By Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T.N. Vijaykumar, *and* Ankit Jalote

DETECTION AND PREVENTION OF STACK BUFFER OVERFLOW ATTACKS

How to mitigate remote attacks that exploit buffer overflow vulnerabilities on the stack and enable attackers to take control of the program.

The July 2005 announcement by computer security researcher Michael Lynn at the Black Hat security conference of a software flaw in Cisco Systems routers grabbed media attention worldwide. The flaw was an instance of a buffer overflow, a security vulnerability that has been discussed for 40 years yet remains one of the most frequently reported types of remote attack against computer systems. In 2004, the national cyber-security vulnerability database (nvd.nist.gov) reported 323 buffer overflow vulnerabilities, an average of more than 27 new instances per month. For the first six months of 2005, it reported 331 buffer overflow vulnerabilities. Meanwhile, security researchers have sought to develop techniques to prevent or detect the exploitation of these vulnerabilities. Here, we discuss what buffer overflow attacks are and survey the various tools and techniques that can be used to mitigate their threat to computer systems.

ILLUSTRATION BY LISA HANEY

A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected. Buffer overflow attacks can take place in processes that use a stack during program execution. (Another type of overflow attack can occur in the heap, but here we stick to stacks.) The left-hand side of the figure on the next page outlines the three logical areas of memory—text (instructions), data, and stack—used by a process. During program execution, when a function is called, a "stack frame" is allocated 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 pops, or deallocates, the stack frame currently on top of the stack. The return address in the frame points to the next instruction to execute after the current function returns. This storage and retrieval of the address of the next instruction on the stack introduces a vulnerability that allows an attacker to cause a program to execute arbitrary code.

To illustrate how this might happen, 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`, the copied data continues to be written toward the high end of memory (higher up in the figure), eventually overwriting other data on the stack, including the stored return address (RA) value. Overwriting causes 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 (such as a "segmentation fault" or "bus error" message). However, attackers can select the value to place in the return address in order to redirect execution to the location of their choosing. If it 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 and the address where that code is stored on the stack. The data used to overflow is often 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 attack strategy requires the attacker to know exactly where the executable code is stored; otherwise, the attack will fail. Attackers get around this requirement by prepending a sequence of unneeded instructions (such as `NOP`) to their string. Prepending a sequence creates a "ramp" or "sled" leading to the executable code. In such cases, the modified RA needs to point only somewhere in the ramp to enable a successful attack. While it still takes some effort to find the proper range, an attacker needs to make only a close guess to be able to hit the target.

Successfully modifying the return address allows the attacker to execute instructions with the same privileges as that of the attacked program. If the compromised program is running as root, the attacker might use the injected code to spawn a super-user 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.

Redirecting Program Execution

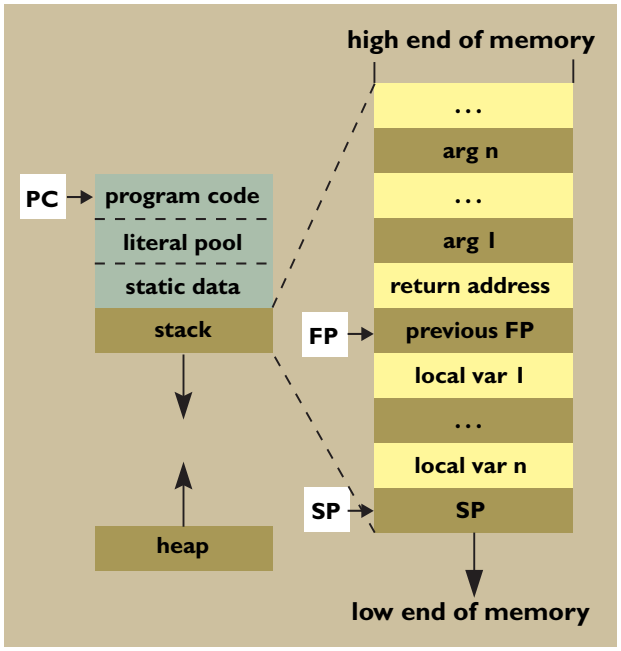
As prevention methods have been developed and attacks have become more sophisticated over the past 20 years, many variants of the basic buffer overflow attack have been developed by both attackers and researchers to bypass protection methods. In addition to the buffer overflow attack, a format string attack in C can be used to overwrite the return address. Format string attacks are relatively new and thought to have gained widespread notice through postings to the BUGTRAQ computer security mailing list [9].

