OpenSSH/Print version

Overview

The OpenSSH (http://www.openssh.org/) suite provides secure remote access and file transfer. Since its initial release, it has grown to become the most widely used implementation of the SSH protocol. During the first ten years of its existence, ssh has largely replaced older corresponding unencrypted tools and protocols. The OpenSSH client is included by default in most operating system distributions, including OS X, Linux, BSD and Solaris. Any day you use the Internet, you are using and relying on dozens if not hundreds of machines operated and maintained using OpenSSH. A survey in 2008 showed that of the SSH servers found running, just over 80% were OpenSSH. [1]

OpenSSH was first released towards the end of 1999. It is the latest step in a very long and useful history of networked commuting, remote access and telecommuting.

History of OpenSSH

The first release of OpenSSH was in December 1999 as part of OpenBSD 2.6. The source code was originally derived from a re-write of the last available open version, ssh 1.2.12 specifically, of SSH[2]. SSH went on to become Tectia SSH.

Ongoing development of OpenSSH is done by the OpenBSD group. Core development occurs first on OpenBSD, then portability teams bring the changes to other platforms. OpenSSH is an integral part of as good as all server systems today and a good many network appliances such as routers, switches and networked storage. The first steps were in many ways the biggest.

The Early Days of Remote Access

Some of the tools that inspired the need for SSH have been around since the beginning, too, or very near the beginning of the Internet. Remote access has been a fundamental part of the concept since the idea stage and the nature and capabilities of this access has evolved as the network has evolved in scale, scope and usage. See the web version of the *Levnez Unix Timeline*[3] by Éric Lévénez for an overview of systems development and the web version of *Hobbes' Internet Timeline*[4] by Robert H Zakon for an overview of the development of the Internet.

1969

- Telnet was one of the original ARPAnet application protocols, named in RFC 97. It was used to access a host at a remote site locally. Telnet was described starting two years later in RFC 137, RFC 139, RFC 318 and others, including the lost RFC 97. That is as good a turning point as any to delineate Telnet.

1971

- Thompson Shell, by Ken Thompson, was an improvement on the old text-based user interface, the shell. This new one allowed redirects but was only a user interface and not for scripting.
- In the same year FTP, the file transfer protocol, was described in RFC 114. A key goal was to promote use of computers over the net by allowing users at any host on the network to use the file system of

any cooperating host.

1978

- Bill Joy created BSD's C shell which is named for the C-like syntax it uses. It allows job control, history substitution, and aliases, which features we find in today's interfaces.
- In the same year, the Bourne Shell by Steve Bourne at Bell Labs [5] was created. It is the progenitor to the default shells used in most distros today: **ksh** and **bash**.

1983

- The remote file copy utility, **rcp**, appeared in 4.2 BSD. rcp copied files across the net to other hosts using **rsh**, which also appeared staring 4.2 BSD, to perform its operations. Like **telnet** and **ftp**, all passwords, user names and data are transmitted unencrypted in clear text. Both **rsh** and **rcp** were part of the **rlogin** suite.

1991

- PGP, written at MIT by Philip Zimmermann[6], charted new waters for encrypted electronic communications with the goals of preserving civil liberties online, ensuring individual privacy, keeping encryption legal in the USA, and protecting business communications. Like SSH it uses asymmetric encryption with public / private key pairs.

1993

- Kerberos V (RFC 1510) authentication service from MIT's project Athena [7] provides a means for authentication over an open, unsecure network. Kerberos got its original start in 1988.

SSH - open then closed

1995

- Tatu Ylönen at the Helsinki University of Technology developed the first SSH protocol and programs, releasing them under an open license[8] as per the norm in computer science, software engineering and advanced development. [9]

1995?

- Björn Grönvall dug out the most recent open version of ssh, version 1.2.12[10] [11]. He and Holger Trapp did the initial work to free the distribution, resulting in OSSH

1996

- SSH2 protocol is defined

OpenSSH

1999

- OpenSSH begins based on OSSH. Niels Provos, Theo de Raadt, Markus Friedl developed the cryptographic components during the port to OpenBSD which became the OpenSSH we know today. Dug Song, Aaron Campbell and many others provided various non-crypto contributions. openssl library issues were sorted by Bob Beck. Damien Miller, Philip Hands, and others started porting

OpenSSH to Linux

2000

- Markus Friedl added SSH 2 protocol support to OpenSSH version 2.0, which was released in June.[12] OpenSSH 2.0 shipped with OpenBSD 2.7. Niels Provos and Theo de Raadt did most of the checking. Bob Beck updated OpenSSL. Markus also added support for the SFTP protocol later that same year.
- In September of 2000, the long wait in the USA was over for the patents on the RSA algorithms to expire. In the European Union, the European Patent Convention of 1972, frees software, algorithms, business methods or literature, unlike the unfortunate, anti-business situation in the USA. This freedom in Europe hangs by a thread at the moment.
- SSH Tectia changes licenses again.

2001

- Damien Miller completed the SFTP client which was released in February.
- SSH2 became the default protocol

2008

- Built-in chroot support for **sshd**.

2010

- As of OpenSSH 5.4, the legacy protocol SSH1 is finally disabled by default.

Note: OpenSSH can be used anywhere in the whole world because it uses only algorithms unencumbered by software patents, business method patents, algorithm patents, and so on. These types of patents do not apply in Europe, only physical inventions can be patented in Europe, but there are regions of the world where these problems do occur. Small and medium businesses in Europe have been active in politics to keep the advantage.

2014

- As of OpenSSH 6.7, both the base and the portable versions of OpenSSH can build against LibreSSL instead of OpenSSL for certain cryptographic functions.

Why Use OpenSSH?

A lot has changed since the commercialization of the Internet began in 1996. It was once a University and Government research network and if you were on the net back then, odds were you were supposed to be there. Though it was far from being utopia, any misbehavior could usually be quickly narrowed down to the individuals involved and dealt with easily, usually with no more than a phone call or a few e-mails. Few, if any, sessions back then were encrypted and both passwords and user names were passed in clear text.

By then, the WWW was more than a few years under way and undergoing explosive growth. The estimated number of web servers online in 1996 grew from 100,000 at the beginning of the year to close to 650,000 by the end of the same year[13]. When other types of servers are included in those figures, the estimated year-end number is over 16,000,000 hosts, representing approximately 828,000 domains.[13]

Nowadays, hosts are subject to hostile scans from the moment they are connected to the network. Any and all unencrypted traffic is scanned and parsed for user names, passwords and other sensitive information. Currently, the biggest espionage threats come from private companies, but governments, individuals, and

organized crime are not without a presences.

Each connection goes through many networks and each packet may take the same or a different route there and back again. Thirteen hops among three organizations in this example from a student computer to a search engine:

```
% /usr/sbin/traceroute -n www.google.com
traceroute: Warning: www.google.com has multiple addresses; using 74.125.95.106
traceroute to www.l.google.com (74.125.95.106), 30 hops max, 40 byte packets
 1 xx.xx.xx.xx           0.419 ms       0.220 ms        0.213 ms        University of Michigan
 2 xx.xx.xx.xx           0.446 ms       0.349 ms        0.315 ms        Merit Network, Inc.
 3 xx.xx.xx.xx           0.572 ms       0.513 ms        0.525 ms        University of Michigan
 4 xx.xx.xx.xx           0.472 ms       0.425 ms        0.402 ms        University of Michigan
 5 xx.xx.xx.xx           0.647 ms       0.551 ms        0.561 ms        University of Michigan
 6 xx.xx.xx.xx           0.945 ms       0.912 ms        0.865 ms        University of Michigan
 7 xx.xx.xx.xx           6.478 ms       6.503 ms        6.489 ms        Merit Network, Inc.
 8 xx.xx.xx.xx           6.597 ms       6.590 ms        6.604 ms        Merit Network, Inc.
 9 216.239.48.154       64.935 ms       6.848 ms        6.793 ms        Google, Inc.
10 72.14.232.141        17.606 ms       17.581 ms       17.680 ms       Google, Inc.
11 209.85.241.27        17.736 ms       17.592 ms       17.519 ms       Google, Inc.
12 72.14.239.193        17.767 ms       17.778 ms       17.930 ms       Google, Inc.
13 74.125.95.106        17.903 ms       17.835 ms       17.867 ms       Google, Inc.:
```

The net is big. It is not uncommon to find a trail of 15 to 20 hops between client and server nowadays. Any machine on any of the subnets the packets travel over can eavesdrop with little difficulty if the packets are not well encrypted.

What OpenSSH Does

The OpenSSH suite gives the following:

- Encrypted remote access, including tunneling insecure protocols.
- Encrypted file transfer
- Run remote commands, programs or scripts and, as mentioned,
- Replacement for **rsh**, **rlogin**, **telnet** and **ftp**

More concretely, that means that the following undesirable activities are prevented:

- Eavesdropping of data transmitted over the network.
- Manipulation of data at intermediate elements in the network (e.g. routers).
- Address spoofing where an attack hosts pretends to be a trusted host by sending packets with the source address of the trusted host.
- IP source routing

As a free software project, OpenSSH provides:

- Open Standards
- Flexible License - freedom emphasized for developers
- Strong Encryption using these ciphers:
  - AES
  - ChaCha20-Poly1305[14]
- Strong Authentication
  - Public Key
  - Single Use Passwords
  - Kerberos
  - Dongles

- Built-in SFTP
- Data Compression
- Port Forwarding
  - Encrypt legacy protocols
  - Encrypted X11 forwarding for X Window System
- Key Agents
- Single Sign-on using
  - Authentication Keys
  - Agent Forwarding
  - Ticket Passing
  - Kerberos
  - AFS

What OpenSSH Doesn't Do

OpenSSH is a very useful tool, but much of its effectiveness depends on correct use. It cannot protect from any of the following situations.

- Misconfiguration, misuse or abuse.
- Compromised systems, particularly where the root account is compromised.
- Insecure or inappropriate directory settings, particularly home directory settings.

OpenSSH must be properly configured and on a properly configured system in order to be of benefit. Arranging both is not difficult, but since each system is unique, there is no one-size-fits-all solution. The right configuration is dependent on the uses the system and OpenSSH are put to.

If you login from a host to a server and an attacker has control of root on either side, he can listen to your session by reading from the pseudo-terminal device, even though SSH is encrypted on the network SSH must communicate in clear text with the terminal device.

If an attacker can change files in your home directory, for example via a networked file system, he may be able to fool SSH.

Last but not least, if OpenSSH is set to allow everyone in, whether on purpose or by accident, it will.

References

1. "Statistics from the current scan results". OpenSSH.org. 2008. http://www.openssh.com/usage/ssh-stats.html.
2. "OpenSSH History". OpenSSH. http://openssh.org/history.html. Retrieved 2012-11-17.
3. "UNIX History Timeline". Éric Lévénez. http://www.levenez.com/unix/. Retrieved 2011-02-17.
4. "Hobbes' Internet Timeline". Robert H'obbes' Zakon. http://www.zakon.org/robert/internet/timeline/. Retrieved 2011-02-17.
5. Howard Dahdah (2009). "The A-Z of Programming Languages: Bourne shell, or sh". Computerworld. http://www.computerworld.com.au/article/279011/a-z_programming_languages_bourne_shell_sh/. Retrieved 2011-02-18.
6. Phil Zimmermann (1991). "Why I Wrote PGP". Massachusetts Institute of Technology. http://www.mit.edu/~prz/EN/essays/WhyIWrotePGP.html. Retrieved 2011-02-18.
7. "Designing an Authentication System: a Dialogue in Four Scenes.". 1988. http://web.mit.edu/Kerberos/dialogue.html. Retrieved 2011-02-17.
8. "Help:SSH 1.0.0 license". FUNET. ftp://ftp.funet.fi/pub/mirrors/ftp.cs.hut.fi/pub/ssh/old/ssh-

1.0.0.tar.gz. Retrieved 2013-04-13.
9. Tatu Ylönen (1995-07-12). "ANNOUNCEMENT: Ssh (Secure Shell) Remote Login Program". news://comp.security.unix. https://groups.google.com/group/comp.security.unix/msg/67079d812a19f499?dmode=source&hl=en&pli=1. Retrieved 2011-11-26.
10. "Help:SSH 1.2.12 license". friedl. http://wwwcip.informatik.uni-erlangen.de/~msfriedl/LIC/ssh-1.2.12/COPYING. Retrieved 2011-02-17.
11. "Help:SSH 1.2.12.92 license". friedl. http://wwwcip.informatik.uni-erlangen.de/~msfriedl/LIC/ssh-1.2.12.92/COPYING. Retrieved 2011-02-17.
12. "OpenSSH Project History and Credits". OpenSSH. http://www.openssh.com/history.html. Retrieved 2011-03-10.
13. Robert H'obbes' Zakon. "Hobbes' Internet Timeline". Zakon Group LLC. http://www.zakon.org/robert/internet/timeline/. Retrieved 2011-02-17.
14. Damien Miller (2013-11-29). "ChaCha20 and Poly1305 in OpenSSH". http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html. Retrieved 2014-04-26.

Why Use Encryption

Encryption has been a hot topic in computing for a long time. It became a high priority item in national and international politics in 1991 when Dr. Phil Zimmermann at MIT first published Pretty Good Privacy (PGP). Encryption went from a specialty to a requirement with the arrival the first web shops and increasing volumes of money changed hands online. By 1996 encryption became essential for e-business. By 2000 it became recognized as a general, essential prerequisite in electronic communication. Currently, 2010, there is almost no chance of maintaining control over or integrity of any networked machine for more than a few minutes without the help of encryption.

Currently much communication over computer networks is still done without encryption. That would be most communication, if inadequate encryption is also taken into account. This is despite years of warnings, government recommendations, best practice guidelines and incidents. As a result, any machine connected to the network can intercept communication that passes over that network. The eavesdroppers are many and varied. They include administrators, staff, employers, criminals, corporate spies, and even governments. Corporate espionage alone has become an enormous burden and barrier.

Businesses are well aware of dumpster diving and take precautions to shred all paper documents. But what about electronic information? Contracts and negotiations, trade secrets, patent applications, decisions and minutes, customer data and invoicing, personnel data, financial and tax records, calendars and schedules, product designs and production notes, training materials, and even regular correspondence go over the net daily. Archived materials, even if they are not accessed directly, are usually on machines that are available and accessed for other reasons.

Many company managers and executives are still unaware that their communications and documents are so easily intercepted, in spite of apparent and expensive access restrictions. In many cases these can be show to be ineffectual and at best purely cosmetic. Security Theater is one aspect and in the field of security it is more common to find snake oil than authentic solutions. Still, there is little public demonstration of awareness of the magnitude of corporate espionage nowadays or the cost of failure. Even failure to act has its costs. Not only is sensitive data available if left unencrypted, but also trends in less sensitive data can be spotted with a large enough sampling. A very large amount of information can be inferred even from lesser communications. Data mining is now a well-known concept as is the so-called wireless wiretap. With the increase in online material and activity, encryption is more relevant than ever even if many years have

passed since the issues were first brought into the limelight.

Excerpt of ssh-1.0.0 README from July 12, 1995

Tatu Ylönen, then at the Helsinki University of Technology, wrote the README[1] accompanying the early versions of his Open Source software, SSH. The following is an excerpt about why encryption is important.

> ssh-1.0.0 README 1995-07-12
>
> ...
>
> WHY TO USE SECURE SHELL
>
> Currently, almost all communications in computer networks are done without encryption. As a consequence, anyone who has access to any machine connected to the network can listen in on any communication. This is being done by hackers, curious administrators, employers, criminals, industrial spies, and governments. Some networks leak off enough electromagnetic radiation that data may be captured even from a distance.
>
> When you log in, your password goes in the network in plain text. Thus, any listener can then use your account to do any evil he likes. Many incidents have been encountered worldwide where crackers have started programs on workstations without the owners knowledge just to listen to the network and collect passwords. Programs for doing this are available on the Internet, or can be built by a competent programmer in a few days.
>
> Any information that you type or is printed on your screen can be monitored, recorded, and analyzed. For example, an intruder who has penetrated a host connected to a major network can start a program that listens to all data flowing in the network, and whenever it encounters a 16-digit string, it checks if it is a valid credit card number (using the check digit), and saves the number plus any surrounding text (to catch expiration date and holder) in a file. When the intruder has collected a few thousand credit card numbers, he makes smallish mail-order purchases from a few thousand stores around the world, and disappears when the goods arrive but before anyone suspects anything.
>
> Businesses have trade secrets, patent applications in preparation, pricing information, subcontractor information, client data, personnel data, financial information, etc. Currently, anyone with access to the network (any machine on the network) can listen to anything that goes in the network, without any regard to normal access restrictions.
>
> Many companies are not aware that information can so easily be recovered from the network. They trust that their data is safe since nobody is supposed to know that there is sensitive information in the network, or because so much other data is transferred in the network. This is not a safe policy.
>
> Individual persons also have confidential information, such as diaries, love letters, health care documents, information about their personal interests and habits, professional data, job applications, tax reports, political documents, unpublished manuscripts, etc.
>
> There is also another frightening aspect about the poor security of communications. Computer storage and analysis capability has increased so much that it is feasible for governments, major companies, and criminal organizations to automatically analyze, identify, classify, and file information about millions of people over the years. Because most of the work can be automated, the cost of collecting this information is getting very low.

Government agencies may be able to monitor major communication systems, telephones, fax, computer networks, etc., and passively collect huge amounts of information about all people with any significant position in the society. Most of this information is not sensitive, and many people would say there is no harm in someone getting that information. However, the information starts to get sensitive when someone has enough of it. You may not mind someone knowing what you bought from the shop one random day, but you might not like someone knowing every small thing you have bought in the last ten years.

If the government some day starts to move into a more totalitarian direction, there is considerable danger of an ultimate totalitarian state. With enough information (the automatically collected records of an individual can be manually analyzed when the person becomes interesting), one can form a very detailed picture of the individual's interests, opinions, beliefs, habits, friends, lovers, weaknesses, etc. This information can be used to 1) locate any persons who might oppose the new system 2) use deception to disturb any organizations which might rise against the government 3) eliminate difficult individuals without anyone understanding what happened. Additionally, if the government can monitor communications too effectively, it becomes too easy to locate and eliminate any persons distributing information contrary to the official truth.

Fighting crime and terrorism are often used as grounds for domestic surveillance and restricting encryption. These are good goals, but there is considerable danger that the surveillance data starts to get used for questionable purposes. I find that it is better to tolerate a small amount of crime in the society than to let the society become fully controlled. I am in favor of a fairly strong state, but the state must never get so strong that people become unable to spread contra-offical information and unable to overturn the government if it is bad. The danger is that when you notice that the government is too powerful, it is too late. Also, the real power may not be where the official government is.

For these reasons (privacy, protecting trade secrets, and making it more difficult to create a totalitarian state), I think that strong cryptography should be integrated to the tools we use every day. Using it causes no harm (except for those who wish to monitor everything), but not using it can cause huge problems. If the society changes in undesirable ways, then it will be to late to start encrypting.

Encryption has had a "military" or "classified" flavor to it. There are no longer any grounds for this. The military can and will use its own encryption; that is no excuse to prevent the civilians from protecting their privacy and secrets. Information on strong encryption is available in every major bookstore, scientific library, and patent office around the world, and strong encryption software is available in every country on the Internet.

Some people would like to make it illegal to use encryption, or to force people to use encryption that governments can break. This approach offers no protection if the government turns bad. Also, the "bad guys" will be using true strong encryption anyway. Thus, any "key escrow encryption" or whatever it might be called only serves to help monitor the ordinary people and petty criminals; it does not help against powerful criminals, terrorists, or espionage, because they will know how to use strong encryption anyway.

...

Thanks also go to Philip Zimmermann, whose PGP software and the associated legal battle provided inspiration, motivation, and many useful techniques, and to Bruce Schneier whose book Applied Cryptography has done a great service in widely distributing knowledge about cryptographic methods.

...

ssh-1.0.0 README 1995-07-12

Phil Zimmermann on encryption and privacy, from 1991, updated 1999

Phil Zimmermann wrote the encryption tool Pretty Good Privacy (PGP) in 1991 to promote privacy and to help keep encryption, and thus privacy, legal around the world. Considerable difficulty occurred in the United States until PGP was published outside and re-imported in a very visible, public manner.

Why I Wrote PGP
Part of the Original 1991 PGP User's Guide (updated in 1999)

"Whatever you do will be insignificant, but it is very important that you do it."

–Mahatma Gandhi.

It's personal. It's private. And it's no one's business but yours. You may be planning a political campaign, discussing your taxes, or having a secret romance. Or you may be communicating with a political dissident in a repressive country. Whatever it is, you don't want your private electronic mail (email) or confidential documents read by anyone else. There's nothing wrong with asserting your privacy. Privacy is as apple-pie as the Constitution.

The right to privacy is spread implicitly throughout the Bill of Rights. But when the United States Constitution was framed, the Founding Fathers saw no need to explicitly spell out the right to a private conversation. That would have been silly. Two hundred years ago, all conversations were private. If someone else was within earshot, you could just go out behind the barn and have your conversation there. No one could listen in without your knowledge. The right to a private conversation was a natural right, not just in a philosophical sense, but in a law-of-physics sense, given the technology of the time.

But with the coming of the information age, starting with the invention of the telephone, all that has changed. Now most of our conversations are conducted electronically. This allows our most intimate conversations to be exposed without our knowledge. Cellular phone calls may be monitored by anyone with a radio. Electronic mail, sent across the Internet, is no more secure than cellular phone calls. Email is rapidly replacing postal mail, becoming the norm for everyone, not the novelty it was in the past.

Until recently, if the government wanted to violate the privacy of ordinary citizens, they had to expend a certain amount of expense and labor to intercept and steam open and read paper mail. Or they had to listen to and possibly transcribe spoken telephone conversation, at least before automatic voice recognition technology became available. This kind of labor-intensive monitoring was not practical on a large scale. It was only done in important cases when it seemed worthwhile. This is like catching one fish at a time, with a hook and line. Today, email can be routinely and automatically scanned for interesting keywords, on a vast scale, without detection. This is like driftnet fishing. And exponential growth in computer power is making the same thing possible with voice traffic.

Perhaps you think your email is legitimate enough that encryption is unwarranted. If you really are a law-abiding citizen with nothing to hide, then why don't you always send your paper mail on postcards? Why not submit to drug testing on demand? Why require a warrant for police searches of your house? Are you trying to hide something? If you hide your mail inside

envelopes, does that mean you must be a subversive or a drug dealer, or maybe a paranoid nut? Do law-abiding citizens have any need to encrypt their email?

What if everyone believed that law-abiding citizens should use postcards for their mail? If a nonconformist tried to assert his privacy by using an envelope for his mail, it would draw suspicion. Perhaps the authorities would open his mail to see what he's hiding. Fortunately, we don't live in that kind of world, because everyone protects most of their mail with envelopes. So no one draws suspicion by asserting their privacy with an envelope. There's safety in numbers. Analogously, it would be nice if everyone routinely used encryption for all their email, innocent or not, so that no one drew suspicion by asserting their email privacy with encryption. Think of it as a form of solidarity.

Senate Bill 266, a 1991 omnibus anticrime bill, had an unsettling measure buried in it. If this non-binding resolution had become real law, it would have forced manufacturers of secure communications equipment to insert special "trap doors" in their products, so that the government could read anyone's encrypted messages. It reads, "It is the sense of Congress that providers of electronic communications services and manufacturers of electronic communications service equipment shall ensure that communications systems permit the government to obtain the plain text contents of voice, data, and other communications when appropriately authorized by law." It was this bill that led me to publish PGP electronically for free that year, shortly before the measure was defeated after vigorous protest by civil libertarians and industry groups.

The 1994 Communications Assistance for Law Enforcement Act (CALEA) mandated that phone companies install remote wiretapping ports into their central office digital switches, creating a new technology infrastructure for "point-and-click" wiretapping, so that federal agents no longer have to go out and attach alligator clips to phone lines. Now they will be able to sit in their headquarters in Washington and listen in on your phone calls. Of course, the law still requires a court order for a wiretap. But while technology infrastructures can persist for generations, laws and policies can change overnight. Once a communications infrastructure optimized for surveillance becomes entrenched, a shift in political conditions may lead to abuse of this new-found power. Political conditions may shift with the election of a new government, or perhaps more abruptly from the bombing of a federal building.

A year after the CALEA passed, the FBI disclosed plans to require the phone companies to build into their infrastructure the capacity to simultaneously wiretap 1 percent of all phone calls in all major U.S. cities. This would represent more than a thousandfold increase over previous levels in the number of phones that could be wiretapped. In previous years, there were only about a thousand court-ordered wiretaps in the United States per year, at the federal, state, and local levels combined. It's hard to see how the government could even employ enough judges to sign enough wiretap orders to wiretap 1 percent of all our phone calls, much less hire enough federal agents to sit and listen to all that traffic in real time. The only plausible way of processing that amount of traffic is a massive Orwellian application of automated voice recognition technology to sift through it all, searching for interesting keywords or searching for a particular speaker's voice. If the government doesn't find the target in the first 1 percent sample, the wiretaps can be shifted over to a different 1 percent until the target is found, or until everyone's phone line has been checked for subversive traffic. The FBI said they need this capacity to plan for the future. This plan sparked such outrage that it was defeated in Congress. But the mere fact that the FBI even asked for these broad powers is revealing of their agenda.

