Matt Jibson
ECE 501
Research Project 2

Buffer Overflows in OpenBSD

Abstract

This paper describes security practices and methods that the OpenBSD operating system uses to prevent against and mitigate buffer overflow expolits. OpenBSD has many techniques, enabled by default, to prevent or mitigate various types of attacks, like buffer overflows. The techniques exist at all levels of the operating system and development process. The application to systems engineering principles is described.

Introduction

Operating systems are among the class of systems that are continuously attacked in a large variety of ways. Attacks can come in the form of protocol weakness, race conditions, implementation bugs, and others. The OpenBSD project has managed a highly successful operating system that is historically highly secure. To understand why they have had success, we will look at their practices and methods.

Main Body

Throughout our course we repeatedly discuss the importance of documenting requirements, and how they affect downstream work and products. This principle is in full effect in OpenBSD. Although, as noted in my previous papers, OpenBSD does not specify new system requirements, they do specify broad project requirements like being "number one in the industry for security" [1]. One way they have done this is to pioneer a number of techniques to increase protection against certian types of attacks.

Buffer Overflow Mitigation Techniques

Buffer overflows are among the most common exploit [2]. When exploited they have the ability to execute arbitrary code on the comprimised machine. They can occur when using unsafe languages (C, C++), if a buffer is written to without bounds checking. This can cause memory at the end of the buffer to be overwritten. An attacker can specialize the overwritten data, causing the program to execute arbitrary code. In order to mitigate buffer overflows, OpenBSD uses many technologies [3] that are effective and have negligible side effects on performance. Some of them are detailed below.

The first technique is stack-gap randomization. A simple type of buffer overflow involves the stack. The stack stores information about functions of a program, including return addresses, which dictate where execution where next go. Thus, by overwriting this return address, an attacker can dictate where execution will continue. An easy solution to this problem is to insert a randomly sized gap at the beginning of the stack, which means that all functions start at a random offset to their default position. This will make the return address unknown to the attacker, and thus decrease greatly the chance of exploit. [4]

The second technique is ProPolice. It is a modification to the compiler (gcc) that "catches most common stack-smashing problems" [4]. A stack-smashing problem is one in which data past the end of a buffer is modified. A technique used by ProPolice (and other stack-smashing protection) is to insert a canary after buffers with a known value at the beginning of a function. At the end of the function the value of the canary is checked. If it is not the same, something has overwritten it. This technique introduces a negligible performance impact and is able to find bugs and make them unexploitable.

The third technique is called W^X, which is pronounced write xor read. Some bugs are exploitable because the address space has memory that is both writeable and executable (W|X). A policy was implemented to fix this: a memory page may be either writeable or executable, but not both. This required numerous, complex changes. To start, it was desirable to make the stack non-executable. For this to happen the signal trampoline was moved away from the top page of the stack. (The signal trampoline allows for the kernel

return control to a program after an interrupt [6].) Next, they modified the way shared (dynamic) libraries functioned. There were many data segments that had write and/or execute permissions that did not need them or could be changed to work without them. These data segments were moved around and given restricted permissions. At this point no memory page had both W and X. Unfortunately, however, not all machines have a processor with support for per-page X. To solve this, they put a "line" in the address space, with one side not executable, where the non-executable pages are stored. Finally, the shared library order was randomized every load. As a bonus (in addition to the added security), performance improved on some machine types, and no machine types were broken. [4]

Two more similar techniques also rely on randomizing the address space. They are the mmap() and malloc() functions, which involve allocating memory. Some recent exploits involve assumptions about the behavior of these functions allocating memory in predictable locations. To prevent this, the functions were changed to allocate to random locations. In addition, gap pages were inserted after allocated blocks. Buffer overflows (read or write) could then be detected when the gap pages were accessed. [4]

A further technique involved removing execute permissions (so only read was left) to strings and pointers that should be readonly. A new data segment was added to the executables do this, called .rodata. [4]

Buffer Overflow Removals

In addition and in response to the techniques presented above, many potential buffer overflows were removed from the system. Some of the side-effects of the techniques were that some classes of bugs became evident, and caused crashes and other problems that allowed them to be found and fixed (some over 30 years old) [7].

