Prevention and Detection of Stack Buffer Overflow Attacks

Benjamin A. Kuperman
Computer Science
Swarthmore College
`kuperman@cs.swarthmore.edu`

Carla E. Brodley
Computer Science
Tufts University
`brodley@cs.tufts.edu`

Hilmi Özdoğanoğlu, T.N. Vijaykumar, and Ankit Jalote
School of Electrical and Computer Engineering
Purdue University
{cyprian,vijay,jalote}@ecn.purdue.edu

August 12, 2005

Introduction

The recent announcement by Michael Lynn at Black Hat 2005 of a software flaw in Cisco routers has grabbed the attention of many technology news sources. The flaw is an instance of a buffer overflow, a type of security vulnerability that has been discussed since the 1960s, yet remains one of the most frequently reported type of remote attack against computer systems. The CVE[1] lists 303 buffer overflow vulnerabilities reported during the year 2004, an average of more than 25 new instances each month. For the first six months of 2005, 331 buffer overflow vulnerabilities were reported – clearly a problem not going away in the near future.

However, security researchers have been working for many years on developing techniques to prevent or detect the exploitation of these vulnerabilities. This article discusses what buffer overflow attacks are and briefly surveys the various tools and techniques that can be used to mitigate their threat.

What is a "buffer overflow"?

In basic terms, a *buffer overflow* occurs during program execution when a fixed size buffer has too much data copied into it. This causes the data to overwrite into adjacent memory locations, and depending on what is stored at these locations, this can affect the behavior of the program itself.

Buffer overflow attacks can take place in processes that use a stack during program execution. (Another type can occur in the heap, but this article looks at the former.) On the left-hand side of Figure 1 we show the three logical areas of memory used by a process. During program execution, when a function is called, a *stack frame* is allocated containing the function arguments, return address, previous frame pointer, and local variables. Each function prologue pushes a stack frame onto the top of the stack, and each function epilogue

[1]A list of Common Vulnerabilities and Exposures: `http://www.cve.mitre.org/`

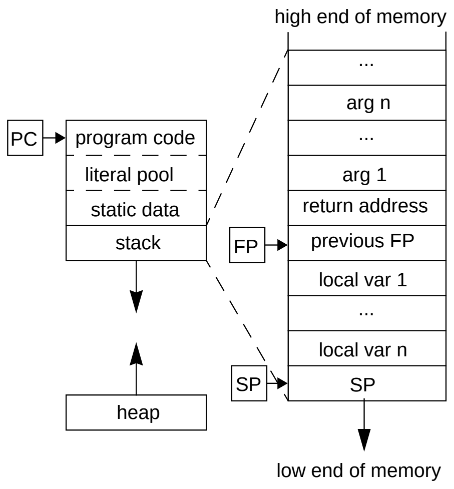

Figure 1: The memory layout of a stack frame after a function has been called.

pops the stack frame currently on the top of the stack. The return address in the frame points to the next instruction to execute after the current function returns. This introduces a vulnerability that allows an attacker to cause a program to execute arbitrary code.

Consider the following C function:

```
int foo(int a, int b) {

    char homedir[100];

    ...

    strcpy(homedir,getenv("HOME"));
    ...
    return(1);
}
```

If an overflow occurs when `strcpy()` copies the result from `getenv()` into the local variable `homedir`, then the copied data continues to be written toward the high end of memory (up in Figure 1), eventually overwriting other data on the stack, including the stored return address (RA) value. This will cause function `foo()` to return execution to whatever address happens to lie in the RA storage location. In most cases, this type of corruption results in a program crash (e.g., a "segmentation fault" or a "bus error" message). But an attacker can carefully select the value to place in the return address in order to redirect execution to the location of her choosing. If that location contains machine code, the attacker causes the program to execute any arbitrary set of instructions – essentially taking control of the process.

A buffer overflow usually contains both executable code as well as the address of where that code is stored on the stack. Frequently, this is a single string constructed by the attacker with the executable code first followed by enough repetitions of the target address that the RA is overwritten. This requires knowing exactly where the executable code will be stored or else the attack will fail. Attackers get around this by prepending a sequence of unneeded instructions (such as `NOP`) to their string. This creates a *ramp* or *sled* leading to the executable code. Now the modified RA only needs to point somewhere in the ramp to cause a successful attack. While it still takes some effort to find the proper range, an attacker only needs a close guess to hit the target.

On successful modification of the return address, the attacker can execute commands with the same level of privilege as that of the attacked program. If the compromised program is running as root then the attacker can use the injected code to spawn a superuser shell and take control of the machine. In the case of worms, a copy of the worm program is installed and the system begins looking for more machines to infect.