Advances in technology will not permit the maintenance of the status quo, as far as privacy is concerned. The status quo is unstable. If we do nothing, new technologies will give the government new automatic surveillance capabilities that Stalin could never have dreamed of. The only way to hold the line on privacy in the information age is strong cryptography.

You don't have to distrust the government to want to use cryptography. Your business can be wiretapped by business rivals, organized crime, or foreign governments. Several foreign governments, for example, admit to using their signals intelligence against companies from other countries to give their own corporations a competitive edge. Ironically, the United States government's restrictions on cryptography in the 1990's have weakened U.S. corporate defenses against foreign intelligence and organized crime.

The government knows what a pivotal role cryptography is destined to play in the power relationship with its people. In April 1993, the Clinton administration unveiled a bold new encryption policy initiative, which had been under development at the National Security Agency (NSA) since the start of the Bush administration. The centerpiece of this initiative was a government-built encryption device, called the Clipper chip, containing a new classified NSA encryption algorithm. The government tried to encourage private industry to design it into all their secure communication products, such as secure phones, secure faxes, and so on. AT&T put Clipper into its secure voice products. The catch: At the time of manufacture, each Clipper chip is loaded with its own unique key, and the government gets to keep a copy, placed in escrow. Not to worry, though–the government promises that they will use these keys to read your traffic only "when duly authorized by law." Of course, to make Clipper completely effective, the next logical step would be to outlaw other forms of cryptography.

The government initially claimed that using Clipper would be voluntary, that no one would be forced to use it instead of other types of cryptography. But the public reaction against the Clipper chip was strong, stronger than the government anticipated. The computer industry monolithically proclaimed its opposition to using Clipper. FBI director Louis Freeh responded to a question in a press conference in 1994 by saying that if Clipper failed to gain public support, and FBI wiretaps were shut out by non-government-controlled cryptography, his office would have no choice but to seek legislative relief. Later, in the aftermath of the Oklahoma City tragedy, Mr. Freeh testified before the Senate Judiciary Committee that public availability of strong cryptography must be curtailed by the government (although no one had suggested that cryptography was used by the bombers).

The government has a track record that does not inspire confidence that they will never abuse our civil liberties. The FBI's COINTELPRO program targeted groups that opposed government policies. They spied on the antiwar movement and the civil rights movement. They wiretapped the phone of Martin Luther King. Nixon had his enemies list. Then there was the Watergate mess. More recently, Congress has either attempted to or succeeded in passing laws curtailing our civil liberties on the Internet. Some elements of the Clinton White House collected confidential FBI files on Republican civil servants, conceivably for political exploitation. And some overzealous prosecutors have shown a willingness to go to the ends of the Earth in pursuit of exposing sexual indiscretions of political enemies. At no time in the past century has public distrust of the government been so broadly distributed across the political spectrum, as it is today.

Throughout the 1990s, I figured that if we want to resist this unsettling trend in the government to outlaw cryptography, one measure we can apply is to use cryptography as much as we can now while it's still legal. When use of strong cryptography becomes popular, it's harder for the government to criminalize it. Therefore, using PGP is good for preserving democracy. If privacy is outlawed, only outlaws will have privacy.

It appears that the deployment of PGP must have worked, along with years of steady public outcry and industry pressure to relax the export controls. In the closing months of 1999, the Clinton administration announced a radical shift in export policy for crypto technology. They essentially threw out the whole export control regime. Now, we are finally able to export strong cryptography, with no upper limits on strength. It has been a long struggle, but we have finally

won, at least on the export control front in the US. Now we must continue our efforts to deploy strong crypto, to blunt the effects increasing surveillance efforts on the Internet by various governments. And we still need to entrench our right to use it domestically over the objections of the FBI.

PGP empowers people to take their privacy into their own hands. There has been a growing social need for it. That's why I wrote it.

Philip R. Zimmermann
Boulder, Colorado
June 1991 (updated 1999)[2]

Original Press Release for OpenSSH

Below is the original press release for OpenSSH sent back in 1999.[3]

Date: Mon, 25 Oct 1999 00:04:29 -0600 (MDT)
From: Louis Bertrand <louis@cvs.openbsd.org>
To: Liz Coolbaugh <lwn@lwn.net>
Subject: OpenBSD Press Release: OpenSSH integrated into operating system

PRESS RELEASE

OpenSSH: Secure Shell integrated into OpenBSD operating system

Source: OpenBSD
Contacts:

Louis Bertrand, OpenBSD media relations
Bertrand Technical Services
Tel: (905) 623-8925 Fax: (905) 623-3852
louis@openbsd.org

Theo de Raadt, OpenBSD lead developer
deraadt@openbsd.org

Project Web site: http://www.openbsd.org/

OpenSSH: Secure Shell integrated into OpenBSD Secure communications package no longer third-party add-on

[October 25, 1999: Calgary, Canada] -- The OpenBSD developers are pleased to announce the release of OpenSSH, a free implementation of the popular Secure Shell encrypted communications package. OpenSSH, to be released with OpenBSD 2.6, is compatible with both SSH 1.3 and 1.5 protocols and dodges most restrictions on the free distribution of strong cryptography.

OpenSSH is based on a free release of SSH by Tatu Ylonen, with major changes to remove proprietary code and bring it up to current security and functionality standards. Secure Shell operates like the popular TELNET remote terminal package but with an encrypted link between the user and the remote server. SSH also allows "tunnelling" of network services through the scrambled connection for added privacy. OpenSSH has been tested to interoperate with ssh-1.2.27 from SSH Communications, and the TTSSH and SecureCRT Windows clients.

"Network sessions involving strong cryptographic security are a requirement in the modern world," says lead developer Theo de Raadt. "Everyone needs this. People using the telnet or rlogin protocols are not aware of the extreme danger posed by password sniffing and session hijacking."

In previous releases of OpenBSD, users were urged to download SSH as soon as possible after installing the OS. Without SSH, terminal sessions transmitted in clear text allow eavesdroppers on the Internet to capture user names and password combinations and thus bypass the security measures in the operating system.

"I asked everyone `what is the first thing you do after installing OpenBSD?' Everyone gave me the same answer: they installed ssh," says de Raadt. "That's a pain, so we've made it much easier."

All proprietary code in the original distribution was replaced, along with some libraries burdened with the restrictive GNU Public License (GPL). Much of of the actual cryptographic code was replaced by calls to the crypto libraries built into OpenBSD. The source code is now completely freely re-useable, and vendors are encouraged to re-use it if they need ssh functionality.

OpenSSH relies on the Secure Sockets Layer library (libssl) which incorporates the RSA public-key cryptography system. RSA is patented in the US and OpenBSD developers must work around the patent restrictions. Users outside the US may download a libssl file based on the patent-free OpenSSL implementation. For US non-commercial users, OpenBSD is preparing a libssl based on the patented RSAREF code. Unfortunately, the US legal framework effectively bans US commercial users from using OpenSSH, and curtails freedom of choice in that market.

OpenSSH was developed and integrated into OpenBSD by Niels Provos, Theo de Raadt, Markus Friedl for cryptographic work; Dug Song, Aaron Campbell, and others for various non-crypto contributions; and Bob Beck for helping with the openssl library issues. The original SSH was written by Tatu Ylonen. Bjoern Groenvall and Holger Trapp did the initial work to free the distribution.

OpenBSD is an Internet-based volunteer effort to produce a secure multi-platform operating system with built-in support for cryptography. It has been described in the press as the world's most secure operating system. For more information about OpenSSH and OpenBSD, see the project Web pages at http://www.OpenBSD.org/.

Source: OpenBSD
http://lwn.net/1999/1028/a/openssh.html

The European Union (EU) on Encryption

During 2000, the European Commission investigated the state of international and industrial electronic espionage. Counter-measures and solutions were investigated as well as the risks. The result was a resolution containing a summary of the findings and a series of recommended actions for Member States to carry out and goals to meet. Recommendations to EU Member States from the European Parliament resolution ECHELON, A5-0264/2001 (emphasis added):

"29. Urges the Commission and Member States to devise appropriate measures to promote, develop and manufacture European encryption technology and software and above all to support projects at developing user-friendly **open-source encryption** software;"

. . .

> "33. Calls on the Community institutions and the public administrations of the Member States to provide training for their staff and make their staff familiar with new **encryption technologies and techniques** by means of the necessary practical training and courses;"[4]

It was found during the investigation that businesses were the most at risk and the most vulnerable and that widespread use of open source encryption technology is to be encouraged. The same can be said even today.

References

1. "SSH 1.0.0 README". FUNET. 1995. ftp://ftp.funet.fi/pub/mirrors/ftp.cs.hut.fi/pub/ssh/old/.
2. Phil Zimmermann (1991). "Why I Wrote PGP". Massachusetts Institute of Technology. http://www.mit.edu/~prz/EN/essays/WhyIWrotePGP.html. Retrieved 2011-02-18.
3. "OpenSSH: Secure Shell integrated into OpenBSD operating system". LWN. 1999. http://lwn.net/1999/1028/a/openssh.html. Retrieved 2011-02-18.
4. "European Parliament resolution on the existence of a global system for the interception of private and commercial communications (ECHELON interception system) (2001/2098(INI))". European Parliament. 2001. http://www.europarl.europa.eu/sides/getDoc.do?type=TA&reference=P5-TA-2001-0441&format=XML&language=EN. Retrieved 2011-02-18.

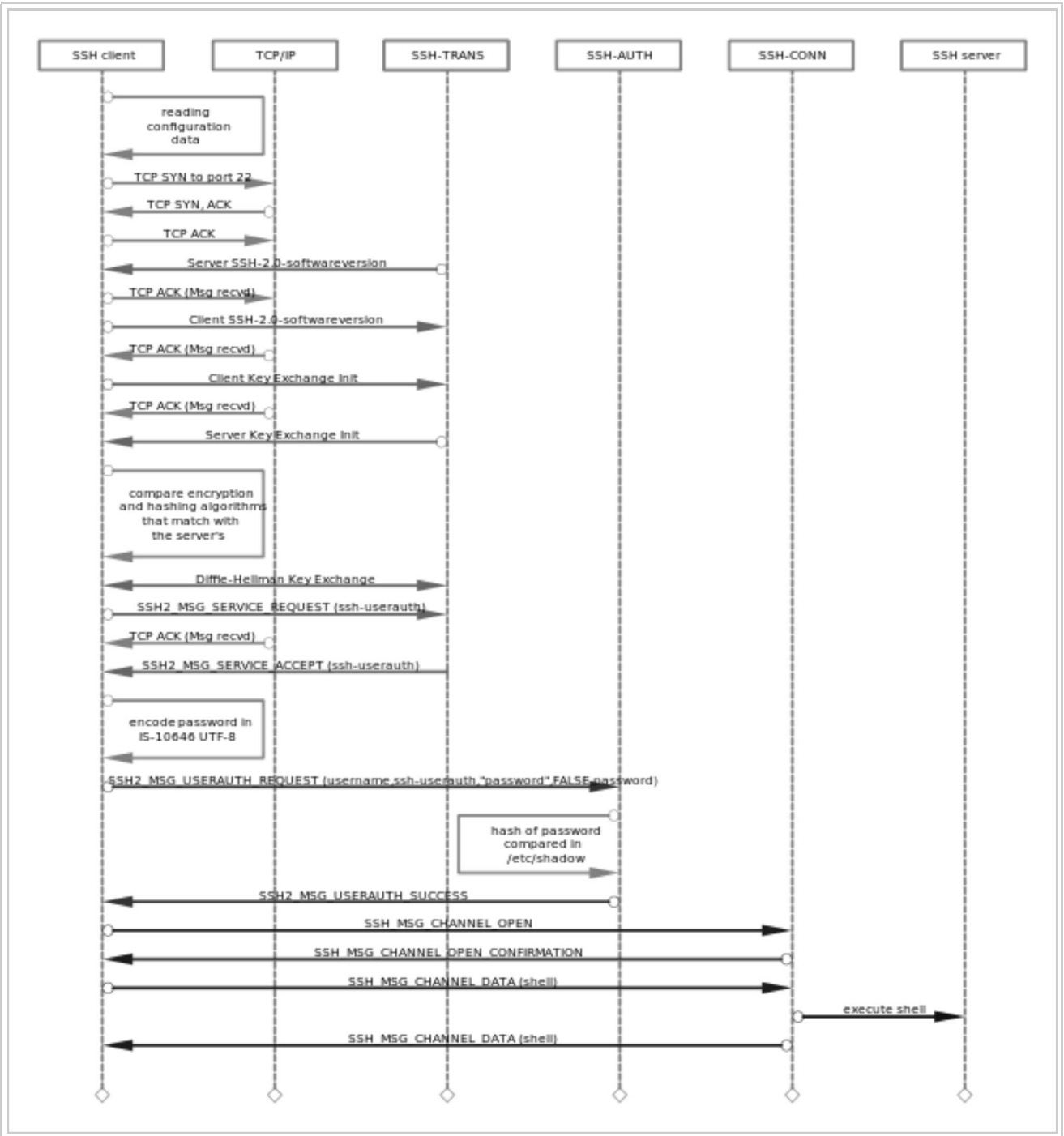
SSH Protocols

The current set of Secure Shell protocols is SSH2. It is a rewrite of the old, deprecated SSH1 protocol. It contains significant improvements in security, performance, and portability. The default is now SSH2. OpenSSH uses the SSH protocol, connecting over TCP. Normally, one SSH session per TCP connection is made, but multiple sessions can be multiplexed over a single TCP connection if planned that way.

The Secure Shell protocol is an open standard. As such, it is vendor-neutral and maintained by the Internet Engineering Task Force (IETF). The current protocol is described in RFC 4250 through RFC 4256 and standardized by the IETF secsh working group. The overall structure of SSH2 is described in RFC 4251, The Secure Shell (SSH) Protocol Architecture.

The SSH protocol is composed of three layers: the transport layer, the authentication layer, and the connection layer.

> **SSH-CONNECT** – The connection layer runs over the user authentication protocol. It multiplexes many different concurrent encrypted channels into logical channels over the authenticated connection. It allows for tunneling of login sessions and TCP-forwarding. It provides a flow control service for these channels. Additionally, various channel-specific options can be negotiated. This layer manages the SSH session, session multiplexing, X11 forwarding, TCP forwarding, shell, remote program execution, invoking SFTP subsystem.
>
> **SSH-USERAUTH** – The user authentication layer authenticates the client-side to the server. It uses the established connection and runs on top of the transport layer. It provides several mechanisms for user authentication. These include password authentication, public-key or host-based authentication mechanisms, challenge-response, pluggable authentication modules (PAM), Generic Security Services API (GSSAPI) and even dongles.
>
> **SSH-TRANS** – The transport layer provides server authentication, confidentiality and data integrity



Sequence Diagram for SSH Password Authentication

over TCP. It does this through algorithm negotiation and a key exchange. The key exchange includes server authentication and results in a cryptographically secured connection: it provides integrity, confidentiality and optional compression. [1]

Among the differences between the current protocol, SSH2, and the deprecated protocol, SSH1, are that SSH2 uses host keys for authentication. Whereas SSH1 used both server and host keys to authenticate. There's not much which can be added about the protocols which is not already covered with more detail and authority in RFC 4251 [2].

SSH File Transfer Protocol (SFTP)

The SSH File Transfer Protocol (SFTP) is a binary protocol to provide secure file transfer, access and management.

SFTP on the server side was added by Markus Friedl in time for the 2.3.0 release of OpenSSH in November 2000. Damien Miller added support for SFTP to the client side in time for 2.5.0. Since then, many have added to both the client and the server.

SFTP is not FTPS

For basic file transfer, nothing more is needed than an account on the machine with the OpenSSH server. SFTP support is built into the OpenSSH server package. The SFTP protocol, in contrast to old FTP, has been designed from the ground up to be as secure as possible for both login and data transfer.

Unless the use-case calls for publicly available, read-only, downloads, don't worry about trying to fiddle with FTP. It is the protocol FTP itself that is inherently insecure. It's great for read-only, public data transfer. The programs vsftpd and proftpd, for example, are secure insofar as the server software itself goes, although the protocol is insecure. In other words the program itself is more or less fine and if you need to provide read-only, publicly available downloads then FTP maybe the right tool. Otherwise forget about FTP. Nearly always when users ask for "FTP" they don't mean specifically the old file transfer protocol from 1971 as described in RFC 114, but a generic means of file transfer and there are many ways to solve that problem. This is especially true since the next part of their request is usually how to make it secure. The name "FTP" is frequently mis-used generically to mean any file transfer utility, much the same way as the term "Coke" is used in some of the Southern States to mean any carbonated soft drink, not just Coca-Cola. Consider SFTP or, for larger groups, even SSHFS, Samba or AFS. While old FTP succeeded very well in achieving its main goal to promote use of networked computers by allowing users at any host on the network to use the file system of any cooperating host, it cannot be made secure. There's nothing to be done about that, so it is time to get over it.

Again, it is the protocol itself, FTP, which is the problem.[3] With FTP, the data, passwords and user name are all sent back and forth unencrypted.[4] Anyone on the client's subnet, the server's subnet or any subnet in between can 'sniff' the passwords and data when FTP is used. With extra effort it is possible to wrap FTP inside SSL or TLS, thus creating FTPS. However, tunneling FTP over SSL/TLS is complex to do and far from an optimum solution.

Unfortunately name confusion combined with the large number of posts and discussions created by complex, nit-picky tasks like wrapping FTP in SSL to provide FTPS, the wrong way still turns up commonly in web searches regarding file transfer. In contrast, easy, relatively painless solutions vanish because it is rarely necessary to post how to do those. Also, an easy solution can be summed up in very few lines and maybe a single answer. Thus, there is still a lot of talk online about 'securing' FTP and very little mention of using SFTP. It's a vicious cycle that this book hopes to help break: Difficult task means lots of discussion and noise, lots of discussion and noise means strong web presence, strong web presence means high Google ranking.

SFTP tools are very common, but might be taken for granted and thus overlooked. SFTP tools are just as easy to use and more functional than old FTP clients. In fact a lot of improvements have been made in usability. There is no shortage of common, GUI-based SFTP clients to transfer files: Filezilla, Konqueror, Dolphin, Nautilus, Cyberduck, Fugu, and Fetch top the list but there are many more. Most are Free Software. These SFTP clients are very easy to use. For example, in Konqueror, just type in the URL to the sftp server, where the server name or address is xx.yy.zz.aa.

```
sftp://xx.yy.zz.aa
```

If it is desirable to start with a specific directory, then that too can be specified.

```
sftp://xx.yy.zz.aa/var/www/pictures/
```

One special client worth knowing about is sshfs. With sshfs as an SFTP client the other machine is accessible as an open folder on your machine's local file system. In that way any program you normally have to work with files, such as LibreOffice, Inkscape or Gimp can access the remote machine via that folder.

Background of FTP

FTP is from the 1970s. It's a well proven workhorse, but from an era when if you were on the net you were supposed to be there and if there was trouble it could usually be cleared up with a short phone call or an e-mail or two. It sends the login name, password and all of the data unencrypted for anyone to intercept. FTP clients can connect to the FTP server in either passive or active modes. Both active and passive modes for FTP use two ports, one for control and one for data. In FTP Active mode, after the client makes a connection to the FTP server it then allows an incoming connection to be initiated from the server to for data transfer. In FTP Passive mode, after the client makes a connection to the FTP server, the server then responds with information about a second port for data transfer and the client initiates the second connection. Thus FTP is most relevant now as Anonymous FTP, which is still excellent for read-only downloads without login. To transfer read-only data, FTP is still one way to go, as would be using the web (HTTP or HTTPS) or a P2P protocol like bittorrent are other options for offering read-only downloads.
Using tcpdump to show FTP activity

An illustration of how the old protocol, FTP, is insecure can be had from the utility **tcpdump**. It can show what is going over the network during an Anonymous FTP session, or for that matter any FTP session. Look at the manual page for **tcpdump** for an explanation of the individual arguments, but the overall result of the usage below is that it displays the first FTP or FTP-Data packets going from the client to the server and vice versa.

The output below hows an excerpt from the output of **tcpdump** which captured packets between an FTP client and the FTP server, one line per packet.

```
$ sudo tcpdump -q -s 0 -c 10 -A -i eth0 \
"tcp and ( port ftp or port ftp-data) "
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
…
:18:36.010820 IP desk.55227 > server.ftp: tcp 16 E..D..@.@.....1.[.X.....G.r. ........l..... ."......USE
:18:36.073192 IP server.ftp > desk.55227: tcp 0 E..4jX@.7.3.[.X...1..... ...G.r#.......... ....."..
:18:36.074019 IP server.ftp > desk.55227: tcp 34 E..VjY@.7.3.[.X...1..... ...G.r#....Y..... ....."..331
:18:36.074042 IP desk.55227 > server.ftp: tcp 0 E..4..@.@..+..1.[.X.....G.r# ..)..........  ."......
:18:42.098941 IP desk.55227 > server.ftp: tcp 23 E..K..@.@.....1.[.X.....G.r# ..)....gv..... ."..w....PAS
:18:42.162692 IP server.ftp > desk.55227: tcp 23 E..KjZ@.7.3.[.X...1..... ..)G.r:..........  ....."..w230
…
:18:43.431827 IP server.ftp > desk.55227: tcp 14 E..Bj\@.7.3.[.X...1..... ..SG.rF.....j..... ....."..221
…
```

As can be seen in lines 3 and 7, data, such as text, from the server is visible. In lines 1 and 5, text entered by the user is visible, in this case including the user name and password used to log in. Fortunately the session is Anonymous FTP, which is read-only and used for downloading. Anonymous FTP is a rather efficient way to publish material for download. For Anonymous FTP, the user name is always "anonymous" and the password the user’s e-mail address and the server's data always read-only.

If you have the package openssh-server already installed, no further configuration of the server is needed to start using SFTP for file transfers. Though comparatively speaking, FTPS is significantly more secure than FTP. If you want remote remote login access, then both FTP and FTPS should be avoided. A very large reason to avoid both is to save work.

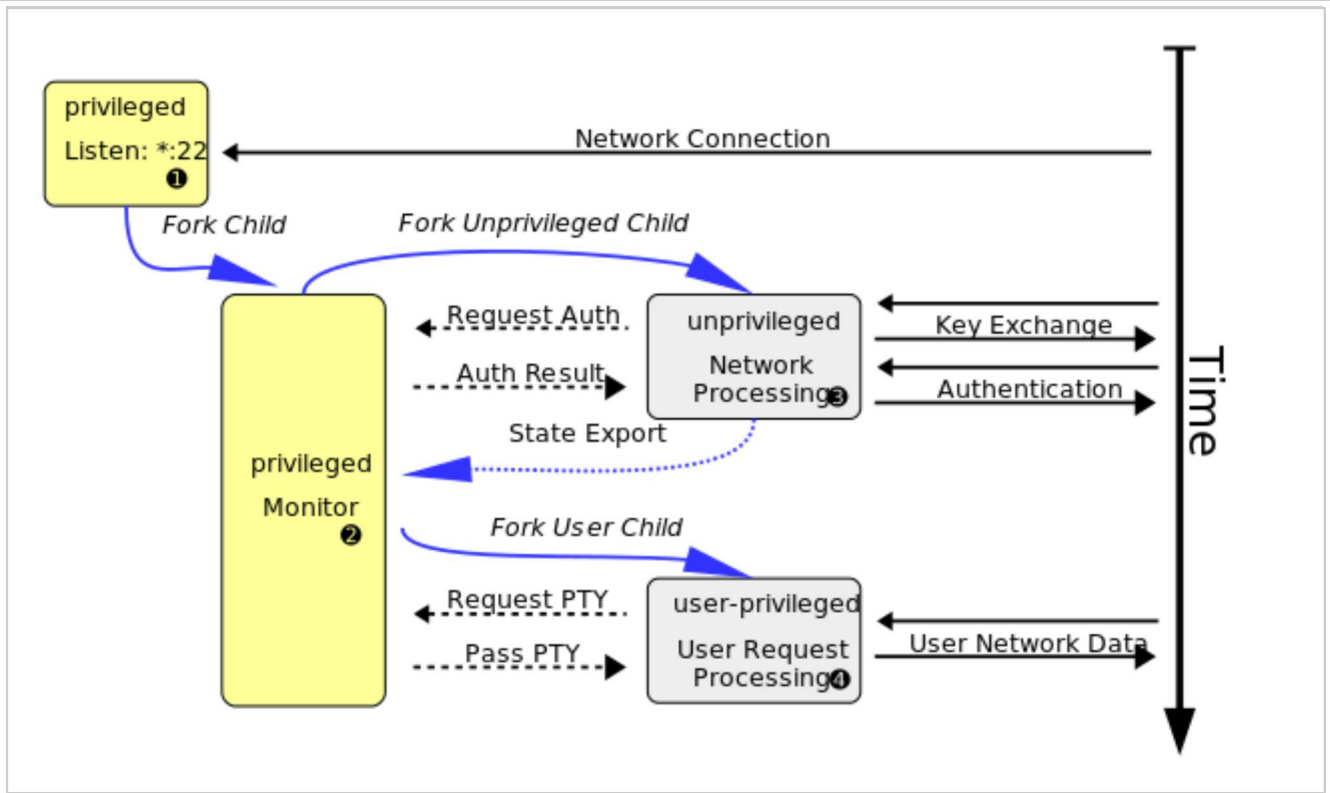
On FTPS

FTPS is FTP tunneled over SSL or TLS. A goal of FTP was to encourage the use of remote computers. It,

along with the web, has succeeded. A goal of FTPS was to secure logins and transfers, and was a necessary step in securing file transfers with the legacy protocol.

