analysis, and detailed knowledge about the specifics of the implementation you choose to use.

Table 7.1 describes the general levels that can be attacked in a VoIP infrastructure.

Table 7.1 VoIP Vulnerabilities

Vulnerability	Description
IP infrastructure	Vulnerabilities on related non-VoIP systems can lead to compromise of VoIP infrastructure.
Underlying operating system	VoIP devices inherit the same vulnerabilities as the operating system or firmware they run on. Operating systems are Windows and Linux.
Configuration	In their default configuration most VoIP devices ship with a surfeit of open services. The default services running on the open ports may be vulnerable to DoS attacks, buffer overflows, or authentication bypass.
Application level	Immature technologies can be attacked to disrupt or manipulate service. Legacy applications (DNS, for example) have known problems.

Denial-of-Service or VoIP Service Disruption

Denial-of-service (DoS) attacks can affect any IP-based network service. The impact of a DoS attack can range from mild service degradation to complete loss of service. There are several classes of DoS attacks. One type of attack in which packets can simply be flooded into or at the target network from multiple external sources is called a distributed denial-of-service (DDoS) attack (see Figures 7.1 and 7.2).

In this figure, traffic flows normally between internal and external hosts and servers. In Figure 7.2, a network of computers (e.g., a botnet) directs IP traffic at the interface of the firewall.

Figure 7.1 Typical Internet Access

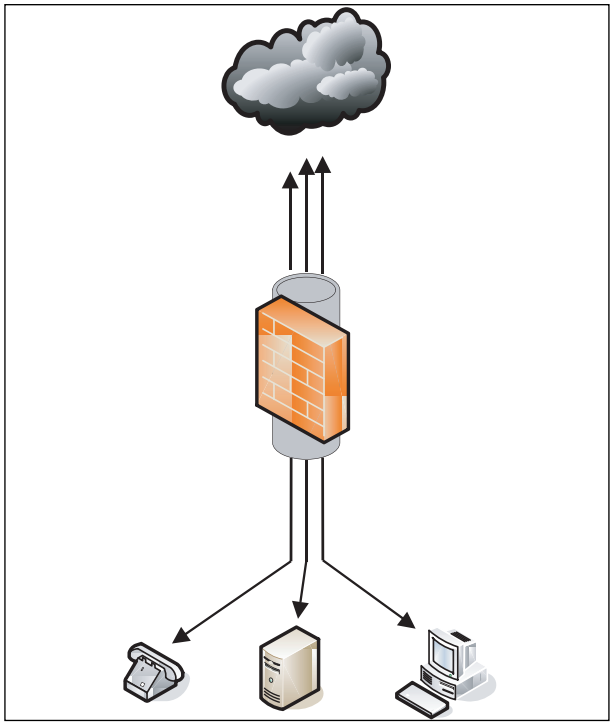
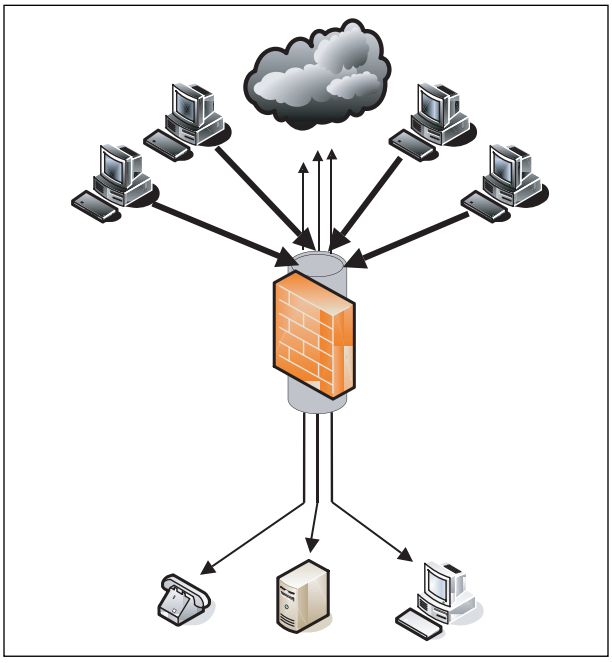
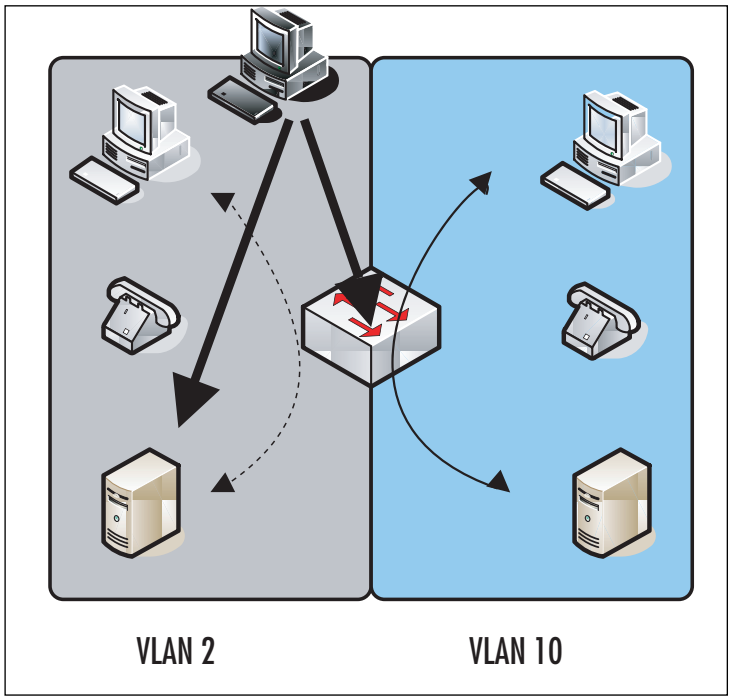


