Security measures in OpenSSH

Damien Miller
OpenSSH Project
djm@openssh.com

ABSTRACT

This paper examines several security measures that have been implemented in OpenSSH. OpenSSH's popularity, and the necessity for the server to wield root privileges, have made it a high-value target for attack. Despite initial and ongoing code audits, OpenSSH has suffered from a number of security vulnerabilities over its 7.5 year life. This has prompted the developers to implement several defensive measures, intended to reduce both the likelihood of exploitable errors and the consequences of exploitation should they occur.

This paper examines these defensive measures; each measure is described and assessed for implementation effort, attack surface reduction, effectiveness in preventing or mitigating attacks, applicability to other network software and possible improvements.

General Terms

Security

Keywords

SSH, OpenSSH, Security, Attack Surface, Network Applications

1 Introduction

OpenSSH [22] is a popular implementation of the SSH protocol [32]. It is a network application that supports remote login, command execution, file transfer and forwarding of TCP connections between a client and server. It is designed to be safely used over untrusted networks and includes cryptographic authentication, confidentiality and integrity protection.

Since its release in 1999, OpenSSH quickly gained popularity and rapidly became the most popular SSH implementation on the Internet [23]. Today it is installed by default on almost all modern Unix and Unix-like operating systems, as well as many network appliances and embedded devices.

OpenSSH is has been developed to run on Unix-like operating systems and must operate within the traditional Unix security model. Notably, the OpenSSH server, *sshd*, requires root privileges to authenticate users, access the host private key, allocate TTYs and write records of logins. OpenSSH is also based on a legacy code-base, that of ssh-1.2.16 [33]

OpenSSH's popularity, and the knowledge that a successful compromise gives an attacker a chance to gain super-user privileges on their victim's host has made it an attractive target for both research and attack. Since its initial release in 1999, a number of security bugs have been found in OpenSSH. Furthermore some of the libraries that OpenSSH depends on have suffered from bugs that were exposed through OpenSSH's use of them.

Some of these errors have been found despite OpenSSH being manually audited on several occasions. This, and the occurrence of vulnerabilities in dependant libraries, have caused the developers to implement a number of proactive measures to reduce the likelihood of exploitable errors, make the attacker's work more difficult and to limit the consequences of a successful exploit. These measures include replacement of unsafe APIs, avoidance of complex or error-prone code in dependant libraries, privilege separation of the server, protocol changes to eliminate pre-authentication complexity and a mechanism to maximise the benefit of OS-provided attack mitigation measures.

A key consideration in implementing these measures has been their effect on reducing OpenSSH's *attack surface*. Attack surface [17] is a qualitative measure of an application's "attackability" based on the amount of application code exposed to an attacker. This quantity is scaled by the ease with which an attacker can exercise the code – for example, code exposed to unauthenticated users would be weighted higher than that accessible only by authenticated users. A further weighting is given to code that holds privilege during its execution, as an attacker is likely to inherit this privilege in the event of a successful compromise. Attack surface may therefore be considered as a measure of how well developers have applied Saltzer and Schroeder's *Economy of Mechanism* and *Least Privilege* design principles [7].

This paper examines these security measures in OpenSSH's server daemon, *sshd*. Each measure is considered for implementation ease, applicability to other network applications, attack surface reduction, actual attacks prevented and possible improvements.

2 Critical vulnerabilities

Table 1 lists and characterises several critical vulnerabilities found in OpenSSH since 1999. We consider a vulnerability *critical* if it has a moderate to high likelihood of successful remote exploitation.

File	*Problem*	*Found*
session.c	sanitisation error	Friedl, 2000 [15]
deattack.c	integer overflow	Zalewski, 2001 [18]
radix.c	stack overflow	Fodor, 2002 [11]
channels.c	array overflow	Pol, 2002 [8]
auth2-chall.c	array overflow	Dowd, 2002 [12]
buffer.c	integer overflow	Solar Designer, 2003 [13]
auth-chall.c	logic error	OUSPG, 2003 [21]

Table 1: Critical vulnerabilities in OpenSSH

OpenSSH has also been susceptible to bugs in libraries it depends on. Over the same period, zlib [9] and OpenSSL [24] have suffered from a number of vulnerabilities that could be exploited through OpenSSH's use of them. These include heap corruption [14] and buffer overflows [27] [16] in zlib, and multiple overflows in the OpenSSL ASN.1 parser [20].