Since SFTP is so much easier to deploy and most systems now include both graphical and text-based SFTP clients, FTPS can really be considered deprecated for most occasions. Some good background material can be found in the Request for Comments (RFCs) for FTP and FTPS. There, SFTP and even HTTPS are better matches and largely supercede FTPS. See the section on Client Applications for an idea of the SFTP clients available.

Privilege Separation



Sequence Diagram for OpenSSH Privilege Separation

Privilege separation is when a process is divided into sub-processes, each of which have just enough access to just the right services to do their part of the job. An underlying principle is that of least privilege, which is where each process has exactly enough privileges to accomplish a task, neither more nor less. The goal of privilege separation is to compartmentalize any corruption and prevent a corrupt process from accessing the entire system. Privilege separation is applied in OpenSSH by using several levels of access, some higher some lower, to run sshd(8) (http://man.openbsd.org/sshd.8) and its subsystems and components. The server ❶ starts out with a privileged process ❷ which then creates an unprivileged process ❸ to work with the network traffic. Once the user has authenticated, another unprivileged process is created ❹ with the privileges of that authenticated user. See the "Sequ So as seen in the diagram, a total of four processes get run to create an SSH session. One, the server, remains and listens for new connections and spawn child processes. Here it is seen waiting for a connection.

```
$ ps -ax -o user,pid,ppid,state,start,command | awk '/sshd/ || NR==1'
USER       PID  PPID S  STARTED COMMAND
root      1473     1 S 05:44:12 /usr/sbin/sshd
```

It is this privileged process that listens for the initial connection from clients.

```
$ netstat -ntlp | awk '/sshd/ || NR<=2'
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1473/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      1473/sshd
```

After the initial connection while waiting for password authentication from user 'fred', a privileged monitor process supervises an unprivileged process by user 'sshd' which handles contact with the remote user's client.

```
$ ps -ax -o user,pid,ppid,state,start,command | awk '/sshd/ || NR==1'
USER       PID  PPID S  STARTED COMMAND
root      1473     1 S 05:44:12 /usr/sbin/sshd
root      9481  1473 S 14:40:37 sshd: fred [priv]
sshd      9482  9481 S 14:40:37 sshd: fred [net]
```

Then after authentication is completed, an established session for user 'fred', a new privieged monitor process supervises a process running as user 'fred'.

```
$ ps -ax -o user,pid,ppid,state,start,command | awk '/sshd/ || NR==1'
USER       PID  PPID S  STARTED COMMAND
root      1473     1 S 05:44:12 /usr/sbin/sshd
root      9481  1473 S 14:40:37 sshd: fred [priv]
fred      9579  9481 S 14:42:02 sshd: fred@pts/30
```

Privilege separation has been the default in OpenSSH since version 3.3[5] Since version 5.9, privilege separation further applies mandatory restrictions on the system calls the privilege separated child can perform. The intent is to prevent a compromised privilege separated child from being used to attack other hosts either by opening sockets and proxying or by probing local kernel attack surface. [6] Since version 6.1, this sandboxing is the default.

References

1. "OpenSSH Manual Pages". OpenSSH. http://www.openssh.com/manual.html. Retrieved 2011-02-17.
2. "RFC 4251: The Secure Shell (SSH) Protocol Architecture". 2006-01. http://tools.ietf.org/html/rfc4251. Retrieved 2013-10-31.
3. "Why You Need To Stop Using FTP". JDPFu.com. 2011-07-10. http://blog.jdpfu.com/2011/07/10/why-you-need-to-stop-using-ftp. Retrieved 2012-01-09.
4. Manolis Tzanidakis (2011-09-09). "Stop Using FTP! How to Transfer Files Securely". Wazi. http://olex.openlogic.com/wazi/2011/stop-using-ftp-how-to-transfer-files-securely/. Retrieved 2012-01-09.
5. Nils Provos. "Privilege Separated OpenSSH". University of Michigan. http://www.citi.umich.edu/u/provos/ssh/privsep.html. Retrieved 2011-02-17.
6. "OpenSSH 5.9 Release Notes". OpenSSH. 2011-09-06. http://www.openssh.com/txt/release-5.9. Retrieved 2012-11-17.

Other SSH Implementations

Dropbear

Dropbear (https://matt.ucc.asn.au/dropbear/dropbear.html) is a smaller, modular, open source SSH2 client and server available for all regular POSIX platforms. Dropbear is partially a derivative of OpenSSH. It is often used in embedded systems because very small binaries can be produced. Functions that are not needed can be left out of the binary, leaving a lean executable. Thus a working server can be boiled down to 110KB by trimming away various functions.

Many distributions and products use Dropbear. This includes OpenWRT, gumstix, Tomato Firmware, PSPSSH, DSLinux, Meego, OpenMoko, Ångström (for Zaurus), ttylinux, Sisela, Trinux, SliTaz, Netcomm, US Robotics, some Motorola phones, and many, many more.

- https://matt.ucc.asn.au/dropbear/dropbear.html

Tectia

Tectia (http://www.tectia.com/en.iw3) is from SSH Communications Security Corporation which is based in Finland. It is a closed-source SSH client and server with FIPS support.

- http://www.ssh.com/?option=com_content&view=article&id=236&Itemid=364

Solaris Secure Shell (SunSSH)

Sun SSH (http://wikis.sun.com/display/SunSSH/SunSSH+FAQ) is fork of OpenSSH 2.3, with many subsequent changes.

- http://hub.opensolaris.org/bin/view/Community+Group+security/SSH

GlobalSCAPE EFT Server

EFT Server (http://www.globalscape.com/eft/) is a closed binary that can include SSH and SFTP modules as extensions.

- http://www.globalscape.com/eft/

Client Applications

On the client side, **ssh**, **scp** and **sftp** provide a wide range of capabilities. Interactive logins and file transfers are just the tip of the iceberg.

ssh(1) (http://man.openbsd.org/ssh.1) - The basic login shell-like client program.
sftp(1) (http://man.openbsd.org/sftp.1) - FTP-like program that works using the SSH protocol.
scp(1) (http://man.openbsd.org/scp.1) - File copy program that acts like rcp(1).

ssh_config(5) (http://man.openbsd.org/ssh_config.5) - The client configuration file.

The SSH client

ssh is a program which provides the client side for secure, encrypted communications between hosts over an insecure network. Its main use is for logging into and running programs on a remote host. It can also be used

to secure remote X11 connections and and forward arbitrary TCP ports to secure legacy protocols. ssh was made, in part, to replace insecure tools like **rsh** and **telnet**. It has largely succeeded at this goal. **rsh** and **telnet** are rarely seen anymore for interactive sessions or anywhere else. ssh can authenticate using regular passwords or with the help of a public-private key pair. More options, such as use of Kerberos, smartcards, or one-time passwords can be configured.

Remote login, authenticating via password:

```
$ ssh fred@somehost.example.org
```

Another way of logging in to the same account:

```
$ ssh -l fred somehost.example.org
```

Remote programs can be run interactively when the client is run via the shell on the remote host. Or they can be run directly when passed as an argument to the SSH client. They can even be pre-configured in the authentication key or the server configuration.

Run **uname** on the remote machine:

```
$ ssh -l fred  somehost.example.org  "uname -a"
```

See what file systems are mounted and how much space is used there:

```
$ ssh -l fred  somehost.example.org  "mount; df -h"
```

It is possible to configure in great detail which programs are allowed by which accounts. There are many combinations of options that give extra capabilities, such as re-using a single connection for multiple sessions or passing through intermediary machines. The level of granularity can be increased even more with the help of sudo(8) (http://linux.die.net/man/8/sudo).

ssh client environment variables

Of course the foundation of most SSH activity is based upon the shell. Upon a successful connection, OpenSSH sets several environment variables.

```
SSH_CLIENT='192.168.223.17 36673 22'
SSH_CONNECTION='192.168.223.17 36673 192.168.223.229 22'
SSH_TTY=/dev/pts/6
```

SSH_CLIENT shows the address of the client system, the outgoing port number on the client system and the incoming port on the server. **SSH_CONNECTION** shows the address of the client, the outgoing port on the client, the address of the server and the incoming port on the server. **SSH_TTY** names the pseudo-terminal device, abbreviated pty, on the server used by the connection. For more information on pseudo-terminals see ptm(4) (http://man.openbsd.org/ptm.4), tty(1) (http://man.openbsd.org/tty.1) and tty(4) (http://man.openbsd.org/tty.4).

The login session can be constrained to a single program with a predetermined set of parameters using **ForceCommand** in the server configuration or **Command=** in the authorized keys file. In that case an additional environment variable gets set.

```
SSH_ORIGINAL_COMMAND=echo "hello, world"
```

Other variables are set depending on the user's shell settings and the system's own settings.

SSH client configuration options

Configuration options can be passed to **ssh** as arguments, see ssh(1) (http://man.openbsd.org/ssh.1) for the full list.

Connect very verbose output, GSSAPI authention:

```
$ ssh -vv -K -l account host.example.org
```

A subset of options can be defined on the server host in the user's own authorized keys file, in conjunction with specific keys. See sshd(8) (http://man.openbsd.org/sshd.8) for which subset exactly.

```
command="/usr/local/sbin/backup.sh",no-pty ssh-rsa AAAAB3NzaC1yc2EAAAQEAsY6u71N...
command="/usr/games/wump",no-port-forwarding,no-pty ssh-dss AAAAB3NzaC1kc3MAeELb...
environment="gtm_dist=/usr/local/gtm/utf8",environment="gtm_principal_editing=NOINSERT:EDITING" ssh-rsa
```

Note that some directives, like setting the environment variables, are disabled by default and must be changed in the server configuration before available to the client. More configuration directives can be set by the user in **~/.ssh/config** or by the system administrator in **/etc/ssh/ssh_config**. These same configuration directives can be passed as arguments using **-o**. See ssh_config(5) (http://man.openbsd.org/ssh_config.5) for the full list with descriptions.

```
$ ssh -o "ServerAliveInterval=60" -o "Compression=yes" -l fred server.example.org
```

The system administrators of the client host can set some defaults in **/etc/ssh/config**. Some of these global settings can be targeted per specific group or per user.

For example, if a particular SSH server is available via port 2022, it may be convenient to have the client use that port automatically. Some of OpenBSD's anoncvs servers accept SSH connections on this port. However, compression should not be used in this case because CVS already uses compression. So that should be turned off. So, one could specify something like the following in the **$HOME/.ssh/config** configuration file so that the default port is 2022 and the connection is made without compression:

```
Host server.example.org
        Compression no
        Port 2022
```

See ssh_config(5) (http://man.openbsd.org/ssh_config.5) for the client side and sshd_config(5) (http://man.openbsd.org/sshd_config.5) for the server side for the full lists with descriptions.

The SFTP client

sftp is an interactive file transfer program which performs all operations over an encrypted SSH transport. It may also use many features of ssh(1) (http://man.openbsd.org/ssh.1), such as public key authentication and compression. It is also the name of the protocol used.

The SFTP protocol is similar in some ways to the now venerable File Transfer Protocol (FTP), except that

the entire session, including the login, is encrypted. However, SFTP is not FTPS. The latter old-fashioned FTP tunneled over SSH/SSL. In contrast, SFTP is actually a whole new protocol. **sftp** can also be made to start in a specific directory on the remote host.

```
$ sftp fred@server.example.org:/var/www
```

Frequently, SFTP is used to connect and log into a specified host and enter an interactive command mode. See the manual page for sftp(1) (http://man.openbsd.org/sftp.1) for the available interactive commands. Also, the same configuration options that work for **ssh** also apply to **sftp**. **sftp** accepts all **ssh_config** options and these can be passed along as arguments at run time. Some have explicit shortcuts, others can be specified by naming them in full using the **-o** option.

```
$ sftp -o "ServerAliveInterval=60" -o "Compression=yes" fred@server.example.org
```

Another way to transfer is to write or retrieve files automatically. If a non-interactive authentication method is used, the whole process can be automatic using batch mode.

```
$ sftp -b session.batch -i ~/.ssh/some_key_rsa fred@server.example.org
```

Batch processing only works with non-interactive authentication.

The SCP client

scp is used for encrypted transfers of files between hosts and is used a lot like regular **cp**. It is based on and a replacement for **rcp** from the original Berkeley Software Distribution (BSD), but uses ssh to encrypt the connection.

The **scp** client, unlike the SFTP client, is not based on any formal standard. It has aimed at doing more or less what old **rcp** does and responding the same way. Since the same program must be used at both ends of the connection and interoperability is required with other implementations of **ssh**. Changes in functionality would probably break that interoperability, so new features are more likely to be added to **sftp** if at all. Thus, it is best to lean towards using **sftp** instead when possible.

Copy from remote to local:

```
$ scp myaccount@sftp.example.org:*.txt .
```

Copy from local to remote, recursively:

```
$ scp -r /etc myaccount@sftp.example.org:.
```

See also the **sftp** client above.

GUI Clients

There are a great many graphical utilities that support SFTP and SSH. Many started out as transfer utilities with the outdated legacy protocol FTP and grew with the times to include SSH and SFTP support. Sadly, many retain the epithet FTP program despite modernization. Others are more general file managers that include SFTP support as one means of network transparency. Most if not all provide full SFTP support

including kerberos authentication.

Below is a partial list to give an idea of the range of options available.

Bluefish is a website managment tool and web page editor with built in support for SFTP. Closed source competitors XMetaL and Dreamweaver are said to have at least partial support for SFTP. Nothing for Quanta+ or Kompozer as of this writing. http://bluefish.openoffice.nl/

Cyberduck is a remote file browser for the Macintosh. It supports an impressive range of protocols in addition to SFTP. http://cyberduck.ch/

Dolphin is a highly functional file manager for the KDE desktop, but can also be run in other environments. It includes SFTP support

Fetch, by Fetch Softworks, is a reliable and well-known SFTP client for the Macintosh. It's been around since 1989, starting life as just an FTP client, and has many useful features combined with ease of use. It is closed source, but academic institutions are eligible for a free of charge site license. http://fetchsoftworks.com/fetch/

Filezilla is presented as a FTP utility, but it has built in support for SFTP. It is available for multiple platforms under the GPL. http://filezilla-project.org/

FireFTP is a SFTP plugin for Mozilla Firefox. Though it is presented as an FTP add-on, it supports SFTP. It is available under both the MIT license and the GPL. http://fireftp.mozdev.org/

Fugu, developed by the University of Michigan research systems unix group, is a graphical front-end for the Macintosh. http://rsug.itd.umich.edu/software/fugu/

gFTP is a multi-threaded file transfer client http://www.gftp.org/

JuiceSSH is an SSH Client for Android/Linux. It uses the jsch (http://www.jcraft.com/jsch/) Java implementation of SSH2. https://juicessh.com/

Konqueror is a file manager and universal document viewer for the KDE desktop, but can also be run in other environments. It includes SFTP support. http://www.konqueror.org/

lftp is a file transfer program that supports multiple protocols. http://lftp.yar.ru/

Nautilus is the default file manager for the GNOME desktop, but can also be run in other environments. It includes SFTP support

PCManFM is an extremely fast, lightweight, yet feature-rich file manager with tabbed browsing which is the default for LXDE. It includes SFTP support. http://wiki.lxde.org/en/PCManFM

PuTTY is another FOSS implementation of Telnet and SSH for legacy and Unix platforms, released under the MIT license. It includes an SFTP client, PSFTP, in addition to an xterm terminal emulator and other tools like a key agent, Paegent. It is written and maintained primarily by Simon Tatham. http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

SecPanel is a GUI for managing and running SSH and scp connections. It is not a new implementation of the protocol or software-suite, but sits on top of either of the SSH software-suites http://themediahost.de /secpanel/

Thunar is the default file manager for the XFCE desktop. It includes SFTP support. http://docs.xfce.org /xfce/thunar/start

Yafc is Yet Another FTP Client and despite the name supports SFTP. http://yafc.sourceforge.net/

Client Configuration Files

Client configuration files can be per user or system wide, with the former taking precedence over the latter and run-time arguments in the shell overriding both. In these configuration files, one parameter per line is allowed with the parameter name followed by its value or values. Empty lines and lines starting with the hash (#) are ignored. An equal sign (=) can be used instead of whitespace between the parameter name and the values.

Values are case-sensitive, but parameter names are not.

System-wide Client Configuration Files

System-wide client files set the default configuration for all users of OpenSSH clients on that system. These defaults can be overridden, in many cases, by the user's own default settings in a local configuration file. Both can be overridden, in many cases, by specifying various options or parameters at run time. The prioritization is as follows:

1. run time arguments via the shell
2. user's own configuration
3. system-wide configuration

The first value obtained is used.

/etc/ssh/ssh_config

This file defines all the default settings for the client utilities for all users on that system. it must be readable by all users. The configuration options are described in detail in ssh_config(5) (http://man.openbsd.org/ssh_config.5).

Below a shortcut is made for connecting to *arc.example.org*. It is enough to enter `ssh arc` and the rest of the information gets filled in automatically.

```
Host arc
        Port 2022
        HostName arc.example.org
        User fred
        IdentityFile ~/.ssh/id_rsa_arc
```

/etc/ssh/ssh_known_hosts

This contains the system-wide list of known host keys used to verify the identity of the remote host and thus hinder impersonation or eavesdropping. This file should be prepared by the system administrator to contain the public host keys of all necessary hosts. It should be world-readable.

See **~/.ssh/known_hosts** below for more explanation or see sshd(8) (http://man.openbsd.org/sshd.8) for further details of the format of this file.

/etc/ssh/sshrc

This file resides on the server and programs in this file are executed there by ssh(1) (http://man.openbsd.org/ssh.1) when the user logs in, just before the user's shell or designated program is started. It is not run as root, but instead as the user who is logging in. See the sshd(8) (http://man.openbsd.org/sshd.8) manual page in the section SSHRC for more information.

User-specific Client Configuration Files

Users can override the default system-wide client settings and choose their own defaults. For situations where the same change is made repeatedly it can make save work to add them to the user's local configuration.

Client Side Files

These files reside on the client machine.

~/.ssh/config

The user's own configuration file which, where applicable, overrides the settings in the global client configuration file, **/etc/ssh/ssh_config**. The configuration options are described in detail in ssh_config(5) (http://man.openbsd.org/ssh_config.5).

This file must *not* be accessible to other users in any way. Set strict permissions: read/write for the user, and not accessible by others. It may group-writable if and only if that user is the only member of the group in question.

Local override of defaults

Not all options can be set or overriden by the user. Those options which may be set or overridden by the user can by defined at the end of the user's configuration, **~/.ssh/config**, file by matching all hosts. Those options which may not be set or overridden will be ignored.

```
Host *
        ExitOnForwardFailure    yes
        Protocol        2
        ServerAliveInterval     400
```

~/.ssh/known_hosts

As with **/etc/ssh/ssh_known_hosts**, this file contains the known keys for hosts that have been connected to in the past and is used to verify the identity of the remote host and protect against impersonation or man-in-the-middle attacks. With each subsequent connection the key will be compared to the key provided by the remote server. If there is a match, the connection will proceed. If the match fails, ssh(1) (http://man.openbsd.org/ssh.1) will fail with an error message. If there is no key at all listed for that remote host, then the key's fingerprint will be displayed and there will be the option to automatically add the key to the file. This file can be created and edited manually, but if it does not exist it will be created automatically by ssh(1) (http://man.openbsd.org/ssh.1) when it first connects to a remote host.

Server Side Files

These files reside on the server, by default in the user's directory. However, the server can be configured to

look for them in other locations, if needed.

~/.ssh/authorized_keys

authorized_keys is a one-key-per-line register of public ECDSA, RSA and ED25519 keys that this account can use to log in with. The file's contents are not highly sensitive, but the recommended permissions are read/write for the user and not accessible by others. The whole key including options and comments must be on a single, unbroken line.

```
ssh-rsa AAAAB3NzaC1yc2EAAA...41Ev521Ei2hvz7S2QNr1zAiVaOFy5Lwc8Lo+Jk=
```

Lines starting with a hash (#) are ignored. Whitespace is used to separate the key's fields. They are, in sequence, an optional list of login options, the key type (usually ssh-dss or ssh-rsa), the key itself encoded as base64, and an optional comment.

If a key is followed by an annotation, the comment does not need to be wrapped in quotes. It has no effect on what the key does or how it works.

```
# an annotated key
ssh-rsa AAAAB3NzaC1yc2EAAA...zAiVaOFy5Lwc8Lo+Jk=  Fred @ Project FOOBAR
```

Keys can be preceded by a comma-separated list of options to affect what happens upon successful login. See the next section, Available key login options, for details.

```
# launch tinyfugue automatically
command="/usr/bin/tinyfugue" ssh-rsa AAAAB3NzaC1yc2EAAA...OFy5Lwc8Lo+Jk=

# set the PATH
environment="PATH=/bin:/usr/bin/:/opt/gtm/bin" ssh-rsa AAAAB3N...4Y2t1j=
```

The format of **authorized_keys** is described in the sshd(8) (http://man.openbsd.org/sshd.8) manual page. Old keys should be deleted from the file when no longer needed. The server can specify multiple locations for **authorized_keys**.

~/.ssh/authorized_principals

By default this file does not exist. If it is specified in sshd_config(5) (http://man.openbsd.org/sshd_config.5), it contains a list of names which can be used in place of the username when authorizing a certificate. This option is useful for role accounts, disjoint account namespaces and "user@realm"-style naming policies in certificates. Principals can also be specified in **authorized_keys**.

~/.ssh/environment

If the server is configured to accept user-supplied, automatic changes to environment variables as part of the login process, then these changes can be set in this file.

If the server, the environment file and an authorization key all try to change the same variable, the file environment takes precedence over what a key might contain. Either one will override any environment variables that might have been passed by ssh(1) (http://man.openbsd.org/ssh.1) using **SendEnv**. See also the **AcceptEnv** and **PermitUserEnvironment** directives in the manual page for sshd_config(5) (http://man.openbsd.org/sshd_config.5).

Authentication keys stored in **authorized_keys** can also be used to set variables.

~/.ssh/rc

This is a script which is executed by sh(1) (http://man.openbsd.org/sh.1) just before the user's shell or command is started. It is not run if **ForceCommand** is used. The script is run after reading the environment variables. The corresponding global file, **/etc/ssh/sshrc**, is not run if the user's **rc** script exists.

Legacy Files

~/.shosts

~/.rhosts

.rhosts is a legacy from rsh containing a local list of trusted host-user pairs that are allowed to log in. Login requests matching an entry are granted access.

See also the global list of trusted host-user pairs, **/etc/hosts.equiv**

rhosts can be used as part of host-based authentication. Otherwise it is recommended not to use rhosts for authentication, there are a lot of ways to misconfigure the rhosts file.

Available key login options

The login options available for use in the local user authorized keys file might be overridden or blocked by the server's own settings.

cert-authority

Specifies that the listed key is a certification authority (CA) trusted to validate signed certificates for user authentication. Certificates may encode access restrictions similar to key options. If both certificate restrictions and key restrictions are present, then the most restrictive union of the two is applied.

command="*program*"

Specifies a program, with options, that is executed when this key is used for authentication.

The program is run on a PTY if the client requests it, otherwise the default is to run without a TTY. The default, running without a TTY, provides an 8-bit clean channel. If the default has been changed, specify **no-pty** to get an 8-bit clean channel.

```
no-pty,command="" ssh-rsa AAAAB3NzaC1yc2EAAA...OFy5Lwc8Lo+Jk=
```

Quotes provided in the program's options must be escaped using a backslash. ('\')

```
command="sh -c \"mysqldump db1 -u fred1 -p\"" ssh-rsa AAAAB3NzaC1yc...Lwc8OFy5Lo+kU=
```

This option applies to execution of the shell, another program or subsystem. Thus any other program specified by the user is ignored, but the program originally specified by the client remains available as the environment variable **SSH_ORIGINAL_COMMAND**.

