Pro OpenSSH

Michael Stahnke

Apress®

Pro OpenSSH

ISBN (pbk): 1-59059-476-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Darren Tucker
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Beckie Stones and Laura Brown
Copy Edit Manager: Nicole LeClerc
Copy Editors: Ami Knox and Damon Larson
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

Contents at a Glance

Contents

PART 4 Administration with OpenSSH

About the Author

MICHAEL STAHNKE works as a UNIX Security Administrator at a Fortune 100 company in the Midwest. He has headed implementations of Secure Shell for his corporate IT group and provided consultation and assistance with production rollouts around the globe. Additionally, he has led several studies and projects to improve the security state of his large-scale UNIX/Linux environment, utilizing SSH, mandatory access control, configuration management integration, and automation techniques. When not devoting his time to improving security at work, Michael spends time researching and applying new open source technologies and practices on his ever-changing home network. Michael has also done contract programming to create content management solutions utilizing PHP, Perl, MySQL, and C++. Michael earned a CS degree from Ball State University in 2002. He also recently became a CISSP.

About the Technical Reviewer

DARREN TUCKER is an independent consultant in Sydney, Australia. He has worked on a variety of systems and networks for over 10 years, many of those with OpenSSH and other SSH products. He has been a member of the OpenSSH development team since 2003. He likes cricket and dislikes talking about himself in the third person.

Acknowledgments

I would like to thank a few individuals who helped me make this book a reality. First and foremost, I would like to thank my wife Jaime for putting up with an absent and almost nonexistent husband on several occasions during the authoring process. I would also like to thank John Traenkenschuh for encouraging me to write down my experiences with SSH. Brian Tower also deserves many thanks for allowing me to share my ideas and for helping correct a few oddly worded sentences.

Finally, I would like to thank the open source community for so many projects that I use everyday; but I especially want to thank the OpenSSH development team, including my technical reviewer Darren Tucker, for delivering a high-quality, secure connectivity solution at a price that everyone can afford.

Introduction

What Is SSH?

In 1995, Helsinki University of Technology researcher Tatu Ylönen learned that system passwords were being retrieved through network monitoring, resulting in a compromise of the university network. Simple tools allowed adversaries of the university to gain access to several account names and passwords. Mr. Ylönen responded by creating the Secure Shell (SSH), a security solution intended to resolve the deficiencies of legacy protocols by encrypting data, account names, and passwords while they are in transit, thus rendering network sniffing useless.

Mr. Ylönen's creation resolved many issues that had haunted legacy protocols for years. The initial release of SSH quickly became popular at his university and throughout the Internet. Users from around the globe were soon asking for copies of the software, support on that software, and of course new features. Later in 1995, he founded SSH Communications Security (`http://www.ssh.com`). With this new tool, security/system administrators could replace legacy system access applications such as `rsh`, `rlogin`, `rcp`, Telnet, and FTP with secure alternatives. SSH could also provide solutions for many common user-friendliness issues—for example, through simplifying passwords by moving to digital credentials in the form of public keys.

According to `http://www.openssh.org`, Mr. Ylönen released his SSH originally as Free SSH, but it contained a few proprietary libraries and licenses that prevented it from being widely adopted by major projects. As the Free SSH project progressed, newer versions came with more restrictive licenses. Some early licenses attached to Free SSH forbade developers to create Windows or DOS versions. Later licenses restricted the use of Free SSH in commercial operating environments.

In 1999, Björn Grönvall, a Swedish computer scientist, created a new version of SSH called OSSH. When the OpenBSD project became aware of Grönvall's work a few short months before the scheduled release of OpenBSD 2.6, the decision was made to include an SSH implementation in OpenBSD. The OpenBSD community forked OSSH to enable rapid development and control over the project, and continued developing and refining it on a very rigorous schedule—upon release, the product was renamed OpenSSH. Shortly after the release of OpenBSD 2.6, Linux advocates and other UNIX programmers saw the need for SSH on their systems. OpenSSH was then split into two versions: baseline (for OpenBSD systems) and portable (for Linux, UNIX, and other operating systems). OpenSSH is freely usable and redistributable by everyone under a BSD license. It has since been included in nearly every major UNIX and Linux release, and is regularly integrated into embedded devices and appliances.

OpenSSH creates value in a network of any size by protecting data. In the enterprise, UNIX administrators will find using key-based authentication allows them to perform tasks and script as quickly as with `rsh`/`rlogin`, but with added security and logging. Security administrators will be happy to remove legacy clear-text protocols and applications.

Auditors can streamline their workflow as they will only require understanding of one service instead of several. The list of benefits for implementing OpenSSH is long, and this book provides instruction on many of the benefits and best practices.