Note that many of these vulnerabilities stem from memory management errors. It follows that measures that reduce the likelihood of memory management problems occurring, or that make their exploitation more difficult are likely to yield a security benefit.

3 OpenSSH security measures

OpenSSH will naturally have a raised attack surface because of its need to accept connections from unauthenticated users, while retaining the *root* privileges it needs to record login and logout events, open TTY devices and authenticate users.

The approaches used to reduce this attack surface or otherwise frustrate attacks generally fall into the following categories: defensive programming, avoiding complexity in dependant libraries, privilege separation and better use of operating system attack mitigation measures.

3.1 Defensive programming

Defensive programming seeks to prevent errors through the insertion of additional checks [29]. An expansive interpretation of this approach should also include avoidance or replacement of APIs that are ambiguous or difficult to use correctly. In OpenSSH's case, this has included replacement of unsafe string manipulation functions with the safer `strlcpy` and `strlcat` [30] and the replacement of the traditional Unix `setuid` with the less ambiguous [2] `setresuid` family of calls.

A source of of potential errors may be traced to POSIX's tendency to overload return codes; using `-1` to indicate an error condition, but zero for success and positive values as a result indicator (a good example of this is the `read` system call). This practice leads to a natural mixing of unsigned and signed quantities, often when dealing with I/O. Integer wrapping and signed-vs-unsigned integer confusion have caused a number of OpenSSH security bugs, so this is of some concern. OpenSSH performs most I/O calls through a "retry on interrupt" function, `atomicio`. This function was modified to always return an unsigned quantity and to instead report its error via `errno`. Making this API change did not uncover any bugs, but reducing the need to use signed types it made it easier to enable the compiler's signed/unsigned comparison warnings and fix all of the issues that it reported.

Integer overflow errors are often found in dynamic array code. A common C language idiom is to allocate an array using `malloc` or `calloc`, but attacker-controlled arguments to these functions may wrap past the maximum expressible `size_t`, resulting in an exploitable condition [1]. `malloc` and array resizing using `realloc` are especially prone to this, as their argument is often a product, e.g. `array = malloc(n * sizeof(*array))`.

OpenSSH replaced all array allocations with an error-checking `calloc` variant (derived from the OpenBSD implementation) that takes as arguments a number of elements to allocate and a per-element size in bytes. These functions check that the product of these quantities does not overflow before performing an allocation. The `realloc` function, which has no calloc-like counterpart in the standard library (i.e. accepting arguments representing a number of elements and an element size), was replaced with a calloc-like error-checking array reallocator. Implementing this change added only 17 lines of code to OpenSSH, but has not yet uncovered any previously-exploitable overflows. A similar change was subsequently made to many other programs in the

OpenBSD source tree.

Once valid criticism of API replacements is that they make a program more difficult to read by an new developer, as they must frequently recurse into unfamiliar APIs. In OpenSSH's case, effort has been made to use standardised APIs as replacements wherever possible as well as using logical and consistent naming for non-standard replacements (e.g. `calloc` → `xcalloc`)

3.2 Avoiding complexity in dependant libraries

Significant complexity, and thus attack surface, can lurk behind simple library calls. If there is sufficient risk, it may be worthwhile to replace them with more simple, or limited versions. Replacing important API calls is not without risk or cost; it represents additional development and maintenance work and it provides the opportunities for new errors to be made in critical code-paths. If replacement is considered to risky, simply avoiding the call may still be an option – OpenSSH avoids the use of regular expression libraries for this reason.

An example of this approach is OpenSSH's replacement of RSA and DSA cryptographic signature verification code. Prior to late 2002, OpenSSH used the OpenSSL `RSA_verify` and `DSA_verify` functions to verify signatures for user- and host-based public-key authentication. The OpenSSL implementations use a general ASN.1 parser to unpack the decrypted signature object. This adds substantial complexity – in the case of `RSA_verify` at least 282 lines of code, not including calls to the raw OpenSSL cryptographic primitives, or its custom memory allocation, error handling and binary I/O functions.

These calls were replaced with minimal implementations that avoided generic ASN.1 parsing in favour of a simple comparison of the structure of the decrypted signature to an expected form. The replacement `openssh_RSA_verify` function was implemented in 63 lines of code, of much simpler structure (basically decrypt then compare) and with no calls to complex subroutines other than the necessary cryptographic operations.

