Global Information Assurance Certification Paper

Analysis of a Secure Shell Vulnerability in Support of the Cyber Defense Initiative

GCIH Practical Assignment version 2.1

James M. "Mike" Anthis

September, 2001

Abstract

On June 20, 2002, a defect in the OpenSSH implementation of the Secure Shell was discovered. Secure Shell is used to permit authorized computer users to safely access computing resources over insecure networks, such as the Internet. Known as the "OpenSSH Challenge-Response Buffer Overflow Vulnerability," the defect can be exploited by a tool that is constructed from modified OpenSSH source code. This paper describes the Secure Shell service, the defect that made OpenSSH vulnerable, the tool that is used to exploit the vulnerability, and measures that can be taken to repair the defect and protect against attacks that exploit it.

Table of Contents

Introduction: The Internet Storm Center and the Cyber-Defense Initiative

The Internet Storm Center

The System and Network Security Institute (SANS) established the Internet Storm Center (formerly known as the Consensus Intrusion Database) to provide early warning and response to widespread computer security incidents. The first dramatic success of the Internet Storm Center was on March 22, 2001, when an automated attack known as the "Lion Worm" began to spread around the world. The Lion Worm was breaking into systems on the Internet, stealing passwords from these systems, and establishing a foothold for further attacks. It was programmed to spread rapidly from each computer to additional computers, and thousands of systems had already been compromised.

Computing analysts associated with the Internet Storm Center received early warning of this incident, which enabled them to plan an effective response and notify computer administrators and law enforcement authorities within a day. The worldwide effort to stop the spread of the attack and block it's damaging effects was unprecedented in its effectiveness because it was prompt and coordinated. The Internet Storm Center's monitoring and early warning capabilities made this possible. (Ullrich, http://isc.incidents.org/about.html)

The Internet Storm Center constantly monitors automated intrusion detection systems around the world, and provides up-to-date information about computer intrusion attempts. The daily "Top 10 Ports" report is one of the ways such information is provided. It indicates the ten specific Internet services that are being attacked most frequently.

The Cyber Defense Initiative

In addition to the Internet Storm Center, the SANS Institute supports a project called the Cyber Defense Initiative (CDI).

> CDI's specific objective is to help provide the information, resources, and tools organizations need to improve the security of their sites. One way to do this is through knowledge sharing.
>
> (Cole, http://www.sans.org/CDI.htm)

One of the methods CDI uses to promote information sharing is to publish research by candidates for the Global Information Assurance Certification (GIAC) program. As part of the program, this paper analyzes an Internet service that is currently a primary target for intrusion attempts, the Secure Shell service, and a recently exploited vulnerability in that service. By addressing one of the numerous Internet vulnerabilities that have recently emerged, the author seeks to join the ranks GIAC scholars in assisting the computer security community.

Part I: Secure Shell – One of The Top 10 Targeted Ports on the Internet

Introduction to Internet Services and Protocols

The global computer network known as the Internet began as an experimental computer network, called ARPAnet, which was constructed by the United States Department of Defense's Advanced Research Projects Agency (DARPA) (Albitz and Liu, p.1). In the early 1980's, protocols on the ARPAnet became standardized, such as the Transmission Control Protocol and the Internet Protocol, now commonly known as TCP/IP.

Among the essential functions served by TCP/IP are the identification of computers on the internet, and the identification of each of the services provided by any given computer on the internet.

Servers and Clients

Networked computers typically operate application programs called "servers." The function of a server application is to receive requests from other computer programs, called "clients," and perform some kind of computing task in response to the request. The client and server applications may be operating on the same computer or on different computers. Servers typically wait for requests to come from clients, and clients typically initiate actions by the servers by sending them requests.

Some services may involve calculating or looking up a piece of information that the client application requires to accomplish its task. Other services may require the server application to store and retain a piece of information provided by the client.

The TCP/IP protocol provides the method used by both the client and server to locate each other. The client and server each have an IP address, indicating the identity of the computer where they are operating, and a port number, indicating the identity of the program running on the computer.

The Internet Protocol identifies computers by their Internet Addresses, and the TCP protocol identifies services by their port numbers. Internet addresses commonly appear as four decimal numbers separated by periods, (for example, "192.168.12.25"). Port numbers are typically denoted as a decimal number, (such as "22"). Occasionally, to identify a specific service on a specific computer (also known as a "host"), the notations are combined with a colon (combining the previous examples in this way results in the notation "192.168.12.25:22").

A specific list of Internet port numbers identifies a number of particular kinds of computer services that are common on the Internet. Services such as file transfer (ports 20 and 21), web page display (port 80), Internet directory service (port 53), clock service (port 37), and so on, are distinguished primarily by the

standard port numbers assigned to each service. Historically, port numbers were used to identify services because it is much more efficient for computers to transmit and interpret small numbers than alphabetic service names. Even so, the most commonly used ports also have names (usually acronyms) that can be used by humans to more easily understand and discuss the services.

Frequently Attacked Internet Services

The Internet Storm Center issues a daily report on the ten most frequently attacked services, by port number. As it turns out, about half of the Top 10 Ports appear on the report every day. These are the ports that attackers find very fruitful to scan using automated tools. There are a few interesting reasons for this.

Some ports provide attack opportunities because of the complexity of the service provided via the port. An example of such a service is the Hypertext Transfer Protocol (HTTP) service on port 80. HTTP supports many of the different kinds of information services known as the World Wide Web. The large numbers of complex technologies that provide information services over HTTP make it very difficult to implement such services securely. This difficulty also makes the HTTP service very attractive to attacks over the Internet.

Other times, the prevalence of old, particularly vulnerable implementations of common services makes them attractive targets. Attackers know that many computer owners will take a long time to discover and repair a vulnerable implementation of a popular service. During that time, attackers can use automated tools to search out and penetrate such vulnerable systems. Some examples of applications for which old, vulnerable versions are prevalent include:

- File Transfer Protocol (FTP)
- Berkely Internet Name Domain (BIND) Service
- Simple Mail Transfer Protocol (SMTP)
- Microsoft's SQL Server database
- Microsoft Internet Information Service (IIS)
- Microsoft Windows File Sharing service

The excellent usability of the Microsoft products can work against security, as they are very accessible to inexperienced computer users. Many Microsoft customers do not have the training or experience to practice good network security. The accessibility of these products, combined with the underlying complexity of the services provided by the product, and the sheer size of the population of Microsoft customers, makes these products very attractive targets for attack.

Automated tools exist that contain databases of services and vulnerabilities. Two popular examples of such tools are Nmap and Nessus. Skoudis provides a list of numerous vulnerability scanners in his book, Hack Attack (Skoudis, p. 229).

Such tools invite the cliché of the sword with two edges: they can be used by computer owners to discover vulnerabilities in their own systems, but they can also be used by attackers to identify potential victims.

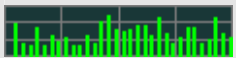
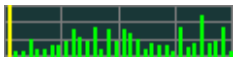
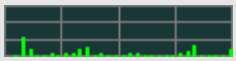
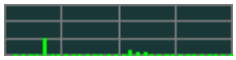
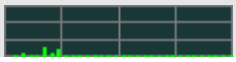
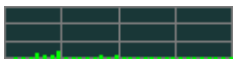
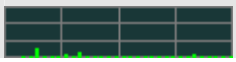
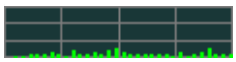
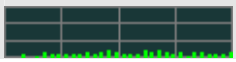
There are also clandestine services that have been installed by attackers on computers scattered around the Internet. These services are sometimes associated with particular ports. For example, a tool known as the "Sub Seven Trojan" is frequently found on port number 27374. One of the objectives of many attackers is to install their own remotely controlled tools, known as "Trojan Horses," in the computers of unwitting users. Such tools are then used to launch attacks on other computers, by amplifying the computing power of the attacker's computer, by abusing compromised computers to identify additional targets that are likely to be vulnerable, or by masking the origin of a particular attack by routing it through a chain of other "trojanned" computers.

Ports associated with abusive tools appear on the Top 10 Ports list for two reasons: either they are being actively used by the original perpetrator, or they are being searched out by opportunists who also know how to use these tools.

The following Top 10 Ports report was issued on July 16th from the Internet Storm Center (Ullrich, http://isc.incidents.org/top10.html)

Last update July 16, 2002 17:29 pm GMT

Top 10 Ports

Service Name	Port Number	30 day history	Explanation
http	80		HTTP Web server
ms-sql-s	1433		Microsoft SQL Server
ftp	21		FTP servers typically run on this port
ssh	22		Secure Shell, old versions are vulnerable
???	4665		eDonkey P2P software
netbios-ssn	139		Windows File Sharing Probe
asp	27374		Scan for Windows SubSeven Trojan
???	6112		
smtp	25		Mail server listens on this port.
???	43981		

In this Top 10 Ports report, there are six common internet services identified (HTTP, SQL Server, FTP, SSH, NETBIOS, SMTP) and four uncommon port numbers (4665, 27374, 6112, 43981). At least one of these latter port numbers is known to be used by the “SubSeven” Trojan horse.

A Common Target: Port 22 and The Secure Shell Service

TCP Port 22 is associated with one of the most commonly attacked services, the Secure Shell protocol. Applications that use this port are commonly referred to as “Secure Shell” or “SSH.” In truth, there are a number of products and services that are called “Secure Shell.” Their common purpose is to provide safe, reliable access between computers.

The Secure Shell protocol protects communications between computers by providing authentication and encryption services. The object of these services is

to protect the data passing between the computers from tampering, as well as to keep it private from eavesdroppers. The sender and recipient of the protected data can be assured that their messages arrive unaltered and undisclosed to third parties.

Protocols are often layered over other protocols; functions provided by a lower-level protocol can be used by other protocols. The Secure Shell uses the lower-level TCP protocol on port 22 because it needs the error correction and sequencing functions that TCP provides.

Secure Shell Applications

Secure Shell applications provide authentic, private communications across insecure networks. These services are useful for a number of applications, most commonly command-line sessions with remote computers, and file transfer between computers.

When communicating across a wide-area network such as the global Internet, messages pass through numerous computers and network devices. Typically, most of these devices do not belong to the owners of the computers at the origin and destination of the messages. Because of the diversity of ownership along the path of an Internet message, there is great potential for potentially hostile agents to observe sensitive information such as passwords or trade secrets. There is also a risk of a hostile change to messages in transit, such as bank account numbers or commands that control computers or other machinery.

Secure Shell uses encryption to protect communication across these untrusted communication links. Modern cryptographic methods simultaneously provide two forms of protection: they make it impossible for adversaries to eavesdrop on the exchange of data, and they prevent tampering with the data that is exchanged. These two forms of protection are known as “privacy” and “integrity.” The primary purpose of Secure Shell is to provide these protections for data exchanged across the Internet.

Computers also need to be reliably identified. Before networks were common, computers were reliably identified by their physical presence. Modern computers are often accessed across a network that may be shared with other, untrusted computers and users. It is possible for a user to be tricked into using an untrusted computer, and divulge secret information to adversaries.

For example, while unknowingly connecting to a malicious user’s computer, instead of her own, a user might divulge a password that controls sensitive information or valuable resources in her own computer. Even though the message would be transmitted securely, the recipient would be an untrusted imposter. The imposter could then use the password to abuse the unwitting user’s computer and data. That’s why it’s important for a user to be able to verify that they are connected to the particular computer they want to use, not an imposter.

Secure Shell verifies the identity of remote computers for its users. It automatically generates special passwords, called "keys," for computers to exchange in order to verify each computer's identity. When a user connects to a remote computer using Secure Shell, the keys are compared before communication starts, and unless the remote computer is proven to be the one the user expects, communication with the imposter is blocked and the user is notified of the discrepancy.

Secure Shell Protocols

The term Secure Shell, or SSH, is often loosely used to refer to applications that provide authentication and encryption over TCP port 22. Strictly speaking, Secure Shell is not a product or application. "SSH is a protocol, not a product. It is a specification of how to conduct secure communication over a network." (Barrett, Silverman, p. 4)

Furthermore, there are two versions of the SSH protocol, SSH-1 and SSH-2. The SSH-2 protocol was developed to eliminate weaknesses that were discovered in the SSH-1 protocol. Although SSH-1 and SSH2 protocols are incompatible with each other, modern Secure Shell applications can use both protocols, in order to provide backwards compatibility to older applications that only use SSH-1.

The SSH protocols incorporate cryptographic protocols in order to accomplish their functions. SSH employs a collection of mathematical protocols known as *Public Key Cryptography*. Also known as *asymmetric* cryptography, these techniques allow computerized passwords known as "keys" to be split into two parts, so that one part can be transmitted over a (vulnerable) public network, while the other part remains a protected secret. Special mathematical procedures recombine the public and private portions of the key to enable the authentication and encryption functions that SSH provides. (Singh, 1999) (Schneier, 1996) (Stallings, 1995)

The mathematical operations required to use the two-part keys is lengthy and complex. For efficiency, SSH combines this split-key strategy with simpler protocols known as *symmetric* cryptography, which use one-part keys. The combination provide the security of asymmetric cryptography with the efficiency of symmetric cryptography. (Note that these cryptographic protocols are mathematical procedures that govern the encryption process, but they are not Internet protocols.)

Internet protocols other than SSH can be protected by the SSH protocol using a process called "tunneling." Tunneling is a way of layering other protocols over SSH so that they enjoy the protections that Secure Shell provides. It is as if the SSH protocol provided an armored pipe, or "tunnel," for other protocols to travel through. For example, the X-Windows terminal system uses an unencrypted protocol called "X." A user typing in an X-Windows session might unwittingly expose his passwords to an eavesdropper, who would then gain access to the computer accounts controlled by those passwords. Using Secure Shell, the user

can “tunnel” the X protocol through Secure Shell, thwarting eavesdroppers with the encryption Secure Shell provides. The term “tunneling” refers to the rerouting of the data that would go between the ports dedicated to the X protocol, and transmitting the data in encrypted form between the SSH ports instead. SSH provides the same tunneling service for other protocols and ports as it does with the X protocol.

SSH Vulnerabilities

The original Secure Shell protocol, SSH-1, had weaknesses in its design and implementation. It was replaced by the improved SSH-2 protocol, which was also designed to support future plug-in extensions for authentication and encryption. The “plug-in” feature allows newly-discovered vulnerabilities to be repaired by replacing the extensions. The protocol would not have to change in order to make such repairs.

SSH-1 Protocol Vulnerabilities

SSH-1 is vulnerable to an “insertion attack” because it uses the weak CRC-32 integrity mechanism (Barrett, Silverman, p.103).

“CRC-32” stands for “Cyclic Redundancy Check – 32 bit checksum.” It is a calculation that is made on the data stream between the client and server, in order to detect any changes to the data during transit. If an attacker tries to change the data, the checksum will not match, and the secure shell application can reject the data and alert the user.

The weakness relates to the specific combination of the CRC-32 integrity check protocol and the SSH-1 encryption protocol. Combining these protocols weakens both. The combined weakness allows an attacker to insert arbitrary text into the data stream (Arce, http://www.landfield.com/isn/mail-archive/1998/Jun/0052.html).

The weakness allows an attacker to perform a “known plaintext attack” that permits the insertion of encrypted packets with any chosen plaintext in the client to server stream. This type of attack will subvert the integrity checks on the server and can allow an attacker to execute arbitrary commands on the server. This ability can be combined with other attack methods to gain privileged access to the system running the SSH-1 server.

SSH-2 provides for a number of stronger integrity checks, known as Message Authentication Codes (MAC’s). Modern MAC’s use mathematical calculations similar to the best encryption techniques. SSH-2 also allows future SSH implementations to “plug in” new MAC’s and encryption methods, which makes it possible to repair any newly-discovered vulnerabilities without changing the SSH-2 protocol.

SSH-2 Implementation Vulnerabilities

Secure Shell (both SSH-1 and SSH-2) is a protocol, not a computer program. Various computer programs can be written that interact with each other using the Secure Shell protocol. (In common parlance, any of these programs are typically called "the secure shell".) A notable feature of the SSH-2 protocol is its ability to support a variety of methods for encryption, integrity checking and authentication.

Particular implementations of the SSH protocol can have their own vulnerabilities that are unrelated to the protocol itself. Buffer overflow vulnerabilities are particularly notorious, and a recent vulnerability of this type was discovered in the OpenSSH implementation of the Secure Shell Protocol.
(OpenSSH, http://www.openssh.org/txt/preauth.adv)
(Common Vulnerabilities and Exposures, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0640)
(Security Focus, http://online.securityfocus.com/bid/5093)

Stack-based buffer overflow vulnerabilities are well documented by Ed Skoudis, in his book *Counter Hack* (Skoudis, p. 256).

Buffer overflow attacks target the way that computer programs manage the layout of data in the computer's memory. Typically, a vulnerable program allocates some memory for messages from the network, and places the messages alongside other memory areas that are used by the program to control its operations. By providing a carefully constructed message that overflows the memory set aside for the message, an attacker can write malicious data over the program's control areas. The malicious data causes the program to malfunction in ways that are useful to the attacker. In this way, the attacker hijacks the vulnerable program and causes it to execute malicious commands.

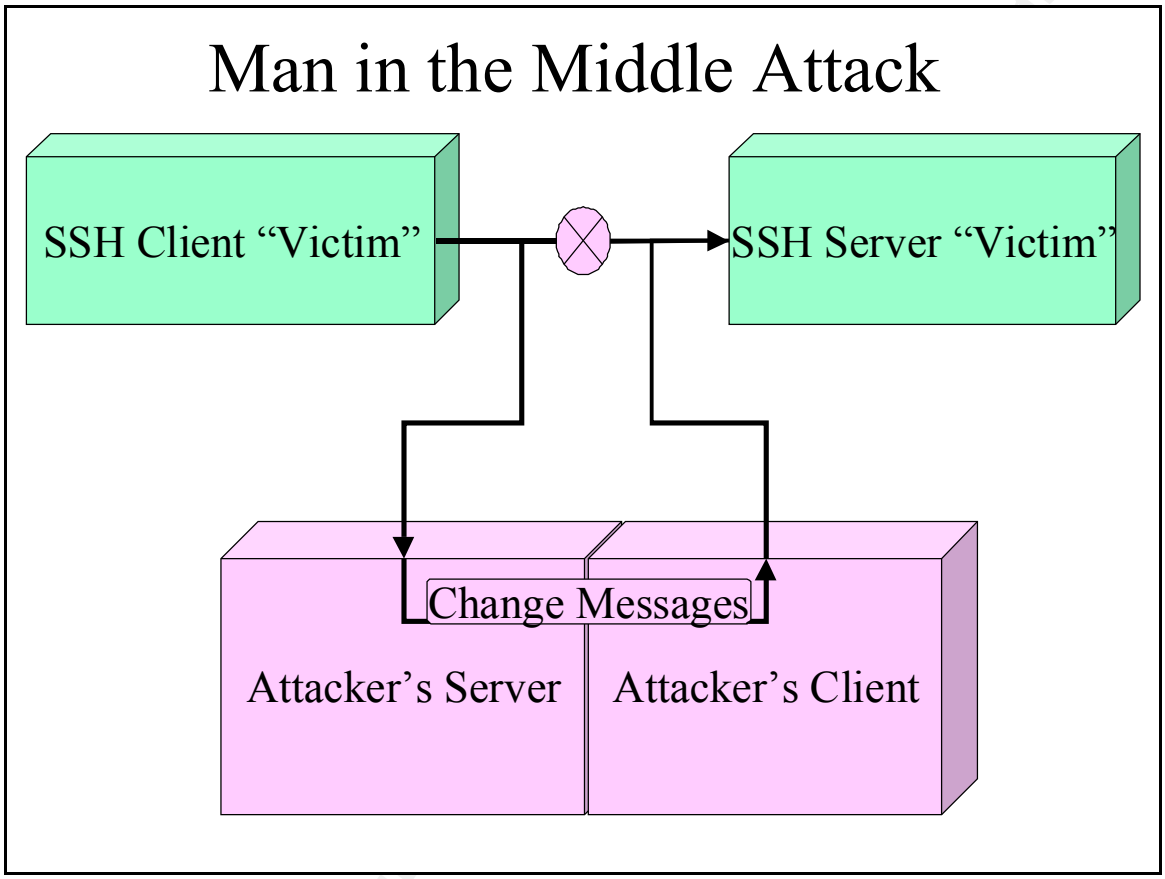
This paper focuses on a vulnerable implementation of the "Challenge Response" authentication method in OpenSSH implementations prior to version 3.4. These versions were susceptible to a combination of overflow conditions in the OpenSSH server when a malicious client sends too many responses to an authentication request. For example, the server may ask for the user's password from the client, but the malicious client will send a huge list of fake passwords. These fake passwords contain code that is designed to fool the server program into providing an interactive session to the user of the malicious client. Since the server program runs with system privileges, the interactive session enables the malicious user to take over the computer where the server program is running.

Human Engineering Issues

Protection against the Man-in-the-Middle Attack

The "man in the middle attack" is a hostile strategy that doesn't need to break encryption protocols or vulnerable implementations. The method is to intercept the original request from an SSH client, and respond to it with an SSH server belonging to the attacker. The attacker follows up by using his own SSH client to

connect to the SSH server that was intended to receive the original request. Typically, the attacker will pass along the client's and server's messages to fool both the client and the server into thinking that they are communicating with each other. However, the attacker can collect and/or change sensitive information as it passes through his own server and client.