An attacker might use two general methods—direct and indirect—to change the stored return address in the stack. We earlier described a direct attack in which a local buffer is overwritten, continuing until an RA value is 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 involve far more dependencies but were specially developed to avoid methods used to prevent direct attacks. By making the assignment via a pointer, attackers are able to jump over any protection or boundary that might exist between the buffer and the stored return address.

Program control is sometimes directed through function pointers (such as continuations and error handlers). Attackers modifying the pointer value as

part of an overflow attack can redirect program execution without modifying an RA. Function pointers are vulnerable to both direct and indirect attacks as well.

The final characteristic of an attack is based on where in memory the execution is redirected. The best-known approach is to write shellcode in one of three ways: into the buffer being overflowed, above the RA on the stack, or in the heap. A second approach is the return-to-libc attack, which was invented to deliberately bypass protection methods preventing user data from being treated as program code. This attack strategy bypasses some protection methods by redirecting execution to the `system()` library function in `libc` with the argument "/bin/sh" (placed on the stack by the attacker). For more on how buffer overflows are written, see [1, 10].



Memory layout of a stack frame after a function has been called.

Vulnerability Prevention Techniques

Fortunately for security-minded application developers, as well as for end users, extensive research has focused on tools and techniques for preventing (and detecting) buffer overflow vulnerabilities. Techniques are categorized into four basic groups—static analysis, compiler modifications, operating system modifications, and hardware modifications—that can often be combined to provide a layered approach to the problem.

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 to use, usually by performing some sort of static analysis on either the source code or on the compiled binaries.

A proven technique for uncovering flaws in software is source code review, also known as source code auditing. Among the various efforts along these lines, the best known is the OpenBSD project (www.openbsd.org). Since 1996, the OpenBSD group has assigned as many as 12 volunteer developers to audit the source code of a free, BSD-based operating system. This analysis requires much time, and its effectiveness depends on the expertise of the auditors. However, the payoff can be noticeable, as reflected in the fact that OpenBSD has one of the best reputations for security and historically lowest rates of remote vulnerabilities, as calculated by the statistics of reported vulnerabilities (via postings to the BUGTRAQ mailing list, ww.securitymap.net/sdm/docs/general/Bugraq-stat/stats.html).

Tools designed for automatic source code analysis complement manual audits by identifying potential security violations, including functions that perform unbounded string copying. Some of the best-known tools are ITS4 (www.cigital.com/its4/), RATS (www.securesw.com/rats/), and LCLint [7]. An extensive list of auditing tools is provided by the Sardonix portal at 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 (such as Java or Pascal) that performs bound checking. However, such 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 additional security features. Cyclone [5] is one such C-language variant. Unfortunately, the performance cost of bounds checking (reported in [5]) involves up to an additional 100% overhead.

These solutions assume the analyst has access to and can modify a program's source code. However, this assumption does not hold in all circumstances (such as in legacy applications and commercial software). A technique described in [11] makes it possible to rewrite an existing binary to keep track of return addresses and verify they have not been changed without needing the source code. The worst reported overhead of this technique was 3.44% in [11] for instrumenting Microsoft PowerPoint.

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 to use, usually by performing some sort of static analysis on either the source code or on the compiled binaries.

Compiler Modifications

If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler; four such compilers are StackGuard, ProPolice, StackShield, and Return Address Defender (RAD). One technique for preventing 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 made by the compiler (or designer of the modified compiler) is that a buffer overflow attack is detectable, because in order to reach the stored address, it had to first overwrite the canary. StackGuard uses a special fixed value (called a terminating canary) composed of the four bytes—`NULL`, `CR`, `LF`, and `EOF`—most commonly used to terminate some sort of string copy. It would be difficult, if not impossible, for attackers to insert this value as part of their exploit string; such attacks are thus easily detected.