The replacement functions clearly reduce the attack surface of public key authentication in OpenSSH and have avoided a number of critical bugs in the OpenSSL implementations, notably an overflow in the ASN.1 parsing [20] and a signature forgery bug [3], both of which were demonstrated to be remotely exploitable.

3.3 Protocol changes to reduce attack surface

The SSH protocol includes a compression facility that is intended to improve throughput over low-bandwidth links. Compression is negotiated during the initial key exchange phase of the protocol and activated, along with encryption and message authentication, as soon as the key exchange has finished. The next phase of the protocol is user authentication, but by this time compression is already enabled and any bugs in the underlying zlib code have been exposed to an unauthenticated attacker.

OpenSSH introduced a new compression method *zlib@openssh.com* [5] as a protocol extension (the SSH protocol has a nice extension mechanism that allows the use of arbitrary extension method names under the developer's domain name – unadorned names are reserved for standardised protocol methods). The zlib@openssh.com compression method uses exactly the same underlying compression algorithm (zlib's *deflate*), it merely delays its activation until successful completion of user authentication. This eliminates all zlib exposure to unauthenticated users.

An alternate solution to this problem that does not a require protocol change is to refuse compression in the initial key exchange proposal, but then offer it in a re-exchange immediately after user authentication has completed. This approach was rejected, as key exchange is a heavyweight operation in the SSH protocol; usually consisting of a Diffie-Hellman [4] key agreement with a large modulus. Performing a re-exchange to effectively flip a bit was considered too expensive.

The benefit of delayed compression is clear, despite there not having been any zlib vulnerabilities published since it was implemented. Network application developers considering making non-standard protocol changes to reduce attack surface should consider interoperability carefully, especially if the protocol they are implementing lacks a orthogonal extension mechanism like SSH's.

3.4 Privilege separation

[*Privilege separation in OpenSSH is described in detail in [25], this is a brief summary*].

The design principle of *Least Privilege* [7] requires that privilege be relinquished as soon as it is no longer required, but what should application developers do in cases where privilege is required sporadically through an applications life? sshd is such an application; it must retain root privileges after the user has authenticated and logged in as it needs to record login and logout records and allocate TTYs. Furthermore the SSH protocol allows multiple sessions over a single SSH transport and these sessions may be started any time after user authentication is complete.

OpenSSH 3.3 implemented *privilege separation* (a.k.a *privsep*), where the daemon is split into a privileged *monitor* and an unprivileged *slave* process. Before authentication (*pre-auth*), the slave runs as a unique, nonprivileged user. After authentication (*post-auth*) the slave runs with the privileges of the authenticated user. In all cases, the slave process is jailed (via the `chroot` system call) into an empty directory, typically `/var/empty`.

The slave is responsible for the SSH transport, including cryptography, packet parsing and managing open "channels" (login sessions, forwarded TCP ports, etc.). When the slave needs to perform an action that requires privilege, or any interaction with the wider system, it messages the monitor, which performs the requested action and returns the results.

The monitor is structured as a state-machine, enforcing constraints over which actions a slave may request at its stage in the protocol (e.g. opening login sessions before user authentication is complete is not permitted). The monitor is intended to be as small (code-wise) as possible; the initial implementation removed privilege from just over two thirds of the OpenSSH application [25].

OpenSSH's privsep implementation is complicated somewhat by the need to offer compression before authentication. Once user authentication is complete, the pre-auth slave must serialise and export its connection state for use by the post-auth slave, including cryptographic keys, initialisation vectors (IVs), I/O buffers and compression state. Unfortunately the zlib library offers no functions to serialise compression state. However it does support allocation hooks that it will use instead of the standard `malloc` and `free` functions. OpenSSH's privsep includes a memory manager that is used by zlib. This manager uses anonymous memory mappings that are shared between the pre-auth slave and the monitor. The post-auth slave inherits this memory from the monitor when it is started. Since the monitor treats these allocations as completely opaque and never invokes zlib functions, there is no risk of monitor compromise through deliberately corrupted zlib state.

The OpenSSH privsep implementation builds the monitor and both the pre- and post-authentication slaves into the one executable. This may be contrasted with the Postfix MTA [31], which uses separate cooperating executables that run at various privilege levels. The OpenSSH implementation could probably be simplified if it the monitor were split into a dedicated executable that in turn executed separate slave executables. An additional benefit to this model would be the slaves would no longer automatically inherit the same address space layout as the monitor (further discussed in section 3.5), but it would carry some cost: it would no longer be possible to disable privsep, and it would be impossible to support the standard compression mode though the above shared memory allocator – zlib would have to be modified to allow state export, or pre-authentication compression would have to be abandoned.