Figure 7.2 A Distributed Denial-of-Service Attack



The second large class of Denial of Service (DoS) conditions occurs when devices within the internal network are targeted by a flood of packets so that they fail—taking out related parts of the infrastructure with them. As in the DdoS scenarios described earlier in this chapter, service disruption occurs to resource depletion—primarily bandwidth and CPU resource starvation (see Figure 7.3). For example, some IP telephones will stop working if they receive a UDP packet larger than 65534 bytes on port 5060.

Figure 7.3 An Internal Denial-of-Service Attack



Neither integrity checks nor encryption can prevent these attacks. DoS or DDoS attacks are characterized simply by the volume of packets sent toward the victim computer; whether those packets are signed by a server, contain real or spoofed source IP addresses, or are encrypted with a fictitious key—none of these are relevant to the attack.

DoS attacks are difficult to defend against, and because VoIP is just another IP network service, it is just as susceptible to DoS attack as any other IP network services. Additionally, DoS attacks are particularly effective against services such as VoIP and other real-time services, because these services are most sensitive to adverse net-

work status. Viruses and worms are included in this category as they often cause DoS or DDoS due to the increased network traffic that they generate as part of their efforts to replicate and propagate.

How do we defend against these DoS conditions (we won't use the term attack here because some DoS conditions are simply the unintended result of other unrelated actions)? Let's begin with internal DoS. Note in Figure 7.3 that VLAN 10 on the right is not affected by the service disruption on the left in VLAN 2. This illustrates one critical weapon the security administrator has in thwarting DoS conditions—logical segregation of network domains in separate compartments. Each compartment can be configured to be relatively immune to the results of DoS in the others. This is described in more detail in Chapter 8.

Point solutions will also be effective in limiting the consequences of DoS conditions. For example, because strong authentication is seldom used in VoIP environments, the message processing components must trust and process messages from possible attackers. The additional processing of bogus messages exhausts server resources and leads to a DoS. SIP or H.323 Registration Flooding is an example of this, described in the list of DoS threats, later. In that case, message processing servers can mitigate this specific threat by limiting the number of registrations it will accept per minute for a particular address (and/or from a specific IP address). An intrusion prevention system (IPS) may be useful in fending off certain types of DoS attacks. These devices sit on the datapath and monitor passing traffic. When anomalous traffic is detected (either by matching against a database of attack signatures or by matching the results of an anomaly-detection algorithm) the IPS blocks the suspicious traffic. One problem I have seen with these devices—particularly in environments with high availability requirements—is that they sometimes block normal traffic, thus creating their own type of DoS.

Additionally, security administrators can minimize the chances of DoS by ensuring that IP telephones and servers are updated to the latest stable version and release. Typically, when a DoS warning is announced by bugtraq, the vendor quickly responds by fixing the offending software.

NOTE

VoIP endpoints can be infected with new VoIP device or protocol-specific viruses. WinCE, PalmOS, SymbianOS, and POSIX-based softphones are especially vulnerable because they typically do not run antivirus software and have less robust operating systems. Several Symbian worms already have

been detected in the wild. Infected VoIP devices then create a new "weak link" vector for attacking other network resources.

Compromised devices can be used to launch attacks against other systems in the same network, particularly if the compromised device is trusted (i.e., inside the firewall). Malicious programs installed by an attacker on compromised devices can capture user input, capture traffic, and relay user data over a "back channel" to the attacker. This is especially worrisome for softphone users.

VoIP systems must meet stringent service availability requirements. Following are some example DoS threats can cause the VoIP service to be partially or entirely unavailable by preventing successful call placement (including emergency/911), disconnecting existing calls, or preventing use of related services like voicemail. Note that this list is not exhaustive but illustrates some attack scenarios.