Other Methods to Redirect Program Execution

As prevention methods were developed and attacks became more sophisticated, many variants of the basic buffer overflow attack were developed to bypass protection methods. In addition to the buffer overflow attack described above, a format string attack in C can be used to overwrite the return address. Format string attacks are relatively new and are first thought to have gained notice in mid-2000 [9].

Regardless of the function involved, there are two general methods an attacker can use to change the stored return address in the stack: **direct** and **indirect**. The previous section described a **direct attack** where a local buffer is overwritten, continuing until a RA value has been changed. In an *indirect attack*, a pointer to the stored return address is used to cause the modification of only the stored value, not the surrounding data. Indirect attacks have a greater number of dependencies, but were developed to avoid methods used to prevent direct attacks. By making the assignment via a pointer, an attacker is able to jump over any protection or boundary that might exist between the buffer and the stored return address.

Sometimes, program control is directed through the use of a *function pointer* (e.g., continuations or error handlers). If an attacker modifies the pointer value as part of an overflow attack, then she can redirect program execution without modifying a RA. Function pointers are vulnerable to both direct and indirect attacks as well.

The final characteristic of an attack is based on where execution is redirected. The most well-known approach is to write shellcode either a) into the buffer being overflowed, b) above the RA on the stack, or c) in the heap. A second approach is called the return-to-libc attack. It was invented to bypass protection methods that prevent user data from being treated as program code. It does so by redirecting execution to the `system()` library function in `libc` with the argument "/bin/sh" (placed on the stack by the attacker).

For more information on how buffer overflows are written, see [1].

Prevention Techniques

Fortunately, there has been extensive research into tools and techniques that can be used to prevent (or detect) buffer overflow vulnerabilities. There are four basic groups of techniques: static analysis, compiler modifications, operating system modifications, and hardware modifications. Many of these can be composed together to provide a layered approach to the problem. We examine each in turn.

Static Techniques

One of the best ways to prevent the exploitation of buffer overflow vulnerabilities is to detect and eliminate them from the source code before the software is put into use. Usually this is done by performing some sort of *static analysis* on either the source code or compiled binaries.

An effective technique for uncovering flaws in software is that of source code review or *source code auditing*. While there are a number of efforts along these lines, the best known is the OpenBSD project.[2] Since 1996, the OpenBSD group has had six to twelve developers auditing the source code of a free, BSD-based operating system. This type of analysis requires substantial expenditure of time, and its effectiveness depends upon the expertise of the auditors. However, the payoff can be noticeable as evidenced by the fact that OpenBSD has one of the best reputations for security and one of the historically lowest rates of remote vulnerabilities (`http://www.securitymap.net/sdm/docs/general/Bugtraq-stat/stats.html`).

Tools designed to perform automatic *source code analysis* complement the act of a manual audit by identifying potential security violations including functions that perform unbounded string copying. Some of the best known tools are its4[3], RATS[4], and LCLint [7]. An extensive list of auditing tools is provided by the Sardonix portal at `https:`

[2]`http://www.openbsd.org/`
[3]`http://www.cigital.com/its4/`
[4]`http://www.securesw.com/rats/`

`//sardonix.org/Auditing_Resources.html`.

Most buffer overflow vulnerabilities are due to the presence of unbounded copying functions or unchecked buffer lengths in programming languages like C. One way to prevent programs from having such vulnerabilities is to write them using a language that does perform bound checking such as Java or Pascal. However, these languages often lack the low-level data manipulation needed by some applications. Therefore, researchers have produced "more secure" versions of C that are mostly compatible with existing programs but add in additional security features. Cyclone [5] is one such C-language variant. Unfortunately, the performance cost of bounds checking was reported to be up to a 100% overhead.

The solutions presented up to this point assume that the analyst has access to, and can modify, the program's source code. There are circumstances in which this assumption does not hold (e.g., legacy applications and commercial software). However, Prasad and Chiueh [11] present a technique by which an existing binary can be re-written to keep track of return addresses and verify they have not been changed without needing the source code. Their worst reported overhead was 3.44% for instrumenting Microsoft PowerPoint.

Compiler Modifications

If the source code is available, individual programs can have buffer overflow detection automatically added in to the program binary through the use of a modified compiler. StackGuard, ProPolice, StackShield, and RAD are four such compilers.

