Buffer Overflows:

Attacks and Defenses for the Vulnerability of the Decade*

Crispin Cowan, Perry Wagle, Calton Pu,
Steve Beattie, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(crispin@cse.ogi.edu)
http://www.cse.ogi.edu/DISC/projects/immunix

Abstract

Buffer overflows have been the most common form of security vulnerability for the last ten years. More over, buffer overflow vulnerabilities dominate the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. If buffer overflow vulnerabilities could be effectively eliminated, a very large portion of the most serious security threats would also be eliminated. In this paper, we survey the various types of buffer overflow vulnerabilities and attacks, and survey the various defensive measures that mitigate buffer overflow vulnerabilities, including our own StackGuard method. We then consider which combinations of techniques can eliminate the problem of buffer overflow vulnerabilities, while preserving the functionality and performance of existing systems.

1 Introduction

Buffer overflows have been the most common form of security vulnerability in the last ten years. More over, buffer overflow vulnerabilities *dominate* in the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. Because these kinds of attacks enable anyone to take total control of a host, they represent one of the most serious classes security threats.

Buffer overflow attacks form a substantial portion of all security attacks simply because buffer overflow vulnerabilities are so common [15] and so easy to exploit [30, 28, 35, 20]. However, buffer overflow vulnerabilities *particularly* dominate in the class of remote penetration attacks because a buffer overflow vulnerability presents the attacker with exactly what they need: the ability to inject and execute attack code. The injected attack code runs with the privileges of the vulnerable program, and allows the attacker to bootstrap whatever other functionality is needed to control ("own" in the underground vernacular) the host computer.

For instance, among the five "new" "remote to local" attacks used in the 1998 Lincoln Labs intrusion detection evaluation, three were essentially social engineering attacks that snooped user credentials, and two were buffer overflows. 9 of 13 CERT advisories from 1998 involved buffer overflows [34] and at least half of 1999 CERT advisories involve buffer overflows [5]. An informal survey on the Bugtraq security vulnerability mailing list [29] showed that approximately 2/3 of respondents felt that buffer overflows are the leading cause of security vulnerability.[1]

Buffer overflow vulnerabilities and attacks come in a variety of forms, which we describe and classify in Section 2. Defenses against buffer overflow attacks similarly come in a variety of forms, which we describe in Section 3, including which kinds of attacks and vulnerabilities these defenses are effective against. The Immunix project has developed the StackGuard defensive mechanism [14, 11], which has been shown to be highly effective at resisting attacks without compromising system compatibility or performance [9]. Section 4 discusses which combinations of defenses complement each other. Section 5 presents our conclusions.

2 Buffer Overflow Vulnerabilities and Attacks

The overall goal of a buffer overflow attack is to subvert the function of a privileged program so that the attacker can take control of that program, and if the program is sufficiently privileged, thence control the host. Typically the attacker is attacking a `root` program, and immediately executes code similar to "exec(sh)" to get a `root` shell, but not always. To achieve this goal, the attacker must achieve two sub-goals:

*. This work supported in part by DARPA grant F30602-96-1-0331, and DARPA contract DAAH01-99-C-R206.

To appear as an invited talk at SANS 2000 (System Administration and Network Security), http://www.sans.org/newlook/events/sans2000.htm

1. The remaining 1/3 of respondants identified "misconfiguration" as the leading cause of security vulnerability.

1. Arrange for suitable code to be available in the program's address space.
2. Get the program to jump to that code, with suitable parameters loaded into registers & memory.

We categorize buffer overflow attacks is in terms of achieving these two sub-goals. Section 2.1 describes how the attack code is placed in the victim program's address space (which is where the "buffer" part comes from). Section 2.2 describes how the attacker overflows a program buffer to alter adjacent program state (which is where the "overflow" part comes from) to induce the victim program to jump to the attack code. Section 2.3 discusses some issues in combining the code injection techniques from Section 2.1 with the control flow corruption techniques from Section 2.2.

2.1 Ways to Arrange for Suitable Code to Be in the Program's Address Space

There are two ways to arrange for the attack code to be in the victim program's address space: either inject it, or use what is already there.

Inject it: The attacker provides a string as input to the program, which the program stores in a buffer. The string contains bytes that are actually native CPU instructions for the platform being attacked. Here the attacker is (ab)using the victim program's buffers to store the attack code. Some nuances on this method:

- The attacker does not have to overflow any buffers to do this; sufficient payload can be injected into perfectly reasonable buffers.
- The buffer can be located anywhere:
  - on the stack (automatic variables)
  - on the heap (malloc'd variables)
  - in the static data area (initialized or uninitialized)

It is already there: Often, the code to do what the attacker wants is already present in the program's address space. The attacker need only parameterize the code, and then cause the program to jump to it. For instance, if the attack code needs to execute "`exec("/bin/sh")`", and there exists code in libc that executes "`exec(arg)`" where "arg" is a string pointer argument, then the attacker need only change a pointer to point to "`/bin/sh`" and jump to the appropriate instructions in the `libc` library [41].