- **TLS Connection Reset** It's not hard to force a connection reset on a TLS connection (often used for signaling security between phones and gateways)—just send the right kind of junk packet and the TLS connection will be reset, interrupting the signaling channel between the phone and call server.
- **VoIP Packet Replay Attack** Capture and resend out-of-sequence VoIP packets (e.g., RTP SSRC—SSRC is an RTP header field that stands for Synchronization Source) to endpoints, adding delay to call in progress and degrading call quality.
- **Data Tunneling** Not exactly an attack; rather tunneling data through voice calls creates, essentially, a new form of unauthorized modem. By transporting modem signals through a packet network by using pulse code modulation (PCM) encoded packets or by residing within header information, VoIP can be used to support a modem call over an IP network. This technique may be used to bypass or undermine a desktop modem policy and hide the existence of unauthorized data connections. This is similar in concept to the so-called "IP over HTTP" threat (i.e., "Firewall Enhancement Protocol" RFC 3093)—a classic problem for any ports opened on a firewall from internal sources.

- **QoS Modification Attack** Modify non-VoIP-specific protocol control information fields in VoIP data packets to and from endpoints to degrade or deny voice service. For example, if an attacker were to change 802.1Q VLAN tag or IP packet ToS bits, either as a man-in-the-middle or by compromising endpoint device configuration, the attacker could disrupt the quality of service "engineered" for a VoIP network. By subordinating voice traffic to data traffic, for example, the attacker might substantially delay delivery of voice packets.
- **VoIP Packet Injection** Send forged VoIP packets to endpoints, injecting speech or noise or gaps into active call. For example, when RTP is used without authentication of RTCP packets (and without SSRC sampling), an attacker can inject RTCP packets into a multicast group, each with a different SSRC, which can grow the group size exponentially.
- **DoS against Supplementary Services** Initiate a DoS attack against other network services upon which the VoIP service depends (e.g., DHCP, DNS, BOOTP). For example, in networks where VoIP endpoints rely on DHCP-assigned addresses, disabling the DHCP server prevents endpoints (soft- and hardphones) from acquiring addressing and routing information they need to make use of the VoIP service.
- **Control Packet Flood** Flood VoIP servers or endpoints with unauthenticated call control packets, (e.g., H.323 GRQ, RRQ, URQ packets sent to UDP/1719). The attacker's intent is to deplete/exhaust device, system, or network resources to the extent that VoIP service is unusable. Any open administrative and maintenance port on call processing and VoIP-related servers can be a target for this DoS attack.
- **Wireless DoS** Initiate a DoS attack against wireless VoIP endpoints by sending 802.11 or 802.1X frames that cause network disconnection (e.g., 802.11 Deauthenticate flood, 802.1X EAP-Failure, WPA MIC attack, radio spectrum jamming). For example, a Message Integrity Code attack exploits a standard countermeasure whereby a wireless access point disassociates stations when it receives two invalid frames within 60 seconds, causing loss of network connectivity for 60 seconds. In a VoIP environment, a 60-second service interruption is rather extreme.

- **Bogus Message DoS** Send VoIP servers or endpoints valid-but-forged VoIP protocol packets to cause call disconnection or busy condition (e.g., RTP SSRC collision, forged RTCP BYE, forged CCMS, spoofed endpoint button push). Such attacks cause the phone to process a bogus message and incorrectly terminate a call, or mislead a calling party into believing the called party's line is busy.
- **Invalid Packet DoS** Send VoIP servers or endpoints invalid packets that exploit device OS and TCP/IP implementation denial-of-service CVEs. For example, the exploit described in CAN-2002-0880 crashes Cisco IP phones using jolt, jolt2, and other common fragmentation-based DoS attack methods. CAN-2002-0835 crashes certain VoIP phones by exploiting DHCP DoS CVEs. Avaya IP phones may be vulnerable to port zero attacks.
- **Immature Software DoS** PDA/handheld softphones and first generation VoIP hardphones are especially vulnerable because they are not as mature or intensely scrutinized. VoIP call servers and IP PBXs also run on OS platforms with many known CVEs. Any open administrative/maintenance port (e.g., HTTP, SNMP, Telnet) or vulnerable interface (e.g., XML, Java) can become an attack vector.
- **VoIP Protocol Implementation DoS** Send VoIP servers or endpoints invalid packets to exploit a VoIP protocol implementation vulnerability to a DoS attack. Several such exploits are identified in the MITRE CVE database (http://cve.mitre.org). For example, CVE-2001-00546 uses malformed H.323 packets to exploit Windows ISA memory leak and exhaust resources. CAN-2004-0056 uses malformed H.323 packets to exploit Nortel BCM DoS vulnerabilities. Lax software update practices (failure to install CVE patches) exacerbate risk.
- **Packet of Death DoS** Flood VoIP servers or endpoints with random TCP, UDP, or ICMP packets or fragments to exhaust device CPU, bandwidth, TCP sessions, and so on. For example, an attacker can initiate a TCP Out of Band DoS attack by sending a large volume of TCP packets marked "priority delivery" (the TCP Urgent flag). During any flood, increased processing load interferes with the receiving system's ability to process real traffic, initially delaying voice traffic processing but ultimately disrupting service entirely.