One technique to prevent buffer overflow attacks is a modified C-language compiler that automatically inserts detection code into a program when compiled. *StackGuard* [4] detects direct attacks against the stored RA by inserting a marker (called a *canary*) between the frame pointer and the return address on the stack. Before a function returns, the canary is read off the stack and tested for modification. The assumption is that a buffer overflow attack can be detected because in order to reach the stored address, it had to overwrite the canary first. StackGuard uses a special fixed value called a *terminating canary* that is composed of the four bytes most commonly used to terminate some sort of string copy (`NULL`, `CR`, `LF`, and `EOF`). It would be difficult, if not impossible, for an attacker to insert this value as part of their exploit string, therefore such attacks will be detected.

The ProPolice compiler[5] (also known as the stack-smashing protector or SSP) protects against direct attacks with a mechanism similar to that of StackGuard. In addition, ProPolice reorders the memory locations of variables such that pointers are below arrays and pointers from arguments are before local variables. The first helps prevent indirect attacks and the latter makes it more likely that a buffer overflow will be detected.

StackShield[6] is a Linux development security add-on (specifically an assembler file preprocessor) that can work with the gcc compiler to add protection from both direct and indirect buffer overflow attacks. It operates by adding instructions during compilation that causes a program to maintain a separate stack of return addresses in a different data

[5] `http://www.research.ibm.com/trl/projects/security/ssp/`
[6] `http://www.angelfire.com/sk/stackshield/`

segment. It would be difficult for an attacker to modify both the return address in the stack segment and the copy in the data segment through a single unbounded string copy. During a function return, the two values are compared and an alert is raised if they do not match.

StackShield also provides a secondary protection mechanism – it can implement a range check on both function call and function return addresses. If a program attempts to make a function call outside of a predefined range, or a function returns to a location outside of that range, then the software presumes an attack has taken place and terminates the process. This also allows software to protect against function pointer attacks.

The *Return Address Defender* or RAD [3] is a patch to gcc that automatically adds protection code into the prologues and epilogues of function calls. RAD stores a second copy of return addresses in a repository (similar to StackShield) but then uses OS memory protection functions to detect attacks against this repository. RAD either makes the entire repository read-only (causing significant performance degradation) or by marking neighboring pages as being read-only (minor overhead, but avoidable by an indirect attack).

Operating System Modifications

Several protection mechanisms operate by a modification of some aspect of the operating system. Because many buffer overflow attacks take place by loading executable code onto the stack and redirecting execution there, one of the simpler approaches is to modify the stack segment to be non-executable. This prevents an attacker from directing control to code they have uploaded into the stack. However, an attacker can still direct execution to either code uploaded in the heap or to an existing function (such as `system()` in libc). Most Unix-like operating systems have an optional patch or configuration switch that will remove execute permissions from the program stack.

A library modification called *Libsafe* [2] intercepts all calls to functions known to be vulnerable and executes a "safe version" of those calls. The safe versions estimate an upper limit for the size of the target buffer. Since it is highly unlikely that a program would deliberately overwrite a frame boundary, copies into buffers are bounded by the top of the frame in which they reside. Libsafe doesn't require programs to be recompiled. Unfortunately, library modifications only add in protection for a subset of functions, and only in dynamically linked programs. Many security critical applications are compiled statically, and it is possible in some instances for a determined attacker to bypass the modified libraries.

Perhaps the most comprehensive set of changes to an operating system to detect and prevent buffer overflows were introduced in the release of OpenBSD 3.3[7] with a multi-layered approach. First, they modify binaries to make it harder to have a successful exploit. They combine *stack-gap randomization* with the ProPolice compiler to make it harder for scripted attacks to work and add in detection capabilities. Secondly, they modify the memory segments allocated by the OS to remove execute permissions from as many places as possible and ensure that no segment is both writable and executable when in

[7]`http://www.openbsd.org/33.html`

user mode. This makes it much harder for an attacker to find code to run that already is present, and impossible for them to upload their own.

Microsoft recently has been putting increased emphasis on their developers to perform both source code auditing and use of automated bounds checking tools. They have stated that beginning with Service Pack 2 for Windows XP[8] there will be a number of security protections built into the operating system as well.

There are also techniques based on restricting the control flow of a program. While these do not detect buffer overflow attacks, they can mitigate their effects by restricting what can be executed afterwards. One technique is known as *proof-carrying code* [8]. In this, binary programs are bundled with a machine verifiable "proof" of what the program is going to do. As the program executes, the behavior is observed and compared against the proof by a new addition to the OS kernel. Any deviations are noticed, and the program can be killed. Another technique is called *program shepherding* [6]. Shepherding requires the verification of every branch instruction and verifying they match a given security policy. This is done by restricting where executable code can be located in memory, restricting where control transfers (e.g., jump, call, return) can take place and what their destinations are, and adding "sandboxing" on other operations. Their implementation is for the RIO runtime system on IA-32 platforms. Their reported worst case performance on SPEC2000 benchmarks was over 70% on Linux and 660% on Windows NT.