Note `http://www.openssh.com` is the official home page for OpenSSH. You can find the latest source code, FAQs, credits, history, and goals of the project there.

Who Should Read This Book?

This question has been foremost in my mind since the onset of this project, as my goal is to compile and offer the right information for the right audience. While architecting and deploying OpenSSH implementations for Fortune 100 enterprises and home networks alike, I have referred to SSH resources both online and in print, and I came to the conclusion that most information on the topic is scattered, disorganized, or pretty dated. This presents a problem to most overworked system administrators. Maybe you are looking for an immediate response to a security incident, or perhaps you need practical information you can use now—not security theory or outdated implementation details and obscure encryption trivia—if so, this book is for you.

Maybe you are a stressed system administrator responsible for UNIX security, stuck in endless Sarbanes-Oxley, HIPAA, or compliance audit du jour meetings; inundated with conference calls, procedural reviews, and emails at all hours; and even forced to carry the support pager off-shift. Like you, I needed immediate answers to the problems with the plain-text protocols, and searched to learn concepts and best practices that helped me make the most of OpenSSH. The time you spend weighing and ultimately implementing this transition to SSH or improving your existing SSH environment should not be wasted, given the importance of this security solution. If you share my sentiments, I think you will find this book valuable. Along the way I will draw upon my experience implementing large-scale SSH systems, and provide real-world examples, scenarios, and best practices.

How This Book Is Organized

Secure communication is not a new goal among information technology professionals, but as new topics, tools, and exploits are created, new measures are taken to obstruct the adversaries and provide assurance of risk mitigation to management professionals. This is where the Secure Shell protocol comes into play, and in this book I will discuss OpenSSH, the most popular implementation of that protocol.

If you are system administrator, security professional, or home user of UNIX/Linux, then this book will provide value to you. Chances are if you are picking up this book, you have some idea what OpenSSH is and what you can do with it. Your exposure to OpenSSH certainly will not be a hindrance as you continue to work your way through the book. Whether you have introductory knowledge of OpenSSH or are simply looking to hone your skill set in a certain area, I am confident that by the time you complete this book, you will be able to piece together an effective, secure SSH network solution.

Part 1: Quick and Secure

The first part of the book deals with introductory topics, including reasons for eliminating legacy protocols, how OpenSSH can help you do so, and basic connectivity. Installation and compilation of OpenSSH on Linux is also covered.

Part 2: Configuring OpenSSH

Part 2 of the book is devoted to in-depth analysis of command-line options, configuration files, and settings. These settings become critical when trying to achieve maximum usability while not compromising the secure infrastructure you are striving to achieve. After a detailed explanation of configuration settings, authentication is covered. Authentication in OpenSSH depends upon several complex settings and concepts, as opposed to traditional password authentication. The power of public key authentication will be introduced and explored, as well as host-based authentication, when appropriate.

Part 3: Advanced Topics

In Part 3, the real power of OpenSSH starts to take shape. Tunneling less secure but still useful protocols such as X-Windows, VNC (Virtual Network Computing), and `rsync` is introduced. Port forwarding and tunneling of most generic TCP protocols is covered as well.

Chapter 8 discusses best practices for securing OpenSSH on both the server and client sides. Chapter 8 relies on the knowledge presented throughout the book to build an OpenSSH secure backbone utilizing an administrative gateway as the focal point for administration. Managing hundreds or even thousands of servers in this fashion becomes a less daunting task when the right scripts and security settings are employed.

Part 4: Administration with OpenSSH

OpenSSH can be used for automating nearly every administration task on a UNIX/Linux operating system. Chapter 9 provides an introduction to scripting with OpenSSH, which includes coverage of shell scripting, Perl, and PHP options. These scripts provide a foundation on which administrators can develop their own automation processes using OpenSSH.

Chapter 10 provides a look at the SSH Tectia Server from SSH Communications Security. This popular commercial SSH implementation has some differences from OpenSSH, which are explored in this final chapter. Working a heterogeneous environment is also discussed—for example, when moving back and forth from SSH Tectia Server and OpenSSH, certain incompatibilities will arise. This chapter aims to minimize the impact of those incompatibilities and show you how to manage a complex environment.

The first appendix provides information about several popular Microsoft Windows SSH clients, including PuTTY and WinSCP. The second appendix shows you how to get an SSH server running on a Windows platform.

What's Not in This Book?