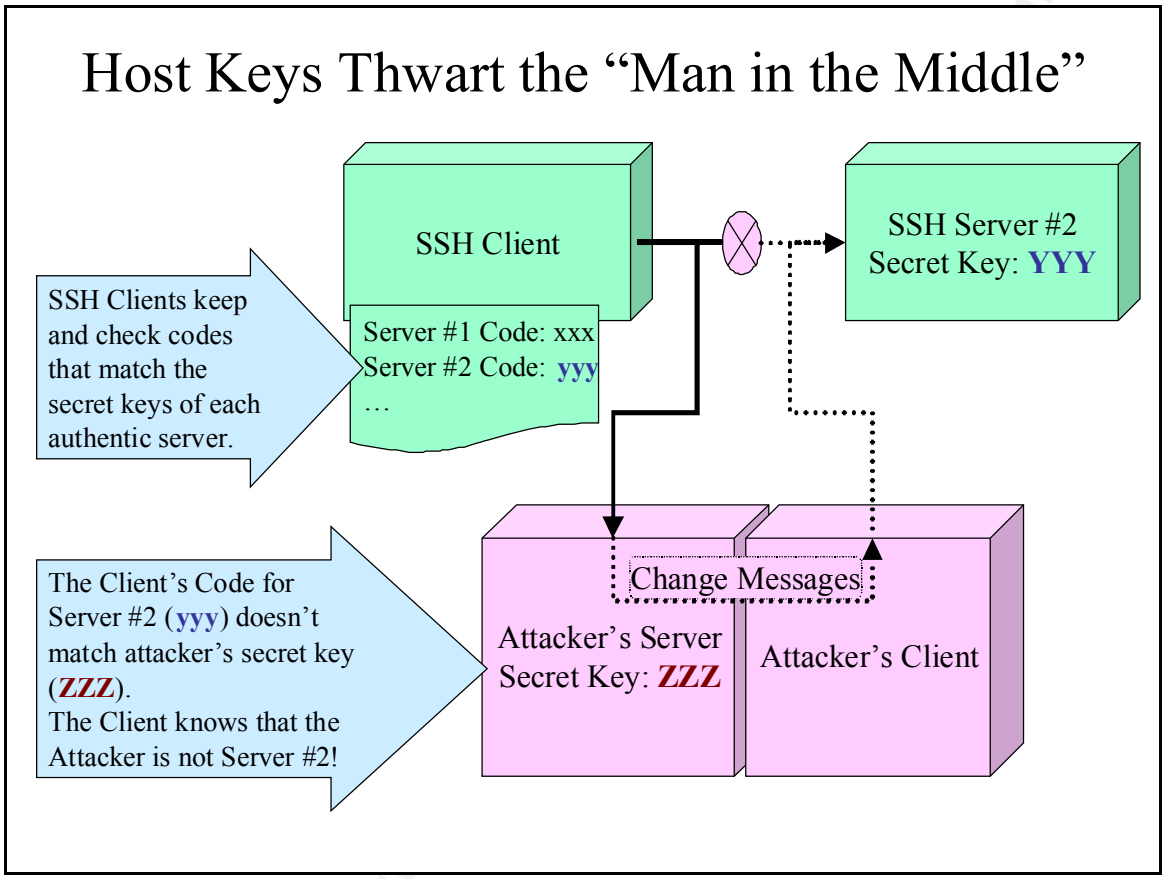
Secure Shell defends against the "man in the middle attack" by giving each server system a secret "key" that uniquely identifies it. In order to defend against this attack, the would-be victim must collect a set of keys that match the secret "host" keys of each of the servers that are known to be trustworthy.

The user of the Secure Shell client does not collect the actual host keys, which are kept secret to the computer running the Secure Shell server program. Instead, the keys that are collected for Secure Shell clients are the "public" keys of "public-private" key pairs, so that the host keys can remain secret, while still providing the ability for clients to verify the key.

Public-private key cryptography, or *asymmetric* cryptography, is used to implement a number of methods used for secure communication. William Stallings (1995) and Bruce Schneier (1996) provide a thorough treatment of the mathematics of asymmetric cryptography, while Simon Singh (1999) addresses the subject with less mathematical rigor.

When used to defend against the Man-in-the-Middle attack, the public key works like a "code" that can translate the server's private key into an "Okay to proceed"

indication. By sending the client's public portion of the server's public-private key pair, the server uses mathematical operations with the public and private key to prove its identity to the client. The man-in-the-middle can intercept the public key "code," but cannot fake the "okay to proceed" message because he never sees the private key of the server.



To summarize, the server's public key is provided by the SSH client whenever a session is established, in order to verify that the host the user wants to communicate with is authentic. The public key held by the user is cryptographically compared with the host's private key as part of the SSH protocol. If the keys match, the host is known to be the one the user wants to communicate with. If the keys do not match, SSH warns the user that there might be a man-in-the-middle attack underway.

The Attacker cannot reproduce the secret key. If the SSH client connects to the attacker's server, the secret key of the attacker's server will not match the key that the client originally collected from the trustworthy server, and by refusing the connection, the client thwarts the man-in-the-middle attack.

Exploiting Human Weaknesses Regarding Host Keys

A human engineering issue with SSH involves the need for users to obtain the public key of each host that they intend to communicate with, prior to initiating the first connection. One way is for the client to obtain the key through "out of band"

communications, that is, by some other means than using a network connection with the trusted server. This could be via secure e-mail (which introduces its own key exchange problem), or by obtaining a key in-person on a removable media. A telephone call could be used to read the key to the user of the client system, but this would be difficult because of the lengthy and unintelligible (to humans) strings that form the keys. However, there are techniques known as "key fingerprinting" that can be used to combine a short, easy-to-recite verifier with a longer, e-mailed key, so that key exchanges can be done without personal contact between the participants (Garfinkel, p. 238).

SSH applications typically provide a very convenient, but somewhat risky feature, to help the user collect the public key of each server the first time they connect to it. When the user has no public key for a server, SSH offers to retrieve the key from the server, and save it, while the first session with the server is being established. It provides a cautionary message to the user, explaining that SSH can only verify that future connections are with the same server, but not that the server is the one the user intends to communicate with.

Risks of Collecting Keys with Secure Shell

Most users will assume that they are not being attacked on their first connection with a server, and accept the risk of collecting the server's key in this manner. If their assumption is correct, they will indeed be getting a safe connection to that server in the future.

If a user unwittingly accepts a key from an attacker's server, he may not find out that he has been fooled until the attacker's host becomes inaccessible. In such a case, the first connection with the authentic server will generate an alerting message to the user. By this time, the user may have already compromised sensitive information. Worse, the authentic server now appears to be the attacker, unless and until the confused user takes steps to verify the host key by some other means than using the secure shell itself.

The Secure Shell client also provides a warning message when a server's private key does not match the public key retained by the client. There are legitimate occasions where an authentic host may have a new private key; for example, when the operating system is reloaded and the administrators neglect to preserve the old SSH keys. After the new keys are created on the server, the next connection by a Secure Shell user will generate an alert message that the host may not be authentic. (This can be frequent in computer laboratories.)

This feature can also play off of human weaknesses: after warning the user, Secure Shell then offers to save the new keys! This is meant to make it more convenient for the user to cope with environments where the host keys change frequently, but it can also make it convenient for the user to fall victim to a real attack.

Both of these key-collection features are intended to help the user avoid the inconvenience of establishing personal contact with the owners of the computers running the Secure Shell servers. Unfortunately, the more often the features pay

off with legitimate connections, the less effective the warnings become in generating justifiable suspicion with the SSH user. While Secure Shell has options for turning off these convenience features, the human tendency is to leave them on in order to make communications more friendly to the users. This trade-off between convenience and security is often at the root of human-engineering vulnerabilities.

Part II: A Specific Exploit of the Challenge-Response Authentication Vulnerability of OpenSSH

Exploit Details

The Challenge-Response Authentication vulnerability was discovered on or around June 24th, 2002 (Milford, http://marc.theaimsgroup.com/?l=openbsd-misc&m=102507468017147&w=2).

It was publicized on June 26th (Internet Security Systems X-Force, http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20584) and an a method for exploiting the vulnerability was published on July 1st (GOBBLES Security, http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz).

Name:	OpenSSH Challenge-Response Authentication Buffer Overflow
CVE Number:	CAN-2002-0640 (under review)
CERT Number:	CA-2002-18
Variants:	GOBBLES, obsdexp
Operating Systems Affected:	OpenSSH 3.3 and earlier, affects numerous Unix and Linux Releases (see appendix), MacOS X
Protocols/Services:	SSH-2 with keyboard-interactive authentication
Brief Description:	A hostile modification to the SSH client floods the server with authentication responses. A buffer overflows and permits hostile code from the client to provide a command shell with the privileges of the server, typically root.

(CERT Coordination Center, http://www.cert.org/advisories/CA-2002-18.html) (Common Vulnerabilities and Exposures, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0640)

Exploit Variants

SecurityFocus provides two proof-of-concept exploits for the Challenge-Response Authentication vulnerability.

Both of the exploit variants involve patching and compiling the source code for OpenSSH. Compiling the patched code produces a malicious SSH client that exploits the Challenge-Response Authentication process.

GOBBLES Exploit

GOBBLES Security calls itself “largest nonprofit (and active) security group in existence” (GOBBLES http://www.immunitysec.com/GOBBLES/about.html). Their OpenSSH exploit is tuned to attack the SSH server that is bundled with OpenBSD 3.1 operating system. Here are two sources of the GOBBLES source code and documentation for this exploit:

http://online.securityfocus.com/data/vulnerabilities/exploits/sshutup-theo.tar.gz

http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz

The GOBBLES exploit is discussed in detail in this paper.

obsdexp-Style Exploit

Christophe Devine provided a simpler proof-of-concept patch to the SSH client (http://online.securityfocus.com/archive/1/279610/2002-06-28/2002-07-04/0).

This patch to the sshconnect2.c program in the portable version of OpenSSH (version 3.2.2p1) produces a malicious SSH client that opens a “backdoor” command interpreter on TCP port 128 of the compromised system.

Differences Between the GOBBLES and obsdexp Exploits

- obsdexp contains the source code for the malicious command shell that executes the attacker’s commands. The GOBBLES exploit provides binary code for the malicious command shell. This code appears as a hexadecimal string in the patch to the OpenSSH client.
- obsdexp connects a root command interpreter to port 128 of the computer that is running the vulnerable SSH server. This is commonly known as a backdoor. The attacker must use a program like netcat (Skoudis, 359) to connect a command line on his attacking computer to the backdoor port on the server. The GOBBLES exploit connects the root command interpreter directly to the attacking SSH client.
- The GOBBLES exploit code has features for adapting the exploit to various different vulnerable servers, by varying the size and number of malicious responses sent to the server. Options for using these adaptations make the GOBBLES exploit something of a hacking tool, in addition to its stated purpose as a proof-of-concept demonstration. The obsdexp code uses fixed quantities for the size and number of responses. It is more a demonstration than a tool.

Details of the Exploited SSH-2 Protocol

Objectives of the Secure Shell Protocols

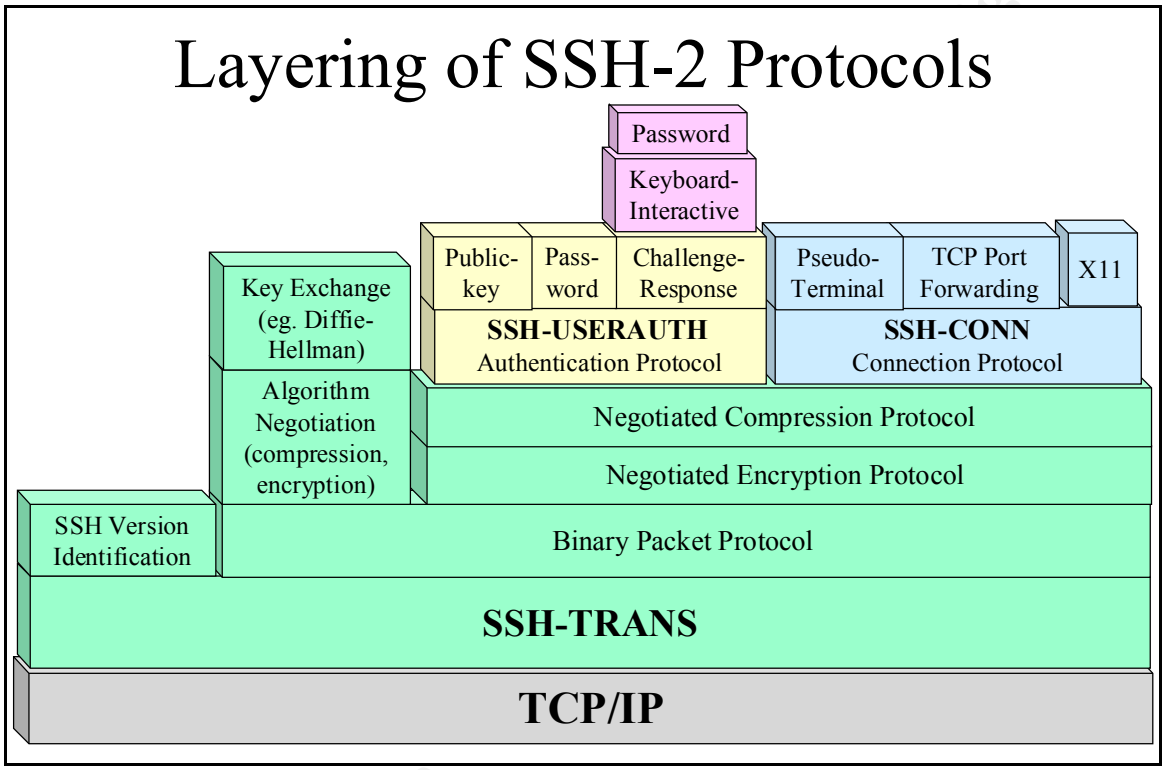
The SSH-2 Protocol provides a secure channel for communication between two computers over an insecure connection, such as the Internet.

To accomplish these objectives, Secure Shell uses authentication and encryption.

- Authentication is the verification of the identity of a computer or computer user. Most computer users have experience using a password to obtain access to a computer. The process of verifying such a password is an example of authentication.
- Encryption is the translation of a message using a "secret code." The translated message cannot be interpreted without using the secret code to retranslate it back to its original form. (Singh, 1999) (Schneier, 1996) (Stallings, 1995) In truth, the "secret code" is typically a combination of a well-known mathematical process with a very large, secret number, or "key." Only the legitimate sender and receiver of a message have access to its contents, because only they know the "key" used to encrypt and decrypt the message. (In asymmetric cryptography, where the "secret code" involves the combination of "public" and "private" keys, techniques exist where only the receiver, and not even the sender, can decrypt the message.)

Structure of the SSH-2 Protocol

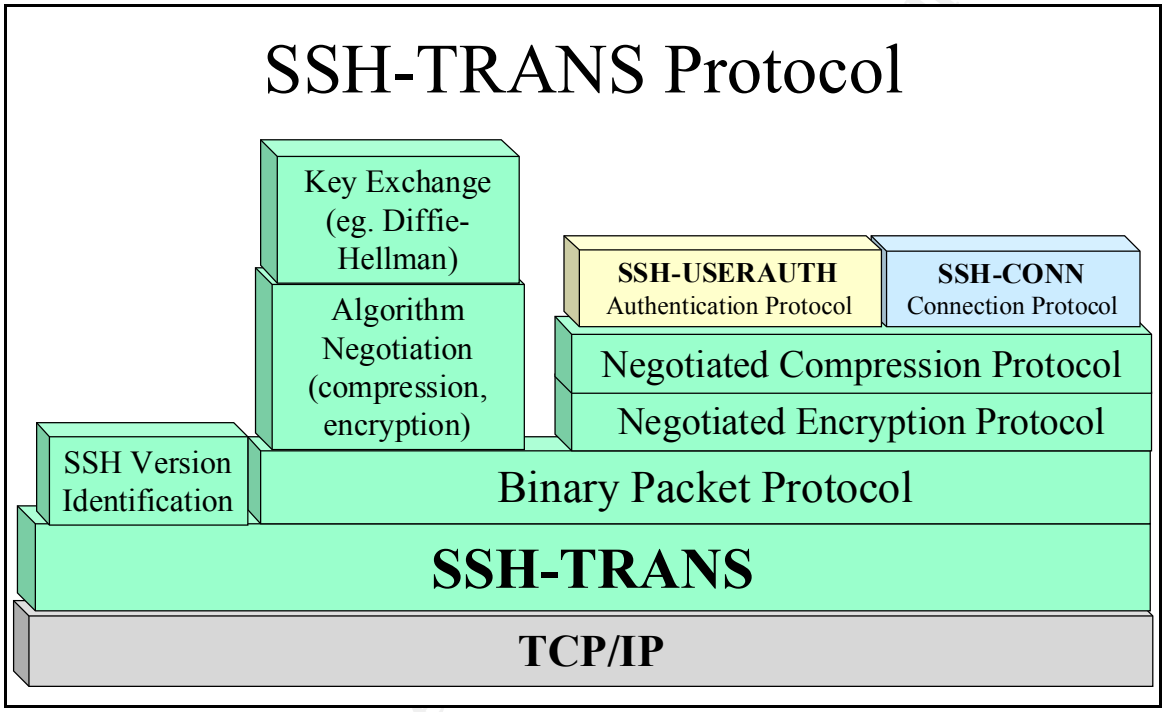
SSH-2 is really a collection of layered protocols working together. Each protocol implements one or more of the mechanisms are employed by the Secure Shell protocols to verify passwords, exchange secret keys, and apply encryption to protect the exchange of data (Ylonen, Kivenen, Saarinen, Rinne, Lehtinen, http://www.openssh.org/txt/draft-ietf-secsh-architecture-12.txt).



- **SSH-TRANS**, or Transport Layer Protocol forms provides the foundation for the collection of SSH protocols. It incorporates the Binary Packet Protocol, which specifies how messages between SSH clients and SSH servers are constructed.
- The Authentication Protocol, **SSH-USERAUTH**, verifies the user's identity at the beginning of an SSH-2 session, by exchanging authentication data such as passwords. The protocol provides a "plug-in" interface for different sub-protocols, such as traditional Unix password authentication, public-private-key authentication, and methods based on authentication devices such as badges or "smart cards".
- The Connection Protocol, **SSH-CONN**, permits a number of different services to exchange data through the secure channel provided by SSH-TRANS. These can include interactive command-line sessions, file copying, and secure forwarding of other protocols, such as X-Windows sessions.

Details of the Transport Layer Protocol

The Transport Layer Protocol, SSH-TRANS, provides the foundation for data exchange between the SSH-2 client and server processes. It provides the method for different Secure Shell implementations to negotiate protocols for the authentication and encryption functions. It also governs the initiation of service-based connections that will utilize the protection features of the secure shell. (Ylonen, Kivenen, Saarinen, Rinne, Lehtinen, http://www.openssh.org/txt/draft-ietf-secsh-transport-14.txt]



The Transport Layer Protocol, SSH-TRANS, initiates an SSH-2 session by taking the following steps:

1. **SSH Version Identification**

   The client initiates a TCP connection to port 22 on the server, and presents an identification string announcing the client's version number.

   The server responds with it's own version identification string.

2. The client and server begin using the "**Binary Packet Protocol**," which is an application-layer format for the data portion of each TCP packet. The purpose of this format is to provide random padding and message authentication for each block of data exchanged between client and server. Random padding makes it harder for attackers to break the encryption codes, and message authentication exposes any tampering with the content of the message.

3. **Algorithm Negotiation**

   The server sends lists of its supported key exchange, encryption, compression, and message authentication protocols to the client.

   The client sends lists of its corresponding lists of supported protocols to the server.

   The client and server then choose a protocol that both support, and prepare to use that common protocol for data compression and encryption.

4. **Key exchange**

   The client and server exchange encryption keys for the session. Currently, SSH-2 uses "Diffie-Hellman" key exchange. The client and server exchange specially-chosen numbers and use mathematical procedures to compute the encryption keys. This method allows the client and server to calculate their keys secretly, even if an eavesdropper observes the numbers exchanged by the client and server.

   A clear explanation of how Diffie-Hellman key exchange works is provided by Simon Singh (1999, p. 257). William Stallings provides a detailed mathematical description of the process (1995, P. 191)

   Key exchange also provides host authentication, so that the user of the SSH client can be assured that they are connecting with the desired computer, not an imposter. This is accomplished by including the server's public and private host keys in the computations. (Friedl, Provos and Simpson, http://www.ietf.org/internet-drafts/draft-ietf-secsh-dh-group-exchange-02.txt).

5. The client and server begin compressing and encrypting the data that they exchange using the **Negotiated Compression Protocol** and the **Negotiated Encryption Protocol**.

6. The client requests the "ssh-userauth" service and completes the **SSH-USERAUTH** protocol with the server.

7. The client uses the SSH Connection Protocol, **SSH-CONN**, to request an SSH connection for one of the protected services (such as a command interpreter, or a file transfer operation), and starts exchanging application data with the server.

   The client may set up additional connections, each of which will benefit from the authentication, compression and encryption steps that have already been accomplished.

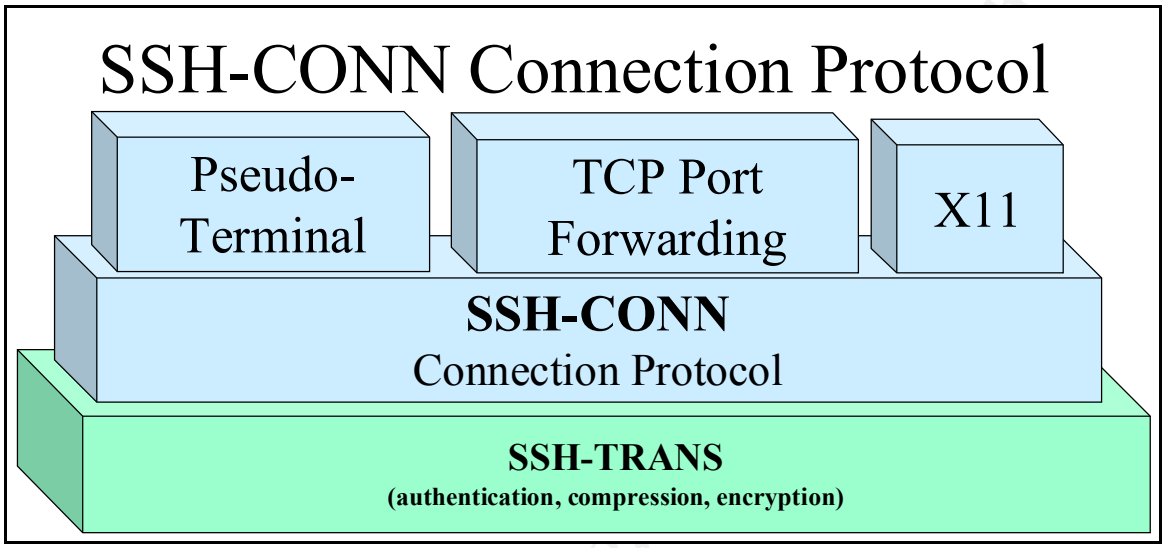
8. Eventually, either the client or server sends a "disconnect" request, terminating the session.

Details of the SSH-CONN Connection Protocol

The SSH Connection Protocol, SSH-CONN, is used by the client to provide some number of simultaneous interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X-Windows

connections. (Ylonen, Kivenen, Saarinen, Rinne, Lehtinen, http://www.openssh.org/txt/draft-ietf-secsh-connect-15.txt)

The primary objective of SSH is to protect communications channels by layering them on top of the Connection Protocol and the lower-level protocols that support it.