This way of forcing a program is useful to restrict a key to a single, specific operation, such as a remote backup. However, TCP and X11 forwarding are still allowed unless explicitly disabled elsewhere.

environment="*NAME=value*"

Sets the value of an environment variable when this key is used to log in. It overrides default values of the variable, if it exists. This option can be repeated to set multiple variables. This option is only allowed if the **PermitUserEnvironment** option is set. The default is disabled, and this option becomes disabled automatically when **UseLogin** is enabled.

from="*pattern-list*"

The canonical name of the remote host or its IP address required in addition to the key. Addresses and hostnames can be listed using a comma-separated list of patterns, see PATTERNS in ssh_config(5) (http://man.openbsd.org/ssh_config.5) for more information on patterns, or the CIDR address/masklen notation.

no-agent-forwarding

Forbids authentication agent forwarding when this key is used for authentication.

no-port-forwarding

Forbids TCP forwarding and any port forward requests by the client will return an error when this key is used for authentication.

no-pty

TTY allocation is prohibited and any request to allocate a PTY will fail.

no-user-rc

Use the **no-user-rc** option in **authorized_keys** to disable execution of **~/.ssh/rc**

no-X11-forwarding

Prevents X11 forwarding when this key is used for authentication, and requests to forward X11 will return an error.

permitopen="*host:port*"

Limits local port forwarding (`ssh -L`) to only the specified host and port. IPv6 addresses can be specified with an alternative syntax: host/port. Multiple permitopen options may be used and must be separated by commas. No pattern matching is performed on the specified host names, they must be literal host names or IP addresses. Used in conjunction with agent-forwarding.

principals="*name1*[,*name2*,...]"

Specify a list of names that may be used in place of the username when authorizing a certificate trusted via the sshd_config(5) (http://man.openbsd.org/sshd_config.5) **TrustedCAKeys** option.

tunnel="*n*"

Select a specific tun(4) (http://man.openbsd.org/tun.4) device on the server. Otherwise when a tunnel device

is requested without this option the next available device will be used.

User-specific Keys

ssh(1) (http://man.openbsd.org/ssh.1) uses Ed25519, ECDSA, RSA, or DSA key pairs to verify hosts or authenticate users. Individual user accounts can maintain their own list of keys or certificates for authentication or to verify the identity of remote hosts.

Public Keys from other Hosts – ~/.ssh/known_hosts

known_hosts is for verifying the identity of other systems. It contains a list of public keys for all the hosts which the user has logged into. It can also include public keys for hosts that the user plans to log into but are not already in the system-wide list of known host keys. ssh(1) (http://man.openbsd.org/ssh.1) automatically adds keys to the user file, but they can be added manually as well. Usually when connecting to a host for the first time, ssh(1) (http://man.openbsd.org/ssh.1) adds the remote host's public key to the user's **known_hosts** file.

The format is one public key or certificate per unbroken line. Each line in contains a hostname, number of bits, exponent, and modulus. At the beginning of the line is either the host name or a hash representing the host name. An optional comment can follow at the end of the line. These can be preceded by an optional marker to indicate a certificate authority, if a certificate is used instead of a SSH key. These fields are separated by spaces. It is possible to use a comma-separated list of the hostname.

```
# keys with basic hostnames
anoncvs.fr.openbsd.org,93.184.34.123 ssh-rsa AAAA...njvPw==
anoncvs.eu.openbsd.org ssh-rsa AAAAB3Nz...cTqGvaDhgtAhw==
```

Non-standard ports can be indicated by enclosing the hostname with square brackets and followed by a colon and the port number.

```
# key with non-standard port
[ssh.example.org]:2222 ssh-rsa AAAAB3Nz...AKy2R2OE=
[127.0.0.2]:4922 ssh-rsa AAAAB4mV...1d6j=
[anga.funkfeuer.at]:2022,[78.41.115.130]:2022 ssh-rsa AAAAB...fgTHaojQ==
```

Hostname patterns can be created using "*****" and "**?**" as wildcards and "**!**" to indicate negation.

Up to one optional marker per line is allowed. If present it must be either **@cert-authority** or **@revoked**. The former shows that the key is a certificate authority key, the latter flags the key as revoked and not acceptable for use.

See sshd(8) (http://man.openbsd.org/sshd.8) for further details of the format of this file and ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) for managing the keys.

Manually Adding Public Keys to ~/.ssh/known_hosts

Manually adding public host keys to known_hosts is a matter of adding one unbroken line per key. How the key is obtained is not important, as long as it is complete, valid and **guaranteed to be the real key and not a fake**. ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1) can fetch a key and ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) can be used to show the fingerprint for verification. See examples in the cookbook chapter for methods of verification. The corresponding system-wide file is **/etc/ssh /ssh_known_hosts**

Local User's Public / Private Key Pairs

Users might have a variety of their own ECDSA, Ed25519, RSA and DSA keys around for various tasks and projects. The naming convention for keys is only a convention, but recommended to follow anyway. Public keys usually have the same name as the private key, but with **.pub** appended to the name.

Public Keys – ~/.ssh/id_dsa.pub ~/.ssh/id_ecdsa.pub ~/.ssh/id_ed25519.pub ~/.ssh/id_rsa.pub

Public keys are used for authentication. Public keys are not sensitive and are allowed to be readable by anyone, unlike the private keys, but don't need to be.

Private Keys – ~/.ssh/id_dsa ~/.ssh/id_ecdsa ~/.ssh/id_ed25519 ~/.ssh/id_rsa

These private keys are sensitive data and should be readable by the user but not accessible by others, mode 0600. The directory they are in should also have mode 0700 or 0500. If a private key file is accessible by others, ssh(1) (http://man.openbsd.org/ssh.1) will ignore it.

It is possible to specify a passphrase when generating the key which will be used to encrypt the sensitive part of this file using AES128. Until version 5.3, the cipher 3DES was used to encrypt the passphrase. Old keys using 3DES that are given new passphrases will use AES128 when they are modified.

Legacy Protocol Keys ~/.ssh/identity ~/.ssh/identity.pub

identity and identity.pub are for ssh protocol version 1, and thus deprecated.

User-specific Public Key Authentication

Users can authenticate using a private key stored on their system or fetched from a smartcard if the corresponding public key is stored on the system being connected to in **authorized_keys**. It is not highly sensitive, but the recommended permissions are read/write for the user, and not accessible by others.

The keys can be preceded by a comma-separated list of options. The whole key must be on a single, unbroken line. No spaces are permitted, except within double quotes. Any text after the key is considered a comment. See the section above on the **authorized_keys** file for more discussion. The **authorized_keys** file is a one-key-per line register of public RSA and DSA keys that can be used to log in as this user.

User-specified Host-based Authentication Configuration is also possible using the **~/.shosts**, **~/.rhosts**, **~/.ssh/environment**, and **~/.ssh/rc** files.

Managing Keys

When working with keys there are some basic, common sense actions that should take place to prevent problems.

Keys should use strong pass phrases. If autonomous logins are required, then the keys should be first loaded into an agent and used from there. See ssh-add(1) (http://man.openbsd.org/ssh-add.1) to get started there. It uses ssh-agent(1) (http://man.openbsd.org/ssh-agent.1) which many systems have installed and some have running by default.

Keys should always be stored in protected locations, even on the client side. This is especially important for

private keys. The private keys should not have read permissions for any user or group other than the owner. They should also be kept in a directory that is not accessible by anyone other than the owner in order to limit exposure.

Old and unused keys should be removed from the server. In particular, keys without a known, valid purpose should be removed and not allowed to accumulate. Using the comment field in the public key for annotation can help eliminate some of the confusion as to the purpose and owner when time has passed. Along those lines, keys should be rotated at intervals. In practice, rotation means generating new key pairs and removing the old ones. This gives a chance to remove old and unused keys. It is also an opportunity to review access needs, whether access is required and if so at what level.

Following the principle of least privilege can limit the chance for accidents or abuse. If a key is only needed to run a specific application or script, then its login options should be limited to just what is needed. See sshd(8) (http://man.openbsd.org/sshd.8) for the 'AUTHORIZED_KEYS FILE FORMAT' section on key login options. For root level access, it is important to remember to configure **/etc/sudoers** appropriately. Access there can be granted to a specific application and even limit that application to specific options.

Pattern Matching in OpenSSH Configuration

A pattern consists of zero or more non-whitespace characters. An asterisk (*****) matches zero or more characters in a row, and a question mark (**?**) matches exactly one character. For example, to specify a set of declarations that apply to any host in the ".co.uk" set of domains in **ssh_config**, the following pattern could be used:

```
Host *.co.uk
```

The following pattern would match any host in the 192.168.0.1 - 192.168.0.9 range:

```
Host 192.168.0.?
```

A pattern-list is a comma-separated list of patterns. The following list of patterns match hosts in the ".co.uk" or ".ac.uk" domains.

```
Host *.co.uk, *.ac.uk
```

Individual patterns by themselves or as part of a pattern-lists may be negated by preceding them with an exclamation mark (**!**). The following will match any host from *example.org* except for *gamma*.

```
Host *.example.org !gamma.example.org
```

For example, to allow a key to be used from anywhere within an organisation except from the dialup pool, the following entry in **authorized_keys** could be used:

```
from="!*.dialup.example.com,*.example.com"
```

See also glob(7) (http://man.openbsd.org/glob.7)

Utilities

ssh-agent(1) (http://man.openbsd.org/ssh-agent.1) - An authentication agent that can store private keys.
ssh-add(1) (http://man.openbsd.org/ssh-add.1) - Tool which adds or removes keys in the above agent.
ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) - Key generation tool.
ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1) - Utility for gathering public host keys from a number of hosts.
ssh-vulnkey(1) - Check key against blacklist of compromised keys
ssh-copy-id(1) (http://linux.die.net/man/1/ssh-copy-id) - Install a public key in a remote machine's authorized_keys

ssh-agent

ssh-agent is a tool to hold private keys in memory for re-use during a session. Usually it is started at the beginning of a session and subsequent windows or programs run as clients to the agent.

ssh-add

ssh-add is a tool to load key identities into an agent for re-use. It can also be used to remove identities from the agent. The agent holds the private keys used for authentication.

ssh-keyscan

ssh-keyscan has been part of the OpenSSH suite since OpenSSH version 2.5.1 and is used to retrieve public keys. Keys retrieved using **ssh-keyscan**, or any other method, must be verified by checking the key fingerprint to ensure the authenticity of the key and reduce the possibility of a man-in-the-middle attack. The default is to request a ECDSA key using SSH protocol 2. David Mazieres wrote the initial version of **ssh-keyscan** and Wayne Davison added support for SSH protocol version 2.

ssh-keygen

ssh-keygen is used to generate key pairs for use in authentication, update and manage keys or to verify key fingerprints. It can generate either ECDSA, Ed25519, RSA or DSA keys and can do the following:

- generate new key pairs, either ECDSA, Ed25519, RSA or DSA
- remove keys from known hosts
- regenerate a public key from a private key
- change the passphrase of a private key
- change the comment text of a private key
- show the fingerprint of a specific public key
- show ASCII art fingerprint of a specific public key
- load or read a key to or from a smartcard, if the reader is available

If the legacy protocol, SSH1, is used, then **ssh-keygen** can only generate RSA keys.

One important use for key fingerprints is when connecting to a machine for the first time. A fingerprint is a

hash or digest of the public key. Fingerprints can be sent ahead of time out of band via post, fax, SMS or other means not related to the Internet.

The verification data for the key should be sent out of band, not done via the Internet. It can be sent ahead of time by post, fax, SMS or a phone call instead or some other way you can be sure it is an authentic and unchanged key.

```
$ ssh -l fred zaxxon.example.org
The authenticity of host 'zaxxon.example.org (203.0.113.114)' can't be established.
RSA key fingerprint is SHA256:DnCHntWa4jeadiUWLUPGg9FDTAopFPR0c5TgjU/iXfw.
Are you sure you want to continue connecting (yes/no)?
```

If you see that and the key matches what your were given in advance, the connection is probably good. If you see that and the key is different than what you were given in advance, then stop and disconnect and get on the phone or voip to work out the mistake. Once the SSH client has accepted the key from the server, it is saved in **known_hosts**.

```
$ ssh -l fred galaga.example.org
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:QIWi4La8svQSf5ZYow8wBHN4tF0jtRlkIaLCUQRlxRI.
Please contact your system administrator.
Add correct host key in /home/fred/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/fred/.ssh/known_hosts:1
ECDSA host key for galaga.example.org has changed and you have requested strict checking.
Host key verification failed.
```

If you start to connect to a known host and you get an error like the one above, then either the first connection was to an impostor or the current connection is to an impostor, or something very foolish was done to the machine. Regardless, disconnect and don't try to log in. Contact the system administrator out of band to find out what is going on.[1] It is possible that the server was reinstalled, either the whole OS or just the OpenSSH server, without saving the old keys resulting in new keys being generated. Either way, check with the system administrator before connecting to be sure.

It is possible to use more than one key per account or the same key on more than one machine. Single-use keys allow some remote data transfer tasks to be automated fairly securely.

The public keys of the hosts you have verified the connections to are stored in **known_hosts** whereas the public keys of the users you accept logins from are kept in **authorized_keys**. There are also global files kept on each machine though each account has its own local ones.

When generating keys use good, solid passphrases. Good passphrases are 10 to 30 characters in length and are not simple sentences or otherwise easily guessable words or phrases from any language. It should contain a mixture of upper and lowercase letters, numbers, and non-alphanumeric characters. Passphrases can be changed later, but there is no way to recover a lost passphrase. If the passphrase is lost or forgotten, the key is useless and must be replaced with a new key pair. Hashed host names and addresses can be looked up in **known_hosts** using **-F** or **-R** can be used to delete them.

```
$ ssh-keygen -F sftp.example.org -f ~/.ssh/known_hosts
# Host sftp.example.org found: line 7 type RSA
|1|slYCk3msDPyGQ8l0lq82IbUTzBU=|KN7HPqVnJHOFX5LFmTXS6skjK4o= ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA3cqqA6f
```

ssh-vulnkey

ssh-vulnkey is included in some distros to check a key against a blacklist of compromised keys. The blacklist was made necessary when a broken version of OpenSSL was distributed by some distros[2], resulting in bad keys that were easily predicted and compromised. Keys made while that broken version was in use that are found to have been compromised cannot be repaired and must be replaced. The problem has since been fixed and new keys should be all right.

ssh-copy-id

ssh-copy-id is included in some distros to install a public key into a remote machine's **authorized_keys** file. It is a simple shell script and the **authorized_keys** file should still be checked manually after first login to verify that everything went ok and that the key was copied as it should be.

References

1. Brian Hatch (2004). "SSH Host Key Protection". SecurityFocus. http://www.securityfocus.com /infocus/1806. Retrieved 2013-04-14.
2. Jake Edge (2008). "Debian vulnerability has widespread effects". LWN. https://lwn.net/Articles /282230/. Retrieved 2013-04-14.

Logging and Troubleshooting

Both the OpenSSH client and server offer a lot of choice as to where the logs are written and how much information is collected.

A prerequisite for logging is having an accurate system clock using the Network Time Protocol, NTP. Having the NTP server running provides time synchronization with the world. The more accurate the time stamp in the log is, the faster it is to coordinate forensics between machines or sites or service providers. If you have to contact outside parties like a service provider, progress can usually only be made with very exact times.

Server Logs

By default sshd(8) (http://man.openbsd.org/sshd.8) logs to the system logs, with log level INFO and syslog facility AUTH. So the place to look for log data from sshd(8) (http://man.openbsd.org/sshd.8) is in **/var/log /auth.log**. These defaults can be overridden using the **SyslogFacility** and **LogLevel** directives. Below is a typical server startup entry in the authorization log.

```
Mar 19 14:45:40 eee sshd[21157]: Server listening on 0.0.0.0 port 22.
Mar 19 14:45:40 eee sshd[21157]: Server listening on :: port 22.
```

In most cases the default level of logging is fine, but during initial testing of new services or activities it is sometimes necessary to have more information. Debugging info usually goes to **stderr**. The log excerpt below show the same basic server start up with increasing detail. Contrast debug level 1 below with the default above:

```
debug1: sshd version OpenSSH_6.8, LibreSSL 2.1
debug1: private host key #0: ssh-rsa SHA256:X9e6YzNXMmr1O09LVoQLlCau2ej6TBUxi+Y590KVsds
debug1: private host key #1: ssh-dss SHA256:XcPAY4soIxU2IMtYmnErrVOjKEEvCc3l5hOctkbqeJ0
debug1: private host key #2: ecdsa-sha2-nistp256 SHA256:QIWi4La8svQSf5ZYow8wBHN4tF0jtRlkIaLCUQRlxRI
debug1: private host key #3: ssh-ed25519 SHA256:fRWrx5HwM7E5MRcMFTdH95KwaExLzAZqWlwULyIqkVM
debug1: rexec_argv[0]='/usr/sbin/sshd'
debug1: rexec_argv[1]='-d'
debug1: Bind to port 22 on 0.0.0.0.
Server listening on 0.0.0.0 port 22.
debug1: Bind to port 22 on ::.
Server listening on :: port 22.
```

And same at the most verbose level, DEBUG3:

```
debug2: load_server_config: filename /etc/ssh/sshd_config
debug2: load_server_config: done config len = 217
debug2: parse_server_config: config /etc/ssh/sshd_config len 217
debug3: /etc/ssh/sshd_config:52 setting AuthorizedKeysFile .ssh/authorized_keys
debug3: /etc/ssh/sshd_config:86 setting UsePrivilegeSeparation sandbox
debug3: /etc/ssh/sshd_config:104 setting Subsystem sftp internal-sftp
debug1: sshd version OpenSSH_6.8, LibreSSL 2.1
debug1: private host key #0: ssh-rsa SHA256:X9e6YzNXMmr1O09LVoQLlCau2ej6TBUxi+Y590KVsds
debug1: private host key #1: ssh-dss SHA256:XcPAY4soIxU2IMtYmnErrVOjKEEvCc3l5hOctkbqeJ0
debug1: private host key #2: ecdsa-sha2-nistp256 SHA256:QIWi4La8svQSf5ZYow8wBHN4tF0jtRlkIaLCUQRlxRI
debug1: private host key #3: ssh-ed25519 SHA256:fRWrx5HwM7E5MRcMFTdH95KwaExLzAZqWlwULyIqkVM
debug1: rexec_argv[0]='/usr/sbin/sshd'
debug1: rexec_argv[1]='-ddd'
debug2: fd 3 setting O_NONBLOCK
debug1: Bind to port 22 on 0.0.0.0.
Server listening on 0.0.0.0 port 22.
debug2: fd 4 setting O_NONBLOCK
debug1: Bind to port 22 on ::.
```

Note that failed login attempts are not logged until half the value in directive **MaxAuthTries** is exceeded. Below is how the default log looks after some failed attempts:

```
...
Nov 23 20:31:12 server sshd[15798]: Connection from 188.124.3.41 port 32889
Nov 23 20:31:14 server sshd[15798]: Failed password for root from 188.124.3.41 port 32889 ssh2
Nov 23 20:31:14 server sshd[29323]: Received disconnect from 188.124.3.41: 11: Bye Bye
Nov 23 22:04:56 server sshd[25438]: Connection from 200.54.84.233 port 45196
Nov 23 22:04:58 server sshd[25438]: Failed password for root from 200.54.84.233 port 45196 ssh2
Nov 23 22:04:58 server sshd[30487]: Received disconnect from 200.54.84.233: 11: Bye Bye
Nov 23 22:04:59 server sshd[21358]: Connection from 200.54.84.233 port 45528
Nov 23 22:05:01 server sshd[21358]: Failed password for root from 200.54.84.233 port 45528 ssh2
Nov 23 22:05:01 server sshd[2624]: Received disconnect from 200.54.84.233: 11: Bye Bye
...
```

It is usually a good idea not to allow root login. That simplifies log analysis greatly. It in particular eliminates the time consuming question of who is trying to get in and why. Tasks that need root level access can be given it through custom-made entries in **/etc/sudoers**. People that need root level access can gain it through **sudo** or **su**.

Successful logins

By default, the server does not store much information about user transactions. That is a good thing. It is also a good thing to recognize when the system is operating as it should. Here is a successful SSH login:

```
Mar 14 19:50:59 server sshd[18884]: Accepted password for fred from 192.0.2.60 port 6647 ssh2
```

And one using a key for authentication, which shows the SHA256 hash in base64.

```
Mar 14 19:52:04 server sshd[5197]: Accepted publickey for fred from 192.0.2.60 port 59915 ssh2: RSA SHA2
```

Prior to 6.8, the fingerprint was a hexadecimal MD5 hash.

```
Jan 28 11:51:43 server sshd[5104]: Accepted publickey for fred from 192.0.2.60 port 60594 ssh2: RSA e8:3
```

In older versions of OpenSSH, prior to 6.3, the key fingerprint is missing.

```
Jan 28 11:52:05 server sshd[1003]: Accepted publickey for fred from 192.0.2.60 port 20042 ssh2
```

A password authentication for SFTP, using the internal-sftp subsystem and with logging for that subsystem set to INFO.

```
Mar 14 20:14:18 server sshd[19850]: Accepted password for fred from 192.0.2.60 port 59946 ssh2
Mar 14 20:14:18 server internal-sftp[11581]: session opened for local user fred from [192.0.2.60]
```

Here is a successful SFTP login using an RSA key for authentication.

```
Mar 14 20:20:53 server sshd[10091]: Accepted publickey for fred from 192.0.2.60 port 59941 ssh2: RSA SHA
Mar 14 20:20:53 server internal-sftp[31070]: session opened for local user fred from [192.0.2.60]
```

Additional data, such as connection duration, can be logged with the help of **xinetd**.

Logging SFTP File Transfers

SFTP file transfers can be logged using LogLevel INFO or VERBOSE. The LogLevel for the SFTP server can be set in *sshd_config* separately from the general SSH server.

```
Subsystem internal-sftp -l INFO
```

By default the SFTP messages will also end up in *auth.log* but it is possible to filter these messages to their own file by reconfiguring the system logger, usually **rsyslogd(8)** or **syslogd(8)**. Sometimes this is done by changing the log facility code from the default of AUTH. Available options are LOCAL0 through LOCAL7, plus, less usefully, DAEMON and USER.

```
Subsystem internal-sftp -l INFO -f LOCAL6
```

If new system log files are assigned, it is important to remember them in log rotation, too.

The following log excerpts are generated from using LogLevel INFO. A session will start with an open and end with a close. The number in the brackets is the process id for that SFTP session and will be the only way to follow a session in the logs.

```
Oct 22 11:59:45 server internal-sftp[4929]: session opened for local user fred from [192.0.2.33]
...
Oct 22 12:09:10 server internal-sftp[4929]: session closed for local user fred from [192.0.2.33]
```

Here is an SFTP upload of a small file of 928 bytes named *foo* to user *fred*'s home directory.

```
Oct 22 11:59:50 server internal-sftp[4929]: open "/home/fred/foo" flags WRITE,CREATE,TRUNCATE mode 0664
Oct 22 11:59:50 server internal-sftp[4929]: close "/home/fred/foo" bytes read 0 written 928
```

And a directory listing in the same session in the directory */var/www*.

```
Oct 22 12:07:59 server internal-sftp[4929]: opendir "/var/www"
Oct 22 12:07:59 server internal-sftp[4929]: closedir "/var/www"
```

And lastly a download of the same small 928-byte file called *foo* from the user *fred*'s home directory.

```
Oct 22 12:08:03 server internal-sftp[4929]: open "/home/fred/foo" flags READ mode 0666
Oct 22 12:08:03 server internal-sftp[4929]: close "/home/fred/foo" bytes read 928 written 0
```

Successful transfers will be noted by a *close* message. Attempts to download (open) files that do not exist will be followed by a *sent status No such file* message on a line of its own instead of a *close*. Files that exist but that the user is not allowed to read will create a *sent status Permission denied* message.

Logging Chrooted SFTP

Logging the built-in **sftp-subsystem** inside a chroot jail, defined by **ChrootDirectory**, needs a **./dev/log** node to exist inside the jail. This can be done by having the system logger such as **syslogd** add additional log sockets when starting up. On some systems that is as simple as adding more flags, like "**-u -a /chroot /dev/log**", in **/etc/rc.conf.local** or whatever the equivalent startup script may be.

Here is a SFTP login with password to a chroot jail, using log level *Debug3* for the SFTP-subsystem logging a file upload:

```
Jan 28 12:42:41 server sshd[26299]: Connection from 192.0.2.60 port 47366
Jan 28 12:42:42 server sshd[26299]: Failed none for fred from 192.0.2.60 port 47366 ssh2
Jan 28 12:42:44 server sshd[26299]: Accepted password for fred from 192.0.2.60 port 47366 ssh2
Jan 28 12:42:44 server sshd[26299]: User child is on pid 21613
Jan 28 12:42:44 server sshd[21613]: Changed root directory to "/home/fred"
Jan 28 12:42:44 server sshd[21613]: subsystem request for sftp
Jan 28 12:42:44 server internal-sftp[2084]: session opened for local user fred from [192.0.2.60]
Jan 28 12:42:58 server internal-sftp[2084]: open "/docs/somefile.txt" flags WRITE,CREATE,TRUNCATE mode 0
Jan 28 12:42:58 server internal-sftp[2084]: close “/docs/somefile.txt” bytes read 0 written 400
```

Remember that SFTP is a separate subsystem and that like the file creation mode, the log level and facility are set separately from the SSH server in sshd_config(5) (http://man.openbsd.org/sshd_config.5):

```
Subsystem internal-sftp -l ERROR
```

Logging Revoked Keys

If the **RevokedKeys** directive is used to point to a list of public keys that have been revoked, sshd(8) (http://man.openbsd.org/sshd.8) will make a log entry when access is attempted using a revoked key. The entry will be the same whether a plaintext list of public keys is used or if a binary Key Revocation List (KRL) has been generated.