Because of my work with OpenSSH security, I am often asked questions that appear to be related to OpenSSH, but in fact are separate security topics altogether. Several security publications and books cover security from a more theoretical standpoint, which can be helpful to

architects and to those interested in learning a concept at its deepest level. This book takes a different route, opting not to linger on the security theory behind OpenSSH, and instead has been designed from its inception to be a book for hard-working administrators trying to keep their proverbial heads above water. That being said, the topics of OpenSSL, IPSec, VPN, PKI, firewalls, and Kerberos are not really covered in this book. There are several wonderful publications that do cover those topics well—however, most of them do not cover OpenSSH in the detail presented in this material. These and other security topics are mentioned when pertinent to the successful implementation of OpenSSH.

Prerequisites

While this book strives to accommodate beginners and experienced professionals alike, a few assumptions are made on your general understanding. The text covers installation and configuration on Linux in the most distribution-agnostic way possible. If you are able to translate commands from Linux onto your operating system of choice, please feel free. Basic command-line knowledge is assumed, including the ability to edit configuration files and to manage daemon execution. Root access or root-equivalent authority is also a must for editing configurations and installing OpenSSH.

Knowledge of Perl and shell scripting is also certainly helpful, but is not required for learning the scripting and administration topics covered in this book.

For most examples, I connect from one system on my network to another. While all of these examples can work with only one computer, having more than one available to you will make the power and ease of OpenSSH more apparent to you.

My Network

The network used in this book is my own home network. The names of my systems are a bit odd, but I will briefly introduce them here because they are referenced throughout the material. Note that I have `/etc/hosts` and DNS configured on my systems so that using a fully qualified domain name is not needed. Examples in this book that require fully qualified domain names will use `example.com`. Table 1 describes my network naming system.

Table 1. *The Network Utilized in Examples Throughout the Book*

System Name	Description
`rack`	My primary workstation, which is running Linux, is named `rack`. It is normally at least a dual-boot system with Fedora and SUSE Linux running on it. This system is named `rack` because it is inside a rack-mount chassis.
`www`	My primary web server is named `www`. It runs a typical LAMP stack (Linux, Apache, MySQL, PHP/Perl/Python). It also runs LDAP and DNS services.
`zoom`	`zoom` is a Debian Linux system.
`mini`	`mini` is the Microsoft Windows XP workstation used in this book. It is in a small form factor case—hence the name `mini`.
`macmini`	`macmini`, amazingly enough, is an Apple Mac mini computer running Mac OS X.

Downloads

The code, configuration files, and scripts used throughout this book are available for download on the Apress website, `http://www.apress.com`, in the Source Code section.

Read On

If you classify yourself as a rookie with OpenSSH, it may be best to start at the beginning of the book. If you have basic knowledge but still want to further your abilities with keys and scripts, Part 2 is great place to start. If you have a good understanding of OpenSSH and are trying to get maximum value and security out of the tool, feel free to jump from section to section or even topic to topic. The goal for the book is to provide you with a comprehensive tool set with which you can do your work securely and efficiently.

If you wish to contact me, I welcome your comments, feedback (for better or worse), updates, and errata. Feel free to email me at `mastahnke@yahoo.com`.

I hope this book will rest near your main administration workstation and be reopened periodically, as so many system administration books I have relied on in the past.

PART 1

Quick and Secure

CHAPTER 1

Legacy Protocols: Why Replace Telnet, FTP, rsh, rcp, and rlogin with SSH?

If you are reading this book, there is a very good chance you might already be aware of some of the advantages of using the Secure Shell (SSH) protocol over traditional UNIX connectivity offerings such as `telnet`, `ftp`, and `rsh`. This chapter will clearly outline those advantages and hopefully enlighten you on a few more positive features of the Secure Shell protocol.

Before getting into a specific discussion about the advantages of the Secure Shell protocol, I will cover several general security-related topics that will allow you to not only better understand the UNIX/Linux secure connectivity principles, but also understand the issues that ultimately spurred the creation of SSH.

Foundations of Information Security

To understand how effective SSH is in terms of security, a discussion of how information security is measured is warranted.

According to the International Information System Security Certification Consortium, or $(ISC)^2$ (`http://www.isc2.org`), security has three main goals, namely *confidentiality*, *integrity*, and *availability*.

Confidentiality is gained through access control, or ensuring unauthorized persons are unable to gain access to the data. Confidentiality is almost always handled by some form of encryption if the data is in transit.

Integrity deals with authenticity of the messages. Is the reader of the message viewing the message in its intended form without any tampering from outsiders? Integrity can be achieved via strong authentication and digital signatures.

Availability is a security concept often forgotten. If the data is unavailable, it is assumed to be secure, but in that case, the data is also not usable by anyone. Ensuring availability will provide protection against failing hardware and Denial of Service attacks, and utilize disaster recovery.