Pseudo-terminal connections provide traditional teletype emulation, so that the user of the SSH client can interactively exchange text data with the computer running the SSH server.

The X-Windows **X11** protocol allows the user to exchange graphical information with applications running on the same computer as the SSH server. An X-Windows terminal (an "X-server"), running on the same computer as the SSH client, will use the SSH connection to communicate with X11 applications ("X-clients") running on the same computer as the SSH server. The SSH client transparently intercepts the X11 protocol. It is not necessary to make any changes to the X-server, nor to the X-clients, in order for them to communicate securely using SSH.

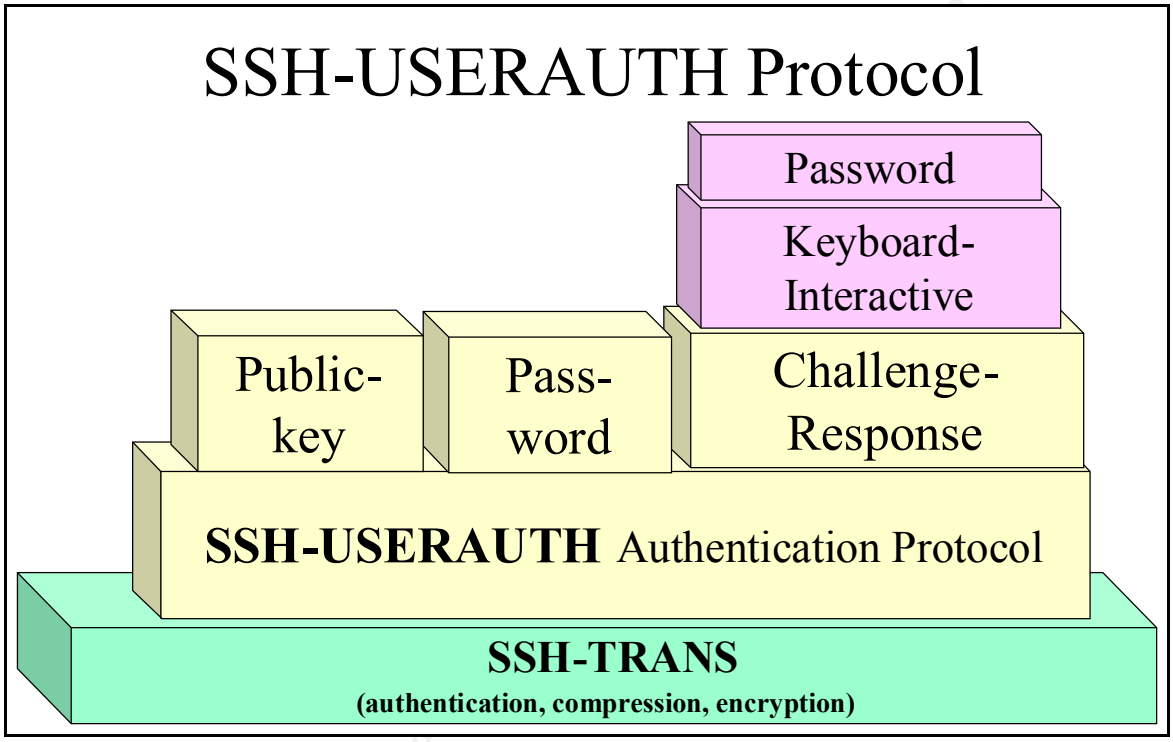
TCP Port Forwarding provides the same kind of protection to any application that communicates using TCP/IP over a specified port. The SSH client intercepts any communication to the port and delivers it to the computer running the SSH server over the SSH connection.

Each connection, or session, exchanges data between the client and server using the Transport Layer Protocol. The **Connection Protocol** provides message types for the client and server to open new connections, close connections, examine the status of connections, and adjust communications parameters.

All connections share the encryption, compression, and authentication services that are established by the Transport and Authentication protocols.

Details of the SSH-USERAUTH Authentication Protocol

The SSH Authentication Protocol, SSH-USERAUTH, is initiated by the server prior to starting the SSH-CONN Connection protocol. It verifies the identity of the user of the SSH client. Before the Connection protocol proceeds, SSH-USERAUTH must complete by successfully verifying the identity of the user (Ylonen, Kivenen, Saarinen, Rinne, Lehtinen, http://www.openssh.org/txt/draft-ietf-secsh-userauth-15.txt).



The SSH-USERAUTH Protocol supports more than one user authentication method. Some authentication methods use simple **password authentication**. With password authentication, the user of the SSH client is prompted to enter a password, and the computer where the SSH server is running verifies the password.

Other authentication methods use cryptographic methods involving a private and a **public key**. Using this alternative, SSH utilities create pairs of keys for a user, and subsequently use these keys to verify the user's identity. Details of this method are described by Barrett and Silverman (p.168).

With **Challenge-response** authentication methods, the server sends one or more requests for information only the authentic user should know. These are called "challenges." In the most simple case, the server may simply prompt for a password (equivalent to "password" authentication). The SSH client collects the requested data, called "responses," from the user, and sends them to the server. The computer running the server provides the means for verifying that the response to each challenge is correct.

Challenge-response authentication supports collection of authentication data from smartcards, token-generating cards, biometric devices, and so forth. It can be used by systems that require more than one of these methods to be used together.

The SSH-USERAUTH protocol operates as follows:

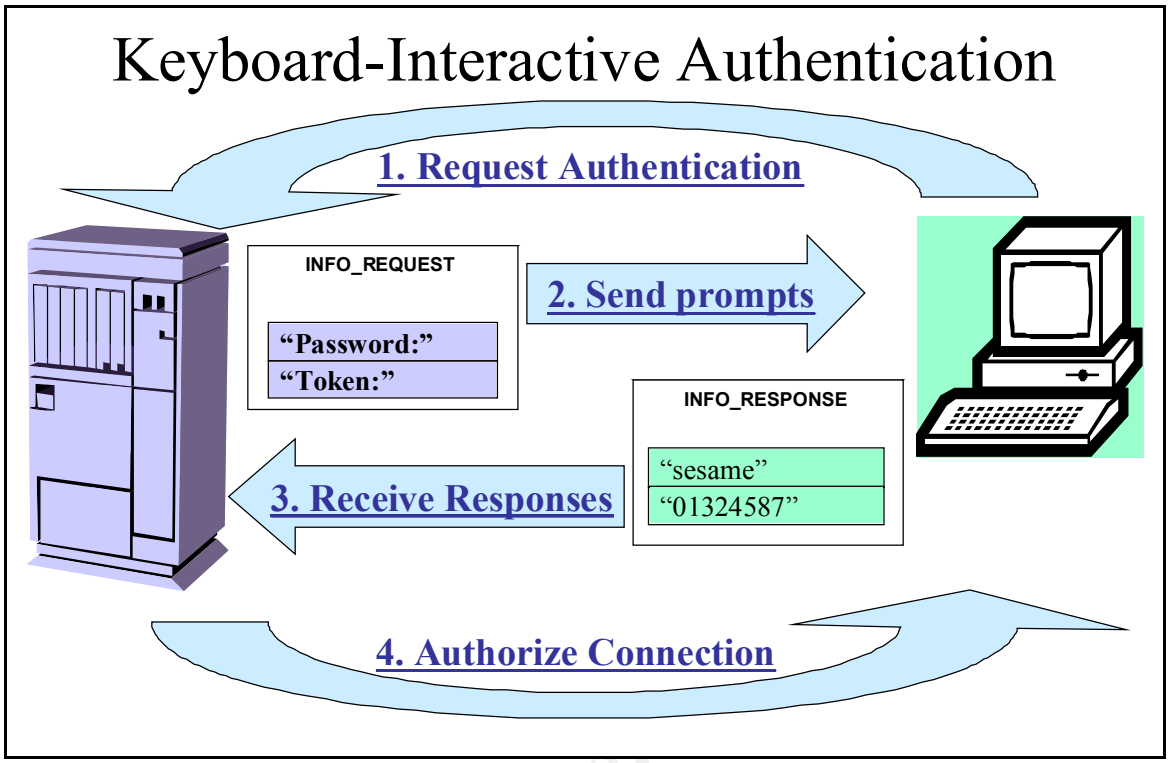
1. The server tells the client which authentication methods it supports.
2. The client sends an authentication request using one of the supported methods.
3. The server may reject the authentication request, and in doing so, may also offer alternate authentication requests for additional attempts by the client. If all alternatives for authentication have been exhausted,
4. Additional messages may be sent between the client and server, depending on the authentication method chosen by the client.
5. The server sends a message to the client indicating that authentication is successful.

The SSH-USERAUTH protocol supports the **Keyboard-Interactive** method of Challenge-Response authentication. The GOBBLES exploit attacks a vulnerability in the Keyboard-Interactive implementations of OpenSSH earlier than version 3.4.

How Keyboard-Interactive Authentication Works

Keyboard-Interactive Authentication is used when the SSH server needs to prompt the user of the SSH client for authentication information. The user then types the requested information on the keyboard of the computer running the SSH client. Using this method, the SSH client program does not have to have any special coding to participate in a challenge-response authentication method provided by the server. (Cusack, Appgate, http://www.openssh.org/txt/draft-ietf-secsh-auth-kbdinteract-02.txt)

Typically, a Unix-style password prompt is provided, a password is collected from the user, transmitted from client to server, and the operating system on the server verifies that the password is valid.



The specific protocol is as follows:

1. The client sends a message of type SSH_MSG_USERAUTH_REQUEST
2. The server requests authentication information with a message of type SSH_MSG_USERAUTH_INFO_REQUEST
3. The client obtains the information from the user and transmits it to the server with a message of type SSH_MSG_USERAUTH_INFO_RESPONSE
4. The server uses the response to verify the identity of the user and authorize the connection.

The SSH_MSG_USERAUTH_INFO_REQUEST message sent by the server may specify multiple prompts, requesting multiple user-provided inputs. The client should then return that number of responses in the SSH_MSG_USERAUTH_INFO_RESPONSE message. However, it is typical for only one prompt for the Unix password to be issued, and one response to be returned (which is encrypted by the SSH Transport Protocol, so that the password is not exposed to eavesdroppers.)

Weaknesses in the Keyboard-Interactive Implementation

Both the SSH_MSG_USERAUTH_INFO_REQUEST and SSH_MSG_USERAUTH_INFO_RESPONSE messages include a count of the number of responses that the user must provide to be successfully authenticated. One requirement of authentication is that the number of responses provided must

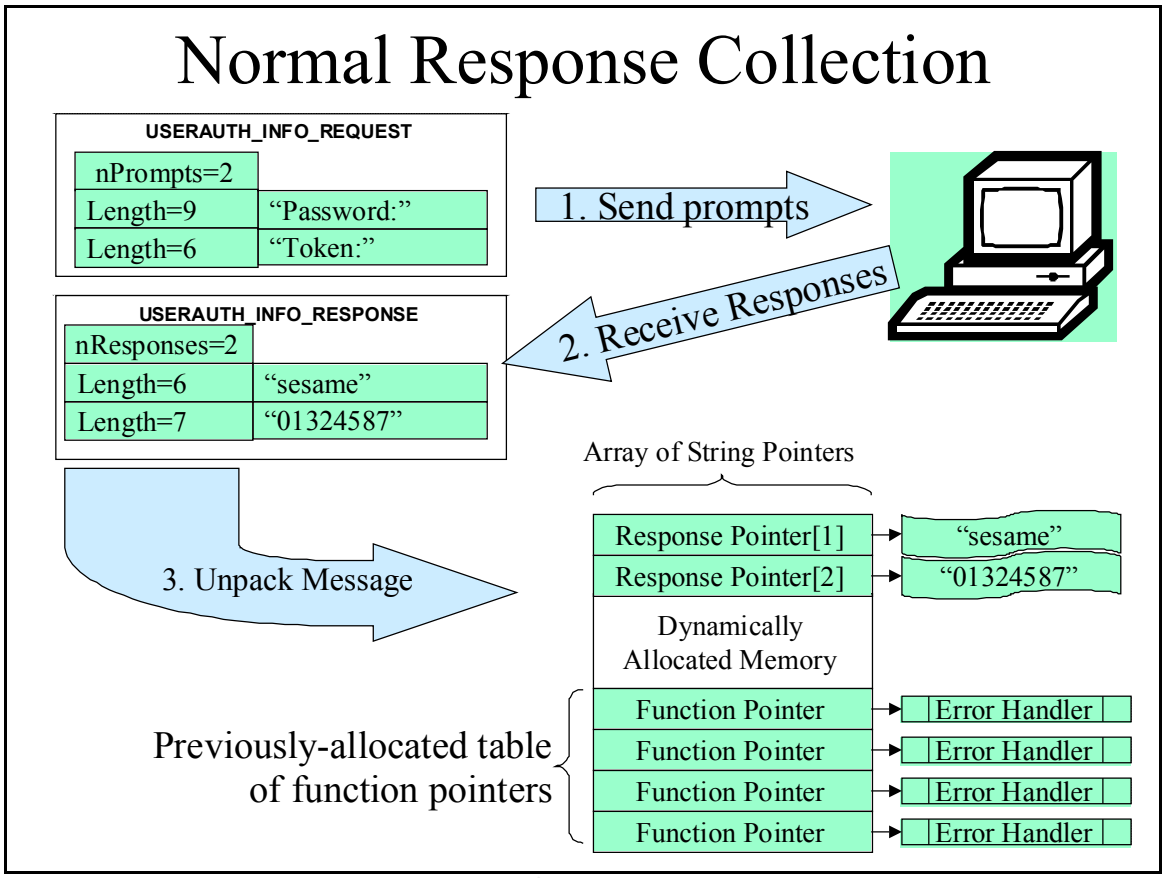
be the same as the number requested. Still, it is possible to send a different number of responses than were requested.

The SSH server must allocate memory to contain the responses sent back from the client. If the memory allocation is not done carefully, a buffer overflow can occur, causing responses to write over data areas adjacent to the buffer. In the vulnerable OpenSSH implementation, data related to error handling is situated near the memory used to store the responses. This data is used by the SSH server program to recover from erroneous responses.

The vulnerable data areas in the SSH server program include string pointers, function pointers and “error handler” functions.

String pointers are used as part of a method for unpacking the responses during keyboard-interactive authentication. As strings of character data are received from the network, the server must store the character strings in memory. In order to keep track of the sequence of responses, the SSH server builds a table of memory locations, or “pointers,” that it uses to find the character strings that were stored.

Although the correct procedure for challenge response authentication requires the same number of prompts and responses to be exchanged, the Challenge-Response authentication protocol allows a greater number of responses to be sent than were requested. If the SSH server program is not implemented carefully, the table of “Response Pointers” can overflow into adjacent areas in memory.



Adjacent to the table of response pointers is another table of "function pointers." These pointers are used by the SSH-server program to find the error-handling procedures when they are needed. Functions of these "error handlers" include: sending replies to the SSH-client indicating that authentication did not succeed, releasing memory that was allocated for responses, and resetting the SSH server to accept new authentication requests.

In vulnerable versions of OpenSSH (prior to version 3.4), the program for interpreting the SSH_MSG_USERAUTH_INFO_RESPONSE message can be attacked by offering a huge number of responses. In these versions, the table of response pointers is near the table of function pointers for the error handlers. An attacker can send a carefully-constructed response that overwrites the error-handling data with hostile information, and cause the SSH server program to give the attacker access to the computer running the server program.

From the Internet Security Systems Advisory (http://www.iss.net/security_center/static/9169.php):

> It is possible for a remote attacker to send a specially-crafted reply that triggers an overflow. This can result in a remote denial of service attack on the OpenSSH daemon or a complete remote compromise. The OpenSSH daemon runs with superuser privilege, so remote attackers can gain superuser access by exploiting this vulnerability.

How the GOBBLES Exploit Works

The exploit code provided by GOBBLES Security (http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz) contains a full explanation of the vulnerable code in OpenSSH version 3.3, as well as how to modify an OpenSSH 3.4 client to exploit the vulnerabilities.

Why the Exploit Works

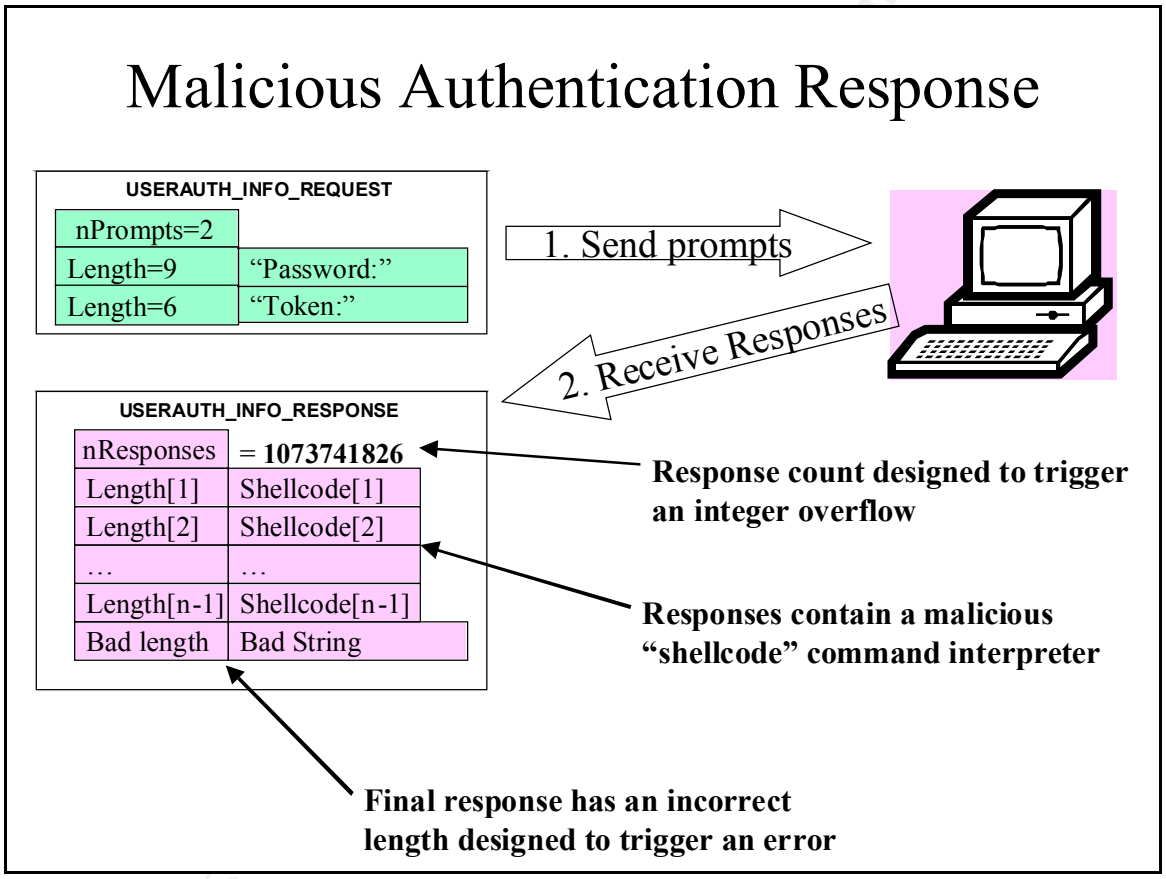
From the Internet Security Systems X-Force Database (http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20584):

> When a challenge is generated, the user is expected to supply a number of responses to verify their identity. The response the user sends supplies an integer that indicates how many responses they are supplying, followed by the responses themselves. By supplying an overly large integer to indicate the number of responses, a remote attacker could overflow a buffer and execute arbitrary code on the system with root privileges.

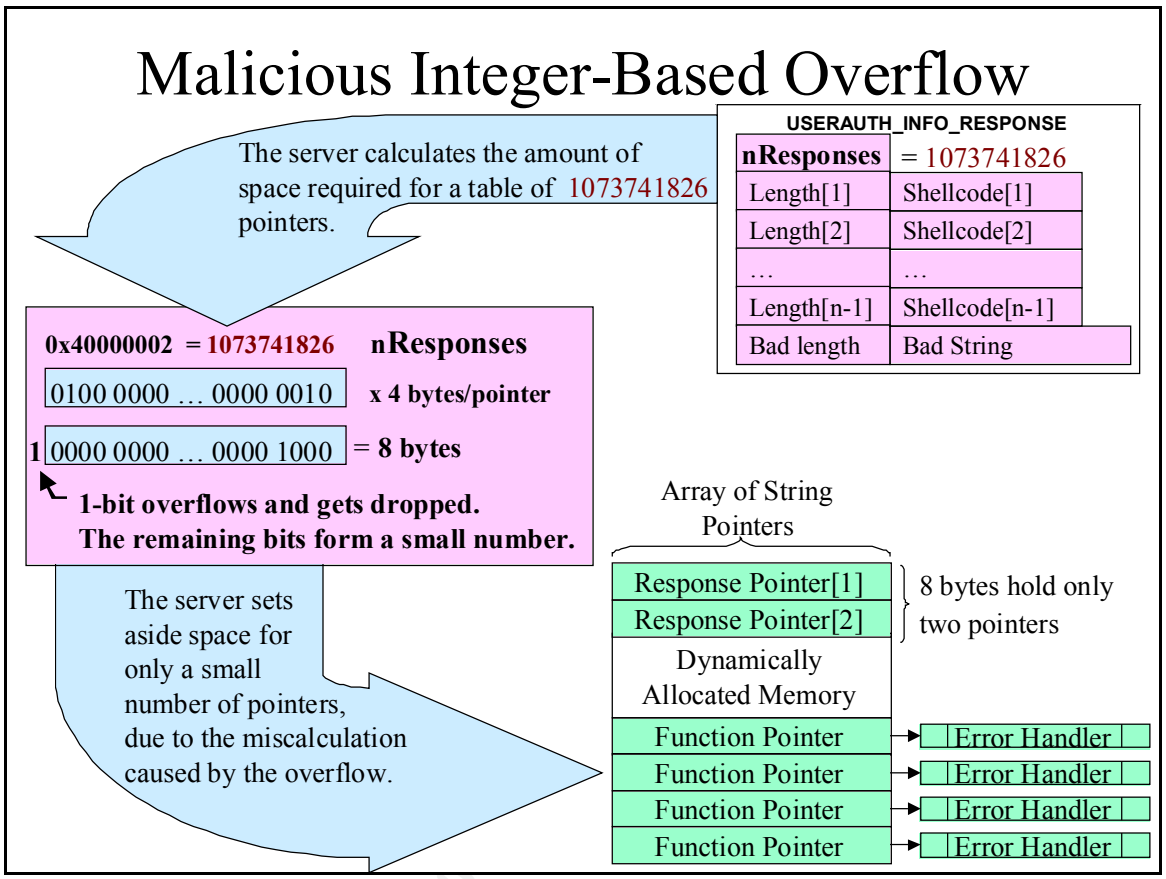
Exploit Actions, Step-by-Step

Here is a simplified description of the GOBBLES exploit:

1. The SSH client is modified to request "keyboard-interactive" authentication from the server. This causes the server to use the Challenge-Response mechanism, which contains the vulnerable code.
2. The vulnerable SSH server sends a USERAUTH_INFO_REQUEST message to the client.