2.2 Ways to Cause the Program to Jump to the Attacker's Code

All of these methods seek to alter the program's control flow so that the program will jump to the attack code. The basic method is to *overflow* a buffer that has weak or non-existent bounds checking on its input with a goal of corrupting the state of an *adjacent* part of the program's state, e.g. adjacent pointers, etc. By overflowing the buffer, the attacker can overwrite the adjacent program state with a near-arbitrary[2] sequence of bytes, resulting in an arbitrary bypass of C's type system[3] and the victim program's logic.

The classification here is the kind of program state that the attacker's buffer overflow seeks to corrupt. In principle, the corrupted state can be *any* kind of state. For instance, the original Morris Worm [37] used a buffer overflow against the `fingerd` program to corrupt the name of a file that `fingerd` would execute. In practice, most buffer overflows found in "the wild" seek to corrupt *code pointers*: program state that points at code. The distinguishing factors among buffer overflow attacks is the kind of state corrupted, and where in the memory layout the state is located.

Activation Records: Each time a function is called, it lays down an *activation record* on the stack [1] that includes, among other things, the return address that the program should jump to when the function exits, i.e. point at the code injected in Section 2.1. Attacks that corrupt activation record return addresses overflow automatic variables, i.e. buffers local to the function, as shown in Figure 1. By corrupting the return address in the activation record, the attacker causes the program to jump to attack code when the victim function returns and dereferences the return address. This form of buffer overflow is called a "stack smashing attack" [14, 30, 28, 35] and constitute a majority of current buffer overflow attacks

Function Pointers: "`void (* foo)()`" declares the variable `foo` which is of type "pointer to function returning `void`." Function pointers can be allocated anywhere (stack, heap, static data area) and so the attacker need only find an overflowable buffer adjacent to a function pointer in any of these areas and overflow it to change the function pointer. Some time later, when the program makes a call through this function pointer, it will instead jump to the attacker's desired location. An example of this kind of attack appeared in an attack against the `superprobe` program for Linux.

Longjmp buffers: C includes a simple checkpoint/rollback system called setjmp/longjmp. The idiom is to say "`setjmp(buffer)`" to checkpoint, and say "`longjmp(buffer)`" to go back to the checkpoint. However, if the attacker can corrupt the state of the buffer, then "`longjmp(buffer)`" will

2. There are some bytes that are hard to inject, such as control characters and null bytes that have special meaning to I/O libraries, and thus may be filtered before they reach the program's memory.
3. That this is possible is an indication of the weakness of C's type system.

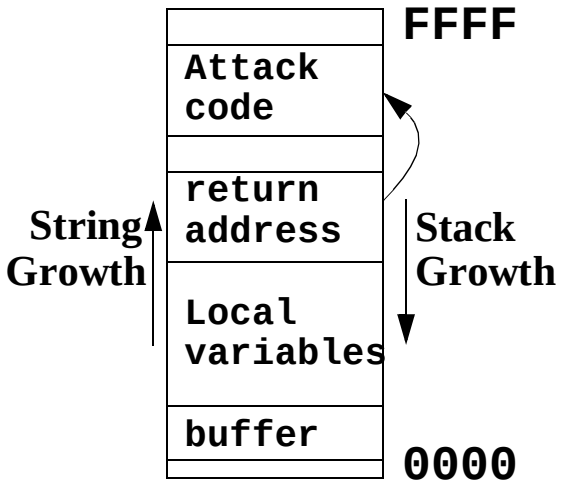

Figure 1: Buffer Overflow Attack Against Activation Record

jump to the attacker's code instead. Like function pointers, `longjmp` buffers can be allocated anywhere, so the attacker need only find an adjacent overflowable buffer. An example of this form of attack appeared against Perl 5.003. The attack first corrupted a `longjmp` buffer used to recover when buffer overflows are detected, and then induces the recovery mode, causing the Perl interpreter to jump to the attack code.

2.3 Combining Code Injection and Control Flow Corruption Techniques

Here we discuss some issues in combining the attack code injection (Section 2.1) and control flow corruption (Section 2.2) techniques.