If password authentication is allowed, and the user tries it, then after the key authentication fails there will be a record of password authentication.

```
Mar 14 20:36:40 server sshd[29235]: error: Authentication key RSA SHA256:jXEPmu4thnubqPUDcKDs31MOVLQJH6F
```

```
...
Mar 14 20:36:45 server sshd[29235]: Accepted password for fred from 192.0.2.10 port 59967 ssh2
```

If password authentication is not allowed, sshd(8) (http://man.openbsd.org/sshd.8) will close the connection as soon as the key fails.

```
Mar 14 20:38:27 server sshd[29163]: error: Authentication key RSA SHA256:jXEPmu4thnubqPUDcKDs31MOVLQJH6F
...
Mar 14 20:38:27 server sshd[29163]: Connection closed by 192.0.2.10 [preauth]
```

The same happens if the user cancels the connection without trying a password after the key attempt fails. In both cases, there will be no clue as to which user tried to log on, so it will be necessary to try to look up the key by its fingerprint from your archive of old keys using `ssh-keygen -lf` and read the key's comments.

On the client side, no warning or error will be given if a revoked key is tried. It will just fail and the next key or method will be tried.

Brute force and Hail Mary attacks

It's fairly common to see failed login attempts almost as soon as the server is connected to the net. Brute force attacks, where one machine hammers on a few accounts trying to find a valid password, are becoming rare. In part this is because packet filters, like IP Tables for Linux and PF for BSD, can limit the number and rate of connection attempts from a single host. The server configuration directive **MaxStartups** can limit the number of simultaneous, unauthenticated connections.

```
...
Nov 23 19:35:42 server sshd[14000]: Connection from 116.28.64.132 port 55680
Nov 23 19:35:44 server sshd[14000]: Failed password for root from 116.28.64.132 port 55680 ssh2
Nov 23 19:35:44 server sshd[11034]: Received disconnect from 116.28.64.132: 11: Bye Bye
Nov 23 19:35:45 server sshd[3096]: Connection from 116.28.64.132 port 55932
Nov 23 19:35:48 server sshd[3096]: Failed password for root from 116.28.64.132 port 55932 ssh2
Nov 23 19:35:48 server sshd[11289]: Received disconnect from 116.28.64.132: 11: Bye Bye
...
```

The way to deal with brute force attacks is to customize the server host's packet filter to limit the attacks or even temporarily block machines that overload the maximum number or rate of connections. Optionally, one should also contact the attacker's net block owner with the IP address and exact date and time of the attacks.

A kind of attack common at the time of this writing is one which is distributed over a large number of compromised machines, each playing only a small role in attacking the server.

To deal with Hail Mary attacks, contact the attacker's net block owner. A form letter with a cut-and-paste excerpt from the log is enough, if it gives the exact times and addresses. Alternately, teams of network or system administrators can work to pool data to identify and blacklist the compromised hosts participating in the attack.

Client Logging

The OpenSSH the client normally sends log information to stderr. The **-y** option can be used to send output to the system logs, managed by syslogd or something similar. Or the **-E** option sends log output to a designated file instead of stderr. Working with the system logs or separate log files are something which can be useful when running ssh in automated scripts. Below is a connection to an interactive shell with the normal level of client logging:

```
$ ssh -l fred server.example.org
fred@server.example.org's password:
Last login: Thu Jan 27 13:21:57 2011 from 192.168.11.1
```

The same connection at the first level of verbosity gives lots of debugging information, 42 lines more.

```
$ ssh -v -l fred server.example.org
OpenSSH_6.8, LibreSSL 2.1
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to server.example.org [198.51.100.20] port 22.
debug1: Connection established.
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_rsa type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_rsa-cert type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_dsa type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_dsa-cert type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_ecdsa type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_ecdsa-cert type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_ed25519 type -1
debug1: key_load_public: No such file or directory
debug1: identity file /home/fred/.ssh/id_ed25519-cert type -1
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_6.8
debug1: Remote protocol version 2.0, remote software version OpenSSH_6.7
debug1: match: OpenSSH_6.7 pat OpenSSH* compat 0x04000000
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: server->client aes128-ctr umac-64-etm@openssh.com none
debug1: kex: client->server aes128-ctr umac-64-etm@openssh.com none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: Server host key: ecdsa-sha2-nistp256 SHA256:CEXGTmrVgeY1qEiwFe2Yy3XqrWdjm98jKmX0LK5mlQg
debug1: Host '198.51.100.20' is known and matches the ECDSA host key.
debug1: Found key in /home/fred/.ssh/known_hosts:2
debug1: SSH2_MSG_NEWKEYS sent
debug1: expecting SSH2_MSG_NEWKEYS
debug1: SSH2_MSG_NEWKEYS received
debug1: Roaming not allowed by server
debug1: SSH2_MSG_SERVICE_REQUEST sent
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey,password,keyboard-interactive
debug1: Next authentication method: publickey
debug1: Trying private key: /home/fred/.ssh/id_rsa
debug1: Trying private key: /home/fred/.ssh/id_dsa
debug1: Trying private key: /home/fred/.ssh/id_ecdsa
debug1: Trying private key: /home/fred/.ssh/id_ed25519
debug1: Next authentication method: keyboard-interactive
debug1: Authentications that can continue: publickey,password,keyboard-interactive
debug1: Next authentication method: password
debug1: Authentication succeeded (password).
Authenticated to server.example.org ([198.51.100.20]:22).
debug1: channel 0: new [client-session]
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: client_input_global_request: rtype hostkeys-00@openssh.com want_reply 0
debug1: client_input_channel_req: channel 0 rtype exit-status reply 0
debug1: client_input_channel_req: channel 0 rtype eow@openssh.com reply 0
debug1: channel 0: free: client-session, nchannels 1
debug1: fd 2 clearing O_NONBLOCK
Last login: Sat Mar 14 21:31:33 2015 from 192.0.2.111

...
```

The same login with the maximum of verbosity, **-vvv**, gives around 150 lines of debugging information. Remember that debugging information is sent to stderr rather than stdout. Regular pipes and redirects work only with stdout so output on stderr must be sent to stdout first if one is going to capture it at the same time. That is done with a redirect, **2>&1**. Mind the spaces, or lack of them:

This will only capture the session in a file, debugging info goes only to the screen, not to the output log:

```
$ ssh -vvv -l fred  somehost.example.org  | tee ~/ssh-output.log
```

The tool tee(1) (http://man.openbsd.org/tee.1) is like a T-pipe and sends output two directions, one to stdout and one to a file.

This will capture both debugging info and session text:

```
$ ssh -vvv -l fred  somehost.example.org  2>&1  | tee ~/ssh-output.log
```

Also, the escape sequences **~v** and **~V** can be used to increase or decrease the verbosity of an existing connection.

```
$ sftp -v -o "IdentityFile=~/.ssh/weblog.key_rsa" fred@server.example.org
```

The debugging verbosity on the client can be increased just like on the server.

```
$ sftp -vvv -o "IdentityFile=~/.ssh/weblog.key_rsa" fred@server.example.org
```

The extra information can be useful to see exactly what is being sent to or requested of the server.

Even when a connection is already established, it is possible to change the verbosity of the client. Using the escape sequences **~v** and **~V** it is possible to raise and lower the logging level for the client in an existing connection. In order, it will raise the log level to VERBOSE, DEBUG, DEBUG2, and DEBUG3, if starting from the default of INFO. When lowering the log level, it will descend through ERROR, FATAL, to QUIET if starting from the default of INFO.

Debugging and Troubleshooting

The server logs are your best friend when troubleshooting. It may be necessary to turn up the log level there temporarily to get more information. It is then also necessary to turn them back to normal after things are fixed to avoid privacy problems or excessively large log files.

For example, the SFTP-subsystem logging defaults to ERROR, reporting only errors. To track transactions made by the client, change the log level to INFO or VERBOSE:

```
Subsystem internal-sftp  -l INFO
```

Caution. Again, operating with elevated logging levels would violate the privacy of users, in addition to filling a lot of disk space, and should generally not be used in production once the changes are figured out.

Also, the manual pages for OpenSSH are very well written and many times problems can be solved by finding the right section within the right manual page. At the very minimum, it is important to skim through the four main manual pages for both the programs and their configuration and become familiar with at least the section headings.

- ssh(1) (http://man.openbsd.org/ssh.1)
- ssh_config(5) (http://man.openbsd.org/ssh_config.5)
- sshd(8) (http://man.openbsd.org/sshd.5)

- sshd_config(5) (http://man.openbsd.org/sshd_config.5)

Then once the right section is found in the manual page, go over it in detail and become familiar with its contents. The same goes for the other OpenSSH manual pages, depending on the activity. Be sure to use the version of OpenSSH available for your system and the corresponding manual pages, preferably those that are installed on your system to avoid a mismatch. In some cases, the client and the server will be of different versions, so the manual pages for each must be looked up separately.

With a few exceptions below, specific examples of troubleshooting are usually given in the cookbook section relevant to a particular activity. So, for example, sorting problems with authentication keys is done in the section on Public Key Authentication itself.

Debugging a script, configuration or key that uses sudo

Usually log levels need only be changed when writing and testing a script, new configurations, new keys, or all three at once. When working with sudo, it is especially important to see exactly what the client is sending so as to enter the right pattern into **/etc/sudoers** for safety. Using the lowest level of verbosity, the exact string being sent by the client to the remote server is shown in the debugging output:

```
$ rsync -e "ssh -v -i /home/webmaint/.ssh/bkup_key -l webmaint" \
        -a server.example.org:/var/www/ var/backup/www/
...
debug1: Authentication succeeded (publickey).
Authenticated to server.example.org ([192.0.2.20]:22).
debug1: channel 0: new [client-session]
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: Sending command: rsync --server --sender -vlogDtpre.if . /var/www/
receiving incremental file list
...
```

What sudoers then needs is something like the following, assuming account *webmaint* is in the group *webmasters*:

```
%webmasters ALL=(ALL) NOPASSWD: /usr/local/bin/rsync --server \
--sender -vlogDtpre.if . /var/www/
```

The same method can be used to debug new server configurations or key logins. Once things are set to run as needed, the log level settings can be lowered back to INFO for sshd(8) (http://man.openbsd.org/sshd.8) and to ERROR for **internal-sftp**. Additionally, once the script is left to run in fully automated mode, the client logging information can be set use the syslog system module instead of **stderr** by setting the **-y** option when it is launched.

Debugging a server configuration

Running the server in debug mode provides a lot of information about the connection and a smaller amount about the server configuration. The server's debugging level (**-d**) can be raised once, twice (**-dd**) or thrice (**-ddd**).

```
$ /usr/sbin/sshd -d
```

Note that the server in this case does not detach and become a daemon, so it will terminate when the SSH connection terminates. To make a subsequent connection from the client, the server must be started again. Though in some ways this is a hassle, it does make sure that session data is a unique set and not mixes of multiple sessions and thus possibly different configurations. Alternately, another option (**-e**) when debugging

sends the debugging data to **stderr** to keep the system logs clean.

In recent versions of OpenSSH, it is also possible to log the debug data from the system logs directly to a separate file and keep noise out of the system logs. Since OpenSSH 6.3, the option **-E** will append the debug data to a particular log file instead of sending it to the system log. This facilitates debugging live systems without cluttering the system logs.

```
$ /usr/sbin/sshd -E /home/fred/sshd.debug.log
```

On older versions of OpenSSH, if you need to save output to a file while still viewing it live on the screen, you can use **tee(1)**.

```
$ /usr/sbin/sshd -ddd 2>&1 | tee /tmp/foo
```

That will save output to the file *foo* by capturing what sshd(8) (http://man.openbsd.org/sshd.8) sent to **stderr**. This works with older versions of OpenSSH, but the **-E** option above is preferable.

If the server is remote and it is important to reduce the risk of getting locked out, the experiments on the configuration file can be done with a second instance of sshd(8) (http://man.openbsd.org/sshd.8) using a separate configuration file and listening to a high port until the settings have been tested.

```
$ /usr/sbin/sshd -dd -p 22222 -f /home/fred/sshd_config.test
```

It is possible to make an extended test (**-T**) of the configuration file. If there is a syntax error, it will be reported, but remember that even sound configurations could still lock you out. The extended test mode can be used by itself, but it is also possible to specify particular connection parameters to use with **-C**. sshd(8) (http://man.openbsd.org/sshd.8) will then process the configuration file in light of the parameters passed to it and output the results. Of particular use, the results of **Match** directives will be shown.

When passing specific connection parameters to sshd(8) (http://man.openbsd.org/sshd.8) for evaluation, *user*, *host*, and *addr* are the minimum required for extended testing. The following will print out the configurations that will be applied if the user *fred* tries to log in to the host *server.example.org* from the address *192.0.2.15*:

```
$ /usr/sbin/sshd -TC user=fred,host=server.example.org,addr=192.0.2.15
```

Two more parameters, *laddr* and *lport*, may also be passed. They refer to the server IP number and port connected to.

```
$ /usr/sbin/sshd -TC user=fred,host=server.example.org,addr=192.0.2.15,laddr=192.0.2.2,lport=2222
```

Sometimes when debugging a server configuration it is necessary to track the client, too. With sftp(1) (http://man.openbsd.org/sftp.1) the options are also passed to ssh(1) (http://man.openbsd.org/ssh.1).

Debugging a client configuration

Since OpenSSH 6.8, the -G option makes ssh(1) (http://man.openbsd.org/ssh.1) print its configuration after evaluating **Host** and **Match** blocks and then exit. That allows viewing of the exact configuration options that will actually be used by the client for a particular connection.

```
$ ssh -G -l fred server.example.org
```

Client configuration is determined by run-time options, the user's own configuration file, or the system-wide client configuration file, in that order, whichever value is first.

Invalid or Outdated Ciphers or MACs

A proper client will show the details of the failure. For a bad Message Authentication Code (MAC), a proper client might show something like the following when trying to foist a bad MAC like hmac-md5-96 onto the server:

```
no matching mac found: client hmac-md5-96 server umac-64-etm@openssh.com,umac-128-etm@openssh.com,hmac-
sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh.com,hmac-sha1-etm@openssh.com,umac-
64@openssh.com,umac-128@openssh.com,hmac-sha2-256,hmac-sha2-512,hmac-sha1
```

And for a bad cipher, a proper client might show something like this when trying to foist an arcfour cipher on the server:

```
no matching cipher found: client arcfour server chacha20-poly1305@openssh.com,aes128-ctr,aes192-
ctr,aes256-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com
```

Sometimes when troubleshooting a problem with the client it is necessary to turn to the server logs. In OpenSSH 6.7 unsafe MACs were removed (http://www.openssh.com/txt/release-6.7) and in OpenSSH 7.2 unsafe ciphers were removed (http://www.openssh.com/txt/release-7.2), but some third-party clients may still try to use them to establish a connection. In that case, the client might not provide much information beyond a vague message that the server unexpectedly closed the network connection. The server logs will, however, show what happened:

```
fatal: no matching mac found: client hmac-sha1,hmac-sha1-96,hmac-md5 server hmac-sha2-512-
etm@openssh.com,hmac-sha2-256-etm@openssh.com,umac-128-etm@openssh.com,hmac-sha2-512,hmac-sha2-256,hmac-
ripemd160 [preauth]
```

Or more recent versions would show a simpler error for a bad MAC.

```
fatal: Unable to negotiate with 192.0.2.37 port 55044: no matching MAC found. Their offer: hmac-md5-96
[preauth]
```

or for a bad cipher.

```
fatal: Unable to negotiate with 192.0.2.37 port 55046: no matching cipher found. Their offer: arcfour
[preauth]
```

The error message in the server log might not say which MACs or ciphers are actually available. For that, the extended test mode can be used to show the server settings, in particular the MACs or ciphers allowed. In its most basic usage the extended test mode would just be -T, as in `/usr/sbin/sshd -T | egrep 'cipher|macs'` with no other options. For more details and options, see the previous section on "Debugging a server configuration" above.

One solution there is to upgrade the client to one that can handle the right ciphers and MACs. Another option is to switch to a different client, one that can handle the modern ciphers or MACs.

Development

It is possible to advance OpenSSH through donations of hardware or money. See the OpenSSH project web site at www.openssh.org (http://www.openssh.org) for details.

OpenSSH is a volunteer project with the goal of making quality software. In that way it relies upon hardware and cash donations to keep the project rolling. Funds are needed for daily operation to cover network line subscriptions and electrical costs. If 2 dollars were given for every download of the OpenSSH source code in 2015 from the master site, ignoring the mirrors, or if a penny was donated for every pf or OpenSSH installed with a mainstream operating system or phone in 2015[1], then funding goals for the year would be met. Hardware is needed for development and porting to new architectures and platforms always requires new hardware.

OpenSSH is currently developed by two teams. The first team works providing code that is as clean, simple and secure as possible as part of the OpenBSD project. The second team works using this core version and ports it to a great many other operating systems. Thus there are two development tracks, the OpenBSD core and the portable version. The work is all done in countries that permit export of cryptography.

Contents

Use the Source, Luke

The main development branch of OpenSSH is part of the OpenBSD project. So the "-current" branch of OpenBSD, available as source code, is where to look for current activity.

The source code for the portable releases of OpenSSH are published using anonymous CVS, so no password is needed to download source from the read-only repository. It is provided and maintained by Damien Miller. Nightly, bleeding-edge snapshots of OpenSSH itself are publicly available from its own CVS tree. Use a mirror when possible.

```
export CVSROOT=anoncvs@anoncvs.mindrot.org:/cvs
export CVS_RSH=/usr/bin/ssh
cvs get openssh
```

The fingerprint for the key used by the OpenSSH source code repository, as of this writing, is:

```
2048 SHA256:UNyCGjDDKB8hPDhrgMRAID6F53TyECEgnMmBN/4ZbuY anoncvs.mindrot.org (RSA)
```

We ask anyone wishing to report security bugs in OpenSSH to please use the contact address given in the source and to practice responsible disclosure.

libssh

libssh is an independend project that provides a mulitplatform C library implementing the SSHv2 and SSHv1 protocol for client and server implementations. With libssh, developers can remotely execute programs, transfer files and use a secure and transparent tunnel for your remote applications.

libssh is available under LGPL 2.1 license, on the web page https://www.libssh.org/

Features:

- Key Exchange Methods: curve25519-sha256@libssh.org, ecdh-sha2-nistp256, diffie-hellman-group1-sha1, diffie-hellman-group14-sha1
- Hostkey Types: ecdsa-sha2-nistp256, ssh-dss, ssh-rsa
- Ciphers: aes256-ctr, aes192-ctr, aes128-ctr, aes256-cbc, aes192-cbc, aes128-cbc, 3des-cbc, des-cbc-ssh1, blowfish-cbc
- Compression Schemes: zlib, zlib@openssh.com, none
- MAC hashes: hmac-sha1, none
- Authentication: none, password, public-key, hostbased, keyboard-interactive, gssapi-with-mic
- Channels: shell, exec (incl. SCP wrapper), direct-tcpip, subsystem, auth-agent-req@openssh.com
- Global Requests: tcpip-forward, forwarded-tcpip
- Channel Requests: x11, pty, exit-status, signal, exit-signal, keepalive@openssh.com, auth-agent-req@openssh.com
- Subsystems: sftp(version 3), publickey(version 2), OpenSSH Extensions
- SFTP: statvfs@openssh.com, fstatvfs@openssh.com
- Thread-safe: Just don’t share sessions
- Non-blocking: it can be used both blocking and non-blocking
- Your sockets: the app hands over the socket, or uses libssh sockets
- OpenSSL or gcrypt: builds with either

Additional Features:

- Client and server support
- SSHv2 and SSHv1 protocol support
- Supports Linux, UNIX, BSD, Solaris, OS/2 and Windows
- Full API documentation and a tutorial
- Automated test cases with nightly tests
- Event model based on poll(2), or a poll(2)-emulation.

libssh2

libssh2 is another independent project providing a lean C library implementing the SSH2 protocol for embedding specific SSH capabilities into other tools. It has a stable, well-documented API for working on the client side with the different SSH subsystems: Session, Userauth, Channel, SFTP, and Public Key. The API can be set to either blocking or non-blocking. The code uses strict name spaces, is C89-compatible and

builds using regular GNU Autotools.

libssh2 is available under a modified BSD license. The functions are each documented in their own manual pages. The project web site contains the documentation, source code and examples: http://www.libssh2.org/

There is a mailing list for **libssh2** in addition to an IRC channel. The project is small, low-key and, as true to the spirit of the Internet, a meritocracy. Hundreds of specific functions allow specific activities and components to be cherry-picked and added to an application:

- Shell and SFTP sessions
- Port forwarding
- Password, public-key, host-based keys, and keyboard-interactive authentication methods.
- Key Exchange Methods diffie-hellman-group1-sha1, diffie-hellman-group14-sha1, diffie-hellman-group-exchange-sha1
- Host Key Types: ssh-rsa and ssh-dss
- Ciphers: aes256-cbc (rijndael-cbc@lysator.liu.se), aes192-cbc, aes128-cbc, 3des-cbc, blowfish-cbc, cast128-cbc, arcfour, or without a cipher.
- Compression Scheme zlib or without compression
- Message Authentication Code (MAC) algorithms for hashes: hmac-sha1, hmac-sha1-96, hmac-md5, hmac-md5-96, hmac-ripemd160 (hmac-ripemd160@openssh.com), or none at all
- Channels: Shell, Exec – including the SCP wrapper, direct TCP/IP, subsystem
  - Channel Requests: x11, pty
- Subsystems: sftp version 3, public-key version 2
- Thread-safe, blocking or non-blocking API
- Your sockets: the app hands over the socket, calls select() etc.
- Builds with either OpenSSL or gcrypt

See also the library **libcurl** which supports SFTP and SCP URLs.

Other language bindings for the SSH protocols

What follows is a list of additional independent resources by programming language:

Perl

- Net::SSH2 (http://search.cpan.org/perldoc?Net::SSH2): a wrapper module for **libssh2**.
- Net::SSH::Perl (http://search.cpan.org/perldoc?Net::SSH::Perl): a full SSH/SFTP implementation in pure Perl. Unfortunately this module is not being maintained any more and has several open bugs. Also, installing it can be a daunting task due to some of its dependencies.
- Net::OpenSSH (http://search.cpan.org/perldoc?Net::OpenSSH): a wrapper for OpenSSH binaries and other handy programs (**scp**, **rsync**, **sshfs**). It uses OpenSSH multiplexing feature in order to reuse connections.
- Net::OpenSSH::Parallel (http://search.cpan.org/perldoc?Net::OpenSSH::Parallel) a module build on top of **Net::OpenSSH** that allows to transfer files and run programs on several machines in parallel efficiently.
- SSH::Batch (http://search.cpan.org/perldoc?SSH::Batch) another module build on top of **Net::OpenSSH** that allows to run programs on several hosts in parallel.
- Net::SSH::Expect (http://search.cpan.org/perldoc?Net::SSH::Expect): this module uses Expect (http://search.cpan.org/search?query=expect) to drive interactive shell sessions run on top of SSH.
- Net::SSH (http://search.cpan.org/perldoc?Net::SSH): a simple wrapper around any SSH client. It does not support password authentication and is very slow as it establishes a new SSH connection for every

remote program invoked.
- Net::SCP (http://search.cpan.org/perldoc?Net::SCP) and Net::SCP::Expect (http://search.cpan.org/perldoc?Net::SCP::Expect): modules wrapping the **scp** program. Note that **Net::SSH2**, **Net::SSH::Perl** and **Net::OpenSSH** already support file transfers via **scp** natively.
- Net::SFTP::Foreign (http://search.cpan.org/perldoc?Net::SFTP::Foreign): a full SFTP client written in Perl with lots of bells and whistles. By default is uses **ssh** to connect to the remote machines but it can also run on top of **Net::SSH2** and **Net::OpenSSH**.
- GRID::Machine (http://search.cpan.org/perldoc?GRID::Machine), IPC::PerlSSH (http://search.cpan.org/perldoc?IPC::PerlSSH) and SSH::RPC (http://search.cpan.org/perldoc?SSH::RPC): these modules allow to distribute and run Perl code on remote machines through SSH.

Python

Paramiko (http://www.lag.net/paramiko/)

- http://www.lag.net/paramiko/

Fabric (http://docs.fabfile.org/)

- http://docs.fabfile.org/

libssh2 (http://www.no-ack.org/2010/11/python-bindings-for-libssh2.html)

- http://www.no-ack.org/2010/11/python-bindings-for-libssh2.html

Ruby

Net::SSH (https://github.com/net-ssh)

- https://github.com/net-ssh

Capistrano (https://github.com/capistrano/capistrano)

- https://github.com/capistrano/capistrano

Java

Jaramiko (http://www.lag.net/jaramiko/)

- http://www.lag.net/jaramiko/

JSch (http://www.jcraft.com/jsch/) - a pure Java implementation of SSH2.

- http://www.jcraft.com/jsch/

References

1. "The OpenBSD Foundation 2016 Fundraising Campaign". The OpenBSD Foundation. 2016. http://www.openbsdfoundation.org/campaign2016.html. Retrieved 2016-03-07.