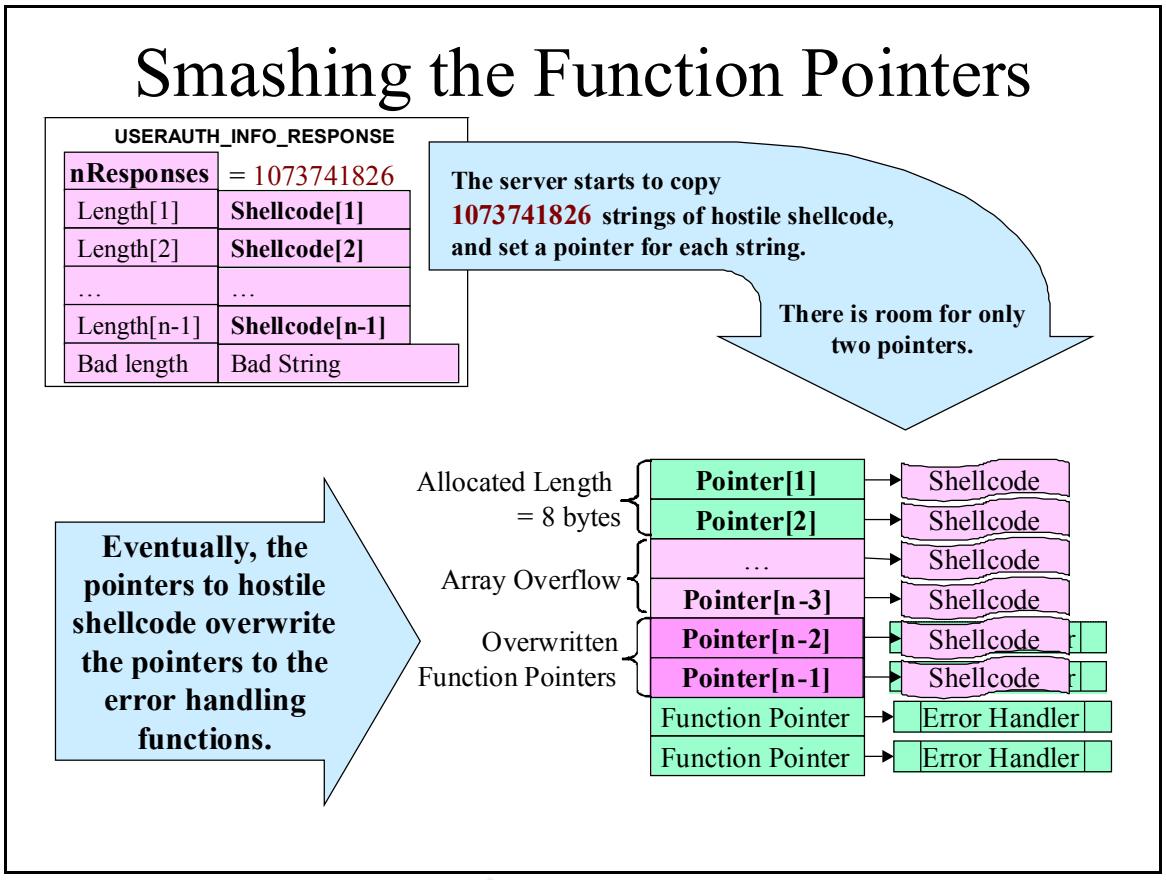
3. When the SSH client receives the authentication request from the server, it ignores the requested number of responses. It constructs a huge list of responses, which includes a count along with fake responses that contain malicious machine code. The count is so big that the server cannot allocate enough memory locations to receive the responses.



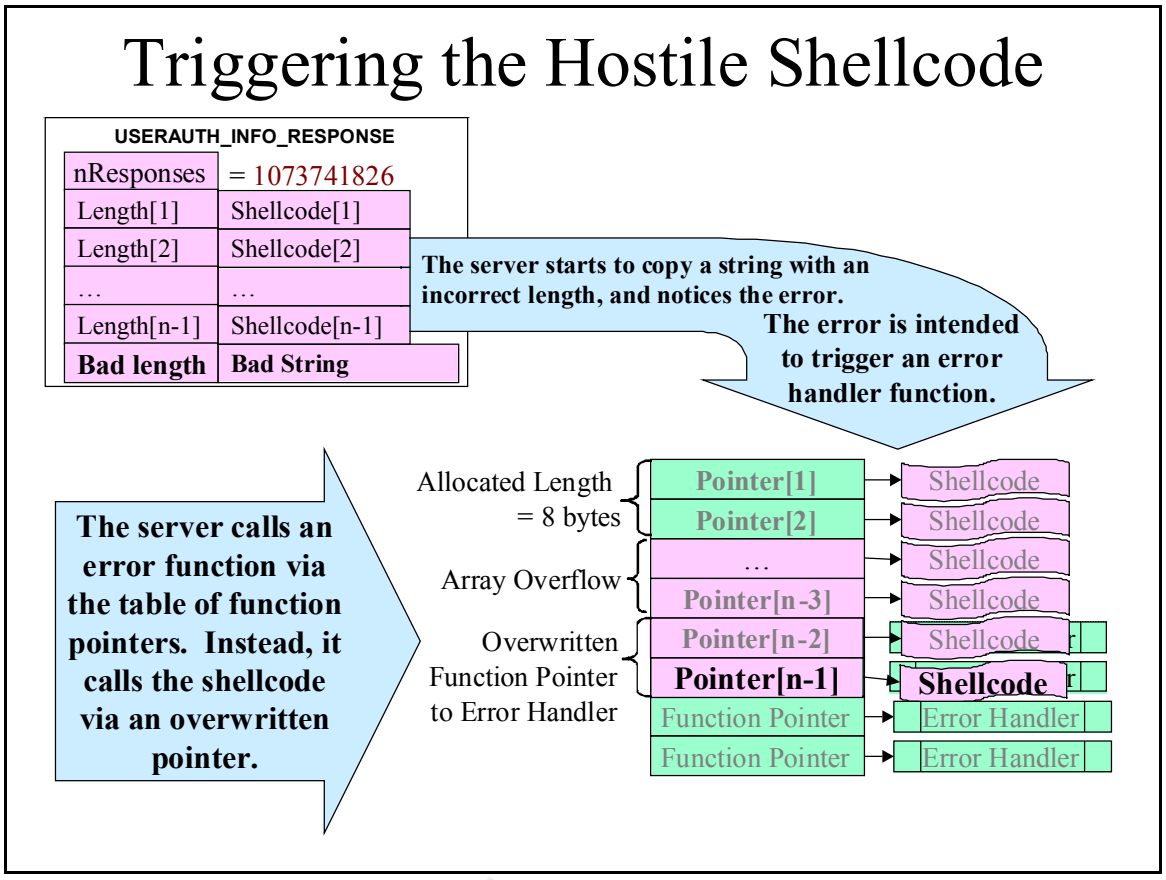
4. The server tries to allocate memory in preparation to receive the huge list of responses. It is vulnerable partly because it does not check to see if the number of responses is reasonable and valid. A valid number of responses would match the number requested. A reasonable number would be feasible for a human to produce. It is also vulnerable because it does not check to see if it can successfully allocate the required amount of memory. Instead, it allocates an insufficient amount of memory, and starts filling that memory with the malicious code from the client.

   Note that the number of response strings actually provided is not really as large as the false number of responses specified. The false number is designed to cause a register to overflow when it is used to allocate memory.

   The amount of memory that is allocated is calculated by multiplying the size of a response pointer by the false number of responses. A response pointer takes up four bytes. The false number of responses is designed to cause an overflow when the calculation is done, so that a binary bit is effectively shifted out of the register that holds the result. This causes the result of the calculation to be much smaller than the correct value; in fact, it is a small number instead of a large one.

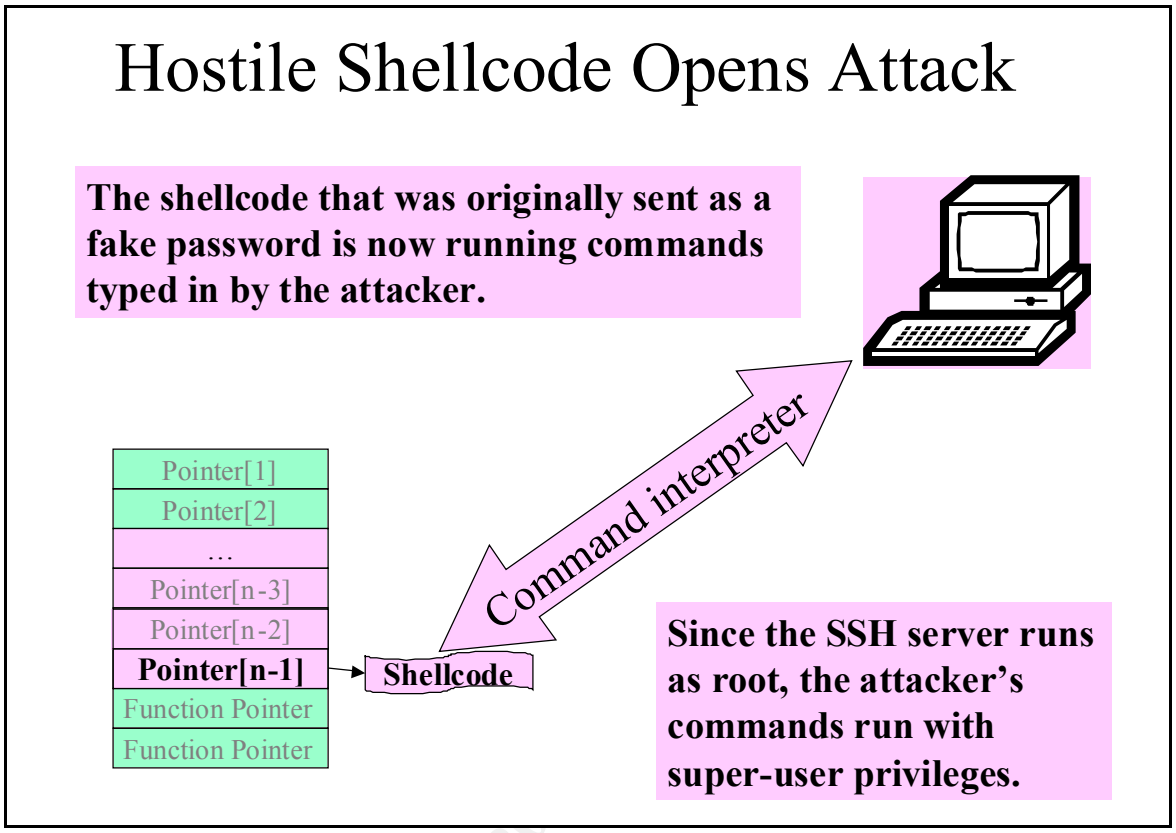


5. After mistakenly allocating insufficient memory for the pointer table, the vulnerable server begins to load the malicious responses and link them into the table. When the pointers to malicious code overflow the memory allocated for the table, it overwrites another table of function pointers that the vulnerable server program uses to handle errors. It replaces the pointers to error handling subroutines with pointers to the malicious code.
6. Each malicious response is constructed of a string length and a string of machine code. The server checks the string length from the response, and invokes an error handler if the length exceeds the size of the buffer allocated to store the string. Most of the responses containing malicious code are associated with correct string lengths, so that the server will continue to load the malicious command interpreter, or "shellcode," into memory.



7. The last string in the malicious response is designed to trigger the attacker's code. It contains an intentionally erroneous string length, and the vulnerable server detects the error. The server's error handler is invoked by using one of the function pointers that was overwritten with a pointer to a malicious subroutine. Instead of shutting down the connection, as the real error handler should, the malicious subroutine connects with the attacking client and executes interactive commands entered by the attacker.

Note that this buffer overflow attack does not involve a classic stack-based overflow. This implies that intrusion detection systems that scan the stack for attack signatures will not be triggered by the hostile shellcode, since the hostile code is stored in allocated memory, not in a temporary variable on the stack. Similarly, intrusion prevention schemes that protect the stack from contamination will not thwart this attack, since the stack is not the region of memory that is overwritten by the exploit.



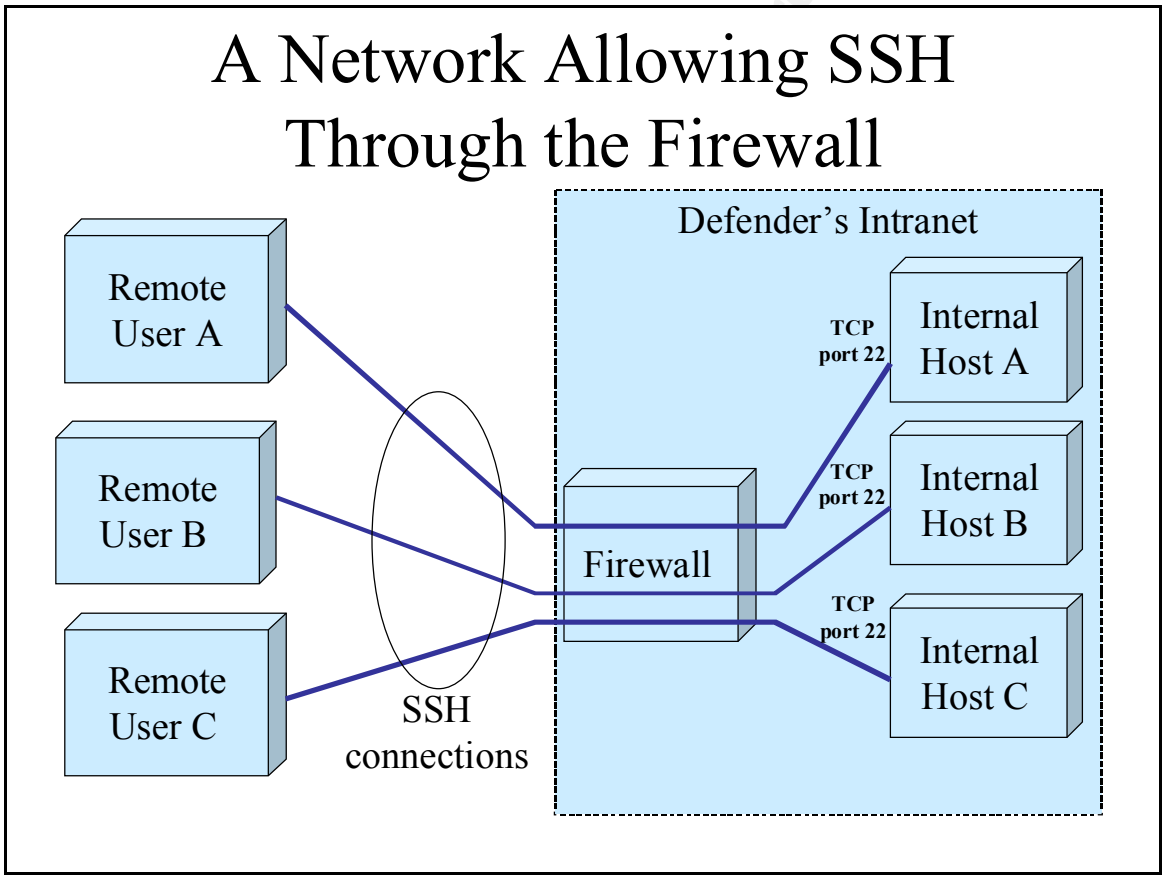
8. Once the malicious command interpreter is running, the attacker can run commands with the same privileges as the SSH server. Typically, administrative access is provided because the SSH server usually runs as "root." A "remote-root" exploit has been accomplished by the attacker.

Visualizing the Exploit in a Network

Many enterprises protect their computing resources by constructing secure perimeter to their network. Special computers called "firewalls" allow authorized data to be exchanged across the perimeter, while preventing malicious connections from outside the perimeter.

Because it is designed to provide secure connections across insecure networks, some enterprises may allow SSH connections to be made through the firewall. Typically this would be done for the purpose of allowing remote users, such as home users, to use the computers inside the security perimeter of an enterprise.

While most connections would be rejected by the firewall as potential attacks, SSH connections might be considered safe because of the authentication and encryption provided by them.



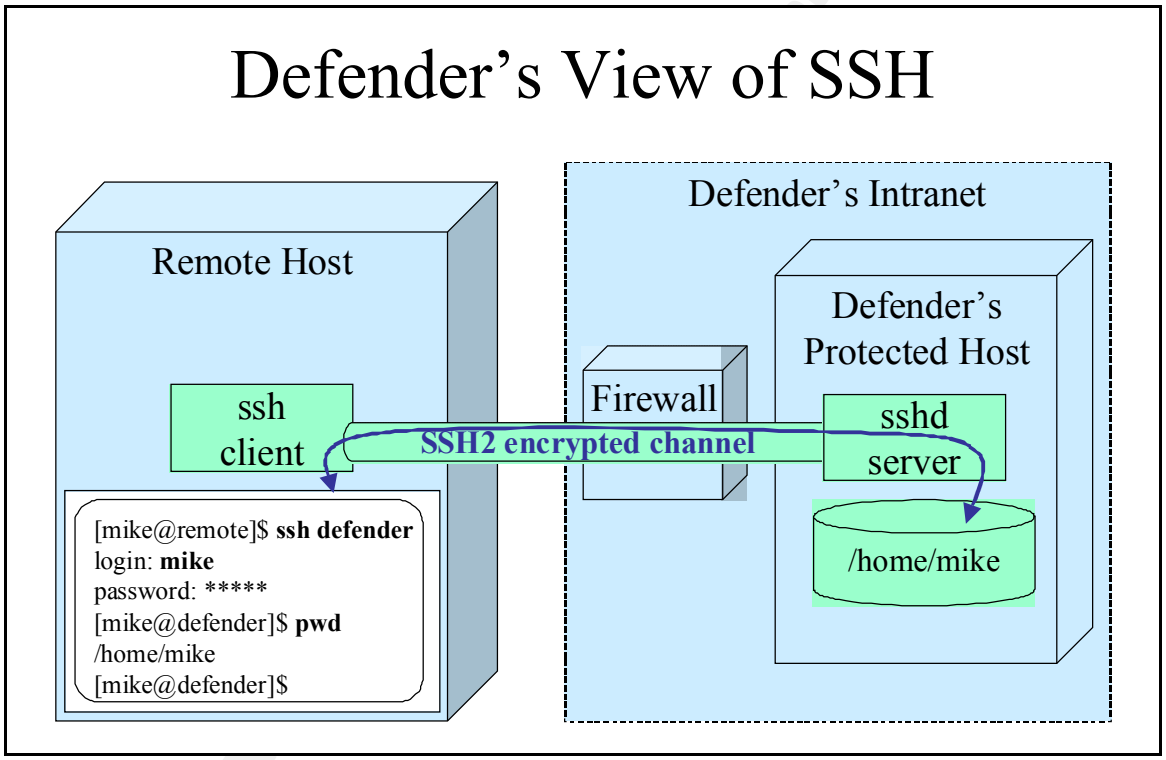
A View From the Defender's Perspective

In a system protected by a firewall and a security perimeter, a trusted remote host might be used by authorized users to connect to computers inside the perimeter.

From the authorized user's perspective, an "encrypted channel" is provided between the remote computer and a computer protected by the firewall.

The encrypted channel is like an armored pipe that does not allow attackers to see or interfere with the traffic flowing through it.

A remote user can use such a connection to issue commands on a computer inside the security perimeter, trusting SSH to prevent eavesdropping of passwords or insertion of unauthorized commands.



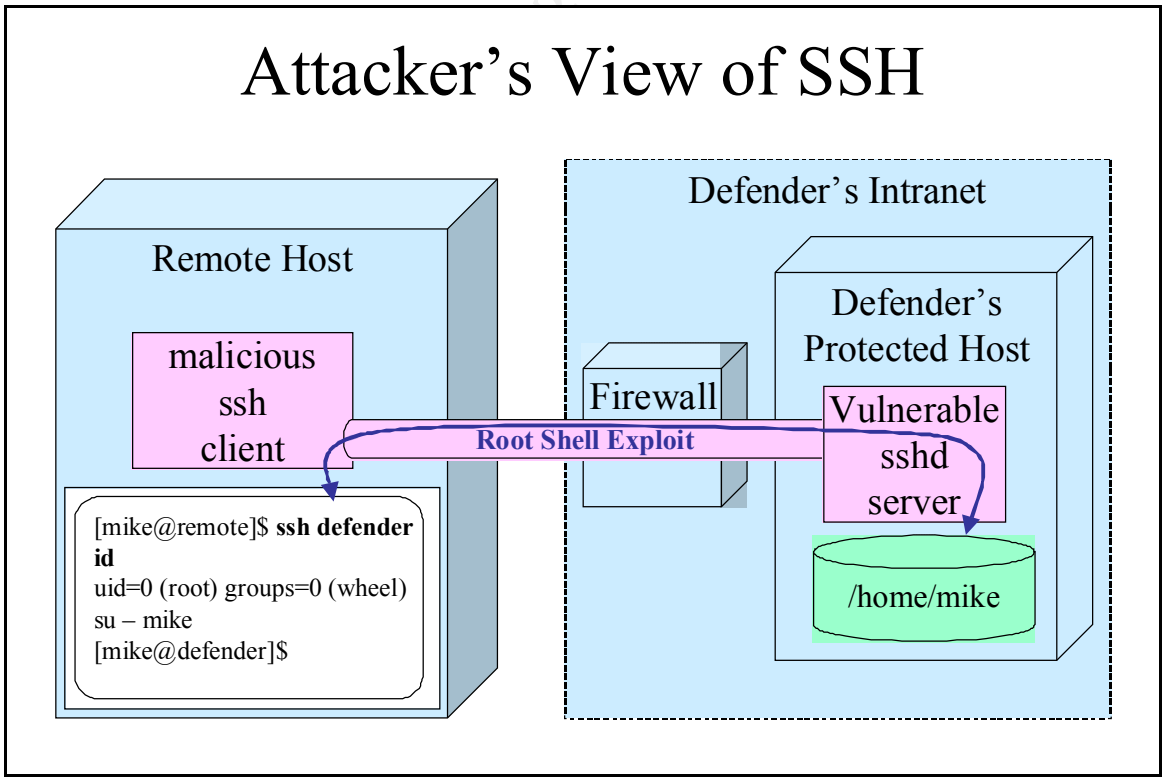
For example, an employee who maintains a computer inside the perimeter may want to connect from home, across the internet, to help solve problems that are reported by users who are working overtime. The firewalls on the perimeter might be set up to allow Secure Shell connections on TCP port 22, while refusing to permit less secure connections. By using a Secure Shell client at home, and a Secure Shell server inside the perimeter, computers inside the perimeter can be safely controlled by the employee from home.