■ **Note** For more information about these three key security goals, check out *CISSP All-in-One Exam Guide, Second Edition* by Shon Harris (McGraw-Hill Osborne Media, 2003).

At this point, it may seem like I have strayed a bit from the goal of introducing the Secure Shell protocol as an excellent security choice, but in reality, the Secure Shell, and in particular OpenSSH (`http://www.openssh.com`), provide all three elements of information security, which is something that the aforementioned traditional UNIX protocols certainly do not.

The final high-level information security concept I wish to cover is the trade-off between usability and security. If a system is thought to be more usable, with less overhead, or less bureaucracy, it usually is assumed to be less secure; in contrast, a system utilizing challenge response authentication, PIN numbers, and other mechanisms is often a headache for end users, but extremely difficult to gain unauthorized access in. This trade-off will require analysis during many phases of your OpenSSH deployment.

Analysis of Legacy Protocols

To understand the distinct advantages that OpenSSH provides over legacy connectivity and administration protocols on UNIX and UNIX-like systems, it is important to understand their strengths and weaknesses.

Common Strengths in Legacy Protocols

Telnet, FTP, and the r-utilities, which include `rlogin`, `rcp`, and `rsh`, have several strengths that can account for their success and explain why they are still so widely used today when more secure alternatives are available.

- *Installation*: `telnetd`, `ftpd`, `rshd`, `rlogind`, and `rcp` are typically enabled on a UNIX system out of the box. The default installs of HP-UX, AIX, and Solaris offer these systems enabled immediately upon installation completion. Often the default settings are enough to get an administrator running with these protocols on new machines. It is appropriate to mention that most Linux systems and some of the BSD variants have these traditional services disabled after a default installation.

- *Cost*: The r-utilities, `ftpd`, and `telnetd` servers are part of the operating system. An SSH server is normally a third-party package, save for Linux and BSD systems, though it could be included on the vendor media. Using third-party tools normally implies a cost either in the form of a software acquisition, training, or support. The stock utilities have no software licensing cost, other than the fees for the operating system, and most experienced UNIX administrators will be familiar with the functionality of these utilities.

- *Ease of use*: The traditional utilities all are easy to use due to their basic authentication practices requiring a username/password or trusted host authentication mechanism. These tools allow for automation of authentication procedures by using trusted hosts/users, which led to a large proliferation of scripts and programs relying on these systems for proper execution. For example, even today several enterprise-grade clustering solutions rely on `rsh` to execute failover and perform several different types of cluster administration.

- *Speed*: A final strength of these protocols is speed. A simple programmatic `for` loop utilizing `rsh` can be orders of magnitude faster than a similar script utilizing SSH. Speed is normally sacrificed to ensure security.

What Makes These Protocols Legacy?

In this context, the term *legacy* is used to describe protocols such as Telnet and FTP because there are better tools available, at least from a security perspective. Legacy also implies that these utilities have been around for a while and are working in many environments. By learning about legacy protocols, their usage and their downfalls, the need for more secure tools will become apparent. Beyond recognizing the need for secure tools, after understanding the tools SSH can replace, you will see how the SSH protocol is designed to handle each of these shortcomings.

Security Analysis of Telnet and FTP

Telnet and FTP often are grouped together as a connectivity solution. Telnet allows remote command execution, while FTP allows for data transfer. Many of their flaws are common, which means that providing protection against the weaknesses in these protocols can be done with single solutions.

Telnet

Telnet is both an application and a protocol. It was designed to replace the hard-wired terminals connected to every computer. It also allowed users at different types of computers to execute their commands on remote machines. Telnet's first and most apparent security issue is that everything, including authentication credentials, are transmitted in clear-text format over the network. This means, using freely available network sniffers such as Ethereal, you can see every piece of information passed over the network.

■ **Note** For more information about the sniffer Ethereal, you can visit `http://www.ethereal.com`. Additionally, it is often included in Linux distributions. Several other network sniffing applications are freely available. In particular, for sniffing on a switched network, Cain & Abel works very nicely. This is available at `http://www.oxid.it/cain.html`.

Many administrators understand that logging in directly as root via Telnet is a bad security practice, but at the same time, many log in as nonprivileged users, and then use `su` or `sudo` (`http://www.courtesan.com/sudo`) to become root all inside of a `telnet` session. This data is easily harvested. Many consider it safe to use Telnet in a situation where the client and server computers all reside on an intranet, yet this is actually a highly insecure practice. Consider that almost every organization has someone who is disgruntled, whether it be a supplier, employee, or partner trying to exploit your system resources. If this is a home network, a friend or family member might attempt to procure your root password to look at files they should not have access to (credit card info, gift receipts, and e-mails) or to hack your system for fun.