Cookbook/Tunnels

In tunneling, or port forwarding, a local port is connected to a port on a remote host or vice versa. So connections to the port on one machine are really connections to a port on the other machine.

The ssh(1) (http://man.openbsd.org/ssh.1) options **-f** (go to background), **-N** (do not execute a remote program) and **-T** (disable pseudo-tty allocation) can be useful for connections that are used only for creation of tunnels.

Tunneling

In regular port forwarding, connections to a local port are forwarded to a port on a remote machine. This is a way of securing an insecure protocol or of making a remote service appear as local. Here we forwarded VNC in two steps. First make the tunnel:

```
$ ssh -L 5901:localhost:5901 -l fred desktop.example.org
```

Then on the local machine, connections to the forwarded port will really be connecting to the remote machine.

Multiple tunnels can be specified at the same time. The tunnels can be of any kind, not just regular forwarding. See the next section below for reverse tunnels. For dynamic forwarding see the section Proxies and Jump Hosts.

```
$ ssh -L 5901:localhost:5901 \
     -L 5432:localhost:5432 \
     -l fred desktop.example.org
```

If a connection is only used to create a tunnel, then it can be told not to execute any remote programs (**-N**), making it a non-interactive session, and to drop to the background (**-f**).

```
$ ssh -fN -L 3128:localhost:3128 -l fred server.example.org
```

Note that **-N** will work even if the **authorized_keys** forces a program using the **command=** option. So a connection using **-N** will stay open instead of running a program and then exiting.

Tunneling via an intermediate host

Tunneling can go via one intermediate host to reach a second host, the latter does not need to be on a publicly accessible network. However, the target port on the second remote machine does have to be accessible on the same network as the first. Here, *192.168.0.101* and *bastion.example.org* must be on the same network and, in addition, *bastion.example.org* has to be directly accessible to the client machine running **ssh(1)**. So, port 80 on *192.168.0.101* has to be available to the machine *bastion.example.org*.

```
$ ssh -fN -L 1880:192.168.0.101:80 -l fred bastion.example.org
```

Thus, to connect to port 80 on *192.168.2.101* via the host *bastion.example.org*, once the tunnel is made, connect to port 1880 on *localhost*. This way works for one or two hosts. It is also possible to chain multiple hosts, using a different method.

For more about passing through intermediate computers, see the Cookbook section on Proxies and Jump Hosts.

Reverse Tunneling

A reverse tunnel goes the opposite direction of a regular tunnel. In a reverse tunnel, a port on the remote host is forwarded to the local host. Once the connection is made, it works the same as a regular tunnel. Connections to the destination port on the local host connect to the remote host's port.

On the machine that will become the remote machine, open a reverse tunnel.

```
$ ssh -fNT -R 2022:localhost:22 -l fred server.example.org
```

Then on the local machine, connecting to the forwarded port as the local host, will open the connection to the machine hosting the reverse tunnel.

```
$ ssh -p 2022 -l fred localhost
```

A common use-case for reverse tunneling is when you have to access a machine that is behind a firewall that blocks incoming ssh but without changing the firewall settings, and you have direct access to a second machine outside the firewall. It is easy to make a reverse tunnel from the machine behind the firewall to the second machine. Then to conect to the first machine from outside, connect to the forwarded port on the second machine. The second machine on the outside acts as a relay server to forward connections to the machine on the inside.

Adding or Removing Tunnels within an Established Connection

It is possible to add or remove tunnels, reverse tunnels, and SOCKS proxies to or from an existing connection using an escape sequence. The default escape character is the tilde (~) and the full range of options is described in the manual page for ssh(1) (http://man.openbsd.org/ssh.1). Escape sequences only work if they are the first characters entered on a line and if followed by a return. When adding or removing a tunnel to or from an existing connection, **~C**, the command line is used.

To add a tunnel in an active SSH session, use the escape sequence to open a command line in SSH and then enter the parameters for the tunnel:

```
~C
L 2022:localhost:22
```

To remove a tunnel from an active SSH session is almost the same. Instead of -L, -R, or -D we have -KL, -KR, and -KD plus the port number. Use the escape sequence to open a command line in SSH and then enter the parameters for removing the tunnel.

```
~C
KL2022
```

Adding or Removing Tunnels within a Multiplexed Connection

There is an additional option for forwarding when multiplexing. More than one SSH connection can be multiplexed over a single TCP connection. Control commands can be passed to the master process to add or drop port forwarding to the master process.

First a master connection is made and a socket path assigned.

```
$ ssh -S '/home/fred/.ssh/%h:%p' -M server.example.org
```

Then using the socket path, it is possible to add port forwarding.

```
$ ssh -O forward -L 2022:localhost:22 -S '/home/fred/.ssh/%h:%p' fred@server.example.org
```

Since OpenSSH 6.0 it is possible to cancel specific port forwarding using a control command.

```
$ ssh -S  "/home/fred/.ssh/%h:%p" -O cancel -L 2022:localhost:22 fred@server.example.org
```

For more about multiplexing, see the Cookbook section on Multiplexing.

Cookbook/Automated Backup

Using OpenSSH with keys can facilitate secure automated backups. It's a myth that remote root access must be allowed. sudo(8) (http://linux.die.net/man/8/sudo) works just fine -- if properly configured. rsync(1) (http://linux.die.net/man/1/rsync)[1], tar(1) (http://man.openbsd.org/tar.1), and dump(8) (http://man.openbsd.org/dump.8) are the foundation for most backup methods. Remember, that until the backup data has been tested and shown to restore reliably, it does not count as a backup copy.

Backup with rsync

rsync(1) (http://linux.die.net/man/1/rsync) now defaults to using SSH. But it still can be specified explicitly:

```
$ rsync --exclude '*~' -avv \
       -e "ssh" \
       fred@server.example.org:./archive \
       /Users/fred/archive/.
```

For some types of data, transfer can be speeded up greatly by using rsync(1) (http://linux.die.net/man/1/rsync) with compression, **-z**.

Rsync with keys

rsync(1) (http://linux.die.net/man/1/rsync) can authenticate using SSH keys. If the key is added to an agent, then the passphrase only needs to be entered once:

```
$ rsync --exclude '*~' -avv \
```

```
        -e "ssh -i ~/.ssh/key_rsa" \
        fred@server.example.org:./archive \
        /Users/fred/archive/.
```

Backup with rsync and sudo

rsync(1) (http://linux.die.net/man/1/rsync) is often used to back up locally or remotely. rsync(1) (http://linux.die.net/man/1/rsync) is fast and flexible and copies incrementally so only the changes are transferred, thus avoiding wasting time re-copying what is already at the destination. It does that through use of its now famous algorithm. When working remotely, it needs a little help with the encryption and the usual practice is to tunnel it over SSH.

Preparation: create an account to use for the backup, create a pair of keys to use only for backup, then make sure you can log in to that account with ssh(1) (http://man.openbsd.org/ssh.1) with and without those keys.

```
$ ssh -t -i ~/.ssh/mybkupkey bkupacct@www.example.org
```

Step 1: Configure sudoers(5) (http://linux.die.net/man/5/sudoers) and test rsync(1) (http://linux.die.net/man/1/rsync) with sudo(8) (http://linux.die.net/man/8/sudo) on the remote host. In this case data is staying on the remote machine.

Step 2: Test rsync(1) (http://linux.die.net/man/1/rsync) with sudo(8) (http://linux.die.net/man/8/sudo) over ssh(1) (http://man.openbsd.org/ssh.1).

```
$ ssh -l bkupacct www.example.org sudo rsync -av:/var/www/ /tmp/
```

It will be necessary to tune **/etc/sudoers** a little at this stage. More refinements may come later. Note that there is an rsync(1) (http://linux.die.net/man/1/rsync) user and an ssh(1) (http://man.openbsd.org/ssh.1) user. The data in this case gets copied from the remote machine to the local **/tmp**.

```
$ rsync -e "ssh -t -l bkupacct" --rsync-path='sudo rsync' \
-av bkupacct@www.example.org:/var/www/ /tmp/
```

Step 3: Use the key.

```
$ rsync -e "ssh -i ~/.ssh/key -t -l bkupacct" --rsync-path='sudo rsync' \
-av bkupacct@www.example.org:/var/www/ /tmp/
```

Step 4: Adjust **/etc/sudoers** so that the backup account has enough access to run rsync(1) (http://linux.die.net/man/1/rsync) but only in the directories it is supposed to run in and without free-rein on the system. Use the first debugging level to see the actual parameters getting passed to the remote host. That provides the basis of what **/etc/sudoers** will need:

```
$ rsync -e "ssh -t -v" --rsync-path='sudo rsync' \
-av bkupacct@www.example.org:/var/www/ /tmp/
...
debug1: Sending command: sudo rsync --server --sender -e.iLs . /var/www
...
```

Be sure that the backed up data is not accessible to others. At this point you are done. However the process can be automated much further.

Step 5: Test rsync(1) (http://linux.die.net/man/1/rsync) with sudo(8) (http://linux.die.net/man/8/sudo) over

ssh(1) (http://man.openbsd.org/ssh.1).

```
$ rsync -e "ssh -t" --rsync-path='sudo rsync' \
-av bkupacct@www.example.org:/var/www/ /tmp/
```

Ok. The account on the server is named 'bkupacct' and the private RSA key is **~/.ssh/key_bkup_rsa** on the client. On the server, the account 'bkupacct' is a member of the group 'autobackup'.

The public key, **~/.ssh/key_bkup_rsa.pub**, has been copied to the account bkupacct on server and placed in **~/.ssh/authorized_keys** there.

The following directories on server are owned by root and belong to the group bkupacct and not group readable, but not group writeable, and definitely not world readable: **~** and **~/.ssh**. Same for the file **~/.ssh /authorized_keys** there. (This assumes you are not also using ACLs) This is one way of many to set permissions on the server:

```
$ sudo chown root:bkupacct ~
$ sudo chown root:bkupacct ~/.ssh/
$ sudo chown root:bkupacct ~/.ssh/authorized_keys
$ sudo chmod u=rwx,g=rx,o= ~
$ sudo chmod u=rwx,g=rx,o= ~/.ssh/
$ sudo chmod u=rwx,g=r,o=  ~/.ssh/authorized_keys
```

Say you're backing up from server to client. rsync(1) (http://linux.die.net/man/1/rsync) on the client uses ssh(1) (http://man.openbsd.org/ssh.1) to make the connection to rsync on the server. rsync(1) (http://linux.die.net/man/1/rsync) is invoked from client like this to see exactly what parameters are being passed to the server:

```
$ rsync \
  -e "ssh \
          -i ~/.ssh/key_bkup_rsa  \
          -t              \
          -l bkupacct"   \
  --rsync-path='sudo rsync' \
  --delete   \
  --archive  \
  --compress \
  --verbose  \
  bkupacct@server:/var/www \
  /media/backups/server/backup/
```

sudo(8) (http://linux.die.net/man/8/sudo) will need to be configured on the server. The argument **--rsync-path** tells the server what to run in place of rsync(1) (http://linux.die.net/man/1/rsync). In this case it runs `sudo rsync`. The argument **-e** says which remote shell tool to use. In this case it is ssh(1) (http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/ssh.1). For the SSH client being called by the rsync(1) (http://linux.die.net/man/1/rsync) client, **-i** says which key, specifically, to use. That is independent of whether or not an authentication agent is used for ssh keys. Having more than one key is a possibility, since it is possible to have different keys for different tasks.

Keep making adjustments to **/etc/sudoers** on the server until it works as it should. You can find the exact settings(s) to use in **/etc/sudoers** by running the SSH in verbose mode (**-v**) on the client. Be careful when working with patterns not to match more than is safe.

```
%autobackup ALL=(ALL) NOPASSWD: /usr/local/bin/rsync --server \
--sender -vlogDtpre.if . /var/www/
```

Backup using tar

The main choice for creating archives is tar(1) (http://man.openbsd.org/tar.1). But since it copies whole files and directories, rsync(1) (http://linux.die.net/man/1/rsync) is usually much more efficient for updates or incremental backups.

The following will make a tarball of the directory **/var/www** and send it via stdout into sdtin via a pipe into ssh(1) (http://man.openbsd.org/ssh.1) where, on the remote machine it is directed into the file called backup.tar. Here tar(1) (http://man.openbsd.org/tar.1) runs on a local machine and stores the tarball remotely:

```
$ tar cf - /var/www/ | ssh -l fred server.example.org "cat > backup.tar"
```

There are really limitless options for that recipe:

```
$ tar zcf - /var/www/ /home/*/www/ \
       |  ssh -l fred server.example.org "cat > $(date +"%Y-%m-%d").tar.gz
```

That will do the same, but also get user www directories, compress the tarball using **gzip**, and label the resulting file according to the current date.

```
$ tar zcf - /var/www/ /home/*/www/ \
       |  ssh -i key -l fred server.example.org "cat > $(date +"%Y-%m-%d").tgz
```

It is just as easy to tar(1) (http://man.openbsd.org/tar.1) what is on a remote machine and store the tarball locally.

```
$ ssh fred@server.example.org "tar zcf - /var/www/" >  backup.tgz
```

Or a fancier example of running tar on the remote machine but storing the tarball locally.

```
$ ssh -i key -l fred server.example.org "tar jcf - /var/www/ /home/*/www/" \
       > $(date +"%Y-%m-%d").tar.bz2
```

The secret to the backup is the use of **stdout** and **stdin** to effect the transfer through judicious use of pipes and redirects.

Backup of files without making a tarball

Sometimes it is necessary to just transfer the files and directories without making a tarball at the destination. In addition to writing to **stdin** on the source machine, tar(1) (http://man.openbsd.org/tar.1) can read from **stdin** on the destination machine to transfer whole directory hierarchies at once.

```
$ tar zcf - /var/www/ | ssh -l fred server.example.org "cd /some/path/; tar zxf -"
```

Or going the opposite direction, it would be the following.

```
$ ssh 'tar zcf - /var/www/' | (cd /some/path/; tar zxf - )
```

However, these still copy everything each time they are run. So rsync(1) (http://linux.die.net/man/1/rsync) described above in the previous section might be a better choice in many situations, since on subsequent runs it only copies the changes.

Backup using dump

Using dump(8) (http://man.openbsd.org/dump.8) remotely is like using tar(1) (http://man.openbsd.org/tar.1). One can copy from the remote server to the local server.

```
$ ssh -t source.example.org 'sudo dump -0an -f - /var/www | gzip -c9' > backup.dump.gz
```

Note that the password prompt for sudo(8) (http://linux.die.net/man/8/sudo) might not be visible and it must be typed blindly.

Or one can go the other direction, copying from the locate server to the remote:

```
$ sudo dump -0an -f - /var/www | gzip -c9 | ssh target.example.org 'cat > backup.dump.gz'
```

Note that here the password prompt might get hidden in the initial output from dump(8) (http://man.openbsd.org/dump.8). It's still there.

References

1. "How Rsync Works". Samba. http://www.samba.org/rsync/how-rsync-works.html.

Cookbook/File Transfer with SFTP

The basic SFTP service requires no additional setup, it is a built-in part of the OpenSSH server. However, the same options and tricks available for the regular SSH client are also available for SFTP clients. Some options may have to be specified with the full option name using the **-o** argument.

For many graphical SFTP clients, it is possible to use a regular URL to point to the target. Many file managers nowadays have built-in support for SFTP. See the section "GUI Clients" above.

The subsystem sftp-server(8) (http://man.openbsd.org/sftp-server.8) implements SFTP file transfer. See the manual page for sftp-server(8) (http://man.openbsd.org/sftp-server.8). The subsystem internal-sftp implements an in-process SFTP server which may simplify configurations using **ChrootDirectory** to force a different filesystem root on clients.

Basic SFTP

Just to say it again, regular SFTP access requires no additional changes from the default configuration. SFTP provides a very easy to use and very easy to configure option for accessing a remote system. The usual clients can be used or special ones like sshfs(1) (http://linux.die.net/man/1/sshfs).

Automated SFTP

SFTP uploads or downloads can be automated. The prerequisite is key-based authentication. Once key-based authentication is working, a batch file can be used to carry out activities via SFTP. See the *batchfile* option **-b** in sftp(1) (http://man.openbsd.org/sftp.1) for details.

```
$ sftp -b /home/fred/cmds.batch -i /home/fred/.ssh/foo_key_rsa server.example.org:/home/fred/logs/
```

If a dash (-) is used as the batchfile name, SFTP commands will be read from **stdin**.

```
$ echo "put /var/log/foobar.log" | sftp -b - -i /home/fred/.ssh/foo_key_rsa server.example.org:/home/fre
```

More than one SFTP command can be sent, but it is better then to use an actual batch file.

```
$ echo -e "put /var/log/foobar.log\nput /var/log/munged.log" | sftp -b - -i /home/fred/.ssh/foo_key_rsa
```

The batch file mode can be very useful in cron jobs and in scripting.

SFTP-only Accounts

Using the **Match** directive in sshd_config(5) (http://man.openbsd.org/sshd_config.5), it is possible to limit members of a specific group to using only SFTP for interaction with the server.

```
Subsystem sftp internal-sftp

Match Group sftp-only
        AllowTCPForwarding no
        X11Forwarding no
        ForceCommand internal-sftp
```

Note that disabling TCP forwarding does not improve security unless users are also denied shell access, as they can in principle install their own forwarders.

See PATTERNS in ssh_config(5) (http://man.openbsd.org/ssh_config.5) for more information on patterns avaiable to **Match**.

Chrooted SFTP-only Accounts

It's common that a group of users need to read and write files to their home directories on the server, but have little or no reason to access the rest of the file system. SFTP provides a very easy to use and very easy to configure chroot. In some cases, it is enough to chroot users to their home directories. Depending on the opensshd version this may not be usable since some versions require that the chroot-target directory and all parent directories are owned by root and not writeable by any others. In most cases home directories aren't owned by root and allow writing by at least one user. One way around this restriction is to have the home directory owned by root, but have it populated with a number of other directories and files that are owned by the actual user to which the user can write.

```
Subsystem sftp internal-sftp

Match Group sftp-only
        ChrootDirectory %h
        AllowTCPForwarding no
        X11Forwarding no
```

```
        ForceCommand internal-sftp
```

If it is not practical to have users' home directories owned by root, a compromise can be made. **ChrootDirectory** can point to **/home**, which must be owned by root anyway, and then **ForceCommand** can then designate the user's home directory as the starting directory using the **-d** option.

```
Subsystem sftp internal-sftp

Match Group sftp-only
        ChrootDirectory /home
        AllowTCPForwarding no
        X11Forwarding no
        ForceCommand internal-sftp -d %u
```

If it is necessary to hide the contents of the home directories from other users, chmod(1) (http://man.openbsd.org/chmod.1) should be used. Permissions could be 0111 for **/home** and 0750 or 0700 for the home directories, be sure to check the group memberships as well.

Another common case is to chroot access to a web server's document root or server root.

```
Subsystem sftp internal-sftp

Match Group webmasters
        ChrootDirectory /var/www
        AllowTCPForwarding no
        X11Forwarding no
        ForceCommand internal-sftp
```

Umask

Starting with OpenSSH 5.4, sftp-server(8) (http://man.openbsd.org/sftp-server.8) can set a umask to override the default one set by the user’s account. The in-process SFTP server, internal-sftp, accepts the same options as the external SFTP subsystem.

```
Subsystem sftp internal-sftp -u 0022
```

Earlier versions can do the same thing through the use of a helper script, but this complicates chrooted directories very much. The helper script can be a regular script or it can be embedded inline in the configuration file though neither works easily in a chroot jail. It’s often easier to get a newer version of sshd(8) (http://man.openbsd.org/sshd.8) which supports umask as part of the server’s configuration. Here is an inline helper script for umask in OpenSSH 5.3 and earler, based on one by gilles@

```
Subsystem sftp /bin/sh -c 'umask 0022; /usr/libexec/openssh/sftp-server'
```

This umask is server-side only. The original file permissions on the client side will usually, but not always, be used when calculating the final file permissions on the server. This depends on the client itself. Most clients pass the file permissions on to the server, FileZilla being a notable exception. As such, permissions can generally be tightened but not loosened. For example, a file that is mode 600 on the client will not be automatically made 664 or anything else less than the original 600 regardless of the server-side umask. That is unless the client does not forward the permissions, in which case only the server's umask will be used. So for most clients, if you want looser permissions on the uploaded file, change them on the client before uploading.

Chrooted SFTP to Shared Directories

Another common case is to chroot a group of users to different levels of the web server they are responsible for. For obvious reasons, symbolic links going from inside the jail to parts of the filesystem outside the chroot jail are not accessible to the chrooted users. So directory hierarchies must be planned more carefully if there are special combinations of access.

```
Subsystem sftp internal-sftp

Match Group webdevel
        ChrootDirectory /var/www/site1
        ForceCommand internal-sftp

Match Group webauthors
        ChrootDirectory /var/www/site1/htdocs
        ForceCommand internal-sftp
```

In these kinds of directories, it may be useful to give different levels of access to more than just one group. In that case, ACLs should be used.

Chrooted SFTP Accounts Accessible Only from Particular Addresses

More complex matching can be done. It is possible to allow a group of users to use SFTP, but not a shell login, only if they log in from a specific address or range of addresses. If they log in from the right addresses, then get SFTP and only SFTP, but if they try to log in from other addresses they will be denied access completely. Both conditions, the affirmative and negative matches, need to be accounted for.

```
Subsystem sftp internal-sftp

Match Group sftp-only, Address 192.0.43.10
        AllowTCPForwarding no
        X11Forwarding no
        ForceCommand internal-sftp
        ChrootDirectory  /home/servers/

Match Group sftp-only, Address *,!192.0.43.10
        DenyGroups sftp-only
```

Note that for negation a wildcard must be specified first and then the address or range to be excluded following it. Mind the spaces or lack thereof. Similar matching can be done for a range of addresses by specifying the addresses in CIDR address/mask format, such as 192.0.32.0/20. Any number of criteria can be specified and only if all of them are met then the directive in the subsequent lines take effect.

Again, the first **Match** block that fits is the one that takes effect, so care must be taken when constructing conditional blocks to make them fit the precise situation desired. Also, any situations that don't fit a **Match** conditional block will fall through the cracks. Those will get the general configuration settings whatever they may be. Specific user and source address combinations can be tested with the configurations using the **-T** and **-C** options with the server for more options. See the section Debugging a Server Configuration for more.

Chrooted SFTP with Logging

The logging daemon must establish a socket in the chroot directory for the sftp-server(8) (http://man.openbsd.org/sftp-server.8) subsystem to access as **/dev/log** See the section on Logging.

Chrooted login shells

The chroot and all its components, must be root-owned directories that are not writable by any other user or group. The **ChrootDirectory** must contain the necessary files and directories to support the user's session. For an interactive session this requires at least a shell, typically bash(1) (http://linux.die.net/man/1/bash),

ksh(1) (http://man.openbsd.org/ksh.1), or sh(1) (http://man.openbsd.org/sh.1), and basic **/dev** nodes such as null(4) (http://man.openbsd.org/null.4), zero(4) (http://man.openbsd.org/zero.4), stdin(4) (http://man.openbsd.org/stdin.4), stdout(4) (http://man.openbsd.org/stdout.4), stderr(4) (http://man.openbsd.org/stderr.4), arandom(4) (http://man.openbsd.org/arandom.4), and tty(4) (http://man.openbsd.org/tty.4) devices. The path may contain the following tokens that are expanded at runtime once the connecting user has been authenticated: **%%** is replaced by a literal '**%**', **%h** is replaced by the home directory of the user being authenticated, and **%u** is replaced by the username of that user.

sshfs - SFTP file transfer via local folders

Another way to transfer files back and forth, or even use them remotely, is to use sshfs(1) (http://linux.die.net/man/1/sshfs) It is a file system client based on SFTP and utilizes the sftp-subsystem. It can make a directory on the remote server accessible as a directory on the local file system which can be accessed by any program just as if it were a local directory. The user must have read-write privileges for mount point to use sshfs(1) (http://linux.die.net/man/1/sshfs).

The following creates the mount point, **mountpoint**, in the home directory if none exists. Then sshfs(1) (http://linux.die.net/man/1/sshfs) mounts the remote server.

```
$ test -e ~/mountpoint || mkdir --mode 700 ~/mountpoint
$ sshfs fred@server.example.org:. ~/mountpoint
```

Reading or writing files to the mount point is actually transferring data to or from the remote system. The amount of bandwidth consumed by the transfers can be reduced using compression. That can be important if the network connection has bandwidth caps or per-unit fees. However, if speed is the only issue, compression can make the transfer slower if the processors on either end are busy or not powerful enough. About the only way to be sure is to test and see which method is faster. Below, compression is specified with **-C**.

```
$ sshfs -C fred@server.example.org:. ~/mountpoint
```

Or try with debugging output:

```
$ sshfs -o sshfs_debug fred@server.example.org:. /home/fred/mountpoint
```

Named pipes will not work over sshfs(1) (http://linux.die.net/man/1/sshfs).

sshfs with a key

The **ssh_command** option is used to pass parameters on to ssh(1) (http://man.openbsd.org/ssh.1). In this example it is used to have ssh(1) (http://man.openbsd.org/ssh.1) point to a key used for authentication to mount a remote directory, **/usr/src**, locally as **/home/fred/src**.

```
$ sshfs -o ssh_command="ssh -i /home/fred/.ssh/id_rsa" fred@server.example.org:/usr/src /home/fred/src/
```

If a usable key is already loaded into the agent, then ssh(1) (http://man.openbsd.org/ssh.1) should find it and use it on behalf of sshfs(1) (http://linux.die.net/man/1/sshfs) without needing intervention.