[8]http://msdn.microsoft.com/security/productinfo/xpsp2/

Hardware Modification

Any technique for buffer overflow detection is going to exact a performance cost on the system employing it. Depending on the technique, this can vary from 4% to over 1000% as reported by the various researchers. One way to reduce the execution time needed is to move operations from software to hardware which can execute the same operations tens or hundreds of times faster.

The *SmashGuard* proposal [10] uses a modification of the the microcoded instructions for the `CALL` and `RET` opcodes in a CPU to enable a transparent protection against buffer overflow attacks.[9] SmashGuard takes advantage of the fact that a modern CPU has substantial memory space on the chip and creates a secondary stack that holds return addresses similar to the RAR employed by StackShield. Unlike StackShield, the SmashGuard modifications to the CPU microcode make it possible to gain protection without needing to modify program software at all.

The `CALL` instruction is modified such that it will transparently store a copy of the return address on a data stack within the processor itself. The `RET` instruction will compare the top of the hardware stack with the address to which the software is trying to redirect execution back to. If the two values do not match, then the processor will raise a hardware exception that will cause the program to terminate in the general case. While this modification has not been fabricated into a CPU, it has been implemented on an architecture simulator.

[9]The authors are members of the **SmashGuard** project.

Performance of applications degraded by 0.02% which is two orders of magnitude better than StackGuard, and four orders better than StackShield. Additionally, issues such as context switches, `setjmp()/longjmp()`, and CPU stack spillage are properly handled.

Another hardware modification that has been proposed is known as *Split Stack and SRAS* [12]. This is a two prong approach. First, programs will be compiled to utilize two software stacks – one for program data and one for control information. This should make it more difficult for an overflow of a data variable to affect the stored control information. Performance costs for this approach varied from 0.01% to 23.77% depending upon the application tested.

A variation is a *secure return address stack* or *SRAS* stored on the processor. The SRAS will store all return addresses after a `CALL` instruction and use it for the next `RET` instruction. Theoretically, this should prevent a buffer overflow from changing the return address (possibly decreasing the effects), but would not actually detect or prevent any buffer overflow from occurring. Based on the discussion in their paper, there are still a number of issues to be worked out with the implementation of SRAS.

Conclusions

Despite the diverse nature of possible solutions, there is no “silver bullet” that can solve the problem of attacks against stored return addresses. No one technique can detect all possible instances, and attackers have a long history of learning how to circumvent prevention and detection mechanisms. Some of the more effective techniques involve training and review, but even the best-trained individuals can make a mistake. Dynamic protection techniques can be costly in terms of overhead, but there is hope to move that functionality into faster, hardware-based protection schemes. As these detection and prevention techniques move out of the academic realm into mainstream software releases, it is important that computer users and professionals are aware of the techniques and limitations. We have collected links to the projects discussed here and many more at our project website `http://www.smashguard.org/`.

References

[1] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49):File 14 of 16, Fall 1997. URL `http://www.phrack.com/`.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[3] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS '01)*, Mesa, AZ, April 2001. SUNY Stony Brook.

[4] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie,

Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, San Antonio, TX, January 1998. USENIX.

[5] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002. URL `http://www.research.att.com/projects/cyclone/`.

[6] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[7] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, August 2001.

[8] George C. Necula. Proof-Carrying Code. In *Proceedings of 24th ACM POPL*, pages 106–119, January 1997.

[9] Tim Newsham. Format String Attacks. Whitepaper, Guardent, Inc., September 2000. URL `http://www.lava.net/~newsham/format-string-attacks.pdf`.

[10] Hilmi Özdoğanoğlu, Carla Brodley, T.N. Vijaykumar, Ankit Jalote, and Benjamin A. Kuperman. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Technical Report TR-ECE 03-13, School of Elec-

trical and Computer Engineering, Purdue University, November 2003. URL `http://www.smashguard.org/`.

[11] Manish Prasad and Tzi-cker Chiueh. A Binary Rewriting Defense Against Stack Based Buffer Overflow Attacks. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[12] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture Support for Defending Against Buffer Overflow Attacks. In *2002 Workshop on Evaluating and Architecting System dependabilitY (EASY-2002)*. University of Illinois at Urbana-Champaign, October 2002.