Matt Jibson
ECE 501
Research Project

OpenBSD's zlib and OpenSSH Vulnerabilities

Abstract

This paper describes two case studies from the OpenBSD operating system. The first case study is an example of how the OpenBSD principles prevented failures. The second is an example of a failure occurring, and an investigation into how it was handled. Throughout, systems engineering principles will describe the success and failure, and how each occurred.

Introduction

Operating systems are among the class of systems that are continuously attacked and at high risk to compromise. Since they are composed purely of software in the millions of lines of code [1] [2], it is a surety that there are bugs (or sometimes called defects) that are, in general, related to the number of lines of code [3]. One measure of the goodness of an operating system is the number of remote exploits found in it, since all operating systems have these exploits. As a case study, looking at OpenBSD presents examples of success where others failed, and its own unique failures. We will look at one example of each.

As background, we start with a primer on remote exploits. A remote exploit is a defect in software that can be accessed and exploited from another machine to execute arbitrary programs on the compromised system. That is, if a machine is connected to a network, like the Internet, an attacker may be able to take control of the machine by exploiting a software defect in the operating system. A common class of defect is a buffer overflow. A buffer overflow occurs when a program stores data beyond the end of a buffer (region of memory), causing the data in that location to change. An appropriately crafted overflow can cause execution of that program to then change, possibly giving control to the attacker. For example, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information

Services [5]. Operating system distributors use many methods to prevent these types of attacks and exploits.

Assuming introductory knowledge as presented in my previous paper about OpenBSD, more details regarding their process are warranted. Their “aspiration is to be [number one] in the industry for security.” As part of their solution for this, they “continue to search for and fix new security holes. . . . The process we follow to increase security is simply a comprehensive file-by-file analysis of every critical software component. We are not so much looking for security holes, as we are looking for basic software bugs, and if years later someone discovers the problem used to be a security issue, and we fixed it because it was just a bug, well, all the better. Flaws have been found in just about every area of the system. Entire new classes of security problems have been found during our audit, and often source code which had been audited earlier needs re-auditing with these new flaws in mind. Code often gets audited multiple times, and by multiple people with different auditing skills.” They have invented or reused and improved many other technologies: strlcpy() and strlcat() (safer string handling), memory protection (like W^X, .rodata segment, guard pages, randomized malloc() (which is the way a program requests memory) and mmap(), atexit() and stdio protection, all of which guard against buffer overflows and other attacks), privilege separation and revocation, chroot jailing, new uids, ProPolice, and others. This vast array of techniques has, as will be shown, provably prevented possible remote exploits before they were discovered. [4]

Main Body

OpenBSD’s practices have prevented many exploits, yet still some have occurred. Now we will look at a case of each, and describe the systems engineering principles that were used, resulting in the prevention of exploit, even without fixing the exploit.

Exploit Prevention

OpenBSD claims on their website that they have had "[o]nly two remote holes in the default install, in more than 10 years" [6]. When discussing their auditing process, they claim that "someone on BUGTRAQ would report that other operating systems were vulnerable to a 'newly discovered problem', and then it would be discovered that OpenBSD had been fixed in a previous release" [4]. One example of this is the double free bug in the zlib compression library. It is described: "If an attacker is able to pass a specially-crafted block of invalid compressed data to a program that includes zlib, the program's attempt to decompress the crafted data can cause the zlib routines to corrupt the internal data structures maintained by malloc. ...Because this bug interferes with the proper allocation and deallocation of dynamic memory, it may be possible for an attacker to influence the operation of programs that include zlib. In most circumstances, this influence will be limited to denial of service or information leakage, but it is theoretically possible for an attacker to insert arbitrary code into a running program." [7].

Many vendors (which is a bit more broad than operating systems) were vulnerable to this exploit: Cisco Systems, most distributions of Linux (Connectiva, Debian, Engarde, Openwall, Red Hat, SuSE), SGI, and others. However, OpenBSD was not vulnerable because their "malloc implementation detects double freeing of memory" [7]. In addition, an OpenBSD developer commented: "I committed a fix for the problem in January. We didn't realize it was a security problem at the time (and neither did the zlib folks)" [8]. This means that OpenBSD was protected in two different ways. The first (the malloc implementation) was also present in some other products (FreeBSD, NetBSD) which prevent against double-free. The second (committed fix), however, was found and fixed before the exploit was found, leaving OpenBSD as one of the few safe distributions.