Using Cain & Abel, I am able to sniff on a switched network. The snippet presented in Listing 1-1 is a sniffer file with just a little cleanup. The reason some characters appear twice is because I sent those to the remote machine, and for each character I sent, it was returned to the display. As you can see, my demonstration password is easily captured, along with my root password. The same principles used in this example on Telnet can be used with FTP and the r-utilities.

Listing 1-1. *A Sniffing Session from Cain & Abel*

```
==========================================
=== Cain's Telnet sniffer generated file ===
==========================================
login: ssttaahhnnkkee
 Password: goldfish
 Last login: Wed Jun  1 18:18:35 from rack
stahnke@www:~
[stahnke@www ~]$ ssuu  --
 Password: myrootpassword$
 root@www ~> uuppttiimmee
  18:21:25 up 23 days, 15:51,  2 users,  load average: 0.24, 0.07, 0.02
root@www ~> eexxiitt
 logout
[stahnke@www ~]$ eexxiitt
 logout
```

Telnet clearly lacks in the confidentiality department because armed with a simple sniffer, an adversary can watch all data transmitted between the server and your `telnet` client window. Integrity can also suffer by having a rogue server setup with the Domain Name Service (DNS) name or IP address of the target machine intercepting information and then forwarding it on (a *man-in-the-middle*, or MITM, attack) either as a pristine copy, or tampered to make some alterations to the intended system. In this sense, integrity suffers because Telnet has no authentication performed by the server. The only supported authentication is user-based Telnet, which is also susceptible to many classic attacks including *ARP Poisoning* (updating a switch's address resolution table with false data, often used to sniff switched networks), and several types of injection or insertion attacks.

Certain issues with the Telnet protocol can be overcome using one-time passwords, or token-based authentication; however, Telnet sessions still can be victim to an insertion/session-hijacking attack where an attacker can insert arbitrary commands and data into an existing Telnet session, or take over the session completely. These attacks sound sophisticated and complex; however, with the emergence of tools such as HUNT (`http://lin.fsid.cvut.cz/~kra/`) and dsniff (`http://www.monkey.org/~dugsong/dsniff/`), these attacks can be executed with ease.

Common organizational security policy often dictates a minimum password length with some variance of punctuation, numeric, uppercase, and lowercase characters. While complex passwords are a wonderful thing, this is negated when Telnet is in use. Even if you have created a 50-character password that is more like a large sentence, it is easily found and harvested using point-and-click tools freely available. Remember that a user does not have to be actively monitoring the output of a network sniffer. It is quite easy to look for the string password and have it e-mail someone the three lines above and below that string.

As for availability, `telnetd` commonly runs out of `inetd` (the Internet services daemon) in most implementations. Normally, this is quite reliable, but I have seen `inetd` die on systems from every major vendor I have worked with, thus leaving a Telnet connection inoperable. A walk or drive to the console (unless you are lucky enough to have remote consoles on systems) is needed to allow network connectivity again.

In summary, Telnet's biggest disadvantages come from confidentiality and integrity, with confidentiality being the easiest to breach. FTP, which is often used in conjunction with Telnet, has remarkably similar weaknesses.

FTP

FTP, or File Transfer Protocol, is much like Telnet in that it is an application and a protocol. It allows a user to store and retrieve files on remote systems As its primary means of authentication, FTP relies on a username and password, just as Telnet does. FTP's primary concern again is the clear-text transfer of data and authentication credentials. FTP opens up a new problem as well. If an FTP password for a user is compromised, the adversary can not only upload/download files as the user, but also log in to the machine interactively (via `telnet`, `ssh`, or `rsh`) because the username/password combination is the same by default.

While there are implementations of FTP daemons that allow for separate passwords or very configurable security, the inherit weakness of FTP still exists, namely that everything transmitted is in clear text.

Using FTP through firewalls can also be difficult to set up. FTP uses multiple ports for its communication and data transfer. Because of this, additional ports must be opened on a firewall. Some firewalls have specific knowledge of the FTP protocol, and can dynamically open ports for an FTP transfer, but these firewalls can sometimes be exploited by attackers spoofing an FTP-like transaction.

■ **Tip** If you are forced to use FTP for legacy applications and scripts, look at `vsftpd` (`http://vsftpd.beasts.org`). `vsftpd` has several advanced security options that take it a step above traditional FTP daemons, and the ability to encrypt communications over SSL (Secure Sockets Layer) has been added, making it a very nice alternative to stock FTP daemons.