A View From the Attacker's Perspective

If the Secure Shell server is vulnerable, an attacker can use it to gain access through the firewall and attack computers inside the perimeter. Once the attacker connects with one computer inside the perimeter, other computing resources inside the perimeter can be attacked without overcoming the obstacle presented by the firewall.

The Secure Shell server typically has administrative privileges in order to carry out its authentication functions, such as verifying passwords. In these typical cases, when the vulnerable OpenSSH server is attacked with the GOBBLES exploit, it provides interactive access with administrative privileges.

An attacker that has administrative privileges inside the security perimeter of an enterprise is in a very good position to compromise many other computers inside the perimeter. Typically, administrative users can change passwords, create and destroy new user accounts, and take on the identity of existing users. The attacker can eavesdrop on other users in the network, especially administrative users, and expand their own scope of administrative access by taking on the identity of other administrative users. Even if other computers inside the perimeter have strong security, the attacker can use the compromised computer to strenuously attack them.

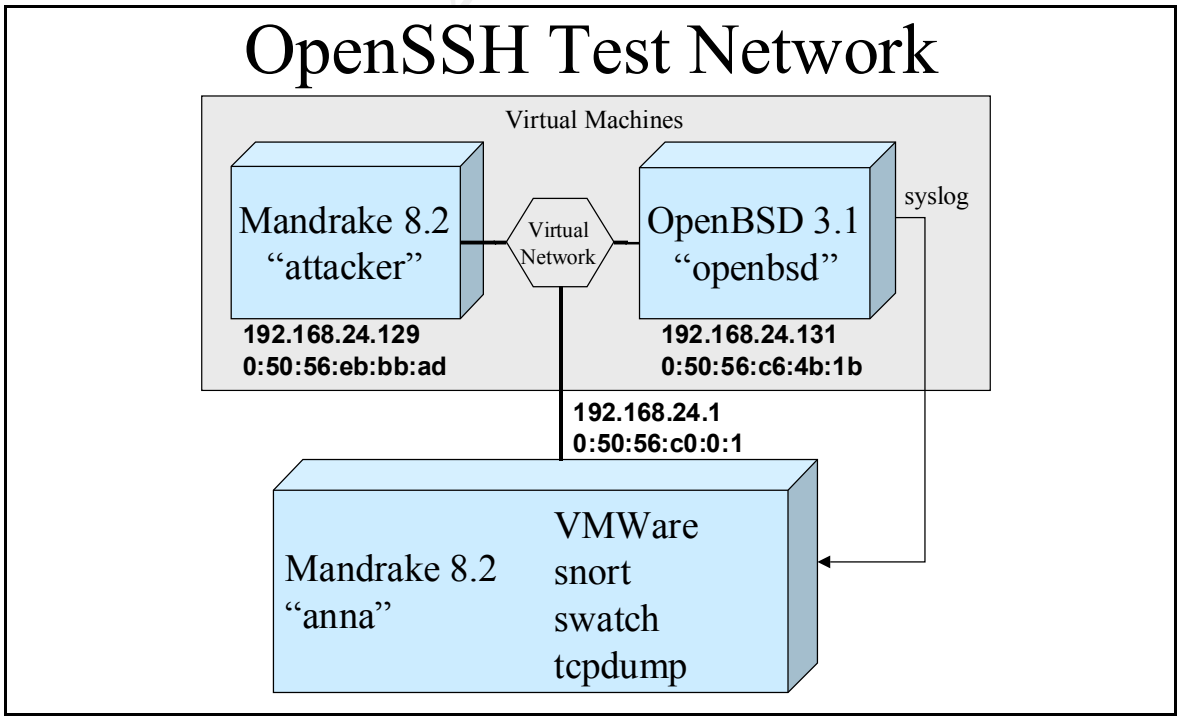


Running the GOBBLES Exploit in a Test Network

To analyze the GOBBLES exploit, an isolated laboratory network was constructed using the following components:

- A "White Box" AMD Duron processor (an x86 architecture computer), which ran the following applications:
    - Mandrake (version 8.2) Linux (http://www.mandrakelinux.com/en/)
    - VMWare Workstation (http://www.vmware.com/products/desktop/ws_features.html)
    - SNORT Intrusion Detection System (Caswell, Roesch, http://www.snort.org)
    - SWATCH log monitoring utility (Atkins; http://www.oit.ucsb.edu/~eta/swatch/)
    - TCPDUMP utility (provided with Mandrake)
- A VMWare virtual machine running Mandrake 8.2 as the "attacker." This machine ran the malicious SSH client implementing the GOBBLES exploit.
- Another VMWare virtual machine running OpenBSD version 3.1 Unix (de Raadt, http://openbsd.groupbsd.org/31.html) as the "defender." This machine ran the vulnerable OpenSSH server.

The "White Box" Mandrake platform was also used to compile the malicious GOBBLES client after applying the GOBBLES patch to OpenSSH 3.4p1 source code.



Screen Shot of a Successful Attack

The following steps reproduced a successful attack using the GOBBLES exploit:

1. Retrieve the source code for OpenSSH version 3.4p1 (ftp://ftp.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-3.4p1.tar.gz)
2. Apply the patch provided by GOBBLES Security (http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz)
3. Compile the OpenSSH source code, producing the malicious SSH client.
4. Transfer the malicious client to the account "mike" on the virtual "attacker" computer.
5. Boot OpenBSD 3.1, which contains the vulnerable OpenSSH server, on the virtual "defender" computer.
6. Invoke the malicious SSH client on the "attacker" computer, requesting a connection to the "defender" computer, by using its IP address.
7. Exercise the resulting command interpreter to examine the access privileges obtained by the exploit.

```
[mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
*GOBBLE*
OpenBSD openbsd 3.1 GENERIC#59 i386
uid=0(root) gid=0(wheel) groups=0(wheel), 2(kmem), 3(sys), 4(tty),
5(operator), 20(staff), 31(guest)
id
uid=0(root) gid=0(wheel) groups=0(wheel), 2(kmem), 3(sys), 4(tty),
5(operator), 20(staff), 31(guest)
uname -a
OpenBSD openbsd 3.1 GENERIC#59 i386
pwd
/
env
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/usr/local/bin
echo $$
12565
ps 12565
  PID TT   STAT      TIME COMMAND
12565 ??  I       0:00.44 //bin/sh
^C
[mike@attacker mike]$
```

In the example session above, modified ssh client displays some progress messages, a success indicator "*GOBBLE*," followed by the output from the 'id' and 'uname –a' commands executed on the attacked host. A prompt-less shell

connection with root privileges is established. A control-C "quit" character terminates the connection.

In this experimental attack, Unix commands illustrate that the attacker has a command interpreter (//bin/sh) and has administrative access (uid=0).

Snooped Packets From an Attack

Appendix C, "Packet Trace of an Attack," details the network traffic exchanged between a malicious SSH client and vulnerable SSH server during an attack. Here is a summary of the traffic in notional form:

The TCP Handshake

Client sends a TCP Syn Packet	→	
	←	Server sends a TCP Syn-Ack Packet
Client sends a TCP Ack Packet	→	

SSH Protocol Version Exchange

	←	The SSH server sends its version identification string, for example, "SSH-1.99-OpenSSH_3.2"
The client sends its identification string, for example, "SSH-2.0-GOBBLES"	→	

Algorithm Negotiation

	←	The server sends an SSH-MSG_KEXINIT message, listing the encryption, authentication, and compression algorithms that it supports.
The client sends an SSH-MSG_KEXINIT message, listing the encryption, authentication, and compression algorithms that it supports.	→	

Diffie-Hellman Key Exchange

The client initiates the key eschange SSH_MSG_KEY_DH_GEX_REQUEST	→	
	←	The server sends mathematical values for the key exchange procedure. SSH_MSG_KEX_DH_GEX_GROUP
The client sends mathematical values for the key exchange procedure SSH_MSG_KEX_DH_GEX_INIT	→	
	←	The server sends mathematical values for the key exchange procedure. SSH_MSG_KEX_DH_GEX_REPLY
The client indicates that it is ready to begin using encryption using the keys that were just exchanged. SSH_MSG_NEWKEYS	→	

Challenge-Response Authentication

The client requests authentication service: SSH-MSG_SERVICE_REQUEST	→	
	←	The server acknowledges the authentication request: SSH-MSG_SERVICE_ACCEPT
The client sends a request for the "none" method of authentication, to obtain the list of acceptable authentication methods: SSH_MSG_USERAUTH_REQUEST	→	
	←	The server lists the acceptable authentication methods while rejecting the "none" method: SSH_MSG_USERAUTH_FAILURE

The client requests authentication using host-based “keyboard interactive” authentication SSH_MSG_USERAUTH_REQUEST	→	
	←	The server sends a “Password” prompt: SSH_MSG_USERAUTH_INFO_REQUEST

The Attack

Instead of sending back a password, the client sends back a misleading count of replies, a large number of replies containing malicious code, and an intentionally erroneous string to trigger a compromised error-handling routine: SSH_MSG_USERAUTH_INFO_RESPONSE	→	The vulnerable server executes the malicious command interpreter in the attacker’s authentication response.
	←	The malicious command interpreter sends a signal to the malicious SSH client program: "GGGG"
The malicious client program sends a signal to the command interpreter: "O"	→	
	←	The command interpreter sends a prompt indicating a successful attack: "*GOBBLE*"
The malicious client sends commands to the compromised server: ”uname.-a;id”	→	
	←	The compromised server sends the results of the malicious commands: ”OpenBSD openbsd.3.1 GENERIC#59 i386”
The malicious client can now continue to execute code on the computer running the vulnerable SSH server.		

Log Entries From a Successfully Attacked System

The following log entries were generated on the system running the vulnerable SSH server program:

```
Aug 17 08:54:44 openbsd sshd[5383]: Connection from 192.168.24.129 port
1062
Aug 17 08:54:44 openbsd sshd[5383]: Enabling compatibility mode for
protocol 2.0
Aug 17 08:54:45 openbsd sshd[5383]: Failed none for mike from
192.168.24.129 port 1062 ssh2
Aug 17 08:54:45 openbsd sshd[5383]: Postponed keyboard-interactive for
mike from 192.168.24.129 port 1062 ssh2
Aug 17 08:54:45 openbsd sshd[5383]: fatal: buffer_get_string: bad
string length 263168
```

The "Failed none for mike" message was generated while the authentication protocol was being negotiated.

The "fatal: buffer_get_string: bad string length" message was generated as the compromised SSH server began to execute the malicious code.

Can the Exploit be Done Without Automation?

In a practical sense, the exploit requires automation to accomplish, primarily for two reasons:

- The negotiation of the encryption algorithms, signatures, and keys requires difficult arithmetic and the construction of complex packets,
- Once the encryption algorithms have been initialized, the encrypted traffic of the Transport Layer Protocol also requires difficult arithmetic.

Attempting to accomplish the task with manual arithmetic and construction of binary packets would be intractable.

Signatures of the Attack

An client attacking an SSH server using the GOBBLES exploit generates a small amount of distinctive traffic. It also generates at least one specific message in the system log.

Recognizing an Attempted Attack

As distributed by GOBBLES Security, the malicious client sends an SSH identification string containing the string "GOBBLES". This string is always the third packet from the client, starting with the TCP SYN packet that requests the connection with the server:

Client sends a TCP Syn Packet	→	
	←	Server sends a TCP Syn-Ack Packet
Client sends a TCP Ack Packet	→	
	←	"SSH-1.99-OpenSSH_3.2"
"SSH-2.0-GOBBLES"	→	

This string appears in the the GOBBLES patch to the OpenSSH source code. It is easy for an attacker to change the source code to remove the string "GOBBLES" and replace it with a string from an uncompromised client. For this reason, this is not a reliable attack signature.

The following message will appear in the system log as a result of an attempted attack:

```
Aug 17 08:54:45 openbsd sshd[5383]: fatal: buffer_get_string: bad
string length 263168
```

This message appears because of the erroneous string length at the end of the malicious response message sent by the attacking SSH client. This bad string length, that is used to trigger the malicious code, also triggers the log message. Messages such as this one, appearing in the system log, suggest that a GOBBLES attack may be underway.

Recognizing a Successful Attack

The command interpreter contained in a malicious client constructed from the GOBBLES patch will generate the following recognizable packets:

… Normal SSH Traffic …		
	←	"GGGG"
"O"	→	
	←	"*GOBBLE*"
uname.-a;id	→	

The "GGGG", "O", and "*GOBBLE*" packets do not appear in the source code of the GOBBLES exploit. They are generated by machine code that is coded into the GOBBLES patch as hexadecimal strings. Neither do the hexadecimal equivalents of these strings appear in the source code. This makes it very difficult for an attacker to change the strings and avoid generating the signature.

These strings would never appear in normal SSH traffic, due to the encryption of user data and the wrapping of the SSH messages by the binary packet protocol.

For these reasons, these strings can form a convenient, reliable signature of a successful attack. When observing the strings "GGGG" or "*GOBBLE*" in

packets from TCP port 22, or the string “O” in packets to TCP port 22, one can reasonably assume that the associated SSH server has been compromised.

The “uname –a; id” string appears in the source code of the GOBBLES patch. Because an attacker can change or remove the string in the source code, it may not be present in an successful attack.

Sniffer Rules for Recognizing an Attack

A class of tools known as Intrusion Detection Systems (IDS) scan network traffic and monitor log files for signs of suspicious activity.

The SNORT intrusion detection system (Caswell, Roesch, http://www.snort.org) is a popular tool for scanning network traffic. It provides a language for specifying signatures of suspicious traffic and generating alert messages when a signature is recognized. It provides numerous alternatives for propagating the alert messages to logfiles, as well as to human interfaces, such as email messages and popup messages in interactive sessions.

Signatures in the SNORT IDS database

The website for the Snort IDS includes a database of attack signatures for numerous publicized exploits. The following signatures have been published in the Snort Signature Database for detection of the GOBBLES OpenSSH exploit (Caswell, Roesch, http://www.snort.org/snort-db).

SID	1810	**message**	EXPERIMENTAL MISC successful gobbles ssh exploit (GOBBLE)
Signature	alert tcp $HOME_NET 22 -> $EXTERNAL_NET any (msg:"EXPERIMENTAL MISC successful gobbles ssh exploit (GOBBLE)"; flow:from_server,established; content:"|2a|GOBBLE|2a|"; reference:bugtraq,5093; classtype:misc-attack; sid:1810; rev:1;)		

Signature 1810 detects outgoing SSH packets with the characters “*GOBBLE*” as the data. This signature does not appear in the source code for the exploit. Apparently, it is generated by the binary shellcode embedded in the exploit source code. This makes it a desirable signature because a casual programmer can’t edit the source code of the exploit to remove the signature.

SID	1811	message	EXPERIMENTAL MISC successful gobbles ssh exploit (uname)
Signature	alert tcp $HOME_NET 22 -> $EXTERNAL_NET any (msg:"EXPERIMENTAL MISC successful gobbles ssh exploit (uname)"; flow:from_server,established; content:"uname"; reference:bugtraq,5093; classtype:misc-attack; sid:1811; rev:1;)		

Signature 1811 detects incoming SSH packets that contain the characters "uname" in the data. The exploit sends the command string "uname –a; id" as the first command to the malicious shellcode. While this signature will detect exploits constructed from the published source code for the exploit, it is easy to modify the published exploit to change or remove the initial commands. For this reason, this signature can be easily circumvented.

SID	1812	message	EXPERIMENTAL MISC gobbles SSH exploit attempt
Signature	alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPERIMENTAL MISC gobbles SSH exploit attempt"; flow:to_server,established; content:"GOBBLES"; reference:bugtraq,5093; classtype:misc-attack; sid:1812; rev:1;)		

Signature 1812 detects incoming SSH packets that contain the characters "GOBBLES" in the SSH-2 Protocol Version Exchange. Since these characters appear as a string in the source code for the exploit, it is easy to circumvent this signature by editing the string while constructing the malicious client.

The following SNORT alerts were generated with rules similar to those above:

```
07/30-12:04:38.432913  [**] [1:0:0] EXPERIMENTAL MISC gobbles SSH
exploit attempt [**] {TCP} 192.168.24.129:1031 -> 192.168.24.131:22
07/30-12:04:39.381008  [**] [1:0:0] EXPERIMENTAL MISC successful
gobbles ssh exploit (GOBBLE) [**] {TCP} 192.168.24.131:22 ->
192.168.24.129:1031
07/30-12:04:39.425051  [**] [1:0:0] EXPERIMENTAL MISC successful
gobbles ssh exploit (uname) [**] {TCP} 192.168.24.129:1031 ->
192.168.24.131:22
```

Additional SNORT IDS Signatures

The following snort rules trigger on signatures that detect additional plaintext traffic between the malicious client and the compromised server.

The first "alert" is issued when a packet containing the characters "GGGG" is transmitted from server to client at the beginning of the clandestine shell session.

The second “alert” is issued when a packet containing the single letter “O” is returned.

```
alert tcp any 22 -> any any (\
        msg:"Mike's gobbles SSH exploit server-side signature"; \
        content:"GGGG"; \
        dsize:4; offset:0; depth:4; \
        session: printable; \
          )
alert tcp any any -> any 22 (\
        msg:"Mike's gobbles SSH exploit client-side signature"; \
        content:"O"; \
        dsize:1; offset:0; depth:1; \
        session: printable; \
        )
```

These signatures do not appear in the published source code for the exploit, so they should form a reliable signature that the exploit is being attempted in the network being monitored.

The following snort alerts appear when using the above rules during an attack:

```
07/30-12:04:39.367965  [**] [1:0:0] Mike's gobbles SSH exploit server-
side signature [**] {TCP} 192.168.24.131:22 -> 192.168.24.129:1031
07/30-12:04:39.369812  [**] [1:0:0] Mike's gobbles SSH exploit client-
side signature [**] {TCP} 192.168.24.129:1031 -> 192.168.24.131:22
```

System Log Messages Issued by the SSH Server Program

The SSH daemon, sshd, logs messages to the Unix system log as connections are requested. (On an OpenBSD system, the messages from sshd typically appear in both of two logfiles: /var/log/messages and /var/log/authlog.)

Normal Log Messages

The following message appears when a normal connection is established:

```
Aug 17 08:40:06 openbsd sshd[12223]: Accepted password for mike from
192.168.24.129 port 1059
```

The “LogLevel” option can be set to “VERBOSE” in the configuration file for sshd (typically /etc/ssh/sshd_config). Additional messages appear during a successful login:

```
Aug 17 08:53:14 openbsd sshd[22637]: Connection from 192.168.24.129
port 1061
Aug 17 08:53:18 openbsd sshd[22637]: Accepted password for mike from
192.168.24.129 port 1061
```

Suspicious Log Messages

In some settings, it may be inappropriate for SSH connections to be accepted from unrecognized computers. When a login is requested from a host with no known hostname, the following message appears in the system log:

```
Aug 17 08:53:18 openbsd sshd[17115]: Could not reverse map address
192.168.24.129.
```

Attacks utilizing the GOBBLES exploit trigger additional messages. Without the VERBOSE option, the following message appears as a result of a successful GOBBLES attack:

```
Aug 17 08:42:36 openbsd sshd[14843]: fatal: buffer_get_string: bad
string length 263168
```

The "bad string length" message is triggered just prior to the invocation of the malicious GOBBLES command shell. As server recognizes the error, it prints the message just before it attempts to call its error recovery routines via the contaminated function pointers.

With the VERBOSE option, logging from an attacked SSH server appears as follows:

```
Aug 17 08:54:44 openbsd sshd[5383]: Enabling compatibility mode for
protocol 2.0
Aug 17 08:54:45 openbsd sshd[5383]: Failed none for mike from
192.168.24.129 port 1062 ssh2
Aug 17 08:54:45 openbsd sshd[5383]: Postponed keyboard-interactive for
mike from 192.168.24.129 port 1062 ssh2
Aug 17 08:54:45 openbsd sshd[5383]: fatal: buffer_get_string: bad
string length 263168
```

Additional activities prior to the activation of the GOBBLES command shell are recorded when the server is using VERBOSE logging.

Log Scanning Rules for Recognizing an Attack

A popular log-scanning tool is SWATCH, the Simple Watcher. (Atkins, http://www.oit.ucsb.edu/~eta/swatch/)

SWATCH uses a rules file for instructions on which messages should trigger an action, and what actions to take for each message. Here is an example of a few SWATCH rules for detecting key system log messages:

```
watchfor /sshd\[[0-9]+\]: fatal: buffer_get_string: bad string length/
echo continue
watchfor /sshd\[[0-9]+\]: Postponed keyboard-interactive/
echo continue
watchfor /sshd\[[0-9]+\]: Failed none for/
echo continue
```

Here is sample invocation of these rules, and the filtered messages during an attack:

```
root@anna swatch]# swatch --config-file=rules --tail-
file=/var/log/messages

*** swatch-3.0.2 (pid:3892) started at Sat Aug 17 18:45:36 PDT 2002

Aug 17 10:34:59 openbsd sshd[21309]: Failed none for mike from
192.168.24.129 port 1074 ssh2
Aug 17 10:34:59 openbsd sshd[21309]: Postponed keyboard-interactive for
mike from 192.168.24.129 port 1074 ssh2
Aug 17 10:34:59 openbsd sshd[21309]: fatal: buffer_get_string: bad
string length 263168
```

SWATCH provides actions that send email alerts or write messages to logged-in users, so that attention may promptly be focused on the attack. Details are provided on the SWATCH homepage at http://www.oit.ucsb.edu/~eta/swatch .