Applied to the principles of systems engineering, we can see how OpenBSD's process helped to identify the issue, removing the subsequent exploit. In this case, requirements identified in the needs analysis phase produced policy that was instrumental in fixing this exploit. As described by their goals: "OpenBSD believes in strong security. ...[W]e find many bugs, and endeavor to fix them even though exploitability is not proven" [4]. This is equivalent to a need, as either OpenBSD defines their needs to meet their goals, or the goals itself qualify as needs. This goal falls into the needs-driven development, which comes about from lacking in the current system. Here, the current system is (more-or-less) always not secure and bug free enough, since it is relatively impossible to fix all bugs in a software product (especially while adding new features), and is thus deficient. In the concept exploration phase, many alternative concepts are researched to meet this need to become more bug free and secure. Normally, one concept will be used in the new system. OpenBSD, however, has a "software development model permits us to take a more uncompromising view towards increased security than Sun, SGI, IBM, HP, or other vendors are able to" [4]. Hence, OpenBSD is able to select many or all of the concepts explored. Evidence of this is above, in the large list of technologies and memory protections described. Once the best concepts are chosen, they are prototyped, developed, and implemented into the system.

Exploit Found

As previously stated, OpenBSD has had two remote holes in the default install. This banner is proudly displayed since it is such a powerful nod to their practices. Hence, it follows that those two holes are among the most significant exploits in OpenBSD. There are certainly others, but they are either not remotely exploitable, or not available in the default install, which enables only a few services. One of these services is sshd, which is an implementation of SSH done by the OpenSSH group, which is part of the OpenBSD project.

In May of 2002, an exploit was found by Internet Security Systems (ISS) in OpenSSH's sshd, in the challenge-response handling [9]. The exploit is a buffer overflow, as previously discussed, that allowed code to be run as root (the superuser on the machine, who has all privileges; this is the most serious type of exploit). Although highly problematic, the information release handling of this vulnerability was done in a manner that, in the end, yielded no known compromised systems.

When the exploit was discovered by ISS, they alerted the OpenSSH (and thus OpenBSD) developers, who verified and fixed the exploit. But, due to the ubiquitousness of OpenSSH [10], they determined that it would be unwise to publish the fix since exploit code would quickly be written and used to break into machines. Thus, they alerted the community that a mjaor security issue was coming, to upgrade, and enable a workaround. The new version to which they recommended upgrade did not have a fix for the serious new issue. However, it did have a workaround, called privilege separation, a technique listed above, which separates a program into two segments, a large one with few privileges, and a small one with many privileges. If an exploit is found in the large segment, it is not serious since that segment has so few privileges. The small segment, with many privileges, is small, and thus easily audited for security issues. This security issue was in the segment with few privileges, and thus if run with privilege separation, would not be exploitable. If they had added the fix to this release, attackers would have easily been able to find the vulnerability. [11]

This theme of attackers finding the vulnerability was repeated throughout the rest of this process. There was another way to solve the problem, which involved disabling a certain feature. However, that would have highlighted that the bug was in about 500 out of 27,000 lines of code. The bug was only present in one of the multiple versions of the SSH protocol, but they could not alert the community to disable that protocol, which would have focused the problem on 5,000 of the 27,000 lines of code. They "did not tell people which versions were vulnerable, since the 2.9 to 2.9.9 transition was largely a rewrite of the ChallengeResponseAuthentication subsystem. This would have highlighted that as the problem area." Based on past experience, they determined not to alert vendors with detailed vulnerability information, since the list of vendors was large (above 80), and they were sure that any disclosure would leak quickly, which had happened before after a previous vulnerability was detailed to vendors. [11]

The timeline moved quickly. On June 21 they released OpenSSH 3.3 [12]. On June 24 they discussed that there was an upcoming vulnerability (but no details), and that some features in the just-released OpenSSH 3.3 were workarounds [13]. On June 26 they released OpenSSH 3.4, with a fix for the bug, which would allow attackers to determine create an exploit [14]. But, because of the caution and recommendation, their "users, including large and significant organizations, who were able to take a security stance by following our instructions about UsePrivilegeSeparation, disabling OpenSSH, filtering port 22, guessing at functional reduction, or preparing themselves for a new release at any time." They report that "[w]e have not heard of a single machine which was broken into as a result of our release announcement method." This is significant, since "[t]he first public attack program for the vulnerability was posted to BUGTRAQ within a day after OpenSSH 3.4 was released, apparently having been written based on the bug description." [11]