- **IP Phone Flood DoS** Send a very large volume of call data toward a single VoIP endpoint to exhaust that device's CPU, bandwidth, TCP sessions, and so on. Interactive voice response systems, telephony gateways, conferencing servers, and voicemail systems are able to generate more call data than a single endpoint can handle and so could be leveraged to flood an endpoint.

Call Hijacking and Interception

Call interception and eavesdropping are other major concerns on VoIP networks. The VOIPSA threat taxonomy (www.voipsa.org/Activities/taxonomy-wiki.php) defines eavesdropping as "a method by which an attacker is able to monitor the entire signaling and/or data stream between two or more VoIP endpoints, but cannot or does not alter the data itself." Successful call interception is akin to wiretapping in that conversations of others can be stolen, recorded, and replayed without their knowledge. Obviously, an attacker who can intercept and store these data can make use of the data in other ways as well.

Tools & Traps…

DNS Poisoning

A DNS A (or address) record is used for storing a domain or hostname mapping to an IP address. SIP makes extensive use of SRV records to locate SIP services such as SIP proxies and registrars. SRV (service) records normally begin with an underscore (_sip.tcpserver.udp.domain.com) and consist of information describing service, transport, host, and other information. SRV records allow administrators to use several servers for a single domain, to move services from host to host with little fuss, and to designate some hosts as primary servers for a service and others as backups.

An attacker's goal, when attempting a DNS Poisoning or spoofing attack, is to replace valid cached DNS A, SRV, or NS records with records that point to the attacker's server(s). This can be accomplished in a number of fairly trivial ways—the easiest being to initiate a zone transfer from the attacker's DNS server to the victim's misconfigured DNS server, by asking the victim's DNS server to resolve a networked device within the attacker's domain. The victim's

Continued

DNS server accepts not only the requested record from the attacker's server, but it also accepts and caches any other records that the attacker's server includes.

Thus, in addition to the A record for www.attacker.com, the victim DNS server may receive a bogus record for www.yourbank.com. The innocent victim will then be redirected to the attacker.com Web site anytime he or she attempts to browse to the yourbank.com Web site, as long as the bogus records are cached. Substitute a SIP URL for a Web site address, and the same scenario can be repeated in a VoIP environment.

This family of threats relies on the absence of cryptographic assurance of a request's originator. Attacks in this category seek to compromise the message integrity of a conversation. This threat demonstrates the need for security services that enable entities to authenticate the originators of requests and to verify that the contents of the message and control streams have not been altered in transit.

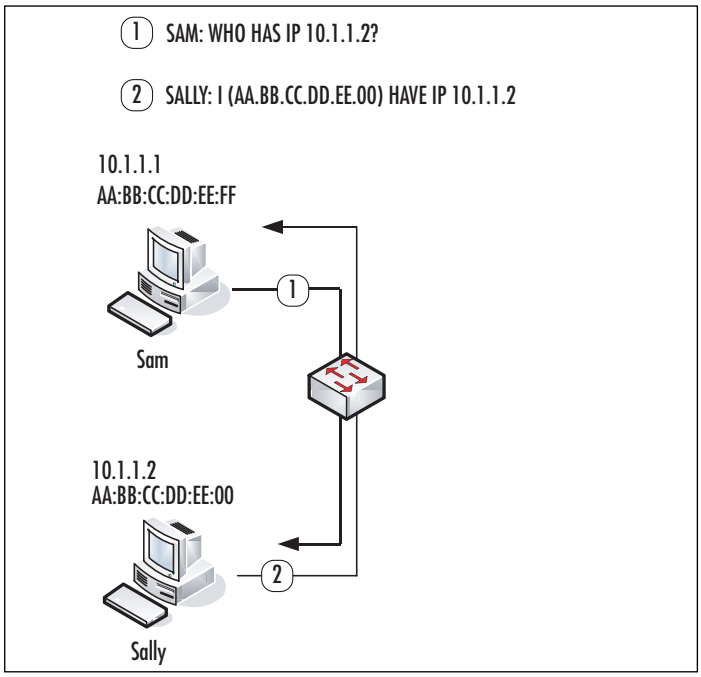
In the past several years, as host PCs have improved their processing power and their ability to process networked information, network administrators have instituted a hierarchical access structure that consists of a single, dedicated switched link for each host PC to distribution or backbone devices. Each networked user benefits from a more reliable, secure connection with guaranteed bandwidth. The use of a switched infrastructure limits the effectiveness of packet capture tools or protocol analyzers as a means to collect VoIP traffic streams. Networks that are switched to the desktop allow normal users' computers to monitor only broadcast and unicast traffic that is destined to their particular MAC address. A user's NIC (network interface card) literally does not see unicast traffic destined for other computers on the network.