Protecting Against the GOBBLES Exploit

These alternatives are available for protecting against the OpenSSH Challenge-Response vulnerabilities:

- Disable SSH entirely

  Advantage: All present and future SSH vulnerabilities are eliminated.

  Disadvantage: No access to the platform without using less secure protocols (eg. telnet, rlogin, X-windows).

- Disable SSH-2 protocol and use SSH-1

  Advantage: Eliminates the vulnerability by removing the Challenge-Response capability.

  Disadvantage: SSH-1 has well-known, inherent design vulnerabilities that were corrected in SSH-2.

- Disable Challenge-Response Authentication

  Advantage: Eliminates the vulnerability by disabling the code in the server program.

  Disadvantage: Some forms of authentication become unavailable, such as s-key authentication.

- Enable Privilege Separation

  Advantage: Mitigates vulnerabilities by severely restricting privileges to a successful attacker.

  Disadvantage: Permits the attacker to connect to the platform, which could assist in exploitation of other vulnerabilities.

- Implement Intrusion Detection Rules that Shut Down Attempted Attacks

Advantage: Protects vulnerable SSH servers that may be present on the network.

Disadvantage: Provides the attacker with information for mapping the network.

- Update SSH to a repaired release

  Advantage: Eliminates known vulnerabilities.

  Disadvantage: Potential vulnerability from as-yet undiscovered defects.

The most practical approach is a combination of Privilege Separation and changing to an up-to-date OpenSSH release. The explicit purpose of Privilege Separation is to limit the risk from undiscovered defects in the SSH protocol and/or implementation. Keeping up-to-date as newly discovered vulnerabilities are repaired is essential to good security, as long as the new releases can be verified as authentic and well tested. Privilege separation can buy the time needed to obtain and test repaired releases as future vulnerabilities are discovered.

Disabling Challenge-Response Authentication

This workaround will disable the vulnerable component of the SSH server program.

From the Internet Security Systems Advisory (http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20584):

> ISS X-Force recommends that system administrators disable unused OpenSSH authentication mechanisms. Administrators can remove this vulnerability by disabling the Challenge-Response authentication parameter within the OpenSSH daemon configuration file. This filename and path is typically: /etc/ssh/sshd_config. To disable this parameter, locate the corresponding line and change it to the line below:
>
> ChallengeResponseAuthentication no
>
> The "sshd" process must be restarted for this change to take effect.

Here is what the attacker sees when Challenge-Response Authentication is disabled:

```
[mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[x] keyboard-interactive method not available
Permission denied (publickey,password).
[mike@attacker mike]$
```

Using Privilege Separation

Niels Provos of the University of Michigan provides a detailed description of how Privilege Separation is designed to work (http://www.citi.umich.edu/u/provos/ssh/privsep.html).

With privilege separation enabled, an attack fails because shellcode in the exploit is isolated in an empty directory. It cannot execute the command shell in the server system because the directory where the command shell program is stored is not visible to the compromised server process.

Specific details about how to use Privilege Separation are found in the README.privsep file of the Portable OpenSSH release 3.4p1 (OpenBSD, 2002) The following procedure is documented there:

```
You should do something like the following to prepare the privsep
preauth environment:

# mkdir /var/empty
# chown root:sys /var/empty
# chmod 755 /var/empty
# groupadd sshd
# useradd -g sshd -c 'sshd privsep' -d /var/empty -s /bin/false sshd

/var/empty should not contain any files.
```

Note: There are compile-time options in OpenSSH that permit variation from the above example. The compile-time defaults for the username, groupname and directory, as shown, can be changed using these options.

Here is what the attacker sees when Privilege Separation is in effect:

```
[mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
*GOBBLE*
echo *
```

Sometimes the attacker may see a dead connection, as above. Other times the connection will be closed by the server, as below:

```
 [mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
Connection closed by 192.168.24.131
[mike@attacker mike]$
```

Fixing the Challenge-Response Buffer Overflow Vulnerability

The "Revised OpenSSH Security Advisory" (OpenSSH, http://www.openssh.org/txt/preauth.adv) contains a patch that repairs the vulnerability in the SSH server program's source code.

The changes are simple:

1. Check for an unreasonable number of responses to the authentication request. In the patch, a limit of 100 responses is implemented.
2. Check that the number of responses returned is the same as the number requested. If the number is different, the response message is rejected.

OpenSSH versions 3.4 and later incorporate these changes in the original source code, so the patch is not necessary for these versions.

From the Internet Security Systems Advisory (http://www.openssh.org/txt/iss.adv):

> X-Force recommends that administrators upgrade to OpenSSH version 3.4 immediately. This version implements privilege separation, contains a patch to block this vulnerability, and contains many additional proactive security fixes.

Blocking the Attack Using Intrusion Detection Rules

The SNORT intrusion detection system provides options for shutting down TCP connections based on patterns matched in the packet data.

The following two snort rules match patterns in an attacker's SSH data stream that would not occur in a normal SSH connection. The "resp: rst_all" options instruct snort to send TCP RESET packets to each side of the attacker's connection. These RESET packets sent by SNORT will close the SSH connection before the malicious command interpreter accepts input from the attacker:

```
alert tcp any 22 -> any any (\
        msg:"Mike's gobbles SSH exploit Server-side SHUTDOWN"; \
        content:"GGGG"; \
        dsize:4; offset:0; depth:4; \
        session: printable; \
        resp: rst_all; \
          )

alert tcp any any -> any 22 (\
        msg:"Mike's gobbles SSH exploit Client-side SHUTDOWN"; \
        content:"O"; \
        dsize:1; offset:0; depth:1; \
        session: printable; \
        resp: rst_all; \
        react: block, msg; \
        )
```

When either or both of these snort rules are applied, the attacker's connection is reset before malicious commands can be issued.

Client sends a "GGGG" Packet or Server sends a "O" Packet	→	IDS monitors the signature, shutdown rule is triggered.
Client Shuts Down TCP connection	←	IDS Spoofs TCP Reset from Server
IDS Spoofs TCP Reset from Client	→	Server shuts down TCP connection

Here is an illustration of what the attacker sees when the the first rule is triggered (the attacker gets an unresponsive connection):

```
[mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
echo a

^C
[mike@attacker mike]$
```

The first rule logs a SNORT alert when the attacker's connection is shut down:

```
08/27-09:39:08.940807  [**] [1:0:0] Mike's gobbles SSH exploit Server-
side SHUTDOWN [**] [Priority: 0] {TCP} 192.168.24.131:22 ->
192.168.24.129:1109
```

Here is a trace of the shutdown activity in the TCP connection. The first packet contains the "GGGG" pattern that signifies a successful attack in progress. Note the MAC addresses of the "openbsd" host and the "attacker" host. In the second and third packets, the MAC address of the host running snort is the originating address, even though the IP addresses of the attacker and openbsd hosts

appear in the packets. The second packet is a TCP Reset sent to the SSH server host, and the third packet is a TCP Reset sent to the attacking SSH client.

```
0:50:56:c6:4b:1b 0:50:56:eb:bb:ad ip 70: openbsd.ssh > attacker.kpop: P
1950:1954(4) ack 20833 win 17376 <nop,nop,timestamp 923266255
315420288> (DF)
0x0000     4500 0038 8c9f 4000 4006 fbcb c0a8 1883 E..8..@.@.......
0x0010     c0a8 1881 0016 0455 dfd2 17ec bdb0 957c .......U.......|
0x0020     8018 43e0 7d72 0000 0101 080a 3707 eccf ..C.}r......7...
0x0030      12cc ee80 4747 4747                     ....GGGG
0:50:56:c0:0:1 0:50:56:c6:4b:1b ip 54: attacker.kpop > openbsd.ssh: R
20833:20833(0) ack 1954 win 0
0x0000     4500 0028 b0f2 0000 ff06 5888 c0a8 1881  E..(......X.....
0x0010     c0a8 1883 0455 0016 bdb0 957c dfd2 17f0  .....U.....|....
0x0020      5014 0000 ae20 0000                      P.......
0:50:56:c0:0:1 0:50:56:eb:bb:ad ip 54: openbsd.ssh > attacker.kpop: R
1950:1950(0) ack 20837 win 0
0x0000     4500 0028 b0f2 0000 ff06 5888 c0a8 1883  E..(......X.....
0x0010     c0a8 1881 0016 0455 dfd2 17ec bdb0 9580  .......U........
0x0020      5014 0000 ae20 0000                      P.......
```

Note the MAC addresses of the "openbsd" host (**0:50:56:c6:4b:1b**) and the "attacker" host (**0:50:56:eb:bb:ad**). These are the addresses that each network adapter uses on the local network, and they correspond to the IP addresses for the "openbsd" and "attacker" hosts, respectively.

In the last two packets, the MAC address of the host running snort (**0:50:56:c0:0:1**) is the originating address, even though the IP addresses of the attacker and openbsd hosts appear in the packets. This is known as "spoofing," and it is typically considered malicious. However, in this case, spoofing is being used by the SNORT intrusion detection system to fool the malicious client and the compromised server into shutting down the attacker's connection.

Here is an illustration of what the attacker sees when the second rule triggers. The attacker gets the "Connection reset by peer" message, due to the TCP reset sent from the intrusion detection host:

```
[mike@attacker mike]$ ./ssh 192.168.24.131
[*] remote host supports ssh2
[*] server_user: mike:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
read(): Connection reset by peer
[mike@attacker mike]$
```

The second rule also logs alerts when the attacker's connection is shut down:

```
08/27-09:36:03.233435  [**] [1:0:0] Mike's gobbles SSH exploit Client-
side SHUTDOWN [**] [Priority: 0] {TCP} 192.168.24.129:1108 ->
192.168.24.131:22
```

Here is a trace of the shutdown activity in the TCP connection. The first packet contains the "GGGG" pattern that signifies a successful attack in progress. The second packet contains the "O" pattern that signifies the client-side response to the "GGGG" packet. This is the pattern that triggers the reset rule. The third packet is a spoofed TCP Reset sent to the attacking client, and the fourth packet

is a spoofed TCP Reset sent to the attacked server. (Either packet alone is sufficient to shut down the attacker's connection.)

```
0:50:56:c6:4b:1b 0:50:56:eb:bb:ad ip 70: openbsd.ssh > attacker.1108: P
1950:1954(4) ack 20833 win 17376 <nop,nop,timestamp 923265898
315402321> (DF)
0x0000     4500 0038 77b4 4000 4006 10b7 c0a8 1883 E..8w.@.@.......
0x0010     c0a8 1881 0016 0454 d25f 73c4 b316 580a .......T._s...X.
0x0020     8018 43e0 beae 0000 0101 080a 3707 eb6a ..C.........7..j
0x0030     12cc a851 4747 4747                     ...QGGGG
0:50:56:eb:bb:ad 0:50:56:c6:4b:1b ip 67: attacker.1108 > openbsd.ssh: P
20833:20834(1) ack 1954 win 8832 <nop,nop,timestamp 315402325
923265898> (DF)
0x0000     4500 0035 a172 4000 4006 e6fb c0a8 1881 E..5.r@.@.......
0x0010     c0a8 1883 0454 0016 b316 580a d25f 73c8 .....T....X.._s.
0x0020     8018 2280 1f98 0000 0101 080a 12cc a855 .."............U
0x0030     3707 eb6a 4f                            7..jO
0:50:56:c0:0:1 0:50:56:eb:bb:ad ip 54: openbsd.ssh > attacker.1108: R
1954:1954(0) ack 20834 win 0
0x0000     4500 0028 b0f2 0000 ff06 5888 c0a8 1883 E..(......X.....
0x0010     c0a8 1881 0016 0454 d25f 73c8 b316 580b .......T._s...X.
0x0020     5014 0000 a7c7 0000                     P.......
0:50:56:c0:0:1 0:50:56:c6:4b:1b ip 54: attacker.1108 > openbsd.ssh: R
20833:20833(0) ack 1955 win 0
0x0000     4500 0028 b0f2 0000 ff06 5888 c0a8 1881 E..(......X.....
0x0010     c0a8 1883 0454 0016 b316 580a d25f 73c9 .....T....X.._s.
0x0020     5014 0000 a7c7 0000                     P.......
```

Conclusions

The Challenge-Response Authentication vulnerability, exploited by the GOBBLES Security patch to the SSH client, represents a severe vulnerability for the following reasons:

1. It provides a "remote root exploit," where an attacker does not need to have a local account in the network to be attacked, and a successful attack provides privileged access to the vulnerable computer.
2. A security-related service is vulnerable. Secure Shell is typically applied to protect connections over hostile networks. A vulnerable server program will expose a computer that it should be protecting.

The appearance of this vulnerability and its exploits illustrates the importance of monitoring online trade journals and vulnerability alerts, and responding promptly to repair reported vulnerabilities. With the exception of the privilege separation feature of OpenSSH, the repairs, workarounds, and intrusion detection methods documented in this paper would not have become apparent without trade reports on the vulnerability and its countermeasures.

The vulnerability also illustrates the value of early implementation of precautionary features, such as privilege separation. Good security demands that countermeasures be implemented as early as possible, preferably in advance of the exploits that they prevent.

Appendix A: Source Code / Pseudo Code

The exploit is well-documented in the source code patch package provided by GOBBLES (GOBBLES Security http://online.securityfocus.com/archive/attachment/280029/2/sshutup-theo.tar.gz)

Source Code for the Exploit

The OpenSSH Source Code version 3.4p1 provides the base for constructing the exploit. It can be obtained from the OpenSSH website, by file transfer protocol ftp://ftp.openbsd.org/pub/OpenBSD/OpenSSH/portable/

The patch can be obtained from the SecurityFocus website, at http://online.securityfocus.com/bid/5093/exploit/

Here is a direct link to an archived copy of the exploit:

http://online.securityfocus.com/data/vulnerabilities/exploits/sshutup-theo.tar.gz

Pseudo Code Illustrating the Vulnerable Challenge-Response Protocol

construct a USERAUTH_INFO_REQUEST message, containing a requested number of user inputs (one), and a password prompt.

send the USERAUTH_INFO_REQUEST message.

receive a USERAUTH_INFO_RESPONSE message, containing a count of the number of responses, and a list of separate responses. Each response has a string length and a string of the specified length.

Allocate a pointer array for the number of responses in the USERAUTH_INFO_RESPONSE message.

For each response in the USERAUTH_INFO_RESPONSE message:

> Examine the string length and allocate a buffer for the string.
>
> Check the number of characters provided in the USERAUTH_INFO_RESPONSE message.
>
> If more characters are provided than indicated by the string length in the message, call an error-handling routine to reject the response.
>
> Otherwise, link the next pointer in the pointer array to the buffered string.

Call an authentication routine that will check the password provided in the response.

If authentication succeeds, proceed with the setup of the secure session,

otherwise reject the authentication attempt.

Pseudo Code Illustrating the Exploit

Initiate an SSH-2 session by sending an identification string to port 22 on the host running the server.

Receive the server's identification string.

Switch to binary packet protocol.

Send a list of supported key exchange, encryption, compression, and message authentication protocols to the server.

Receive a list of supported key exchange, encryption, compression, and message authentication protocols from the server.

Select a common protocol and exchange encryption keys with the server.

Begin using encryption in the binary packet protocol.

Send an authentication request with the "none" authentication method. The server will reject this initial request, and send a list of supported authentication methods.

Receive a list of supported authentication methods from the server.

Send an authentication request for the Keyboard-Interactive method.

Receive an authentication information request containing a number (1) of password prompts and an expected number (1) of replies.

Construct a malicious response with the following contents:

1. A count of replies that is inordinately large. It is designed to cause an integer overflow when the server computes the storage required for a list of pointers to the replies. Because of the overflow, the server will allocate space for a small number of pointers, but will prepare to read in the larger number of replies.
2. A list of reply strings, each of which contains the malicious code that invokes a command shell over the SSH connection. There need only be enough reply strings to cause one of the string pointers to overflow the server's pointer array, and then to overwrite a function pointer in the server's program space.
3. A reply string with a length that is too short for the number of characters. The server will detect this potential overflow, and use an overwritten function pointer to call the malicious code.

Send the malicious response to the server, triggering the malicious command shell contained within the response.

Loop until the malicious client is interrupted:

4. Read a command from the malicious ssh client session
5. Send the command to the malicious (remote) command shell
6. Receive the results of the command from the malicious command shell
7. Display the received results

Appendix B: Additional Information

More Information on the Challenge-Response Buffer Overflow Exploit

The OpenSSH Security Advisory (4th revision) is at http://www.openssh.com/txt/preauth.adv

This advisory contains patches to the auth2-chall.c and auth2-pam.c modules of OpenSSH.

The Internet Security Systems Security Advisory (June 26, 2002) is at ether of these locations:

http://online.securityfocus.com/archive/1/278818/2002-06-23/2002-06-29/0

http://www.openssh.com/txt/iss.adv

This advisory contains a detailed description of the vulnerability and how to defend against it.

An early discussion of the vulnerability of OpenSSH v3.3, by developer Theo de Raadt, is at

http://marc.theaimsgroup.com/?l=openssh-unix-dev&m=102495293705094&w=2

Mr. de Raadt commented on the challenges of maintaining the authentication code in OpenSSH, as well as his experiences in notifying vendors in advance of the public announcement of the vulnerabilities.

An excellent illustrationof Privilege Separation concept, by Niels Provos of the University of Michigan, can be found at

http://www.citi.umich.edu/u/provos/ssh/privsep.html

A particularly succinct procedure for enabling privilege separation, by Nathan Ryan Milford, is at

http://marc.theaimsgroup.com/?l=openbsd-misc&m=102507468017147&w=2

Where to Obtain the Exploit

The GOBBLES exploit, illustrated in detail in this paper, can be obtained from the SecurityFocus website, at http://online.securityfocus.com/bid/5093/exploit/

The above page also includes a link to a related exploit.

Here is a direct link to a copy of the exploit:

http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz

Appendix C: Packet Trace of an Attack

This section contains an annotated TCPDUMP trace of a successful attack using the GOBBLES exploit.

Packets exchanged prior to the malicious challenge-response sequence:

The TCP 3-way handshake

The attacking connection begins with a standard TCP connection handshake.

```
attacker.1031 > openbsd.ssh: S 3603243973:3603243973(0) win 5840 <mss
1460,sackOK,timestamp 81021658 0,nop,wscale 0> (DF)
openbsd.ssh > attacker.1031: S 263199839:263199839(0) ack 3603243974
win 17376 <mss 1460,nop,nop,sackOK,nop,wscale 0,nop,nop,timestamp
1197394595 81021658> (DF)
attacker.1031 > openbsd.ssh: . ack 1 win 5840 <nop,nop,timestamp
81021658 1197394595> (DF)
```

Protocol Version Exchange

The server sends the identification string "SSH-1.99-OpenSSH_3.2" and the attacker's TCP acknowlegement is returned. (Subsequent TCP acknowledgement packets will be omitted from this appendix.)

```
openbsd.ssh > attacker.1031: P 1:22(21) ack 1 win 17376
<nop,nop,timestamp 1197394595 81021658> (DF)
0x0030 04d4 4ada 5353 482d 312e 3939 2d4f 7065  ..J.SSH-1.99-Ope
0x0040 6e53 5348 5f33 2e32 0a                   nSSH_3.2.
attacker.1031 > openbsd.ssh: . ack 22 win 5840 <nop,nop,timestamp
81021660 1197394595> (DF)
```

The attacker sends the identification string "SSH-2.0-GOBBLES"

```
attacker.1031 > openbsd.ssh: P 1:17(16) ack 22 win 5840
<nop,nop,timestamp 81021668 1197394595> (DF)
0x0030 475e caa3 5353 482d 322e 302d 474f 4242  G^..SSH-2.0-GOBB
0x0040 4c45 530a                                LES.
```

Algorithm Negotiation

The client and server begin using the SSH-TRANS binary packet protocol. The first byte of each packet is a "Message Number" that indicates the function of the packet in the SSH-2 protocol.

Using the binary packet protocol, the client and server begin to negotiate encryption, compression, and authentication protocols.

The server starts key exchange by sending an SSH_MSG_KEXINIT message, containing its list of supported algorithms.