So, even though this was the first known remote exploit for OpenBSD and OpenSSH, it was handled well in code and within the community, yielding no known exploited systems. Let us now apply the systems engineering principles to this method. As before, the first and foremost need for this system (OpenSSH) is security. OpenSSH provides secure communication as its primary purpose. Thus, the same argument is made that this need for security propogates down to the concepts and implementations. And, again, the more general policy of adopting security technologies led to a safe transition to prevent the bug. The privilege separation technology which was designed for exactly this type of exploit (but not specifically because of and for this exploit), was one of the many concepts that were explored and implemented. During the announcement of this exploit, good post-production techniques were followed, and vendors were contacted and given instructions on how to best proceed.

The answer to what could have been done to prevent this type of exploit is more difficult to answer, and perhaps moot since the release process was done so well and without casualty. As we are dealing with a large, highly complex, software system, it is nigh impossible to find and remove all bugs. Hence, it is not possible for a systems engineering principle to create a perfect system. But, the adherence to security as a need yielded concepts, solutions, and followups which were successful.

Conclusion

We have seen two case studies from OpenBSD. First, the zlib vulnerability which was fixed in two different ways due to the adherence of security and bug fixing as a need, which affected the system's concepts and solutions as a whole, and allowed multiple solutions to be developed, one of which prevented exploit. Second, the OpenSSH vulnerability, which, while highly serious, was handled in a way that allowed all users to upgrade, without allowing attackers to create an exploit. Again, security as a need drove the principles and solutions which allowed this success.

References

[1] D. Walker-Morgan, "Kernel Log: More than 10 million lines of Linux source files," Oct. 21, 2008. [Online]. Available: http://www.heise-online.co.uk/open/Kernel-Log-More-than-10-million-lines-of-Linux-source-files–/news/111759. [Accessed: Oct. 27, 2008].

[2] S. Lohr, J. Markoff, "Windows Is So Slow, but Why?," Mar. 27, 2006. [Online]. Available: http://www.nytimes.com/2006/03/27/technology/27soft.html. [Accessed: Oct. 27, 2008].

[3] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[4] "OpenBSD Security," Oct. 24, 2008. [Online]. Available: http://openbsd.org/security.html. [Accessed: Oct. 27, 2008].

[5] "CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL," Jan. 17, 2002. [Online]. Available: http://www.cert.org/advisories/CA-2001-19.html. [Accessed: Oct. 27, 2008].

[6] "OpenBSD," Oct. 24, 2008. [Online]. Available: http://openbsd.org/. [Accessed: Oct. 27, 2008].

[7] "CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library," July 20, 2002. [Online]. Available: http://www.cert.org/advisories/CA-2002-07.html. [Accessed: Oct. 27, 2008].

[8] T. Miller, "Re: zlib bug," Mar. 12, 2002. [Online]. Available: http://marc.info/?l=openbsd-misc&m=101594961414449. [Accessed: Oct. 27, 2008].

[9] "OpenSSH Challenge-Response Buffer Overflow Vulnerabilities," Jun. 24, 2002. [Online]. Available: http://www.securityfocus.com/bid/5093/. [Accessed: Oct. 27, 2008].

[10] "SSH usage profiling," July 21, 2008. [Online]. Available: http://openssh.org/usage/graphs.html. [Accessed: Oct. 27, 2008].

[11] "Revised OpenSSH Security Advisory." [Online]. Available: http://www.openssh.com/txt/preauth.adv. [Accessed: Oct. 27, 2008].

[12] M. Friedl. "OpenSSH 3.3 released," June 21, 2002. [Online]. Available: http://marc.info/?l=openbsd-misc&m=102469496408742. [Accessed: Oct. 27, 2008].

[13] T. d. Raadt. "Upcoming OpenSSH vulnerability," June 24, 2002. [Online]. Available: http://marc.info/?l=openbsd-misc&m=102496766218240. [Accessed: Oct. 27, 2008].

[14] M. Friedl. "OpenSSH 3.4 released," June 26, 2002. [Online]. Available: http://marc.info/?l=openbsd-misc&m=102510264109175. [Accessed: Oct. 27, 2008].