The address resolution protocol (ARP) is a method used on IPv4 Ethernet networks to map the IP address (layer 3) to the hardware or MAC (Media Access Control) layer 2 address. (Note that ARP has been replaced in IPv6 by Neighbor Discovery [ND] protocol. The ND protocol is a hybrid of ARP and ICMP.) Two classes of hardware addresses exist: the broadcast address of all ones, and a unique 6 byte identifier that is burned into the PROM of every NIC (Network Interface Card).

Figure 7.4 illustrates a typical ARP address resolution scheme. A host PC (10.1.1.1) that wishes to contact another host (10.1.1.2) on the same subnet issues an ARP broadcast packet (ARPs for the host) containing its own hardware and IP addresses. NICs contain filters that allow them to drop all packets not destined for their unique hardware address or the broadcast address, so all NICs but the query target silently discard the ARP broadcast. The target NIC responds to the query

request by unicasting its IP and hardware address, completing the physical to logical mapping, and allowing communications to proceed at layer 3.

Figure 7.4 Typical ARP Request/Reply



To minimize broadcast traffic, many devices cache ARP addresses for a varying amount of time: The default ARP cache timeout for Linux is one minute; for Windows NT, two minutes, and for Cisco routers, four hours. This value can be trivially modified in most systems. The ARP cache is a table structure that contains IP address, hardware address, and oftentimes, the name of the interface the MAC address is discovered on, the type of media, and the type of ARP response. Depending upon the operating system, the ARP cache may or may not contain an entry for its own addresses.

In Figure 7.4, Sam's ARP cache contains one entry prior to the ARP request/response:

Internet Address	Physical Address	
10.1.1.1	AA:BB:CC:DD:EE:FF	int0

After the ARP request/response completes, Sam's ARP cache now contains two entries:

Internet Address	Physical Address	
10.1.1.1	AA:BB:CC:DD:EE:FF	int0
10.1.1.2	AA:BB:CC:DD:EE:00	int0

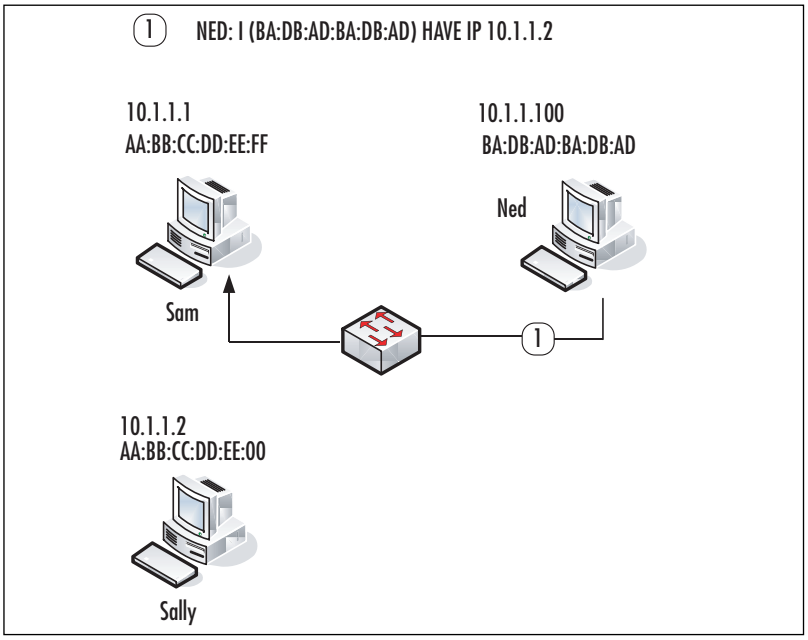
Note that Sally's ARP cache, as a result of the request/response communications, is updated with the hardware:IP mappings for both workstations as well.

ARP Spoofing

ARP is a fundamental Ethernet protocol. Perhaps for this reason, manipulation of ARP packets is a potent and frequent attack mechanism on VoIP networks. Most network administrators assume that deploying a fully switched network to the desktop prevents the ability of network users to sniff network traffic and potentially capture sensitive information traversing the network. Unfortunately, several techniques and tools exist that allow any user to sniff traffic on a switched network because ARP has no provision for authenticating queries or query replies. Additionally, because ARP is a stateless protocol, most operating systems (Solaris is an exception) update their cache when receiving ARP reply, regardless of whether they have sent out an actual request.

Among these techniques, ARP redirection, ARP spoofing, ARP hijacking, and ARP cache poisoning are related methods for disrupting the normal ARP process. These terms frequently are interchanged and confused. For the purpose of this section, we'll refer to ARP cache poisoning and ARP spoofing as the same process. Using freely available tools such as ettercap, Cain, and dsniff, an evil IP device can spoof a normal IP device by sending unsolicited ARP replies to a target host. The bogus ARP reply contains the hardware address of the normal device and the IP address of the malicious device. This "poisons" the host's ARP cache (see Figure 7.5).