The following packet indicates the preferences:

```
Key algorithm:                      diffie-hellman-group-exchange-sha1
Server host key algorithm:                ssh-rsa
Encryption Algorithm, client-to-server:   faes128-cbc
Encryption Algorithm, server-to-client:   faes128-cbc
MAC Algorithm, client-to-server:          Uhmac-md5
MAC Algorithm, server-to-client:          Uhmac-md5
Compression Algorithm, client-to-server:  none
Compression Algorithm, server-to-client:  none
Languages, client-to-server:              (none specified)
Languages, server-to-client:              (none specified)
"first_kex_packet_follows":               false
```

```
openbsd.ssh > attacker.1031: P 22:566(544) ack 17 win 17376
<nop,nop,timestamp 1197394596 81021668> (DF)
0x0030 04d4 4ae4 0000 021c 0914 398c ef66 4fd5  ..J.......9..fO.
0x0040 d888 8448 31fb 85c0 752f 0000 003d 6469  ...H1...u/...=di
0x0050 6666 6965 2d68 656c 6c6d 616e 2d67 726f  ffie-hellman-gro
0x0060 7570 2d65 7863 6861 6e67 652d 7368 6131  up-exchange-sha1
0x0070 2c64 6966 6669 652d 6865 6c6c 6d61 6e2d  ,diffie-hellman-
0x0080 6772 6f75 7031 2d73 6861 3100 0000 0f73  group1-sha1....s
0x0090 7368 2d72 7361 2c73 7368 2d64 7373 0000  sh-rsa,ssh-dss..
0x00a0 0066 6165 7331 3238 2d63 6263 2c33 6465  .faes128-cbc,3de
0x00b0 732d 6362 632c 626c 6f77 6669 7368 2d63  s-cbc,blowfish-c
0x00c0 6263 2c63 6173 7431 3238 2d63 6263 2c61  bc,cast128-cbc,a
0x00d0 7263 666f 7572 2c61 6573 3139 322d 6362  rcfour,aes192-cb
0x00e0 632c 6165 7332 3536 2d63 6263 2c72 696a  c,aes256-cbc,rij
0x00f0 6e64 6165 6c2d 6362 6340 6c79 7361 746f  ndael-cbc@lysato
0x0100 722e 6c69 752e 7365 0000 0066 6165 7331  r.liu.se...faes1
0x0110 3238 2d63 6263 2c33 6465 732d 6362 632c  28-cbc,3des-cbc,
0x0120 626c 6f77 6669 7368 2d63 6263 2c63 6173  blowfish-cbc,cas
0x0130 7431 3238 2d63 6263 2c61 7263 666f 7572  t128-cbc,arcfour
0x0140 2c61 6573 3139 322d 6362 632c 6165 7332  ,aes192-cbc,aes2
0x0150 3536 2d63 6263 2c72 696a 6e64 6165 6c2d  56-cbc,rijndael-
0x0160 6362 6340 6c79 7361 746f 722e 6c69 752e  cbc@lysator.liu.
0x0170 7365 0000 0055 686d 6163 2d6d 6435 2c68  se...Uhmac-md5,h
0x0180 6d61 632d 7368 6131 2c68 6d61 632d 7269  mac-sha1,hmac-ri
0x0190 7065 6d64 3136 302c 686d 6163 2d72 6970  pemd160,hmac-rip
0x01a0 656d 6431 3630 406f 7065 6e73 7368 2e63  emd160@openssh.c
0x01b0 6f6d 2c68 6d61 632d 7368 6131 2d39 362c  om,hmac-sha1-96,
0x01c0 686d 6163 2d6d 6435 2d39 3600 0000 5568  hmac-md5-96...Uh
0x01d0 6d61 632d 6d64 352c 686d 6163 2d73 6861  mac-md5,hmac-sha
0x01e0 312c 686d 6163 2d72 6970 656d 6431 3630  1,hmac-ripemd160
0x01f0 2c68 6d61 632d 7269 7065 6d64 3136 3040  ,hmac-ripemd160@
0x0200 6f70 656e 7373 682e 636f 6d2c 686d 6163  openssh.com,hmac
0x0210 2d73 6861 312d 3936 2c68 6d61 632d 6d64  -sha1-96,hmac-md
0x0220 352d 3936 0000 0009 6e6f 6e65 2c7a 6c69  5-96....none,zli
0x0230 6200 0000 096e 6f6e 652c 7a6c 6962 0000  b....none,zlib..
0x0240 0000 0000 0000 0000 0000 0000 0000 0000  ................
0x0250 0000 0000                                ....
```

The client continues key exchange by sending an SSH_MSG_KEXINIT message, containing its list of supported algorithms. The following packet indicates the preferences:

```
Key algorithm:                        diffie-hellman-group-exchange-sha1
Server host key algorithm:                ssh-rsa
Encryption Algorithm, client-to-server:   faes128-cbc
Encryption Algorithm, server-to-client:   faes128-cbc
MAC Algorithm, client-to-server:          Uhmac-md5
MAC Algorithm, server-to-client:          Uhmac-md5
Compression Algorithm, client-to-server:  none
Compression Algorithm, server-to-client:  none
Languages, client-to-server:              (none specified)
Languages, server-to-client:              (none specified)
"first_kex_packet_follows":               false
```

```
attacker.1031 > openbsd.ssh: P 17:553(536) ack 566 win 6528
<nop,nop,timestamp 81021669 1197394596> (DF)
0x0030 475e caa4 0000 0214 0b14 bcc3 a4ee f77c  G^.............|
0x0040 eeb8 478a c4f8 12b6 ca5d 0000 003d 6469  ..G......]...=di
0x0050 6666 6965 2d68 656c 6c6d 616e 2d67 726f  ffie-hellman-gro
0x0060 7570 2d65 7863 6861 6e67 652d 7368 6131  up-exchange-sha1
0x0070 2c64 6966 6669 652d 6865 6c6c 6d61 6e2d  ,diffie-hellman-
0x0080 6772 6f75 7031 2d73 6861 3100 0000 0f73  group1-sha1....s
0x0090 7368 2d72 7361 2c73 7368 2d64 7373 0000  sh-rsa,ssh-dss..
0x00a0 0066 6165 7331 3238 2d63 6263 2c33 6465  .faes128-cbc,3de
0x00b0 732d 6362 632c 626c 6f77 6669 7368 2d63  s-cbc,blowfish-c
0x00c0 6263 2c63 6173 7431 3238 2d63 6263 2c61  bc,cast128-cbc,a
0x00d0 7263 666f 7572 2c61 6573 3139 322d 6362  rcfour,aes192-cb
0x00e0 632c 6165 7332 3536 2d63 6263 2c72 696a  c,aes256-cbc,rij
0x00f0 6e64 6165 6c2d 6362 6340 6c79 7361 746f  ndael-cbc@lysato
0x0100 722e 6c69 752e 7365 0000 0066 6165 7331  r.liu.se...faes1
0x0110 3238 2d63 6263 2c33 6465 732d 6362 632c  28-cbc,3des-cbc,
0x0120 626c 6f77 6669 7368 2d63 6263 2c63 6173  blowfish-cbc,cas
0x0130 7431 3238 2d63 6263 2c61 7263 666f 7572  t128-cbc,arcfour
0x0140 2c61 6573 3139 322d 6362 632c 6165 7332  ,aes192-cbc,aes2
0x0150 3536 2d63 6263 2c72 696a 6e64 6165 6c2d  56-cbc,rijndael-
0x0160 6362 6340 6c79 7361 746f 722e 6c69 752e  cbc@lysator.liu.
0x0170 7365 0000 0055 686d 6163 2d6d 6435 2c68  se...Uhmac-md5,h
0x0180 6d61 632d 7368 6131 2c68 6d61 632d 7269  mac-sha1,hmac-ri
0x0190 7065 6d64 3136 302c 686d 6163 2d72 6970  pemd160,hmac-rip
0x01a0 656d 6431 3630 406f 7065 6e73 7368 2e63  emd160@openssh.c
0x01b0 6f6d 2c68 6d61 632d 7368 6131 2d39 362c  om,hmac-sha1-96,
0x01c0 686d 6163 2d6d 6435 2d39 3600 0000 5568  hmac-md5-96...Uh
0x01d0 6d61 632d 6d64 352c 686d 6163 2d73 6861  mac-md5,hmac-sha
0x01e0 312c 686d 6163 2d72 6970 656d 6431 3630  1,hmac-ripemd160
0x01f0 2c68 6d61 632d 7269 7065 6d64 3136 3040  ,hmac-ripemd160@
0x0200 6f70 656e 7373 682e 636f 6d2c 686d 6163  openssh.com,hmac
0x0210 2d73 6861 312d 3936 2c68 6d61 632d 6d64  -sha1-96,hmac-md
0x0220 352d 3936 0000 0004 6e6f 6e65 0000 0004  5-96....none....
0x0230 6e6f 6e65 0000 0000 0000 0000 0000 0000  none............
0x0240 0000 0000 0000 0000 0000 0000            ............
```

The Diffie-Hellman Group Exchange

To use the selected encryption method, the client and servers must establish encryption keys using a Diffie-Hellman key exchange algorithm.

First, the client sends:

```
byte      SSH_MSG_KEY_DH_GEX_REQUEST
uint32    min, minimal size in bits of an acceptable group
uint32     n, preferred size in bits of the group the server
          should send
uint32    max, maximal size in bits of an acceptable group
```

In the following packet, min=0x400=1024, n=0x800=2048, and max=0x2000=8192

Message 0x22=34: SSH2_MSG_KEX_DH_GEX_REQUEST

```
attacker.1031 > openbsd.ssh: P 553:577(24) ack 566 win 6528
<nop,nop,timestamp 81021690 1197394596> (DF)
0x0000 4500 004c 6924 4000 4006 1f33 c0a8 1881  E..Li$@.@..3....
0x0010 c0a8 1883 0407 0016 d6c5 25ee 0fb0 1e95  ..........%.....
0x0020 8018 1980 e7aa 0000 0101 080a 04d4 4afa  ..............J.
0x0030 475e caa4 0000 0014 0622 0000 0400 0000  G^.......".....
0x0040 0800 0000 2000 0000 0000 0000            ............
```

The server responds with

```
byte SSH_MSG_KEX_DH_GEX_GROUP
mpint      p, safe prime
mpint      g, generator for subgroup in GF(p)
```

In the following packet, the prime is 0x18f=399 bytes long (3192 bits), and the generator is equal to 5.

Message 0x1f=31: SSH2_MSG_KEX_DH_GEX_GROUP

```
openbsd.ssh > attacker.1031: P 566:990(424) ack 577 win 17376
<nop,nop,timestamp 1197394596 81021690> (DF)
0x0000 4500 01dc 03ee 4000 4006 82d9 c0a8 1883  E.....@.@.......
0x0010 c0a8 1881 0016 0407 0fb0 1e95 d6c5 2606  ..............&.
0x0020 8018 43e0 38c6 0000 0101 080a 475e caa4  ..C.8.......G^..
0x0030 04d4 4afa 0000 01a4 0a1f 0000 018f 669b  ..J...........f.
0x0040 a3ed 661f 226a 090b e564 4a2b b420 9371  ..f."j...dJ+...q
0x0050 b78f c3e6 848a 0958 2199 3f59 084c a5ee  .......X!.?Y.L..
0x0060 1205 2f97 7d01 f066 6f03 f657 3b19 9dfe  ../.}..fo..W;...
0x0070 c9ab 9458 8c2c 60de 3b3e 7cf5 0945 8791  ...X.,`.;>|..E..
0x0080 9fcc 3fb4 0a61 c261 e891 a0f9 1d9f fc8f  ..?..a.a........
0x0090 30ca 12cf 809d d829 0dd7 86fa 8b04 1ffa  0......).........
0x00a0 c579 3c38 f387 57ea 6790 472a c269 2185  .y<8..W.g.G*.i!.
0x00b0 b554 b004 6e8c 065c 983c 0acc 8d2f 85ab  .T..n..\.<.../..
0x00c0 4bed f7ce 2330 0921 8c96 91fe 4426 1580  K...#0.!....D&..
0x00d0 d414 9f1d 4471 b0b5 df79 e224 2524 74eb  ....Dq...y.$%$t.
0x00e0 c3b7 b549 0950 bb43 8bf4 98e7 9f87 9449  ...I.P.C.......I
0x00f0 8b3a 3b5f bb42 829c 3bbe a406 7f28 c23b  .:;_.B..;....(.;
0x0100 e403 77b9 86bd 5443 cccf 0240 5b8c ccaa  ..w...TC...@[...
0x0110 09e8 179f 0168 d496 9994 171a 6ad9 8f81  .....h......j...
0x0120 015b c84e 10a4 4e1e fd2e 0862 c5d1 aafe  .[.N..N....b....
0x0130 9901 4715 a368 00db d9a6 c51c 0226 cc82  ..G..h.......&..
0x0140 a651 dae4 f73d 54c4 d103 c13d 1c15 cf8c  .Q...=T....=....
0x0150 ca67 d5cb 39f0 3c66 f3b7 467f 8ffd cc50  .g..9.<f..F....P
0x0160 74cd 0c1b 2538 fbf9 5697 1bf3 9314 cedd  t...%8..V.......
0x0170 20e1 b10d e16d 86e1 0be7 fa5b 1a70 6aeb  .....m.....[.pj.
0x0180 4c35 6f49 807a 2207 2cd0 0559 af0a 8637  L5oI.z".,..Y...7
0x0190 8895 6651 919e 26a3 15ea d1d2 6e7c 98fc  ..fQ..&.....n|..
0x01a0 4cfa 35a0 f04d d400 a299 1a1f fe5b 271f  L.5..M.......['.
0x01b0 ede5 4375 896a 29f9 68be 1d51 1ba4 66a9  ..Cu.j).h..Q..f.
0x01c0 2ac3 e377 2709 fc81 5b0a b16d af00 0000  *..w'...[..m....
0x01d0 0105 0000 0000 0000 0000                 ............
```

The client responds with:

```
byte SSH_MSG_KEX_DH_GEX_INIT
mpint      numeric Diffie-Hellman "e" parameter
```

In the following packet Diffie-Hellman key exchange value, e, sent by the client, is 3192 bits long.

Message 0x20=32: SSH2_MSG_KEX_DH_GEX_INIT

```
attacker.1031 > openbsd.ssh: P 577:993(416) ack 990 win 7616
<nop,nop,timestamp 81021695 1197394596> (DF)
0x0000 4500 01d4 6926 4000 4006 1da9 c0a8 1881  E...i&@.@.......
0x0010 c0a8 1883 0407 0016 d6c5 2606 0fb0 203d  ..........&....=
0x0020 8018 1dc0 1e85 0000 0101 080a 04d4 4aff  ..............J.
0x0030 475e caa4 0000 019c 0720 0000 018f 3f92  G^..........?.
0x0040 07db eeb1 d82b 57b7 405e e71e 6f8b 2c98  .....+W.@^..o.,.
0x0050 27a3 178a 4e83 be71 425f 222d d523 baba  '...N..qB_"-.#..
0x0060 3929 602b 6b50 5b38 3e30 8e59 173b e7eb  9)`+kP[8>0.Y.;..
0x0070 d6c3 e8a9 96be 0035 bd88 33df 7eb8 24ce  .......5..3.~.$.
0x0080 ce49 c00c f19f acaf b7d5 1ece 35e2 ed43  .I..........5..C
0x0090 32ac 5f0f 3398 37d2 f542 873c 779b 583e  2._.3.7..B.<w.X>
0x00a0 b74c 044f 00ae ed71 5218 cf68 5541 29df  .L.O...qR..hUA).
0x00b0 487c a5ff 23cb 183e 45b1 0827 c5ba 29fd  H|..#..>E..'..).
0x00c0 1552 c6f5 6402 0b9f 9e09 2fe6 7bd2 df45  .R..d...../.{..E
0x00d0 32de 32de 745d e79a d6f8 7ea5 469e e75e  2.2.t]....~.F..^
0x00e0 23d6 52f8 26ae 323b 894c 341d 609e eec7  #.R.&.2;.L4.`...
0x00f0 4831 23f2 0557 d078 8f1a 25a2 034b 34b9  H1#..W.x..%..K4.
0x0100 9030 9cfa 3def b87f 416b 4c9b 5050 ddbf  .0..=...AkL.PP..
0x0110 615d 0462 5da5 d667 1cc3 965f 64b7 6bb3  a].b]..g..._d.k.
0x0120 666d 3876 4949 abd7 af2a f5ea 2ea0 2f8c  fm8vII...*..../.
0x0130 13c9 8671 24c4 e408 33b1 10c8 c9f7 d2ae  ...q$...3.......
0x0140 3051 e01f f762 7396 1344 892f 0455 4b39  0Q...bs..D./.UK9
0x0150 f4e3 64e0 1e2c ed8a 5d36 cd24 b0b7 8ba9  ..d..,..]6.$....
0x0160 d96e 1251 e1a5 1c73 c2b1 fd42 bfa5 2054  .n.Q...s...B...T
0x0170 ccfa e1ca ae78 95ed b5c1 c3f4 aeb0 e2a5  .....x..........
0x0180 b939 4152 a615 eeca 8f9e 0495 eab3 cab2  .9AR............
0x0190 236b 7b0f 6f70 67ef 61e2 34f4 9af3 1a7f  #k{.opg.a.4.....
0x01a0 e947 012a 7105 1b42 d447 84bb 403e c106  .G.*q..B.G..@>..
0x01b0 42cf 023d 1813 62a9 14a8 c78f c401 cff0  B..=..b.........
0x01c0 b092 aaba db99 7714 b74a 9811 1300 0000  ......w..J......
0x01d0 0000 0000                                ....
```

The server responds with:

```
byte SSH_MSG_KEX_DH_GEX_REPLY
string     server public host key and certificates (K_S)
mpint      numeric Diffie-Hellman "f" parameter
string     signature of H
```

In the following packet, the K_S value is 0x95=149 bytes long, the server key exchange value is 3192 bits long, and the signature is 0x8f=143 bytes long.

The K_S key value breaks down into the following fields:

```
string "ssh-rsa"
mpint  numeric Diffie-Hellman "e" parameter
mpint  numeric Diffie-Hellman "n" parameter
```

The signature of H breaks down into the following fields:

```
string    "ssh-rsa"
string    rsa_signature_blob
```

Note that the cleartext string "ssh-rsa" appears twice in the packet.

Message 0x21=33: SSH2_MSG_KEX_DH_GEX_REPLY ....

```
openbsd.ssh > attacker.1031: P 990:1726(736) ack 993 win 17376
<nop,nop,timestamp 1197394596 81021695> (DF)
0x0000 4500 0314 09b9 4000 4006 7bd6 c0a8 1883  E.....@.@.{.....
0x0010 c0a8 1881 0016 0407 0fb0 203d d6c5 27a6  ..........=..'.
0x0020 8018 43e0 e8e7 0000 0101 080a 475e caa4  ..C.........G^..
0x0030 04d4 4aff 0000 02cc 0b21 0000 0095 0000  ..J......!......
0x0040 0007 7373 682d 7273 6100 0000 0123 0000  ..ssh-rsa....#..
0x0050 0081 00d1 f8bc e8f5 f18e 08f8 d043 dc98  ............C..
0x0060 445e e95b 17ae 0014 a791 46e4 fc45 68c9  D^.[......F..Eh.
0x0070 6876 3367 1b0f bdf2 d800 1735 e5b1 bf44  hv3g.......5...D
0x0080 bb3b 9706 def6 557d 55ba ebbd 9413 cf47  .;....U}U......G
0x0090 e4ec 9949 2c68 82a0 d31f 9215 8474 8659  ...I,h.......t.Y
0x00a0 6c61 bfa8 360e f0ba 40c1 8ada 33a0 fec4  la..6...@...3...
0x00b0 ba26 0ed6 e55a 8d70 7f53 d84e 035e 248e  .&...Z.p.S.N.^$.
0x00c0 8c74 a840 0747 95b2 0dd0 3822 6c58 4c6f  .t.@.G....8"lXLo
0x00d0 fdfb 9100 0001 8f61 251d c2d6 7e36 8312  .......a%...~6..
0x00e0 e918 d604 6e80 6482 581a ab53 9d9b 141f  ....n.d.X..S....
0x00f0 f506 be5b 0ce0 57f9 87e2 1eab 3dd1 4d1d  ...[..W.....=.M.
0x0100 c00c 800c a292 6eed 58d1 b390 cf3f 5aeb  ......n.X....?Z.
0x0110 08fc 9066 fa47 06ad f89e 2411 e7bb 3813  ...f.G....$...8.
0x0120 2915 d756 6ea5 1ad1 af28 0aeb 378c 91ef  )..Vn....(..7...
0x0130 cc77 8abb c228 a39e 7171 9ed7 38f6 d35c  .w...(..qq..8..\
0x0140 7a9f 2cb2 710a e410 ace7 dc4b c62e 2ff0  z.,.q......K../.
0x0150 0707 26bc 5d7a d64c 4c25 f022 f7b4 dfac  ..&.]z.LL%."....
0x0160 1b9e b781 ac10 47a0 aff3 287d df51 32e2  ......G...(}.Q2.
0x0170 2fc4 4815 5fc9 29a5 fe50 a45a 42ee 4013  /.H._.)..P.ZB.@.
0x0180 a488 7fd6 6bbd ff40 4d01 05d8 cb2e c507  ....k..@M.......
0x0190 766f b1df b920 ad9d 1baa 9b2d 0fb7 ed76  vo.........-...v
0x01a0 f13d 9691 dd94 aeca c801 42af a6af de0c  .=........B.....
0x01b0 62ed e5f5 76a7 eb06 573b adac 8f4d 8d93  b...v...W;...M..
0x01c0 b1f5 3b24 3344 2f63 786c 5c0d 5fc5 90c5  ..;$3D/cxl\._...
0x01d0 9240 afbd c79a cd4b ebdc 0790 9cfe 2274  .@.....K......"t
0x01e0 fc80 18b8 8487 5f58 15cd 328b a4d6 a9c3  ......_X..2.....
0x01f0 50e3 08d6 a579 d4a2 7115 c965 3ccc b5f7  P....y..q..e<...
0x0200 c412 d427 4317 4321 3b31 5c90 8296 1558  ...'C.C!;1\....X
0x0210 e5a9 1b82 eac2 9553 2051 f6b1 713b 25e7  .......S.Q..q;%.
0x0220 fcd4 b5b6 3a10 eed7 b0ec f88a 528c 42f9  ....:.......R.B.
0x0230 daaa 1da6 6796 9b41 b8eb 5b5b bb54 e620  ....g..A..[[.T..
0x0240 1251 0098 5167 e9e4 14c1 fe34 8b5d 967b  .Q..Qg.....4.].{
0x0250 f93b e788 6da0 0e28 fada 29f6 c2f8 5b09  .;..m..(..)...[.
0x0260 628a 0f87 0e61 0000 008f 0000 0007 7373  b....a........ss
0x0270 682d 7273 6100 0000 804e 7d28 9754 80c9  h-rsa....N}(.T..
0x0280 8243 80d2 5503 034f 226f 419e 7ed4 1a85  .C..U..O"oA.~...
0x0290 3b69 9817 c22d 4b92 11cf 1b4e 6d3c 154d  ;i...-K....Nm<.M
0x02a0 1120 a238 e6d8 ab6b df27 6f3b 7501 ac75  ...8...k.'o;u..u
0x02b0 7b3b 598b 0e4d b3d0 4090 37c1 ac00 36a4  {;Y..M..@.7...6.
0x02c0 666c e670 850b caaf c7ac 6701 e2b9 6b61  fl.p......g...ka
0x02d0 63bd e0ce a2c1 e7ef 014f f00d 675e c7cb  c........O..g^..
0x02e0 5006 36f0 3c78 8075 1f76 2653 a62a b495  P.6.<x.u.v&S.*..
0x02f0 7471 bc99 bc66 0883 4300 0000 0000 0000  tq...f..C.......
0x0300 0000 0000 0000 000c 0a15 0000 0000 0000  ................
0x0310 0000 0000                                ....
```

With this packet, the Diffie-Hellman exchange is completed, and both client and server will now begin to encrypt the binary packets of the SSH-TRANS protocol.

Encrypted Transport-Layer Protocol

Next, the client sends a signal to the server that it is ready to commence encryption with the keys that have just been exchanged:

Message 0x15=21: SSH_MSG_NEWKEYS

```
attacker.1031 > openbsd.ssh: P 993:1009(16) ack 1726 win 8832
<nop,nop,timestamp 81021711 1197394596> (DF)
0x0000 4500 0044 6928 4000 4006 1f37 c0a8 1881  E..Di(@.@..7....
0x0010 c0a8 1883 0407 0016 d6c5 27a6 0fb0 231d  ..........'...#.
0x0020 8018 2280 0073 0000 0101 080a 04d4 4b0f  .."..s........K.
0x0030 475e caa4 0000 000c 0a15 0000 0000 0000  G^..............
0x0040 0000 0000                                ....
```

Next, SSH switches to the Transport Layer Protocol, invoking encryption and compression. From this point until the successful exploit, encryption and compression will mask the exchange. The packet_length, message number, payload, random padding, and MAC fields of the SSH Binary Packet Protocol are all encrypted. Until the exploit starts its command shell, packet contents will appear random.