Cookbook/Public Key Authentication

Authentication keys can improve efficiency, if done properly. As a bonus advantage, the passphrase and private key never leave the client[1]. They are generally recommended for outward facing systems so password authentication can be turned off.

Verify a Host Key by Fingerprint

The first time connecting to a remote host, the key itself should be verified. Usually this is done by comparing the fingerprint or the ASCII art visual host key, metadata about the key, rather than trying to compare the whole key itself.

```
$ ssh -l fred server.example.org
The authenticity of host 'server.example.org (192.0.32.10)' can't be established.
ECDSA key fingerprint is SHA256:LPFiMYrrCYQVsVUPzjOHv+ZjyxCHlVYJMBVFerVCP7k.
Are you sure you want to continue connecting (yes/no)?
```

The fingerprint can be forced to display as an MD5 hash in hexadecimal instead by passing **FingerprintHash** configuration directive as a runtime argument or in **ssh_config**. But the default is now SHA256 in base64.

```
$ ssh -o FingerprintHash=md5 host.example.org
The authenticity of host 'host.example.org (192.0.32.203)' can't be established.
RSA key fingerprint is MD5:10:4a:ec:d2:f1:38:f7:ea:0a:a0:0f:17:57:ea:a6:16.
Are you sure you want to continue connecting (yes/no)?
```

In OpenSSH 6.7 and earlier this fingerprint was a hexadecimal MD5 checksum instead a of the base64-encoded SHA256 checksum currently used:

```
$ ssh -l fred server.example.org
The authenticity of host 'server.example.org (192.0.32.10)' can't be established.
RSA key fingerprint is 4a:11:ef:d3:f2:48:f8:ea:1a:a2:0d:17:57:ea:a6:16.
Are you sure you want to continue connecting (yes/no)?
```

Downloading keys

Usually a host's key is displayed the first time the SSH client tries to connect. The remote host's public keys can also be fetched on demand using ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1):

```
$ ssh-keyscan host.example.org
# host.example.org SSH-2.0-OpenSSH_7.2
host.example.org ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLC2PpBnFrbXh2Y
# host.example.org SSH-2.0-OpenSSH_7.2
host.example.org ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9iViojCZkcpdLju7/3+OaxKs/11TAU4SuvIPTvVYvQO32o4K
# host.example.org SSH-2.0-OpenSSH_7.2
host.example.org ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDDOmBOknpyJ61Qnaeq2s+pHOH6rdMn09iREz2A/yO2m
```

Once the key is acquired, its fingerprint can be shown using ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1). This can be done directly with a pipe.

```
$ ssh-keyscan host.example.org | ssh-keygen -lf -
```

```
# host.example.org SSH-2.0-OpenSSH_7.2
# host.example.org SSH-2.0-OpenSSH_7.2
# host.example.org SSH-2.0-OpenSSH_7.2
256 SHA256:sxh5i6KjXZd8c34mVTBfWk6/q5cC6BzR6Qxep5nBMVo host.example.org (ECDSA)
2048 SHA256:hlPei3IXhkZmo+GBLamiiIaWbeGZMqeTXg15R42yCC0 host.example.org (RSA)
256 SHA256:ZmS+IoHh31CmQZ4NJjv3z58Pfa0zMaOgxu8yAcpuwuw host.example.org (ED25519)
```

If there is more than one public key type is available from the server on the port polled, then ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1) will fetch them. If there is more than one key fed via **stdin** or a file, then ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) will process them. Prior to OpenSSH 7.2 manual fingerprinting was a two step process, the key was read to a file and then processed for its fingerprint.

```
$ ssh-keyscan -t ed25519 host.example.org > key.pub
# host.example.org SSH-2.0-OpenSSH_6.8
$ ssh-keygen -lf key.pub
256 SHA256:ZmS+IoHh31CmQZ4NJjv3z58Pfa0zMaOgxu8yAcpuwuw host.example.org (ED25519)
```

Note that some output from ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1) is sent to **stderr** instead of **stdout**.

The hash can be generated manually with awk(1) (http://linux.die.net/man/1/awk), sed(1) (http://linux.die.net/man/1/sed) and xxd(1) (http://linux.die.net/man/1/xxd), on systems where they are found.

```
$ awk '{print $2}' key.pub | base64 -d | md5sum -b | sed 's/../&:/g; s/: .*$//'
$ awk '{print $2}' key.pub | base64 -d | sha256sum -b | sed 's/ .*$//' | xxd -r -p | base64
```

It is possible to find all hosts from a file which have new or different keys from those in **known_hosts**, if the host names are in clear text and not stored as hashes.

```
$ ssh-keyscan -t rsa,dsa -f ssh_hosts | \
  sort -u - ~/.ssh/known_hosts | \
  diff ~/.ssh/known_hosts -
```

ASCII Art Visual Host Key

An ASCII art representation of the key can be supplied along with the SHA256 base64 fingerprint:

```
$ ssh-keygen -lvf key
256 SHA256:BClQBFAGuz55+tgHM1aazI8FUo8eJiwmMcqg2U3UgWU www.example.org (ED25519)
+--[ED25519 256]--+
|o+=*++Eo         |
|+o .+.o.         |
|B=.oo.  .        |
|*B.=.o .         |
|= B *   S        |
|. .@ .           |
| +..B            |
|  *. o           |
| o.o.            |
+----[SHA256]-----+
```

In OpenSSH 6.7 and earlier the fingerprint is in MD5 hexadecimal form.

```
$ ssh-keygen -lvf key
2048 37:af:05:99:e7:fb:86:6c:98:ee:14:a6:30:06:bc:f0 www.example.net (RSA)
+--[ RSA 2048]----+
|          o      |
|         o .     |
```

```
|        o o      |
|       o +       |
|   .  . S        |
|    E ..         |
|  .o.* ..        |
|  .*=.+o         |
|  ..==+.         |
+-----------------+
```

Key-based authentication

OpenSSH can use public key cryptography for authentication. In public key cryptography, encryption and decryption are asymmetric. The keys are used in pairs, a public key to encrypt and a hidden key to decrypt. ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) can make RSA, DSA, Ed25519, or ECDSA keys for authenticating. DSA keys must be exactly 1024 bits in size. RSA keys are allowed to vary from 768 bits on up. Ed25519 keys have a fixed length. ECDSA can be one of 256, 384 or 521 bits. Shorter keys are faster, but less secure. Longer keys are much slower to work with but provide better protection.

The key files can be named anything, so it is possible to have many keys each for different services or tasks. The comment field at the end of the public key can be useful in helping to keep the keys sorted, if you have many of them or use them infrequently.

With key-based authentication, a copy of the public key is stored on the server and the private key is kept on the client. When the client first contacts the server, the server responds by using the client's public key to encrypt a random number and return that encrypted random number as a challenge to the client. The client uses the matching private key to decrypt the challenge and extract the random number. The client then makes an MD5 hash of the session ID and the random number from the challenge and returns that hash to the server. The server then makes its own hash of the session ID and the random number and compares that to the hash returned by the client. If there is a match, the login is allowed. If there is not a match, then the next public key on the server is tried until either a match is found or all the keys have been tried. [2]

If an agent is used on the client side to manage authentication, the process is similar. It only differs in that ssh(1) (http://man.openbsd.org/ssh.1) passes the challenge off to the agent which then calculates the response and passes it back to ssh(1) (http://man.openbsd.org/ssh.1) which then passes the agent's response back to the server.

Basics of Public Key Authentication

For public key authentication, a key pair is needed. ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) is used to make the key pair. From that pair the public key must be properly stored on the remote host and the private key stored safely on the client. The public key is added to the designated **authorized_keys** file for the remote user account. Once the keys have been prepared they can be used again and again without needing to do anything else.

Preparation of the keys:

0) If either the **authorized_keys** file or **.ssh** directory do not exist on either the client machine or the remote machine, create them and set the permissions correctly:

```
$ mkdir ~/.ssh/
$ touch ~/.ssh/authorized_keys
$ chmod 0700 ~/.ssh/
$ chmod 0600 ~/.ssh/authorized_keys
```

1) Create a key pair. The example here creates a 4096-bit RSA key pair in the directory **~/.ssh**. The option

-b determines the key size, the option **-t** assigns the key type, and the option **-f** assigns the key file a name. It is good to give keys descriptive file names, especially if larger numbers are managed. As a result of the line below, the public key will be seen named **mykey_rsa.pub** and and the private key will be called **mykey_rsa**. Be sure to enter a sound passphrase.

```
$ ssh-keygen -b 4096 -t rsa -f ~/.ssh/mykey_rsa
```

Since 6.5 a new private key format is available using a bcrypt KDF to better protect keys at rest. This new format is always used for Ed25519 keys, and sometime in the future will be the default for all keys. But for right now it may be requested when generating or saving existing keys of other types via the **-o** option in ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1).

```
$ ssh-keygen -o -b 4096 -t rsa -f ~/.ssh/mykey_rsa
```

Details of the new format are found in the source code in the file **PROTOCOL.key**.

2) Transfer both the public and private keys to the client machine, if not already there from the step above. It is usually best to store both the public and private keys in the directory **~/.ssh/** together.

3) Transfer only the public key to remote machine.

```
$ scp ~/.ssh/mykey_rsa.pub fred@remotehost.example.org:.
```

4) Log in to the remote machine and add the new public key to the **authorized_keys** file, whether in **~/.ssh /authorized_keys**, the default, or somewhere else as designated by the server's configuration.

```
$ cat mykey_rsa.pub >> ~/.ssh/authorized_keys
$ nano -w ~/.ssh/authorized_keys
```

Any editor that does not wrap long lines can be used. In the example above, nano(1) (http://linux.die.net /man/1/nano) is started with the **-w** option to prevent wrapping of long lines. The same can also be accomplished permanently by editing nanorc(5) (http://linux.die.net/man/5/nanorc) However it is done, the key must be in the file whole and unbroken, on a single line.

Usage of the keys: Authenticate to the remote machine from the client using the private key. The option **-i** tells ssh(1) (http://man.openbsd.org/ssh.1) which private key to try.

```
$ ssh -i ~/.ssh/mykey_rsa -l fred remotehost.example.org
```

➥ **Troubleshooting of key-based authentication**:

If the server refuses to accept the key and fails over to the next authentication method (eg: "Server refused our key"), then there are several possible mistakes to look for on the server side.

A common error, if a key doesn't work, is that the file permissions are wrong. The authorized key file must be owned by the user in question and not be group writable. Nor may the key file's directory be group or world writable.

```
$ chmod u=rwx,g=rx,o= ~/.ssh
$ chmod u=rw,g=,o=  ~/.ssh/authorized_keys
```

Another mistake that can happen is if the key is broken by line breaks or has other white space in the middle. That can be fixed by joining up the lines and removing the spaces or by recopying the key more carefully.

And, though it should go without saying, the halves of the key pair need to match. The public key on the server needs to match the private key held on the client. If the public key is lost, then a new one can be generated with the **-y** option, but not the other way around. If the private key is lost, then the public key should be erased as it is no longer of any use. If many keys are in use for an account, it might be a good idea to add comments to them. On the client, it can be a good idea to know which server the key is for, either through the file name itself or through the comment field. A comment can be added using the **-C** option.

```
$ ssh-keygen -b 4096 -t rsa -f ~/.ssh/mykey_rsa -C "web server mirror"
```

On the server, it can be important to annotate which client they key is from if there is more than one public key there in an account. There the comment can be added to the authorized key file on the server in the last column if a comment does not already exist. Again, the format of the authorized keys file is given in the manual page for sshd(8) (http://man.openbsd.org/sshd.8) in the section "AUTHORIZED_KEYS FILE FORMAT". If the keys are not labeled they can be hard to match, which might or might not be what you want.

Associating Keys Permanently with a Server

The **Host** directive in ssh_config(5) (http://man.openbsd.org/ssh_config.5) can apply specific settings to a target host. That includes using specific keys with certain hosts. So by changing **~/.ssh/config** it is possible to assign keys to be tried automatically whenever making a connection to that host. Here, all that's needed is to type `ssh` *`web1`* to connect with the key for that server.

```
Host web1
        Hostname 198.51.100.32
        IdentitiesOnly=yes
        IdentityFile=/home/fred/.ssh/web_key_rsa
```

Below **~/.ssh/config** uses different keys for *server* versus *server.example.org*, regardless whether they resolve to the same machine. This is possible because the host name argument given to ssh(1) (http://man.openbsd.org/ssh.1) is not converted to a canonicalized host name before matching.

```
Host server
        IdentityFile /home/fred/.ssh/key_a_rsa

Host server.example.org
        IdentityFile /home/fred/.ssh/key_b_rsa
```

The parser is first-match, so the shorter name is tried first. Of course less ambiguous shortcuts can also be made.

Encrypted Home Directories

With encrypted home directories, the keys must be stored in an unencrypted directory and sshd(8) (http://man.openbsd.org/sshd.8) configured appropriately to find the keys in that special location. Sometimes it is also necessary to add a script or call a program from **/etc/ssh/sshrc** to decrypt the directory after authentication.

One symptom of having an encrypted home directory is that key-based authentication only works when you

are already logged into the same account, but fails when trying to make the first connection and log in for the first time.

Here is one method for solving the access problem. Each user is given a subdirectory under **/etc/ssh/keys/** which they can then use for storing their **authorized_keys** file. This is set in the server's configuration file **/etc/ssh/sshd_config**

```
AuthorizedKeysFile      /etc/ssh/keys/%u/authorized_keys
```

Multiple key file locations can be specified if they are separated by whitespace. The user needs read access to the directory and keys, but write access is not needed. That opens up more possibilities as to how the keys can be managed.

Passwordless login

One way of allowing passwordless logins is to follow the steps above, but do not enter a passphrase. Note that this is very risky, so the key files should be very well protected and kept track of. That includes that they only be used as single-purpose keys as described below. Timely key rotation becomes especially important. In general, it is not a good idea to make a key without a passphrase. A better solution is to work with an authentication agent in conjunction with a single-purpose key.

Requiring Both Keys and a Password

While users should have strong passphrases for their keys, there is no way to enforce or verify that. Indeed, the passphrase for any given key never leaves the client machine so it is nothing that the server can have any influence over. Starting with OpenSSH 6.2, it is possible for the server to require multiple authentication methods for login using the **AuthenticationMethods** directive.

```
AuthenticationMethods publickey,password
```

This example requires that users first authenticate using a key and only queries for a password if that succeeds. It is not possible to get to the system password prompt without first authenticating with a valid key. Changing the order of the arguments changes the order of the authentication methods.

Requiring Two or More Keys

Since OpenSSH 6.8, the server now remembers which public keys have been used for authentication and refuses to accept previously-used keys. This allows a set up requiring that users authenticate using two different public keys.

```
AuthenticationMethods publickey,publickey
```

The **AuthenticationMethods** directive, whether for keys or passwords, can also be set on the server under a **Match** directive to apply only to certain groups or situations.

Requiring Certain Key Types

Also since OpenSSH 6.8, the **PubkeyAcceptedKeyTypes** directive can specify that certain key types are accepted. Those not in the comma-separated pattern list are not allowed.

```
PubkeyAcceptedKeyTypes ssh-ed25519*,ssh-rsa*,ecdsa-sha2*
```

The actual key types or a pattern can be in the list. Spaces are not allowed in the pattern list. The exact list of supported key types can be found by the **-Q** option on the client.

```
$ ssh -Q key
```

For host-based authentication, it is the **HostbasedAcceptedKeyTypes** directive that determines which key types are allowed for authentication.

Key-based authentication using an Agent

The authentication agent, ssh-agent(1) (http://man.openbsd.org/ssh-agent.1), should be started at the beginning of the session and used to launch the login session or X-session so that the environment variables are passed to each subsequent shell pointing to the agent process and its unix-domain socket. Many distros do this automatically upon login or startup. Starting the agent sets a pair of environment variables:

- SSH_AGENT_PID : the process id of the agent
- SSH_AUTH_SOCK : the filename and full path to the unix-domain socket

The various SSH and SFTP clients find these variables automatically and use them to contact the agent if they are set and if one or more keys are loaded. If the shell or desktop session was launched using ssh-agent(1) (http://man.openbsd.org/ssh-agent.1), then these variables are already set and available. If not, then it is necessary to set them manually inside each shell or for each application in order to use the agent.

Once the agent is available, a private key only needs to be loaded once and then can be used many times. Private keys are loaded into the agent with ssh-add(1) (http://man.openbsd.org/ssh-add.1).

```
$ ssh-add /home/fred/.ssh/mykey_ed25519
```

It is possible to list the identities currently in the agent. The option **-l** will list all the fingerprints of the identities in the agent.

```
$ ssh-add -l
256 SHA256:77mfUupj364g1WQ+O8NM1ELj0G1QRx/pHtvzvDvDlOk mykey_ed25519 (ED25519)
```

It is also possible to remove individual identities from the agent using **-d** which will remove them one at a time by name, but only if the name is given. Without the name of a private key, it will fail silently. Using **-D** will remove all of them at once without needing to specify any by name.

Agent forwarding

In agent forwarding, intermediate machines forward challenges and responses back and forth between the client and the final destination. This eliminates the need for passwords or keys on any of these intermediate machines. In other words, an advantage of agent forwarding is that the private key itself is not needed on any remote machine, thus hindering unwanted access to it. [3] Another advantage is that the actual agent to which the user has authenticated does not go anywhere and is thus less susceptible to analysis.

The default configuration files for the server enable authentication agent forwarding, so to use it, nothing needs to be done there. On the client side it is disabled by default and so it must be enabled explicitly. Put the following line in ssh_config(5) (http://man.openbsd.org/ssh_config.5) to enable agent forwarding for a

particular server:

```
Host gateway.example.org
        ForwardAgent yes
```

One risk with agents is that they can be re-used to tailgate in if the permissions allow it. Keys cannot be copied this way, but authentication is possible when there are incorrect permissions. Note that disabling agent forwarding does not improve security unless users are also denied shell access, as they can always install their own forwarders.

The risks of agent forwarding can be mitigated by confirming each use of a key by adding the **-c** option when adding the key to the agent. This requires the SSH_ASKPASS variable be set and available to the agent process, but will generate a prompt on the host running the agent upon each use of the key by a remote system.

Single-purpose keys

Tailored single-purpose keys can eliminate use of remote root logins for administrative activities. A finely tailored sudoers is needed along with a user account. When done right, it gives just enough access to get the job done, following the security principle of Least Privilege. Single-purpose keys are accompanied by use of either the **ForceCommand** directive in sshd_config(5) (http://man.openbsd.org/ssh_config.5) or the **Command=** directive inside the **authorized_keys** file. Further, neither the **authorized_keys** file nor the home directory should be writable. `ssh -v` can show exactly what is being passed to the server so that sudoers can be set up correctly when setting up encrypted remote backups using rsync(1) (http://linux.die.net/man/1/rsync), tar(1) (http://man.openbsd.org/tar.1), mysqldump(1) (http://linux.die.net/man/1/mysqldump), etc. For example, here is what `ssh -v` shows from one particular usage of rsync(1) (http://linux.die.net/man/1/rsync):

```
$ rsync --server -vlHogDtprz --delete --delete-after \
        --ignore-errors . /org/backup
```

Key-based authentication, with limitations on activity

The **authorized_keys** file can force a particular program to run whenever the key is used. This is useful for automating various tasks. The following launches firefox automatically and exits the connection when it is finished.

```
command="/usr/bin/firefox" ssh-rsa AAAAB3NzaC1yc2EAAA...OFy5Lwc8Lo+Jk=
```

The following further restricts the connection to coming from a single domain.

```
command="/usr/bin/firefox",from="*.example.net" ssh-rsa AAAAB3...FLoJk=
```

The manual page sshd(8) (http://man.openbsd.org/sshd.8) has the full list of options for the **authorized_keys** file.

The following key will only echo some text and then exit, unless used non-interactively with the **-N** option. No matter what the user tries, it will echo the text. The **-N** option will disable running the remote program, allowing the connection to stay open.

```
command="/bin/echo do-not-send-commands" ssh-rsa AAAAB3...99ptc=
```

This is very useful for keys that are only used for tunnels.

Read-only access to keys

In some cases it is necessary to prevent the users from changing their own authentication keys. This can be done by putting the key file in an external directory where the user has read-only access to the key file and no write permissions to either the file or the directory. The **AuthorizedKeysFile** directive sets where sshd(8) (http://man.openbsd.org/sshd.8) looks for the keys and can point to the secured location for the keys instead of the default location.

The default location for keys on most systems is usually **~/.ssh/authorized_keys**. A good alternate location could be to create the directory **/etc/ssh/authorized_keys** and store the selected users' key files there. The change can be made to apply to only a group of users by putting it under a **Match** directive.

```
Match Group sftpusers
        AuthorizedKeysFile /etc/ssh/authorized_keys/%u
```

This works even within a chrooted environment.

```
Match Group sftpusers
        ChrootDirectory /home
        ForceCommand internal-sftp -d %u
        AuthorizedKeysFile /etc/ssh/authorized_keys/%u
```

Of course it could be set to affect all users by putting the directive in the main part of the server configuration file.

Mark public keys as revoked

Keys that have been revoked can be stored in **/etc/ssh/revoked_keys**, a file specified in sshd_config(5) (http://man.openbsd.org/ssh_config.5) using the directive **RevokedKeys**, so that sshd(8) (http://man.openbsd.org/sshd.8) will prevent attempts to log in with them. No warning or error on the client side will be given if a revoked key is tried. Authentication will simply progress to the next key or method.

The revoked keys file should contain a list of public keys, one per line, that have been revoked and can no longer be used to connect to the server. The key cannot contain any extras, such as login options or it will be ignored. If one of the revoked keys is tried during a login attempt, the server will simply ignore it and move on to the next authentication method. An entry will be made in the logs of the attempt, including the key's fingerprint. See the section on logging for a little more on that.

```
RevokedKeys /etc/ssh/revoked_keys
```

The **RevokedKeys** configuration directive is not set in sshd_config(5) (http://man.openbsd.org/ssh_config.5) by default. It must be set explicitly if it is to be used.

Key Revocation Lists

A compact, binary form of representing revoked keys and certificates is available as a Key Revocation List (KRL). KRLs are generated with ssh-keygen(1) (http://man.openbsd.org/ssh-keygen.1) and can be created

fresh or an existing one updated.

```
$ ssh-keygen -kf  /etc/ssh/revoked_keys -z 1 ~/.ssh/old_key_rsa.pub
$ ssh-keygen -ukf /etc/ssh/revoked_keys -z 2 ~/.ssh/old_key_dsa.pub
```

It is possible to test if a specific key or certificate is in the revocation list.

```
$ ssh-keygen -Qf  /etc/ssh/revoked_keys ~/.ssh/old_key_rsa.pub
```

Only public keys and certificates will be loaded into the KRL. Corrupt or broken keys will not be loaded and will produce an error message if tried. Like with the regular **RevokedKeys** list, the public key destined for the KRL cannot contain any extras like login options or it will produce an error when an attempt is made to load it into the KRL or search the KRL for it.

More on Verifying SSH Keys

Reliable verification of a host key must be done when first connecting. To know the key fingerprint in advance, it can be necessary to contact the system administrator who can provide it out of band. Use **ssh-keygen** on the local console to be absolutely sure.

Here the server's RSA key is read and its fingerprint shown as SHA256 base64:

```
$ ssh-keygen -lf /etc/ssh/ssh_host_rsa_key.pub
2048 SHA256:hlPei3IXhkZmo+GBLamiiIaWbeGZMqeTXg15R42yCC0 root@server.example.net (RSA)
```

And here the corresponding ECDSA key is read, but shown as an MD5 hexadecimal hash:

```
$ ssh-keygen -E md5 -lf /etc/ssh/ssh_host_ecdsa_key.pub
256 MD5:ed:d2:34:b4:93:fd:0e:eb:08:ee:b3:c4:b3:4f:28:e4 root@server.example.net (ECDSA)
```

Prior to 6.8, the fingerprint was expressed as an MD5 hexadecimal hash:

```
$ ssh-keygen -lf /etc/ssh/ssh_host_rsa_key.pub
2048 MD5:e4:a0:f4:19:46:d7:a4:cc:be:ea:9b:65:a7:62:db:2c root@server.example.net (RSA)
```

It is also possible to use ssh-keyscan(1) (http://man.openbsd.org/ssh-keyscan.1) to get keys from an active SSH server. However, the fingerprint still needs to be verified out of band then.

Warning: Remote Host Identification Has Changed!

If the server's key does not match what has been recorded in either the system's or the local user's **authorized_keys** files, then the SSH client will issue a warning. Two main reasons for the warning exist. One is when the server's key has been changed, maybe the server was reinstalled without backing up the old keys. Another situation is when the connection is made to the wrong machine, one such case would be the during an attempted man in the middle attack or when contacting a server that changes addresses because of dynamic address allocation.