Figure 7.5 ARP Spoofing (Cache Poisoning)



In Figure 7.5, Ned is the attacking computer. When SAM broadcasts an ARP query for Sally's IP address, Ned, the attacker, responds to the query stating that the IP address (10.1.1.2) belongs to Ned's MAC address, BA:DB:AD:BA:DB:AD. Packets sent from Sam supposedly to Sally will be sent to Ned instead. Sam will mistakenly assume that Ned's MAC address corresponds to Sally's IP address and will direct all traffic destined for that IP address to Ned's MAC. In fact, Ned can poison Sam's ARP cache without waiting for an ARP query since on Windows systems (9x/NT/2K), static ARP entries are overwritten whenever a query response is received regardless of whether or not a query was issued.

Sam's ARP cache now looks like this:

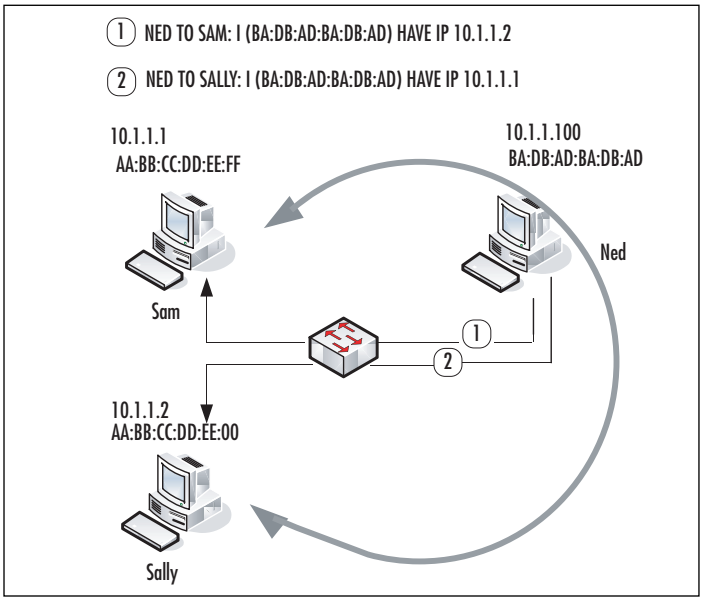
Internet Address	Physical Address	
10.1.1.1	AA:BB:CC:DD:EE:FF	int0
10.1.1.2	BA:DB:AD:BA:DB:AD	int0

This entry will remain until it ages out or a new entry replaces it.

ARP redirection can work bidirectionally, and a spoofing device can insert itself in the middle of a conversation between two IP devices on a switched network (see Figure 7.6). This is probably the most insidious ARP-related attack. By routing

packets on to the devices that should truly be receiving the packets, this insertion (known as a Man/Monkey/Moron in the Middle attack) can remain undetected for some time. An attacker can route packets to /dev/null (nowhere) as well, resulting in a DoS attack.

Figure 7.6 An ARP MITM Attack



Sam's ARP cache:

Internet Address	Physical Address	
10.1.1.1	AA:BB:CC:DD:EE:FF	int0
10.1.1.2	BA:DB:AD:BA:DB:AD	int0

Sally's ARP cache:

Internet Address	Physical Address	
10.1.1.1	BA:DB:AD:BA:DB:AD	int0
10.1.1.2	AA:BB:CC:DD:EE:00	int0

As all IP traffic between the true sender and receiver now passes through the attacker's device, it is trivial for the attacker to sniff that traffic using freely available tools such as Ethereal or tcpdump. Any unencrypted information (including e-mails, usernames and passwords, and web traffic) can be intercepted and viewed.

This interception has potentially drastic implications for VoIP traffic. Freely available tools such as vomit and rtpsniff, as well as private tools such as VoipCrack, allow for the interception and decoding of VoIP traffic. Captured content can include speech, signaling and billing information, multimedia, and PIN numbers. Voice conversations traversing the internal IP network can be intercepted and recorded using this technique.

There are a number of variations of the aforementioned techniques. Instead of imitating a host, the attacker can emulate a gateway. This enables the attacker to intercept numerous packet streams. However, most ARP redirection techniques rely on stealth. The attacker in these scenarios hopes to remain undetected by the users being impersonated. Posing as a gateway may result in alerting users to the attacker's presence due to unanticipated glitches in the network, because frequently switches behave in unexpected ways when attackers manipulate ARP processes. One unintended (much of the time) consequence of these attacks, particularly when switches are heavily loaded, is that the switch CAM (Content-Addressable Memory) table—a finite-sized IP address to MAC address lookup table—becomes disrupted. This leads to the switch forwarding unicast packets out many ports in unpredictable fashion. Penetration testers may want to keep this in mind when using these techniques on production networks.