Note: For the remainder of this appendix, encrypted traffic will be omitted from the packet trace.

The SSH-Userauth Protocol

We can assume that the client will send a SSH-MSG_SERVICE_REQUEST packet, and that the server will reply to it with a SSH-MSG_SERVICE_ACCEPT packet. We can also assume that the service will be "ssh-userauth":

```
attacker.1031 > openbsd.ssh: P 1009:1057(48) ack 1726 win 8832
<nop,nop,timestamp 81021731 1197394597> (DF)
(TCP packet containing 48 bytes of encrypted data)
```

```
openbsd.ssh > attacker.1031: P 1726:1774(48) ack 1057 win 17376
<nop,nop,timestamp 1197394597 81021731> (DF)
(TCP packet containing 48 bytes of encrypted data)
```

Now, we assume that the client sends an SSH_MSG_USERAUTH_REQUEST, asking for the “none” authentication method. This technique is used to request the list of available authentication methods.

```
attacker.1031 > openbsd.ssh: P 1057:1137(80) ack 1774 win 8832
<nop,nop,timestamp 81021731 1197394597> (DF)
(TCP packet containing 80 bytes of encrypted data)
```

The server sends back a SSH_MSG_USERAUTH_FAILURE message, which contains a list of acceptable authentication methods, one of which is "keyboard-interactive".

```
openbsd.ssh > attacker.1031: P 1774:1854(80) ack 1137 win 17376
<nop,nop,timestamp 1197394597 81021731> (DF)
(TCP packet containing 80 bytes of encrypted data)
```

The Attack Sequence

Once the Challenge-Response protocol is underway, the keyboard-interactive vulnerability can be exploited.

The Keyboard-Interactive Protocol

The malicious client sends a SSH_MSG_USERAUTH_REQUEST message, which requests "keyboard-interactive" authentication:

```
attacker.1031 > openbsd.ssh: P 1137:1233(96) ack 1854 win 8832
<nop,nop,timestamp 81021732 1197394597> (DF)
(TCP packet containing 96 bytes of encrypted data)
```

The server sends back an SSH_MSG_USERAUTH_INFO_REQUEST message, containing a password prompt, and requests a password in return.

```
openbsd.ssh > attacker.1031: P 1854:1950(96) ack 1233 win 17376
<nop,nop,timestamp 1197394597 81021732> (DF)
(TCP packet containing 96 bytes of encrypted data)
```

The actual attack begins. The malicious client sends back an enormous SSH_MSG_USERAUTH_INFO_RESPONSE message, containing numerous response strings, instead of the single password requested. The responses contain the malicious code and function pointers to it. The function pointers overwrite a table of error functions in the SSH server program.

Because the contents of the malicious reply is very large, it is broken into separate Binary Protocol packets. The maximum size of each packet is determined by the ethernet medium being used by TCP/IP. In this trace, each packet contains 1448 bytes of encrypted data.

```
attacker.1031 > openbsd.ssh: . 1233:2681(1448) ack 1950 win 8832
<nop,nop,timestamp 81021735 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 2681:4129(1448) ack 1950 win 8832
<nop,nop,timestamp 81021735 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 4129:5577(1448) ack 1950 win 8832
<nop,nop,timestamp 81021736 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 5577:7025(1448) ack 1950 win 8832
<nop,nop,timestamp 81021736 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 7025:8473(1448) ack 1950 win 8832
<nop,nop,timestamp 81021736 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 8473:9921(1448) ack 1950 win 8832
<nop,nop,timestamp 81021736 1197394597> (DF)
attacker.1031 > openbsd.ssh: P 9921:11369(1448) ack 1950 win 8832
<nop,nop,timestamp 81021737 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 11369:12817(1448) ack 1950 win 8832
<nop,nop,timestamp 81021737 1197394597> (DF)
attacker.1031 > openbsd.ssh: P 12817:14265(1448) ack 1950 win 8832
<nop,nop,timestamp 81021737 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 14265:15713(1448) ack 1950 win 8832
<nop,nop,timestamp 81021737 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 15713:17161(1448) ack 1950 win 8832
<nop,nop,timestamp 81021738 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 17161:18609(1448) ack 1950 win 8832
<nop,nop,timestamp 81021738 1197394597> (DF)
attacker.1031 > openbsd.ssh: . 18609:20057(1448) ack 1950 win 8832
<nop,nop,timestamp 81021738 1197394597> (DF)
```

Finally, the last portion of the malicious response is sent. The last response of the message contains the erroneous string length that triggers an error condition. The SSH server program notices the error, and tries to call an error handler via its table of function pointers. But the function pointers have been overwritten with pointers to the malicious code. The server calls the malicious code, thinking it's an error function.

```
attacker.1031 > openbsd.ssh: P 20057:20833(776) ack 1950 win 8832
<nop,nop,timestamp 81021738 1197394597> (DF)
(TCP packet containing 776 bytes of encrypted data)
```

Successful Attack

The compromised server sends the string "GGGG" to the malicious client. Note that the encrypted SSH Transport Protocol is no longer being used.

This appears to be a potential attack signature.

```
openbsd.ssh > attacker.1031: P 1950:1954(4) ack 20833 win 17376
<nop,nop,timestamp 1197394597 81021738> (DF)
0x0000 4500 0038 2b82 4000 4006 5ce9 c0a8 1883  E..8+.@.@.\.....
0x0010 c0a8 1881 0016 0407 0fb0 23fd d6c5 7526  ..........#...u&
0x0020 8018 43e0 0c35 0000 0101 080a 475e caa5  ..C..5......G^..
0x0030 04d4 4b2a 4747 4747                      ..K*GGGG
```

The malicious client sends back the single letter "O", which also appears to be a potential attack signature.

```
attacker.1031 > openbsd.ssh: P 20833:20834(1) ack 1954 win 8832
<nop,nop,timestamp 81021739 1197394597> (DF)
0x0000 4500 0035 693b 4000 4006 1f33 c0a8 1881  E..5i;@.@..3....
0x0010 c0a8 1883 0407 0016 d6c5 7526 0fb0 2401  ..........u&..$.
0x0020 8018 2280 6d21 0000 0101 080a 04d4 4b2b  ..".m!........K+
0x0030 475e caa5 4f                             G^..O
```

Another possible attack signature follows. The "*GOBBLE*" prompt is displayed by the evil client to indicate that the server has been successfully exploited.

```
openbsd.ssh > attacker.1031: P 1954:1963(9) ack 20834 win 17376
<nop,nop,timestamp 1197394597 81021739> (DF)
0x0000 4500 003d 728c 4000 4006 15da c0a8 1883  E..=r.@.@.......
0x0010 c0a8 1881 0016 0407 0fb0 2401 d6c5 7527  ..........$...u'
0x0020 8018 43e0 8fb8 0000 0101 080a 475e caa5  ..C.........G^..
0x0030 04d4 4b2b 2a47 4f42 424c 452a 0a         ..K+*GOBBLE*.
```

The malicious client is coded to give the chain of commands "uname -a;id" when first connecting to the malicious shellcode.

```
attacker.1031 > openbsd.ssh: P 20834:20846(12) ack 1963 win 8832
<nop,nop,timestamp 81021741 1197394597> (DF)
0x0000 4500 0040 693c 4000 4006 1f27 c0a8 1881  E..@i<@.@..'....
0x0010 c0a8 1883 0407 0016 d6c5 7527 0fb0 240a  ..........u'..$.
0x0020 8018 2280 b339 0000 0101 080a 04d4 4b2d  ..".."9........K-
0x0030 475e caa5 756e 616d 6520 2d61 3b69 640a  G^..uname.-a;id.
```

The compromised server returns the results of "uname -a":

```
openbsd.ssh > attacker.1031: P 1963:1999(36) ack 20846 win 17376
<nop,nop,timestamp 1197394598 81021741> (DF)
0x0000 4500 0058 1894 4000 4006 6fb7 c0a8 1883  E..X..@.@.o.....
0x0010 c0a8 1881 0016 0407 0fb0 240a d6c5 7533  ..........$...u3
0x0020 8018 43e0 7596 0000 0101 080a 475e caa6  ..C.u.......G^..
0x0030 04d4 4b2d 4f70 656e 4253 4420 6f70 656e  ..K-OpenBSD.open
0x0040 6273 6420 332e 3120 4745 4e45 5249 4323  bsd.3.1.GENERIC#
0x0050 3539 2069 3338 360a                      59.i386.
```

The compromised server returns the results of "id". Note that the userid is root!

```
openbsd.ssh > attacker.1031: P 1999:2040(41) ack 20846 win 17376
<nop,nop,timestamp 1197394598 81021750> (DF)
0x0000 4500 005d 3fcf 4000 4006 4877 c0a8 1883  E..]?.@.@.Hw....
0x0010 c0a8 1881 0016 0407 0fb0 242e d6c5 7533  ..........$...u3
0x0020 8018 43e0 4216 0000 0101 080a 475e caa6  ..C.B.......G^..
0x0030 04d4 4b36 7569 643d 3028 726f 6f74 2920  ..K6uid=0(root).
0x0040 6769 643d 3028 7768 6565 6c29 2067 726f  gid=0(wheel).gro
0x0050 7570 733d 3028 7768 6565 6c29 0a         ups=0(wheel).
```

The following packets are the commands typed interactively at the client, and the responses sent back from the compromised server:

```
attacker.1031 > openbsd.ssh: P 20846:20849(3) ack 2040 win 8832
<nop,nop,timestamp 81022108 1197394598> (DF)
0x0000 4500 0037 693f 4000 4006 1f2d c0a8 1881  E..7i?@.@..-....
0x0010 c0a8 1883 0407 0016 d6c5 7533 0fb0 2457  ..........u3..$W
0x0020 8018 2280 46e6 0000 0101 080a 04d4 4c9c  ..".F.........L.
0x0030 475e caa6 6964 0a                        G^..id.

openbsd.ssh > attacker.1031: P 2040:2081(41) ack 20849 win 17376
<nop,nop,timestamp 1197394605 81022108> (DF)
0x0000 4500 005d 58b4 4000 4006 2f92 c0a8 1883  E..]X.@.@./.....
0x0010 c0a8 1881 0016 0407 0fb0 2457 d6c5 7536  ..........$W..u6
0x0020 8018 43e0 407d 0000 0101 080a 475e caad  ..C.@}......G^..
0x0030 04d4 4c9c 7569 643d 3028 726f 6f74 2920  ..L.uid=0(root).
0x0040 6769 643d 3028 7768 6565 6c29 2067 726f  gid=0(wheel).gro
0x0050 7570 733d 3028 7768 6565 6c29 0a         ups=0(wheel).

attacker.1031 > openbsd.ssh: P 20849:20858(9) ack 2081 win 8832
<nop,nop,timestamp 81022453 1197394605> (DF)
0x0000 4500 003d 6941 4000 4006 1f25 c0a8 1881  E..=iA@.@..%....
0x0010 c0a8 1883 0407 0016 d6c5 7536 0fb0 2480  ..........u6..$.
0x0020 8018 2280 455b 0000 0101 080a 04d4 4df5  ..".E[........M.
0x0030 475e caad 756e 616d 6520 2d61 0a         G^..uname.-a.

openbsd.ssh > attacker.1031: P 2081:2117(36) ack 20858 win 17376
<nop,nop,timestamp 1197394612 81022453> (DF)
0x0000 4500 0058 447e 4000 4006 43cd c0a8 1883  E..XD~@.@.C.....
0x0010 c0a8 1881 0016 0407 0fb0 2480 d6c5 753f  ..........$...u?
0x0020 8018 43e0 723e 0000 0101 080a 475e cab4  ..C.r>......G^..
0x0030 04d4 4df5 4f70 656e 4253 4420 6f70 656e  ..M.OpenBSD.open
0x0040 6273 6420 332e 3120 4745 4e45 5249 4323  bsd.3.1.GENERIC#
0x0050 3539 2069 3338 360a                      59.i386.
```

Appendix D: Affected OS Releases

(SecurityFocus http://online.securityfocus.com/bid/5093)
(Internet Security Systems http://www.iss.net/security_center/static/9169.php)

Apple MacOS X 10.0

Apple MacOS X 10.0, 10.0.1 – 10.0.4, 10.1, 10.1.1-10.1.5

Caldera OpenLinux Server 3.1, 3.1.1

Caldera OpenLinux Workstation 3.1, 3.1.1

Caldera OpenUnix 8.0

Caldera UnixWare 7.1.1

Conectiva Linux 5.0, 5.1, 6.0, 7.0, 8.0

Conectiva Linux ecommerce

Conectiva Linux graficas

Debian Linux 3.0

EnGarde Secure Linux Community Edition

FreeBSD FreeBSD 4.4 - RELENG

FreeBSD FreeBSD 4.5 - RELEASE

FreeBSD FreeBSD 4.5 - STABLEpre2002-03-07

FreeBSD-current: All Versions

Guardian Digital Engarde Secure Linux 1.0.1

HP HP-UX 11.0

HP HP-UX 11.11

HP Secure OS software for Linux 1.0

IBM AIX 4.3, 4.3.1-4.3.3

Immunix Immunix OS 7.0

Immunix OS 7.0

Immunix OS 7+-beta

Mandrake Linux 7.1, 7.2, 8.0-8.2

Mandrake Linux Corporate Server 1.0.1

Mandrake Single Network Firewall 7.2

MandrakeSoft Corporate Server 1.0.1

MandrakeSoft Linux Mandrake 7.1, 7.2, 8.0, 8.1

MandrakeSoft Linux Mandrake 8.0 ppc

MandrakeSoft Single Network Firewall 7.2

NetBSD 1.4.x, 1.5.x, 1.6.x

NetBSD-current pre20020626

OpenBSD 3.0, 3.1

OpenPKG 1.0

OpenPKG OpenPKG 1.0

OpenSSH 3.0 to 3.2.3

Openwall GNU/*/Linux (Owl)-current

Openwall GNU/*/Linux 0.1 -stable

Red Hat Linux 7.0-7.3

S.u.S.E. eMail Server III: All Versions

S.u.S.E. Linux 6.4, 7.0-7.3, 8.0

S.u.S.E. Linux Database Server

S.u.S.E. Linux Enterprise Server 7

S.u.S.E. Linux Firewall

S.u.S.E. SuSE eMail Server III

SCO Open Server 5.0, 5.0.1-5.0.6, 5.0.6a

Sun Cobalt RaQ 550

Sun Solaris 9.0

Trustix Secure Linux 1.1, 1.2, 1.5

Wirex Immunix OS 6.2

List of References

Albitz, Paul; Liu, Cricket. DNS and Bind. Third Edition. Sebastopol, CA, O'Reilly & Associates, Inc, September, 1998.

Arce, Ivan. "SSH Insertion Attack." CORE SDI S.A. June 6, 1998. URL: http://www.landfield.com/isn/mail-archive/1998/Jun/0052.html (August 27, 2002)

Atkins, E. Todd. "SWATCH – The Simple WATCHer." November 8, 2001. URL: http://www.oit.ucsb.edu/~eta/swatch/ (August 17, 2002)

Barrett, Daniel J., with Silverman, Richard E. SSH, the Secure Shell: The Definitive Guide. Sebastopol, California: O'Reilly & Associates, Inc., February 2001.

[1] Caswell, Brian; Roesch, Brian. "The Open Source Network Intrusion Detection System." Published August 13, 2002. Snort.org. URL: http://www.snort.org (August 17, 2002)

[2] Caswell, Brian; Roesch, Brian. "Snort Signature Database." Published August 9, 2002. Snort.org. URL: http://www.snort.org/snort-db (August 9, 2002)

CERT Coordination Center. "CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling." June 26, 2002, revised August 8, 2002. Pittsburgh, Pennsylvania, Carnegie Mellon Software Engineering Institute. URL: http://www.cert.org/advisories/CA-2002-18.html (September 17, 2002)

Cole, Eric. "Welcome to the SANS Security CDI Conferences!" SANS Security Cyber Defense Initiative East & West, URL: http://www.sans.org/CDI.htm (July 16, 2002)

Common Vulnerabilities and Exposures. "CAN-2002-0640 (under review)." The Mitre Corporation. July 26, 2002. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0640 (August 27, 2002)

Cusack, Frank; Appgate AB, Martin Forssen. "Generic Message Exchange Authentication for SSH." March 1, 2001. Network Working Group. URL: http://www.openssh.org/txt/draft-ietf-secsh-auth-kbdinteract-02.txt (August 9, 2002)

[1] de Raadt, Theo. "OpenBSD 3.1 Release." May 19, 2002. URL: http://openbsd.groupbsd.org/31.html (September 14, 2002)

[2] de Raadt, Theo. "Upcoming OpenSSH vulnerability." Mailing list ARChives. The AIMS Group. June 24, 2002. URL: http://marc.theaimsgroup.com/?l=openssh-unix-dev&m=102495293705094&w=2 (September 16, 2002)

Devine, Christophe. "OpenBSD 3.1 sshd remote root exploit." SecurityFocus HOME Mailing List: BugTraq. June 28, 2002. URL: http://online.securityfocus.com/archive/1/279610/2002-06-28/2002-07-04/0 (August 27, 2002)

Friedl, Markus; Provos, Niels; Simpson, William A. “Diffie-Hellman Group Exchange for the SSH Transport Layer Protocol.” January, 2002. Internet Draft, Network Working Group. URL: http://www.ietf.org/internet-drafts/draft-ietf-secsh-dh-group-exchange-02.txt (August 17, 2002)

Garfinkel, Simon. Pretty Good Privacy. O’Reilly and Associates, Inc. March, 1995.

[1] GOBBLES Security, “Remote OpenSSH exploit for 2.9.9-3.3” SecurityFocus HOME Mailing List, Jul 1, 2002. Message ID <200207011732.g61HW0w84242@mailserver1.hushmail.com>. URL: http://online.securityfocus.com/archive/attachment/280029/2/sshutup-theo.tar.gz (July 19, 2002)

[2] GOBBLES Security, “sshutuptheo.tar.gz” Exploits, Jul 1, 2002. URL: http://www.immunitysec.com/GOBBLES/exploits/sshutup-theo.tar.gz (August 9, 2002)

[3] GOBBLES Security, “About GOBBLES Security.” Immunity, Inc. 2002. URL: http://www.immunitysec.com/GOBBLES/about.html (August 27, 2002)

Internet Security Systems. “OpenSSH "Challenge-Response" authentication buffer overflow.” June 26 2002. X-Force Database Search. Article number 9169. URL: http://www.iss.net/security_center/static/9169.php (July 19, 2002)

Internet Security Systems X-Force. “OpenSSH Remote Challenge Vulnerability.” Internet Security Systems Security Advisory, Internet Security Systems. June 26, 2002. URL: http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20584 (September 1, 2002)

Mandrake Linux. URL: http://www.mandrakelinux.com/en/ (September 14, 2002)

Milford, Nathan Ryan. “Re: Upcoming OpenSSH vulnerability.” June 25, 2002. Mailing List Archives. The AIMS Group. URL: http://marc.theaimsgroup.com/?l=openbsd-misc&m=102507468017147&w=2 (July 20, 2002)

OpenBSD. “OpenSSH.” OpenSSH Portable Release for Linux/Solaris/etc v 1.91 2002/06/26. URL: ftp://ftp.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-3.4p1.tar.gz (July 20, 2002)

OpenSSH. “Revised OpenSSH Security Advisory.” OpenBSD. August 1, 2002. URL: http://www.openssh.org/txt/preauth.adv (August 28, 2002)

Provos, Niels. “Privilege Separated OpenSSH.” Published July 6, 2002. Center for Information Technology Integration, University of Michigan. URL: http://www.citi.umich.edu/u/provos/ssh/privsep.html (July 20, 2002)

Schneier, Bruce. Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc., 1996.

SecurityFocus. “OpenSSH Challenge-Response Buffer Overflow Vulnerabilities.” Published Jun 24, 2002, Updated Jul 15, 2002. Vulnerabilities. Bugtraq number 5093. URL: http://online.securityfocus.com/bid/5093 (July 19, 2002)

Singh, Simon. The Code Book: The Evolutino of Secrecy from Mary Queen of Scots to Quantum Cryptography. New York: Random House, 1999.

Skoudis, Ed. Counter Hack, A Step-By-Step Guide to Computer Attacks and Effective Defenses. Upper Saddle River, New Jersey: Prentice Hall, 2002.

Stallings, William. Cryptography and Network Security: Principles and Practice, Second Edition. Upper Saddle River, New Jersey: Prentice Hall, 1995.

[1] Ullrich, Johannes. "About," The Internet Storm Center, URL: http://isc.incidents.org/about.html (July 16, 2002)

[2] Ullrich, Johannes. "Top 10 Ports, " The Internet Storm Center, July 16, 2002, URL: http://isc.incidents.org/top10.html

VMWare. "VMWare Workstation." VMWare, Inc. 2002. URL: http://www.vmware.com/products/desktop/ws_features.html (September 14, 2002)

[1] Ylonen, Tatu; Kivenen, Tero; Saarinen, Markku-Juhani; Rinne, Timo J; Lehtinen, Sami. "SSH Protocol Architecture." January 31, 2002. The Internet Society. URL: http://www.openssh.org/txt/draft-ietf-secsh-architecture-12.txt (July 20, 2002)

[2] Ylonen, Tatu; Kivenen, Tero; Saarinen, Markku-Juhani; Rinne, Timo J; Lehtinen, Sami. "SSH Transport Layer Protocol." March 20, 2002. The Internet Society. URL: http://www.openssh.org/txt/draft-ietf-secsh-transport-14.txt (July 20, 2002)

[3] Ylonen, Tatu; Kivenen, Tero; Saarinen, Markku-Juhani; Rinne, Timo J; Lehtinen, Sami. "SSH Connection Protocol." January 31, 2002. The Internet Society. URL: http://www.openssh.org/txt/draft-ietf-secsh-connect-15.txt (July 20, 2002)

[4] Ylonen, Tatu; Kivenen, Tero; Saarinen, Markku-Juhani; Rinne, Timo J; Lehtinen, Sami. "SSH Authentication Protocol." February 28, 2002. The Internet Society. URL: http://www.openssh.org/txt/draft-ietf-secsh-userauth-15.txt (July 20, 2002)