In cases where the key has changed there is only one thing to do: contact the system administrator and verify the key. The system administrator may be you yourself in some cases. If so, was OpenSSH-server recently reinstalled, or was the machine restored from an old backup? The new key fingerprint can be sent out by some method where it is possible to verify the integrity and origin of the message, for example via

PGP-signed e-mail.

Then go over to the client machine where you got the error and remove the old key from **~/.ssh /known_hosts**

```
$ ssh-keygen -R host.example.org
```

Then try logging in, but enter the password if and **only** if the key fingerprint matches what you got on the server console. If the key fingerprint matches, then go through with the login process and the key will be automatically added. If the key finger print does not match, stop immediately and it would be a good idea to get on the phone (a real phone, no computer phones) to the remote machine's system administrator or the network administrator.

Multiple keys for a host, multiple hosts for a key in known_hosts

Using pattern matching in known_hosts, it is possible to assign multiple host names or ip addresses to the same key. That goes for both those global keys in **/etc/ssh/ssh_known_hosts** and those local user-specific key lists in **~/.ssh/known_hosts**. They should provide you with an up-to-date set to add to your known hosts file. Otherwise, you must verify the keys by hand.

```
server1,server2,server3 ssh-rsa AAAAB097y0yiblo97gvl...jhvlhjgluibp7y807t08mmniKjug...==
```

You can use globbing to a limited extent in either **/etc/ssh/ssh_known_hosts** or **~/.ssh/known_hosts**.

```
172.19.40.* ssh-rsa AAAAB097y0yiblo97gvl...jhvlhjgluibp7y807t08mmniKjug...==
```

However, to get the effect you want, multiple keys for the same address, make multiple entries in either **/etc/ssh/ssh_known_hosts** or **~/.ssh/known_hosts**, one for each key.

```
server1 ssh-rsa AAAAB097y0yiblo97gvljh...vlhjgluibp7y807t08mmniKjug...==
server1 ssh-rsa AAAAB0liuouibl kuhlhlu...qerf1dcw16twc61c6cw1ryer4t...==
server1 ssh-rsa AAAAB568ijh68uhg63wedx...aq14rdfcvbhu865rfgbvcfrt65...==
```

To get a pool of servers to share a pool of keys, each server-key combination must be added manually to the **known_hosts** file.

```

server1 ssh-rsa AAAAB097y0yiblo97gvljh...07t8mmniKjug...==
server1 ssh-rsa AAAAB0liuouibl kuhlhlu...qerfw1ryer4t...==
server1 ssh-rsa AAAAB568ijh68uhg63wedx...aq14rvcfrt65...==

server2 ssh-rsa AAAAB097y0yiblo97gvljh...07t8mmniKjug...==
server2 ssh-rsa AAAAB0liuouibl kuhlhlu...qerfw1ryer4t...==
server2 ssh-rsa AAAAB568ijh68uhg63wedx...aq14rvcfrt65...==
```

I'm not quite sure how to get that to work with hashed host names.

This will get the rsa key for *server1*, put the rsa key in file (e.g. **z.key**) and give a fingerprint for the key in that file:

```
$ ssh-keyscan -t rsa server1
$ ssh-keyscan -t rsa server1 > z.key
$ ssh-keygen -l -f z.key
```

Another way of Dealing with Dynamic (roaming) IP Addresses

It is possible to manually point to the right key using **HostKeyAlias** either as part of ssh_config(5) (http://man.openbsd.org/ssh_config.5) or as a runtime parameter. Here the key for machine Foobar is used to connect to host 192.168.11.15

```
$ ssh -o CheckIP=no -o StrictHostKeyChecking=no \
      -o HostKeyAlias=foobar   192.168.11.15
```

This is useful when DHCP is not configured to try to keep the same addresses for the same machines over time.

Hostkey Update and Rotation in known_hosts

A protocol extension to rotate weak pubic keys out of **known_hosts** has been in OpenSSH from version 6.8[4] and later. With it the server is able to inform the client of all its host keys and update **known_hosts** with them when at least one trusted key already known. The method still requires the private keys be available to the server [5] so that proofs can be completed. In ssh_config(5) (http://man.openbsd.org /ssh_config.5), the directive **UpdateHostKeys** specifies whether the client should accept updates of additional host keys from the server after authentication is completed and add them to **known_hosts**. A server can offer multiple keys of the same type for a period before removing the deprecated key from those offered, thus allowing an automated option for rotating keys as well as for upfrading from weaker algorithms to stronger ones.

References

1. "The Secure Shell (SSH) Authentication Protocol". IETF. 2006. https://tools.ietf.org /html/rfc4252#section-7. Retrieved 2015-05-06.
2. Steve Friedl (2006-02-22). "An Illustrated Guide to SSH Agent Forwarding". Unixwiz.net. http://www.unixwiz.net/techtips/ssh-agent-forwarding.html#chal. Retrieved 2013-04-27.
3. Daniel Robbins (2002-02-01). "Common threads: OpenSSH key management, Part 3". IBM. http://www.ibm.com/developerworks/library/l-keyc3/. Retrieved 2013-04-27.
4. Damien Miller (2015-02-01). "Key rotation in OpenSSH 6.8+". DJM's Personal Weblog. http://blog.djm.net.au/2015/02/key-rotation-in-openssh-68.html. Retrieved 2016-03-05.
5. Damien Miller (2015-02-17). "Hostkey rotation, redux". DJM's Personal Weblog. http://blog.djm.net.au/2015/02/hostkey-rotation-redux.html. Retrieved 2016-03-05.

Cookbook/Proxies and Jump Hosts

A proxy is an intermediary that forwards requests from clients to other servers. Performance improvement, load balancing, security or access control are some reasons they are used. Some proxies include caching to save bandwidth and increase speed by avoiding going out to the net to keep retrieving the same, unchanged documents.

SOCKS Proxy

It's possible to connect via an intermediate machine using a SOCKS proxy. SOCKS4 and SOCKS5 proxies are currently supported by OpenSSH. SOCKS5[1] allows transparent traversal, by an application, of a firewall or other barrier and can use strong authentication with help of GSS-API.Dynamic application-level port forwarding allows the outgoing port to be allocated on the fly thus creating a proxy at the TCP session level.

Here the web browser can connect to the SOCKS proxy on port 3555 on the local host:

```
$ ssh -D 3555 server.example.org
```

Using ssh(1) (http://man.openbsd.org/ssh.1) as a SOCKS5 proxy, or in any other capacity where forwarding is used, you can specify multiple ports in one action:

```
$ ssh -D 80 -D 8080 -f -C -q -N fred@server.example.org
```

For example in Firefox, you'll also want the DNS requests to go via your proxy, so changing **about:config** needs **network.proxy.socks_remote_dns** set to **true**

It'll be similar for other programs that support SOCKS proxies.

You can tunnel samba over ssh(1) (http://man.openbsd.org/ssh.1), too.

Tunneling SSH Over Tor with Netcat

Instead of using ssh(1) (http://man.openbsd.org/ssh.1) as a SOCKS proxy, it is possible to tunnel the SSH protocol itself over a SOCKS proxy such as Tor (https://www.torproject.org/). Tor is anonymity software and a corresponding network that uses relay hosts to conceal a user's location and network activity. Its architecture is intended to prevent surveillance and traffic analysis. Tor can be used in cases where it is important to conceal the point of origin of the SSH client.

On the end point that the client sees, Tor is a regular SOCKS5 proxy and can be used like any other SOCKS5 proxy. So this is tunneling SSH over a SOCKS proxy. For example, if Tor is installed locally and listening on a port, then SSH can run over Tor [2] using netcat (http://man.openbsd.org/nc.1):

```
$ ssh -o ProxyCommand="nc -X 5 -x localhost:9150 %h %p" server.example.org
```

If the user name on the remote system is different from that on the local system, it is possible to pass along a different user name.

```
$ ssh -o User=fred -o ProxyCommand="nc -X 5 -x localhost:9150 %h %p" server.example.org
```

When attempting a connection like this, it is very important that it does not leak information. In particular, the DNS lookup should occur over Tor and not be done by the client itself. Make sure that if **VerifyHostKeyDNS** is used that it be set to 'no'. The default is 'no' but check to be sure. It can be passed as a run-time argument to remove any doubt or uncertainty.

```
$ ssh -o "VerifyHostKeyDNS=no" -o ProxyCommand="nc -X 5 -x localhost:9150 %h %p" server.example.org
```

Using the netcat-openbsd nc(1) (http://man.openbsd.org/nc.1) package, this seems not to leak any DNS information. Other netcat packages might or might not be the same. It's also not clear if there are other ways

in which this method might leak information. YMMV.

Jump Hosts -- Passing through a gateway or two

It is possible to connect to another host via an intermediary or two so that the client can act as if the connection were direct. **ProxyCommand** works and the utility Netcat fits here, too.

The safest and most straightforward way is to use ssh(1) (http://man.openbsd.org/ssh.1)'s stdio forwarding (**-W**) mode to "bounce" the connection through an intermediate host.

```
$ ssh -o ProxyCommand="ssh -W %h:%p firewall.example.com" server2.example.org
```

This approach is the safest and supports port-forwarding without needed special tricks.

Another trick, usable with clients that don't support the **-W** option, is `ssh -tt`. This forces TTY allocation, so instead of the above you can do the following, connecting to *server2* via *firewall* as the jump host:

```
$ ssh -tt firewall.example.com ssh -tt server2.example.org
```

This opens an ssh terminal to the remote machine. You can also pass commands. For example, to reattach to a remote screen session using screen (http://linux.die.net/man/1/screen) you can do the following:

```
$ ssh -tt firewall.example.com ssh -tt server2.example.org screen -dR
```

The chain can be arbitrarily long and is not limited to just two hosts. The disadvantage of this approach over stdio-forwarding is that your session, any forwarded agent, X11 server or sockets are exposed to the intermediate hosts.

Port Forwarding via an Intermediate Host

Tunneling, also called port forwarding, is when a port on one machine mapped to a connection to a port on another machine. In that way remote services can be accessed as if they were local. Or in the case of reverse port forwarding, vice verse. Forwarding can be done directly from one machine to another or via a machine in the middle.

Below we are setting up a tunnel from the *localhost* to *machine2*, which is behind a firewall, *machine1*. The tunnel will be via *machine1* which is publicly accessible and also has access to *machine2*.

```
$ ssh -L 2222:machine2.example.org:22 machine1.example.org
```

Next connecting to the tunnel will actually connect to the second host, *machine2*.

```
$ ssh -p 2222 remoteuser@localhost
```

That's it.

It is possible to use all the options in this way, such as **-X** for X11 forwarding. Here is an example of running rsync(1) (http://linux.die.net/man/1/rsync) between the two hosts using *machine1* as an intermediary with the above setup.

```
$ rsync -av -e "ssh -p 2222"  /path/to/some/dir/   localhost:/path/to/some/dir/
```

SOCKS proxy via an Intermediate Host

If you want to open a SOCKS proxy via an intermediate host, it is possible:

```
$ ssh -L 8001:localhost:8002 user1@machine1.example.org -t ssh -D 8002 user2@machine2.example.org
```

The client will see a SOCKS proxy on port 8001 on the local host, which is actually a connection to *machine1* and traffic will ultimately enter and leave the net through *machine2*. Port 8001 on the local host connects to port 8002 on *machine1* which is a SOCKS proxy to *machine2*. The port numbers can be chosen to be whatever you needed, but forwarding privileged ports still requires root privileges.

ProxyCommand with Netcat

Another way is to use the **ProxyCommand** configuration directive and netcat (http://man.openbsd.org/nc.1). The utility nc(1) (http://man.openbsd.org/nc.1) is for reading and writing network connections directly. It can be used to pass connections onward to a second machine. In this case, *login* is the final destination reached via the intermediary *jumphost*.

```
$ ssh -o 'ProxyCommand ssh %h nc login.example.edu 22' \
     -o 'HostKeyAlias=login.example.edu' \
     jumphost.example.org
```

Keys and different login names can also be used. Using **ProxyCommand**, ssh(1) (http://man.openbsd.org/ssh.1) will first connect to *jumphost* and then from there to login.example.edu. The **HostKeyAlias** directive is needed to look up the right key for login.example.edu, without it the key for *jumphost* will be tried and that will, of course, fail unless both have the same keys. The account *user2* exists on *jumphost*.

```
$ ssh -o 'ProxyCommand ssh -i key-rsa -l user2 %h nc login.example.edu 22' \
     -o 'HostKeyAlias=login.example.edu' \
     jumphost.example.org
```

It's also possible to make this arrangement more permanent and reduce typing by editing **ssh_config** Here a connection is made to *host2* via *host1*:

```
Host host2.example.org
        ProxyCommand ssh fred@host1.example.org nc %h %p
```

Here a connection is made to *server2* via *server1* using the shortcut name 'jump'.

```
Host jump
        ProxyCommand ssh %h nc server2.example.org 22
        HostKeyAlias server2.example.org
        Hostname server1.example.org
        User fred
```

It can be made more general:

```
Host gateway.example.org
        ProxyCommand none
```

```
Host *.example.org my-private-host
        ProxyCommand ssh myuser@gateway.example.org nc %h %p
```

The same can be done with sftp(1) (http://man.openbsd.org/sftp.1) by passing parameters on to ssh(1) (http://man.openbsd.org/ssh.1). Here is a simple example with sftp(1) (http://man.openbsd.org/sftp.1) where *machine1* is the jump host to connect to *machine2*. The user name is the same for both hosts.

```
$ sftp -o 'ProxyCommand=ssh %h nc machine2.example.edu 22' \
       -o 'HostKeyAlias=machine2.example.edu' \
       fred@machine1.example.edu
```

Here is a more complex example using a key for *server1* but regular password-based login for the SFTP server.

```
$ sftp -o 'ProxyCommand ssh -i /Volumes/Home/fred/.ssh/server1_rsa \
       -l user2 server1.example.edu nc sftp.example.edu 22'   \
       -o 'HostKeyAlias=sftp.example.edu' sftp.example.edu
```

If the user accounts names are different on the two machines, that works, too. Here, 'user2' is the account on the second machine which is the final target. The user 'fred' is the account on the intermediary or jump host.

```
$ ssh -l user2 \
      -o 'ProxyCommand ssh -l fred %h nc machine2.example.org 22' \
      -o 'HostKeyAlias machine2.example.org' \
      machine1.example.org
```

Passing through a gateway using netcat mode

As of OpenSSH 5.4[3], a 'netcat mode' can connect stdio on the client to a single port forwarded on the server. This can also be used to connect using ssh(1) (http://man.openbsd.org/ssh.1), but it needs the **ProxyCommand** option either as a run time parameter or as part of **~/.ssh/config**. However, it no longer needs netcat to be installed on the intermediary machine(s). Here is an example of using it in a run time parameter.

```
$ ssh -o ProxyCommand="ssh -W %h:%p jumphost.example.org" server.example.org
```

In that example, authentication will happen twice, first on the jump host and then on the final host where it will bring up a shell.

The syntax is the same if the gateway is identified in the configuration file. ssh(1) (http://man.openbsd.org/ssh.1) expands the full name of the gateway and the destination from the configuration file. The following allows the destination host to be reached by entering `ssh` *`server`* in the terminal.

```
Host server
        Hostname server.example.org
        ProxyCommand ssh jumphost.example.org -W %h:%p
```

The same can be done for SFTP. Here the destination SFTP server can be reached by entering `sftp` *`sftpserver`* and the configuration file takes care of the rest. If there is a mix up with the final host key, then it is necessary to add in **HostKeyAlias** to explicitly name which key will be used to identify the destination system.

```
Host sftpserver
```

```
        HostName sftpserver.example.org
        HostKeyAlias sftpserver.example.org
        ProxyCommand ssh jumphost.example.org -W %h:%p
```

It is possible to add the key for the gateway to the ssh-agent which you have running or else specify it in the configuration file. The option **User** refers to the user name on the destination. If the user is the same on both the destination and the originating machine, then it does not need to be used. If the user name is different on the gateway, then the **-l** option can be used in the **ProxyCommand** option. Here, the user 'fred' on the local machine, logs into the gateway as 'fred2' and into the destination server as 'fred3'.

```
Host server
        HostName server.example.org
        User fred3
        ProxyCommand ssh -l fred2 -i /home/fred/.ssh/rsa_key jumphost.example.org -W %h:%p
```

If both the gateway and destination are using keys, then the option **IdentityFile** in the config is used to point to the gateway's private key and the option **IdentityFile** specified on the commandline points at the destination's private key.

```
Host jump
        HostName server.example.org
        IdentityFile /home/fred/.ssh/rsa_key_2
        ProxyCommand ssh -i /home/fred/.ssh/rsa_key jumphost.example.org -W %h:%p
```

The old way prior to OpenSSH 5.4 used netcat, **nc(1)**.

```
Host server
        Hostname server.example.org
        ProxyCommand ssh jumphost.example.org nc %h %p
```

But that should not be used anymore and the netcat mode, provided by **-W**, should be used instead. The new way does not require netcat at all on any of the machines.

Recursively chaining gateways

If the route always has the same hosts in the same order, then a straight forward chain can be put in the configuration file. Here three hosts are chained with the destination being given the shortcut *machine3*.

```
Host machine1
        Hostname server.example.org
        User fred
        IdentityFile /home/fred/.ssh/machine1_e25519
        Port 2222

Host machine2
        Hostname 192.168.15.21
        User fred
        IdentityFile /home/fred/.ssh/machine2_e25519
        Port 2222
        ProxyCommand ssh -W %h:%p machine1

Host machine3
        Hostname 10.42.0.144
        User fred
        IdentityFile /home/fred/.ssh/machine3_e25519
        Port 2222
        ProxyCommand ssh -W %h:%p machine2
```

Thus any machine in the chain can be reached with a single line. For example, the final machine can be reached with `ssh` *`machine3`* and worked with as normal. This includes port forwarding and any other

capabilities.

Only **hostname** and, for second and subsequent hosts, **ProxyCommand** are needed for each **Host** directive. If keys are not used, then **IdentityFile** is not needed. If the user is the same for all hosts, the that can be skipped. And if the port is the default, then **Port** can be skipped. If using many keys in the agent at the same time and the error "too many authentication" pops up on the client end, it might be necessary to add **IdentitiesOnly yes** to each host's configuation.

Recursively chaining an arbitrary number of hosts

It is possible to make the configuration more abstract and allow passing through an arbitrary number of gateways. You can set the user name with **-l** thanks to the **%r@**, but that user name will be used for all host that you connect to or through. There are limitations resulbting from using the slash as a separator, as there would be with other symbols. However, it allows use of dirname(1) (http://man.openbsd.org/dirname.1) and basename(1) (http://man.openbsd.org/basename.1) to process the host names.

```
Host */*
        ProxyCommand ssh %r@$(dirname %h) -W $(basename %h):%p
```

In this way hosts are separated with a slash (/) and can be arbitrary in number[4].

```
$ ssh host1/host2/host3/host4
```

If keys are to be used then load them into an agent, then the client figures them out automatically if agent forwarding with the **-A** option is used. However, agent forwarding is considered by many to be a security flaw and a general misfeature. However, because the default for **MaxAuthTries** on the server is 6, using keys normally in an agent will limit the number of keys or hops to 6, with server-side logging getting triggered after half that.

The following configuration uses sed(1) (http://man.openbsd.org/sed.1) to allow different port numbers and user names using the plus sign (+) as the delimiter for hosts, a colon (:) for ports, and an percentage sign (%) for user names. The basic structure is *`ssh -W $() $()`* and where **%h** is substituted for the target host name.

```
Host *+*
        ProxyCommand ssh -W $(echo %h | sed 's/^.*+//;s/^\([^:]*$\)/\1:22/') $(echo %h | sed 's/+[^+]*$/
```

The port can be left off for the default of 22 or delimited with a colon (:) for non-standard values[5].

```
$ ssh host1+host2:2022+host3:2224
```

As-is, the colons confound sftp(1) (http://man.openbsd.org/sftp.1), so the above configuration will only work with it using standard ports. If sftp(1) (http://man.openbsd.org/sftp.1) is needed on non-standard ports then another delimiter, such as an underscore (_), can be configured.

Any user name except the final one can be specified for a given host using the designated delimiter, in the above it is a percentage sign (%). The destination host's user name is specified with **-l** and all others can be joined to their corresponding host name with the delimiter.

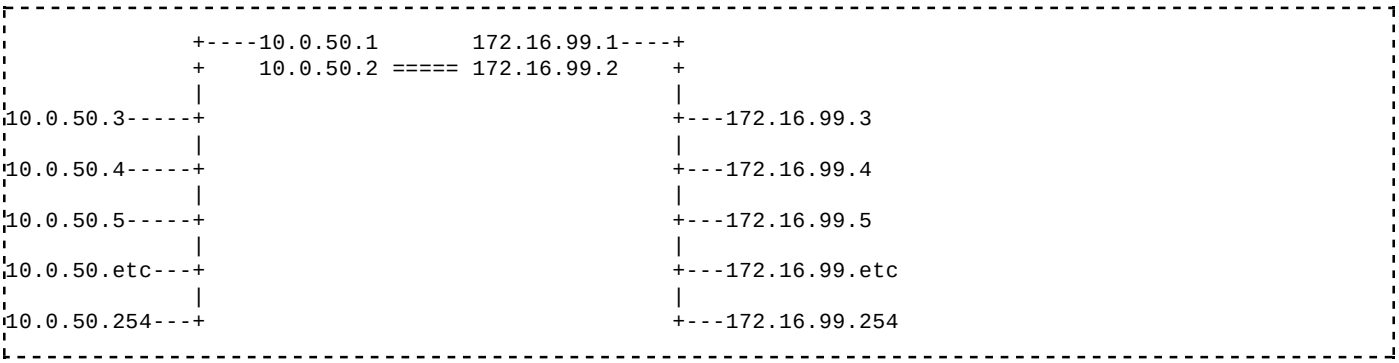
```
$ ssh -l user3 user1%host1+user2%host2+host3
```

If user names are specified, depending on the delimiter, ssh(1) (http://man.openbsd.org/ssh.1) can be unable to match the final host to an IP number and the key fingerprint in *known_hosts*. In such cases, it will ask for verification each time the connection is established, but this should not be a problem if either the equal sign (=) or percentage sign (%) is used.

Passing through a gateway with an ad hoc VPN

Two subnets can be connected over SSH by configuring the network routing on the end points to use the tunnel. The result is a VPN. A drawback is that root access is needed on both hosts, or at least sudo(8) (http://linux.die.net/man/8/sudo) access to ifconfig(8) (http://man.openbsd.org/ifconfig.8) and route(8) (http://man.openbsd.org/route.8). Note, there are very few instances where use of a VPN is legitimately called for, not because VPNs are illegal (quite they opposite, indeed data protection laws in many countries make them absolutely compulsory to protect content in transit) but simply because OpenSSH is usually flexible enough to complete most routine sysadmin and operational tasks using normal SSH methods as and when required. This SSH ad-hoc VPN method is therefore needed only very rarely.

Take this example with two networks. One network has the address range 10.0.50.1 through 10.0.50.254. The other has the address range 172.16.99.1 through 172.16.99.254. Each has a machine, 10.0.50.1 and 172.16.99.1 respectively, that will function as a gateway. Local machine numbering starts with 3 because 2 will be used for the tunnel interfaces on each LAN.



First a tun device is created on each machine, a virtual network device for point-to-point IP tunneling. Then the tun interfaces on these two gateways are then connected by an SSH tunnel. Each tun interface is assigned an IP address.

The tunnel connects machines 10.0.50.1 and 172.16.99.1 to each other, and each are already connected to their own local area network (LAN). Here is a VPN with the client as 10.0.50.0/24, remote as 172.16.99.0/24. First, set on the client:

```
$ ssh -f -w 0:1 192.0.2.15 true
$ ifconfig tun0 10.1.1.1 10.1.1.2 netmask 255.255.255.252
$ route add 172.16.99.0/24 10.1.1.2
```

On the server:

```
$ ifconfig tun1 10.1.1.2 10.1.1.1 netmask 255.255.255.252
$ route add 10.0.50.0/24 10.1.1.1
```

References

1. "SOCKS Protocol version 5". IETF. http://tools.ietf.org/html/rfc1928. Retrieved 2011-02-17.

2. "SSH Over Tor". The Tor Project. 2012-08-28. https://trac.torproject.org/projects/tor/wiki/doc/TorifyHOWTO/ssh. Retrieved 2013-05-04.
3. "OpenSSH 5.4 Release Notes". OpenSSH. 2010-04-07. http://www.openssh.com/txt/release-5.4. Retrieved 2013-04-19.
4. Josh Hoblitt (2011-09-03). "Recursively chaining SSH ProxyCommand". [Read This Fine Material] from Joshua Hoblitt. https://joshua.hoblitt.com/rtfm/2011/09/recursively_chaining_ssh_proxycommand/. Retrieved 2014-02-14.
5. Mike Hommey (2016-02-08). "SSH through jump hosts, revisited". glandium. https://glandium.org/blog/?p=3631. Retrieved 2016-02-09.