In order to limit damage due to ARP manipulation, administrators should implement software tools that monitor MAC to IP address mappings. The freeware tool, Arpwatch, monitors these pairings. At the network level, MAC/IP address mappings can be statically coded on the switch; however, this is often administratively untenable. Dynamic ARP Inspection (DAI) is available on newer Cisco Catalyst 6500 switches. DAI is part of Cisco's Integrated Security (CIS) functionality and is designed to prevent several layer two and layer three spoofing attacks, including ARP redirection attacks. Note that DAI and CIS are available only on Catalyst switches using native mode (Cisco IOS).

The potential risks of decoding intercepted VoIP traffic can be eliminated by implementing encryption. Avaya's Media Encryption feature is an example of this. Using Media Encryption, VoIP conversations between two IP endpoints are encrypted using AES encryption. In highly secure environments, organizations should ensure that Media Encryption is enabled on all IP codec sets in use.

DAI enforces authorized MAC-to-IP address mappings. Media Encryption renders traffic, even if intercepted, unintelligible to an attacker.

The following are some additional examples of call or signal interception and hijacking. This class of threats, though typically more difficult to accomplish than DoS, can result in significant loss or alteration of data. DoS attacks, whether caused by active methods or inadvertently, although important in terms of quality of service, are more often than not irritating to users and administrators. Interception and hijacking attacks, on the other hand, are almost always active attacks with theft of service, information, or money as the goal. Note that this list is not exhaustive but illustrates some attack scenarios.

- **Rogue VoIP Endpoint Attack** Rogue IP endpoint contacts VoIP server by leveraging stolen or guessed identities, credentials, and network access. For example, a rogue endpoint can use an unprotected wall jack and auto-registration of VOIP phones to get onto the network. RAS password guessing can be used to masquerade as a legitimate endpoint. Lax account maintenance (expired user accounts left active) increases risk of exploitation.
- **Registration Hijacking** Registration hijacking occurs when an attacker impersonates a valid UA to a registrar and replaces the registration with its own address. This attack causes all incoming calls to be sent to the attacker.
- **Proxy Impersonation** Proxy impersonation occurs when an attacker tricks a SIP UA or proxy into communicating with a rogue proxy. If an attacker successfully impersonates a proxy, he or she has access to all SIP messages.
- **Toll Fraud** Rogue or legitimate VoIP endpoint uses a VoIP server to place unauthorized toll calls over the PSTN. For example, inadequate access controls can let rogue devices place toll calls by sending VoIP requests to call processing applications. VoIP servers can be hacked into in order to make free calls to outside destinations. Social engineering can be used to obtain outside line prefixes.
- **Message Tampering** Capture, modify, and relay unauthenticated VoIP packets to/from endpoints. For example, a rogue 802.11 AP can exchange frames sent or received by wireless endpoints if no payload integrity check (e.g., WPA MIC, SRTP) is used. Alternatively, these attacks can occur through registration hijacking, proxy impersonation, or an attack on any component trusted to process SIP or H.323 messages, such as the proxy, registration servers, media gateways, or firewalls. These represent non-ARP-based MITM attacks.

- **VoIP Protocol Implementation Attacks** Send VoIP servers or endpoints invalid packets to exploit VoIP protocol implementation CVEs. Such attacks can lead to escalation of privileges, installation and operation of malicious programs, and system compromise. For example, CAN-2004-0054 exploits Cisco IOS H.323 implementation CVEs to execute arbitrary code. CSCed33037 uses unsecured IBM Director agent ports to gain administrative control over IBM servers running Cisco VoIP products.

Notes from the Underground…

ANI/Caller-ID Spoofing

Caller ID is a service provided by most telephone companies (for a monthly cost) that will tell you the name and number of an incoming call. Automatic Number Identification (ANI) is a system used by the telephone company to determine the number of the calling party. To spoof Caller-ID, an attacker sends modem tones over a POTS lines between rings 1 and 2. ANI spoofing is setting the ANI so as to send incorrect ANI information to the PSTN so that the resulting Caller-ID is misleading. Traditionally this has been a complicated process either requiring the assistance of a cooperative phone company operator or an expensive company PBX system.

In ANI/Caller-ID spoofing, an evildoer hijacks phone number and the identity of a trusted party, such as a bank or a government office. The identity appears on the caller ID box of an unsuspecting victim, with the caller hoping to co-opt valuable information, such as account numbers, or otherwise engage in malicious mischief. This is not a VoIP issue, per se. In fact, one of the big drawbacks about VoIP trunks is their inability to send ANI properly because of incomplete standards.

H.323-Specific Attacks

The only existing vulnerabilities that we are aware of at this time take advantage of ASN.1 parsing defects in the first phase of H.225 data exchange. More vulnerabilities can be expected for several reasons: the large number of differing vendor implementations, the complex nature of this collection of protocols, problems with the various implementations of ASN.1/PER encoding/decoding, and the fact that

these protocols—alone and in concert—have not endured the same level of scrutiny that other more common protocols have been subjected to. For example, we have unpublished data that shows that flooding a gateway or media server with GRQ request packets (RAS registration request packets) results in a DoS against certain vendor gateway implementations—basically the phones deregister.