Security Analysis of R-utilities

When working in medium- to large-sized UNIX environments, maintaining synchronized root passwords, updating configuration files, and installing patches can be a very tedious process. To help facilitate such matters, a collection of three utilities known as the r-utilities were designed in BSD4.2 as great features to allow administrators to remove the connectivity and configuration differences between their systems. Namely, the `rsh`, `rlogin`, and `rcp` utilities introduce a way to manage dozens, hundreds, or even thousands of systems utilizing simple programmatic `for` loops. Each is responsible for implementing a specific feature that helps administrators accomplish this goal. Specifically, `rsh` allows the user execution of remote individual commands, `rlogin` gives a terminal window, and `rcp` allows the user to store files to or retrieve files from a remote machine. Administrators and power users have made these utilities

essential to their daily tasks when interacting with UNIX systems. These tools were developed in an age of computing where networks were trusted and used only by computer professionals. Networks were not connected together in situations beyond academia. Eventually, their ease of use caused these tools to proliferate into nearly every UNIX variant in existence.

However, one would be hard-pressed to find a set of tools that has more security problems than these. These tools shine when it comes to speed and stability, but when security comes into question, they obviously tip the scales on the usability side. The inherent problems from the r-utilities lie in the trust mechanism. Most setups of r-utilities involve trusting servers using files called `.rhosts`. If machine A trusts machine B, then Machine A must also trust the authentication mechanisms of machine B. This can lead to several interesting scenarios. Additionally, many times trust relationships were not set up by individual machines, but by IP addresses, or ranges of IP addresses, or even domain names. For example, if I have a machine running the r-utilities daemons, and I trust anything coming from `*.mycompany.com`, anyone who plugs into my network is now allowed to hop from their machine to all the rest of them. This becomes of particular concern with the rise of user-controlled machines that can pass the r-utilities a UID of 0 or that of any other arbitrary user.

If I am a consultant and I walk into My Company and plug in my laptop, most likely I will be assigned an IP address via Dynamic Host Configuration Protocol (DHCP), and it will probably resolve to something in the `*.mycompany.com` domain. Because I have root access on my laptop, I can use `rlogin` to connect to another system, machine B, and machine B trusts that since I have root on my laptop, I should be able to have root on machine B.

The r-utilities have a slew of other concerns that stem from trust relationships among machines, weak (in some cases, virtually nonexistent) authentication, and clear-text transmission of all data.

Again, confidentiality is lost in this setup. There is no encryption, and with many trust relationships, such as those provided with `.rhosts` files, there really is not even any form of access control. Integrity is also very low because I can set up my laptop with the same IP address as machine B and get all of the information intended for machine B, do as I please with it, and then relay it to the real machine B, or just keep it. This attack is also a type of man-in-the-middle attack. Availability becomes questionable and is probably less important when the integrity and confidentiality are at such low levels. If the data is available, can it be trusted?

The discussion of the downside of the r-utilities can be extremely lengthy, but overall the r-utilities are situated on some extremely shaky ground. For more information about UNIX security in general and especially problems with `rsh`, `rlogin`, and `rcp`, you can review *Practical Unix & Internet Security, 3rd Edition* by Simson Garfinkel, Gene Spafford, and Alan Schwartz (O'Reilly and Associates, 2003).

WHERE DO LEGACY PROTOCOLS STILL MAKE SENSE?

Using legacy protocols is a bad idea about 95% of the time. However, in my daily work as an administrator of hundreds of UNIX/Linux systems, I still do from time to time.

I use FTP heavily for patching systems via tools like rpm, apt, pkg, CPAN, and swinstall. The FTP server is configured for read-only access and only has publicly available data, so if any information is compromised it is easily replaced. Additionally, most packaging systems have a way to verify integrity such as RPMs using GNU Privacy Guard (GnuPG, `http://www.gnupg.org`).

Until very recently, if UNIX machines needed to transfer data to or from a mainframe, the only choice was FTP. However, OpenSSH is now available for the mainframe (`http://www-1.ibm.com/servers/eserver/zseries/zos/unix/toys/ssh.html`), and hopefully we can soon eliminate FTP as a data transfer mechanism of mainframe data.

A final usage of legacy protocols involves the use of private networks inside of High Performance Computing (HPC) clusters. I recommend using the r-utilities internal to the cluster only. Normally, two head nodes are on the intranet and the rest of the compute nodes are only on a public network. The r-utility daemons are not enabled on the head nodes, but they are enabled on the compute nodes due to speed and assumed security on a private network. Additionally, most HPC software vendors count on `rsh` being enabled. As an added security measure, `iptables` (`http://www.netfilter.org`) can be used to block the r-utilities ports on the head nodes, in case the r-utilities are ever enabled.