The ProPolice compiler (also known as the stack-smashing protector, or SSP, www.research.ibm.com/trl/projects/security/ssp/) protects against direct attacks with a mechanism similar to StackGuard. In addition, ProPolice reorders the memory locations of variables, so pointers are below arrays and pointers from arguments are before local variables. Having pointers below arrays helps prevent indirect attacks; having pointers from arguments before local variables makes it more likely that a buffer overflow will be detected.

StackShield is a Linux security add-on (an assembler file preprocessor) that works with the gcc compiler to add protection from both direct and indirect buffer overflow attacks (www.angelfire.com/sk/stackshield/). It operates by adding instructions during compilation that cause programs to maintain a separate stack of return addresses in a different data segment. It would be difficult or impossible 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 by the inserted function epilogue; an alert is raised if they do not match.

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

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

Operating System Modifications

Several protection mechanisms operate by modifying 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 to defending against them is to modify the stack segment so it is nonexecutable. This prevents attackers 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 removes 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 the 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 the recompilation of programs. Unfortunately, library modifications add protection for only a subset of functions and only in dynamically linked programs. Many security-critical applications are compiled statically, making it possible in some instances for a determined attacker to bypass the modified libraries.

Perhaps the most comprehensive set of changes to an operating system for detecting and preventing buffer overflows was introduced in May 2003 in the release of OpenBSD 3.3 (www.openbsd.org/33.html). The developer first modifies binaries to make it more difficult for an attacker to be able to exploit a buffer overflow in any system program. The changes combine stack-gap randomization with the ProPolice compiler to make it more difficult for scripted attacks to succeed; detection capabilities were also added. Second, a developer modifies the memory segments allocated by the operating system to remove execute permissions from as many places as possible and ensure that no segment is both writable and executable when in user mode. These memory-segment changes made it much more difficult for attackers to find code to run that is already present and impossible for attackers to upload their own.

Microsoft has been pushing its in-house developers to perform source-code auditing and use automated bounds-checking tools. It announced that beginning with Service Pack 2 for Windows XP (August 2004), a number of security protections would be built into the operating system, including making memory nonexecutable on newer processors and buffer length checks in system programs (msdn.microsoft.com/security/productinfo/xpsp2/).

Other security-related programming techniques are based on restricting a program's control flow. Although they are not designed to detect buffer overflow attacks, they mitigate their effects by restricting what can be executed after an attack takes place. Proof-carrying code is one such technique [8]; binary programs are bundled with a machine-verifiable "proof" of what the program is going to do. As the program executes, that behavior is observed by a security monitor and compared against the proof by a new addition to the operating system kernel. Any deviations are noticed by the monitor (which might be in the kernel), and the program can be killed. Another technique is "program shepherding" [6], which requires the verification of every branch instruction and verifies they match a given security policy. It is done by restricting where executable code can be located in memory, restricting where control transfers (such as jump, call, and return) can take place, along with their destinations, and adding "sandboxing," or access restrictions, on other operations. Shepherding is for the MIT-run Runtime Introspection and Optimization operating system on IA-32 platforms (www.cag.lcs.mit.edu/rio/). Its reported worst-case performance on SPEC2000 benchmarks was over 70% on Linux and 660% on Windows NT.

Hardware Modification

Any technique that performs buffer overflow detection will exact a performance cost from the system employing it. Depending on the technique, it can vary from 4% to over 1,000%, as reported by various researchers. One way to reduce execution time is to move operations from software to hardware able to execute the same operations possibly tens or hundreds of times faster.

The SmashGuard proposal [10] uses a modification of the microcoded instructions for the `CALL` and `RET` opcodes in a CPU to enable transparent protection against buffer overflow attacks. (We are members of the SmashGuard project.) 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 return address repository employed by StackShield. Unlike StackShield, the SmashGuard modifications to the CPU microcode make it possible to add protection without having to modify the software.