SIP-Specific Attacks

Multiple vendors have confirmed vulnerabilities in their respective SIP (Session Initiation Protocol) implementations. The vulnerabilities have been identified in the INVITE message used by two SIP endpoints during the initial call setup. The impact of successful exploitation of the vulnerabilities has not been disclosed but potentially could result in a compromise of a vulnerable device. (CERT: CA-2003-06.) In addition, many recent examples of SIP Denial of Service attacks have been reported.

Recent issues that affect Cisco SIP Proxy Server (SPS) [Bug ID CSCec31901] demonstrate the problems SIP implementers may experience due to the highly modular architecture or this protocol. The SSL implementation in SPS (used to secure SIP sessions) is vulnerable to an ASN.1 BER decoding error similar to the one described for H.323 and other protocols. This example illustrates a general concern with SIP: As the SIP protocol links existing protocols and services together, all the classic vulnerabilities in services such as SSL, HTTP, and SMTP may resurface in the VoIP environment.

Summary

DoS attacks, whether they are intentional or unintended, are the most difficult VoIP-related threat to defend against. The packet switching nature of data networks allows multiple connections to share the same transport medium. Therefore, unlike telephones in circuit-switched networks, an IP terminal endpoint can receive and potentially participate in multiple calls at once. Thus, an endpoint can be used to amplify attacks. On VoIP networks, resources such as bandwidth must be allocated efficiently and fairly to accommodate the maximum number of callers. This property can be violated by attackers who aggressively and abusively obtain an unnecessarily large amount of resources. Alternatively, the attacker simply can flood the network with large number of packets so that resources are unavailable to all other callers.

In addition, viruses and worms create DoS conditions due to the network traffic generated by these agents as they replicate and seek out other hosts to infect. These agents are proven to wreak havoc with even relatively well-secured data networks. VoIP networks, by their nature, are exquisitely sensitive to these types of attacks. Remedies for DoS include logical network partitioning at layers 2 and 3, stateful firewalls with application inspection capabilities, policy enforcement to limit flooded packets, and out-of-band management. Out-of-band management is required so that in the event of a DoS event, system administrators are still able to monitor the network and respond to additional events.

Theft of services and information is also problematic on VoIP networks. These threats are almost always due to active attack. Many of these attacks can be thwarted by implementing additional security controls at layer 2. This includes layer 2 security features such as DHCP Snooping, Dynamic ARP Inspection, IP Source Guard, Port Security, and VLAN ACLs. The fundamental basis for this class of attacks is that the identity of one or more of the devices that participate is not legitimate.

Endpoints must be authenticated, and end users must be validated in order to ensure legitimacy. Hijacking and call interception revolves around the concept of fooling and manipulating weak or nonexistent authentication measures. We are all familiar with different forms of authentication, from the password used to login to your computer to the key that unlocks the front door. The conceptual framework for authentication is made up of three factors: "something you have" (a key or token), "something you know" (a password or secret handshake), or "something you are" (fingerprint or iris pattern). Authentication mechanisms validate users by one or a combination of these. Any type of unauthenticated access, particularly to

key infrastructure components such as the IP PBX or DNS server, for example, can result in disagreeable consequences for both users and administrators.

VoIP relies upon a number of ancillary services as part of the configuration process, as a means to locate users, manage servers and phones, and to ensure favorable transport, among others. DNS, DHCP, HTTP, HTTPS, SNMP, SSH, RSVP, and TFTP services all have been the subject of successful exploitation by attackers. Potential VoIP users may defer transitioning to IP Telephony if they believe it will reduce overall network security by creating new vulnerabilities that could be used to compromise non-VoIP systems and services within the same network. Effective mitigation of these threats to common data networks and services could be considered a security baseline upon which a successful VoIP deployment depends. Firewalls, network and system intrusion detection, authentication systems, anti-virus scanners, and other security controls, which should already be in place, are required to counter attacks that might debilitate any or all IP-based services (including VoIP services).

H.323 and SIP suffer security vulnerabilities based simply upon their encoding schemes, albeit for different reasons. Because SIP is an unstructured text-based protocol, it is impossibly to test all permutations of SIP messages during development for security vulnerabilities. It's fairly straightforward to construct a malformed SIP message or message sequence that results in a DoS for a particular SIP device. This may not be significant for a single UA endpoint, but if this "packet of death" can render all the carrier-class media gateway controllers in a network useless, then this becomes a significant problem. H.323 on the other hand is encoded according to ASN.1 PER encoding rules. The implementation of H.323 message parsers, rather than the encoding rules themselves, results in security vulnerabilities in the H.323 suite.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

T

U

V

W

Z