Learn to Replace Legacy Protocols

As network computing grew in purpose and popularity, these legacy protocols were made even more common on all types of machines. With the surge of an Internet presence in the middle 1990s by most corporations, along came their Telnet, FTP, and r-utilities for all to see. During the latter half of the 1990s, it was commonplace to read about a compromised environment, system, or corporation because of the use of `.rhosts` files or weak passwords. Even Fortune 500 companies with very large information technology budgets were not immune to the problems created by poor choices in connectivity protocols and poor policies covering information security.

Many organizations have created new information security policies requiring heavy restrictions on the usage of the r-utilities, Telnet, and FTP without providing a new solution; these policies are then left unenforceable, allowing virtually everyone to get some form of business exception, which allows someone to accept the risk. This is not always required, however, because a viable replacement solution exists in the form of OpenSSH.

The case for Telnet, FTP, and r-utilities made in the past is simply not viable today. These days, networks are shared via Virtual Private Network (VPN), Demilitarized Zone (DMZ), and extranet access. Consultants, contractors, and business partners are constantly onsite, devoting their resources to the highest bidder, and everyone understands that information is probably your organization's most valuable asset. OpenSSH will enable the replacement of legacy protocols and enforce information security policy.

Improving Security with OpenSSH

To replace these legacy protocols, you need a toolset that can provide terminal emulation, transfer files, and run commands remotely, all while encrypting the data, providing user and host authentication, and have comparable availability, speed, and ease of use.

Replacing `rlogin`, `rsh`, and `rcp` was the original goal for the SSH protocol. It has done so without relying on simple IP addresses or `*.mycompany.com` entries. OpenSSH also provides `sftp` in lieu of FTP, eliminating clear-text authentication, while still offering a very similar command set.

Establishing Security Basics

To understand the power and security embedded within OpenSSH, some key security ideas must be conceptually understood. OpenSSH relies on several security mechanisms to ensure the confidentiality, integrity, and availability of its data. This section will not explain exactly how OpenSSH uses these security mechanisms, but it will provide background information so that when a security focus point is discussed, some background on the topic can be assumed.

Checksums

Checksums are used to verify integrity of data. They are calculated values that are used to ensure the provided data matches the expected data. For example, upon login to a UNIX system, a user is prompted for a password. That password is then encrypted using a checksum and compared against an expected value in the `passwd` or `shadow` file. If the value provided matches the value in the file, the user is permitted to log in. If not, the user must try again.

Checksums are fixed lengths. If a file is 100 bytes or 100 gigabytes, a checksum will be of the same size. This can lead to the potential problem of checksum collisions. In certain sum algorithms the collision rate is high, while with others a collision is nearly impossible. The data inside a file cannot be derived by just having the hash value of a file. Checksums are sometimes called *hash functions*, because they convert data via a one-way hash.

sum

Of the different forms of checksum, the `sum` command is one of the least secure. It is found on UNIX/Linux systems and can be used to check to see if two files are the same; however, programs are available to pad a file until a given `sum` output is reached, thus negating `sum` efforts.

To use `sum`, try checking your `/etc/hosts` file. The output is the checksum followed by the number of blocks in a file.

```
stahnke@rack: ~> sum /etc/hosts
3705       1
```

MD5

md5 is a 128-bit hash function. It is commonly used in UNIX/Linux as the hash mechanism for password encryption. The collision rate on md5 was generally thought to be very good; however, recently several algorithms published and implemented have been able to produce collisions in a few hours. At the time of this writing, in many cases md5 is still reliable and secure, but most people are moving to another hash function.

To create an md5 hash value of a file, use the `md5sum` command. It is installed by default on many different system types, and is available for nearly any operating system. The following example shows how to generate a hash value using the md5 algorithm:

```
stahnke@rack: ~> md5sum /etc/hosts
f935ef656d642e4e3b377e8eba42db66  /etc/hosts
```

SHA-1

SHA-1 is another hash function that uses 160-bit checksums. It is generally thought to be more secure than md5. SHA-1 is used in some password authentication implementations on

UNIX/Linux. It also is approved for use by the United States Government. The collision rate on SHA-1 is significantly lower than that of even MD5; however, recently some theoretical attacks against SHA-1 have been presented. These, too, require weeks or months on supercomputers to duplicate. For maximum security, combining multiple hash algorithms can be used. If someone is able to duplicate an MD5 and SHA-1 checksum, the chances of a collision is believed to be impossible even using today's fastest supercomputers.