Besides those bugs, though, many potential buffer overflows were removed by a brute force code audit. One exploitable function is strcpy(). This function copies data from one string to another, stopping when the end of the source string is reached. If the destination string has not been allocated to sufficient size, an overflow occurs. To prevent this, the

function strncpy() was created [8]. It copies up to N characters of the string, but does not always copy the end-of-string marker. Thus, programmers must check that by hand. Since this is such a common function, these checks are often wrong. In response to this, OpenBSD created a new function, strlcpy(), which acts like strncpy() except that the end-of-string marker is guaranteed to terminate the string. This means the programmer does not have to write special code to do this, and is not prone to making mistakes in the implementation. The OpenBSD team converted all uses of strcpy() and strlcpy() to strncpy() where it made sense, which removed many potential errors [9].

Systems Engineering Principles

To evaluate identified issues that would have been disastrous if not for the use of specific systems engineering principles, we need to take a broad look at operating systems. In general, it is accepted that all Internet-facing operation systems have security vulnerabilities which may or may not be discovered and then exploited. With this assumption, the question posed above needs to change. Instead of assessing one large failure event, we must measure the possibility and severity of classes of exploits, and prevention methods against them. By this metric, OpenBSD has performed well, as they claim to have fixed bugs before they were known to be exploits, which other operating systems were vulnerable against [1]. In the other case, assuming that these exploits were not fixed or mitigated, the worst possible situation is that sensitive data could be retrieved or manipulated by an attacker. Depending on the data, this may be disastrous.

The success OpenBSD has had can be attributed to their following some principles of systems engineering. Above it was noted that OpenBSD has only little requirements documentation. Unlike standard systems procedures, they do not have a typical life cycle, as discussed in my first paper. However, success still followed from systems engineering principles. The requirements for the entire project in general discuss security in many aspects. These general requirements propogate down to all new code that comes in, and affects the auditing of old code. Since security is such a high priority, value is found in engineering new,

complex techniques that advance the state-of-the-art. These techniques prevent possible exploits from occuring, and have prevented bugs from becoming exploitable in the past.

Conclusion

We have seen that OpenBSD has defined security as a requirement. This caused them to advance new techniques like stack randomization and protection, address space randomization, privilege revocation in memory pages, and new, safe string handling functions that mitigate exploits and remove bugs from the system. This has resulted in a secure operating system with proactive security techniques, where possible exploits are controlled or removed without proof of exploitability.

References

[1] "OpenBSD Security," Nov. 02, 2008. [Online]. Available: http://openbsd.org/security.html. [Accessed: Dec. 1, 2008].

[2] S. Chaki & S. Hissam. "Certifying the Absence of Buffer Overflows," Sep. 2006. [Online]. Available: http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn030.pdf. [Accessed: Dec. 1, 2008].

[3] T. d. Raadt. "Advances in OpenBSD," Apr. 2003. [Online]. Available: http://cvs.openbsd.org/papers/csw03/index.html. [Accessed: Dec. 1, 2008].

[4] T. d. Raadt. "Exploit Mitigation Techniques," 2005. [Online]. Available: http://cvs.openbsd.org/papers/ven05-deraadt/index.html. [Accessed: Dec. 1, 2008].

[5] Aleph One. "Smashing The Stack For Fun And Profit." [Online]. Available: http://insecure.org/stf/smashstack.html. [Accessed: Dec. 1, 2008].

[6] E. Dreyfus. "Linux Compatibility on BSD for the PPC Platform: Part 3," June 07, 2001. [Online]. Available: http://www.onlamp.com/pub/a/onlamp/2001/06/07/linux_bsd.html. [Accessed: Dec. 1, 2008].

[7] O. Moerbeek. "Developer blog — otto@: New Otto malloc helps spot ancient bugs," Jul. 9, 2008. [Online]. Available: http://undeadly.org/cgi?action=article&sid=20080708155228. [Accessed: Dec. 1, 2008].

[8] T. C. Miller. "strlcpy and strlcat — consistent, safe, string copy and concatenation." [Online]. Available: http://www.openbsd.org/papers/strlcpy-paper.ps. [Accessed: Dec. 1, 2008].

[9] "OpenBSD 3.3 changes," Aug. 23, 2008. [Online]. Available: http://www.openbsd.org/plus33.html. [Accessed: Dec. 1, 2008].