The `CALL` instruction is modified such that it transparently stores a copy of the return address on a data stack within the processor itself. The `RET` instruction compares 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, the processor raises a hardware exception that causes the program to terminate in the general case. While this modification is not fabricated into a CPU, it has been implemented on an architecture simulator. Application performance was degraded by 0.02%—two orders of magnitude better than StackGuard and four orders better than StackShield. Additionally, the system properly handles `TODO` issues (such as context switches, `setjmp()/longjmp()`, and CPU stack spillage).

A proposed hardware modification is Split Stack and Secure Return Address Stack (SAS) [12]. (Note that several similar solutions are found in the literature; the full list of citations is included on our Web site www.smashguard.org/). It involves a two-pronged approach in which programs are compiled to utilize two software stacks, one for program data, one for control information. This should make it more difficult for an overflow of a data variable to affect the stored control information. The performance cost for this approach, as reported in [12], varies from 0.01% to 23.77%, depending on the application being tested.

A variation of the Split Stack software modification is a Secure Return Address Stack (SRAS) stored on the processor. The SRAS stores all return addresses after a `CALL` instruction, using it for the next `RET` instruction. Theoretically, this storage method should prevent a buffer overflow from changing the return address (possibly decreasing the effects) but would not actually detect or prevent the occurrence of any buffer overflow. However, a number of implementation issues (such as `setjmp()/longjmp()`) must still be worked out concerning SRAS implementation.

Conclusion

Despite the diverse nature of these potential solutions, no silver bullet is available for solving the problem of attacks against stored return addresses, and attackers have a long history of learning how to circumvent detection and prevention mechanisms. Some of the more effective techniques involve training and review, but even the best-trained individuals make mistakes. Dynamic protection techniques can be costly in terms of overhead, but some researchers are trying to move that functionality into faster, hardware-based protection schemes. As these techniques move from academic laboratories into mainstream software releases, computer users and software developers have become aware of what they can do, and what they can't do. We have collected links to the projects discussed here, along with many others, at our Web site www.smashguard.org/. C

References

1. Aleph One. Smashing the stack for fun and Profit. *Phrack Magazine 7,* 49 (Fall 1997); www.phrack.com/.
2. Baratloo, A., Singh, N., and Tsai, T. Transparent fun-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference* (San Diego, CA, June 2000).
3. Chiueh, T. and Hsu, F.-H. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (Mesa, AZ, Apr. 2001).
4. Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Conference* (San Antonio, TX, Jan. 1998).
5. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002), 275–288; www.research.att.com/projects/cyclone/.
6. Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).
7. Larochelle, D. and Evans, D. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium* (Aug. 2001).
8. Necula, G. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (Jan. 1997), 106–119.
9. Newsham, T. *Format String Attacks.* White paper, Guardent, Inc., Sept. 2000; www.lava.net/~newsham/format-string-attacks.pdf.
10. Ozdoganoglu, H., Brodley, C., Vijaykumar, T., Jalote, A., and Kuperman, B. *SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address.* Tech. Rep. TR-ECE 03-13, Purdue University School of Electrical and Computer Engineering, Nov. 2003; www.smashguard.org/.
11. Prasad, M. and Chiueh, T. 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. Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. Architecture support for defending against buffer overflow attacks. In *Proceedings of the 2002 Workshop on Evaluating and Architecting System dependabilitY (EASY-2002)* (University of Illinois at Urbana-Champaign, Oct. 2002).

Benjamin A. Kuperman (kuperman@cs.swarthmore.edu) is a visiting assistant professor in the Department of Computer Science at Swarthmore College, Swarthmore, PA.

Carla E. Brodley (brodley@cs.tufts.edu) is a professor in the Department of Computer Science at Tufts University, Medford, MA.

Hilmi Ozdoganoglu (hilmi.o@gmail.com) earned a master's degree from the School of Electrical and Computer Engineering at Purdue University, West Lafayette, IN.

T.N. Vijaykumar (vijay@ecn.purdue.edu) is an associate professor in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, IN.

Ankit Jalote (jalote@ecn.purdue.edu) is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, IN.