SHA-1 hashes can be generated using `sha1sum`. This utility is available on Linux and several BSD variants by default. It normally is an add-on for Solaris, HP-UX, AIX, and IRIX. In the following example, a `sha1sum` is used to compute a SHA-1 checksum of the `/etc/hosts` file:

```
stahnke@rack: ~> sha1sum /etc/hosts
d538de234634994b2078a34ea49c377108193ce7  /etc/hosts
```

■ **Tip** OpenSSL can be used to generate md5 and SHA-1 sums also. `openssl md5 /etc/hosts` and `openssl sha1 /etc/hosts` will provide hash outputs.

MACs

MACs, in a security sense (not MAC addresses), are Message Authentication Codes. In cryptography, a MAC is a small amount of information used to authenticate a given data set. A MAC algorithm uses a secret key in combination with the data set to create the MAC or tag. This process can protect a file in the form of integrity, because a MAC will change if the data set has been altered. MACs also provide nonrepudiation (proof of origin), because the secret key must be known to generate a valid MAC.

MACs are not normally available from the command line of a UNIX system. These algorithms are used in encryption security products, such as OpenSSH.

Symmetric Ciphers

Encryption is handled in the form of *ciphers*. A cipher can be a simple as a character substitution. For example, if a message is to be sent over an untrusted medium, the message might be, "We attack at dawn." This text is called the *plaintext*, and is human readable. Using a simple cipher called ROT13, the ciphertext, or encrypted text, is "Jr nggnpx ng qnja." This substitution simply replaces each letter of the alphabet with a character 13 places to the right of it. For example, "e" becomes "r," and "w" becomes "j." The party receiving the message then has to replace each character with the letter 13 places to left in the alphabet, which can span the beginning or end of the alphabet. These are called *symmetric ciphers* because the same key is used to encrypt and decrypt a message.

This is obviously an example of weak cryptography, but it illustrates some key points. Poor encryption is trivial to break. Replacing a single letter with another letter often takes very little time to decode. Patterns can easily be discovered in the ciphertext to assist with decoding the message. For example, if a character occurs more often than any other character, and the plain text message is English, that letter probably represents the letter "e." Attacks can be made against encrypted messages.

Encrypted messages can be sniffed on a network; therefore strong cryptographic algorithms should be utilized to thwart potential attacking efforts.

The simple substitution cipher given in this example is keyed by the number of letters to rotate the alphabet by. Because there are only 25 possible values for this key, it is about 4 bits in length, thus extremely weak. Ciphers depend on a shared key. If a shared key is agreed upon, ahead of time by both the sender and receiver of a message, the encryption is much stronger. An attacker will then need a sniffed (intercepted) ciphertext message and a key to decode it properly. Strong ciphers also produce ciphertext that will contain higher degrees of randomness than most normal written languages.

This description of ciphers is fairly high level. To administer OpenSSH, no knowledge of how a cipher is created is required. For performance optimization or security concerns, however, knowing which ciphers to choose and their key length can come into play.

There are two main types of symmetric ciphers utilized in computing today: block and stream ciphers. Both of these types of ciphers are supported in OpenSSH.

Block Ciphers

Block ciphers are used to encrypt data of a fixed length in conjunction with a shared key. For example, if a cipher is using a 128-bit block of clear text, the encrypted text would be 128 bits. The way the encryption occurs is dependent on the key and cipher algorithm. Decryption of block ciphers requires the shared key and using the decryption algorithm. Ciphers are generally a fast solution to encryption, but they present a problem of transmitting a shared key to the receiving party without it being compromised.

DES

DES, or the Data Encryption Standard, was published in the late 1970s as an encryption standard for the United States Government. It uses 56-bit keys, which at the time would have taken years to break; however, with computational power increases, compromising DES-encrypted communication is possible, and with some expensive hardware can be done in hours or sometimes minutes. DES is only supported by the OpenSSH client for compatibility with SSH Protocol 1 servers that do not support any stronger ciphers. OpenSSH does not support any cryptographically weak ciphers by default.

3DES

3DES, pronounced triple-DES, is a derivative of DES that uses separate shared keys (either two or three) to encrypt, decrypt, and then encrypt data again using DES. Because it was designed to be implemented in hardware, the structure of DES makes it relatively slower to implement in comparison with other ciphers. It has not been broken by any known attacks against it.

AES

AES, which stands for Advanced Encryption Standard, was adopted by United States Government as an encryption standard after DES proved to be much too weak. Like DES, it relies on a shared key, but the algorithm used for encryption is much more difficult to break. AES supports several different block sizes as well. Using OpenSSH, AES key sizes can range from 128 bits to 256 bits. AES performs at high speed and is commonly used in hardware and software.

Blowfish

Blowfish is another block cipher that is supported in OpenSSH. Blowfish is a fast cipher that was aimed at replacing aging DES technology. It usage has been decreasing because of the