Another criticism [19] of OpenSSH's privsep implementation is that it uses the same buffer API as the slave to marshal and unmarshal its messages. This renders the monitor susceptible to the same bugs as the slave if they occur in this buffer code (one such bug has already occurred [13]). However, the alternative of reimplementing the buffer API for the monitor is not very attractive either; maintaining two functionally identical buffer implementation raises the attack surface for questionable benefit. A better approach may be to automatically generate the marshalling code (discussed further in section 4).

Privilege separation in OpenSSH has been a great success; it has reduced the severity of all but one of the memory management bugs found in OpenSSH since its implementation, and the second layer of checking that the privsep monitor state machine offers has prevented at least one logic error in the slave from being exploited. It has suffered from only one known bug, that was not exploitable without first having compromised the slave process. OpenSSH is something of a worst-case in terms of the complexity required to implement privilege separation, other network applications seeking to implement it will likely find it substantially easier.

3.5 Assisting OS-level attack mitigation

Modern operating systems are beginning to implement attack mitigation measures [28] intended to reduce the probability that a given attack will succeed. These measures include stack protection as well as a suite of *runtime randomisations* that are applied to stack gaps, executable and library load addresses, as well as to the addresses returned by memory allocation functions. Collectively, these randomisations (often referred to as Address Space Layout Randomisation – ASLR) render useless any exploits that use fixed offsets or return addresses.

These randomisations are typically applied *per-execution*, as it would be very difficult to otherwise re-randomise an application's address space at runtime. Stack protectors such as SSP/Propolice [6] also employ random "canaries" that are initialised per-execution.

A typical Unix daemon that forks a subprocess to service each request will inherit the address space layout of its parent. In the absence of other mitigation measures, an attacker may therefore perform an *exhaustive search*, trying every possible offset or return address until they find one that works. They will be guaranteed success, as there is a finite and unchanging space of possible addresses (as little as 16-bits in some implementations [26]).

To improve this situation, OpenSSH implemented self-re-execution. sshd was modified to fork, then execute itself to service each connection rather than simply forking. Each daemon instance serving a connection is therefore re-randomised, approximately doubling the effort required to guess a correct offset and removing any absolute guarantee of success.

This change, while straightforward to implement, does incur some additional overhead for each connection, and has been criticised as offering little benefit on systems that do not support any ASLR-like measures.

4 Future directions

There are several opportunities to further improve OpenSSH's security and attack resistance. Perhaps the most simple is to disable or deprecate unsafe or seldom-used protocol elements. Removing support for pre-authentication compression (once a delayed compression method is standardised) would permanently remove complexity and significantly simplify the privilege separation implementation. Likewise, deactivating and ultimately removing support for the legacy SSH protocol version 1 would remove a lot of complexity from the code (and may hasten the demise of a protocol with known weaknesses).

Further attack resistance may be gained by including measures to frustrate *return-to-executable* attacks, where the attacker sets up a stack frame with controlled arguments and then returns to a useful point inside the currently executing process. In OpenSSH's case, they may select the `do_exec` function, that is responsible for spawning a subshell as part of session creation. These attacks may be made more difficult by pervasively inserting authentication checks into code that has the potential to be subverted. However these checks have the potential to bloat and obfuscate the code and their effectiveness at preventing this attack is not entirely clear.

Improved attack resistance could also be achieved by having the listener sshd process check the exit status of the privsep monitor and (indirectly) slave processes. If an abnormal exit status is detected, such as a forced termination for a segmentation violation, then the listener could take remedial action such as rate-limiting similar connections as defined by a (source address, username) tuple. This would work especially well on operating systems that support ASLR – exploits will be nondeterministic on these platforms, and an attacker will be forced to make many connections to find working offsets. Attacks may be rendered infeasible or unattractive by limiting the rate at which these attempts can be made. This concept could be extended to form the basis of a denial of service mitigation, where the listener could impose a rate-limit on connections from hosts that repeatedly fail to authenticate within the login grace period, or that experience frequent login failures.

Another potential approach to reducing errors is to generate mechanical parts of the source automatically. Packet parsers are an excellent candidate for automatic generation, and this technique is used by many RPC implementations and at least one SSH implementation already [10]. The channel and privilege separation state-machines could also be represented at a higher level, allowing easier verification.

5 Conclusion

Network software that accepts data from unauthenticated users while requiring privilege to operate presents a significant security challenge to the application developer. This paper has described the OpenSSH project's approach to this problem, has detailed a number of specific measures that have been implemented and explored areas of possible future work.

These measures focus on reducing the attack surface of the application and making better use of any attack mitigation facilities provided by the underlying operating system. They have been shown to be effective in stopping exploitable problems occurring or in reducing their impact when they do occur. Finally, these measures have been shown to be relatively easy to implement and widely applicable to other network software

References

[1] blexim. Basic Integer Overflows. *Phrack, Vol 11, Number 60, Build 3*, December 2002.

[2] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[3] Daniel Bleichenbacher. OpenSSL PKCS Padding RSA Signature Forgery Vulnerability. `www.securityfocus.com/bid/19849`, September 2005.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, 1976.

[5] Markus Friedl and Damien Miller. Delayed compression for the SSH Transport Layer Protocol, 2006. INTERNET-DRAFT draft-miller-secsh-compression-delayed-00.txt.

[6] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `www.trl.ibm.com/projects/security/ssp/`, August 2005.

[7] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 63, number 9*, pages 1278–1308, September 1975.

[8] Joost Pol. OpenSSH Channel Code Off-By-One Vulnerability. `online.securityfocus.com/bid/4241`, March 2002.

[9] Jean loup Gailly and Mark Adler. zlib. `www.zlib.net`.

[10] Anil Madhavapeddy and David Scott. On the Challenge of Delivering High-Performance, Dependable, Model-Checked Internet Servers. In *In the proceedings of the First Workshop on Hot Topics in System Dependability (HotDep)*, pages 37–42, June 2005.

[11] Marcell Fodor. OpenSSH Kerberos 4 TGT/AFS Token Buffer Overflow Vulnerability. `online.securityfocus.com/bid/4560`, April 2002.

[12] Mark Dowd. OpenSSH Challenge-Response Buffer Overflow Vulnerabilities. `online.securityfocus.com/bid/5093`, June 2002.

[13] Mark Dowd and Solar Designer and the OpenSSH team. OpenSSH Buffer Management Vulnerabilities. `online.securityfocus.com/bid/8628`, September 2003.

[14] Mark J. Cox and Matthias Clasen and Owen Taylor. ZLib Compression Library Heap Corruption Vulnerability. `www.securityfocus.com/bid/4267`, March 2002.

[15] Markus Friedl. OpenSSH UseLogin Vulnerability. `online.securityfocus.com/bid/1334`, June 2000.

[16] Markus Oberhumer. Zlib Compression Library Decompression Buffer Overflow Vulnerability. `online.securityfocus.com/bid/14340`, July 2005.

[17] Michael Howard. Fending Off Future Attacks by Reducing Attack Surface. `msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp`, February 2003.

[18] Michal Zalewski. SSH CRC-32 Compensation Attack Detector Vulnerability. `online.securityfocus.com/bid/2347`, February 2001.

[19] Sun Microsystems. Sun's Alternative "Privilege Separation" for OpenSSH. `src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/ssh/README.altprivsep`, 2006.

[20] NISCC and Stephen Henson. OpenSSL ASN.1 Parsing Vulnerabilities. `www.securityfocus.com/bid/8732`, September 2003.

[21] OUSPG. Logic error in PAM authentication code. Private email, September 2003.

[22] The OpenSSH Project. OpenSSH. `www.openssh.org`.

[23] The OpenSSH Project. SSH usage profiling. `www.openssh.org/usage`.

[24] The OpenSSL Project. OpenSSL. `www.openssl.org`.

[25] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, August 2003.

[26] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[27] Tavis Ormandy. Zlib Compression Library Buffer Overflow Vulnerability. `online.securityfocus.com/bid/14162`, July 2005.

[28] Theo de Raadt. Exploit Mitigation Techniques. `www.openbsd.org/papers/ven05-deraadt`, November 2005.

[29] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. Technical report, August 1976.

[30] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference. Monterey, CA*, pages 175–178, June 1999.

[31] Wieste Venema. Postfix MTA. `www.postfix.org`.

[32] T. Ylönen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[33] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, July 1996.