The simplest and most common form of buffer overflow attack combines an injection technique with an activation record corruption in a single string. The attacker locates an overflowable automatic variable, feeds the program a large string that simultaneously overflows the buffer to change the activation record, and contains the injected attack code. This is the template for an attack outlined by Levy [30]. Because the C idiom of allocating a small local buffer to get user or parameter input is so common, there are a lot of instances of code vulnerable to this form of attack.

The injection and the corruption do not have to happen in one action. The attacker can inject code into one buffer without overflowing it, and overflow a different buffer to corrupt a code pointer. This is typically done if the overflowable buffer *does* have bounds checking on it, but gets it wrong, so the buffer is only overflowable up to a certain number of bytes. The attacker does not have room to place code in the vulnerable buffer, so the code is simply inserted into a different buffer of sufficient size.

If the attacker is trying to use already-resident code instead of injecting it, they typically need to parameterize the code. For instance, there are code fragments in `libc` (linked to virtually every C program) that do "`exec(something)`" where "`something`" is a parameter. The attacker then uses buffer overflows to corrupt the argument, and another buffer overflow to corrupt a code pointer to point into `libc` at the appropriate code fragment.

3 Buffer Overflow Defenses

There are four basic approaches to defending against buffer overflow vulnerabilities and attacks. The brute force method of writing correct code is described in Section 3.1. The operating systems approach described in Section 3.2 is to make the storage areas for buffers non-executable, preventing the attacker from injecting attack code This approach stops many buffer overflow attacks, but because attackers do not necessarily need to inject attack code to perpetrate a buffer overflow attack (see Section 2.1) this method leaves substantial vulnerabilities. The direct compiler approach described in Section 3.3 is to perform array bounds checks on all array accesses. This method completely eliminates the buffer overflow problem by making overflows impossible, but imposes substantial costs. The indirect compiler approach described in Section 3.4 is to perform integrity checks on code pointers before dereferencing them. While this technique does not make buffer overflow attacks *impossible*, it does stop most buffer overflow attacks, and the attacks that it does not stop are difficult to create, and the compatibility and performance advantages over array bounds checking are substantial, as described in Section 3.5.

3.1 Writing Correct Code

"To err is human, but to really foul up requires a computer." -- Anon. Writing correct code is a laudable but remarkably expensive proposition [13, 12], especially when writing in a language such as C that has error-prone idioms such as null-terminated strings and a culture that favors performance over correctness. Despite a long history of understanding of how to write secure programs [6] vulnerable programs continue to emerge on a regular basis [15]. Thus some tools and techniques have evolved to help novice developers write programs that are somewhat less likely to contain buffer overflow vulnerabilities.

The simplest method is to `grep` the source code for highly vulnerable library calls such as `strcpy` and `sprintf` that do not check the length of their arguments. Versions of the C standard library have also been developed that complain when a program links to vulnerable functions like `strcpy` and `sprintf`.

Code auditing teams have appeared [16, 2] with an explicit objective of auditing large volumes of code by hand, looking for common security vulnerabilities such as buffer overflows and file system race conditions [7]. However, buffer overflow vulnerabilities can be subtle. Even defensive code that uses safer alternatives such as

`strncpy` and `snprintf` can contain buffer overflow vulnerabilities if the code contains an elementary off-by-one error. For instance, the `lprm` program was found to have a buffer overflow vulnerability [22], despite having been audited for security problems such as buffer overflow vulnerabilities.

To combat the problem of subtle residual bugs, more advanced debugging tools have been developed, such as fault injection tools [23]. The idea is to inject deliberate buffer overflow faults at random to search for vulnerable program components. There are also static analysis tools emerging [40] that can detect many buffer overflow vulnerabilities.

While these tools are helpful in developing more secure programs, C semantics do not permit them to provide total assurance that all buffer overflows have been found. Debugging techniques can only minimize the number of buffer overflow vulnerabilities, and provide no assurances that *all* the buffer overflow vulnerabilities have been eliminated. Thus for high assurance, protective measures such those described in sections 3.2 through 3.4 should be employed unless one is *very* sure that all potential buffer overflow vulnerabilities have been eliminated.

3.2 Non-Executable Buffers

The general concept is to make the *data segment* of the victim program's address space *non-executable*, making it impossible for attackers to execute the code they inject into the victim program's input buffers. This is actually the way that many older computer systems were designed, but more recent UNIX and MS Windows systems have come to depend on the ability to emit dynamic code into program data segments to support various performance optimizations. Thus one cannot make *all* program data segments non-executable without sacrificing substantial program compatibility.

However, one *can* make the *stack segment* non-executable and preserve most program compatibility. Kernel patches are available for both Linux and Solaris [18, 19] that make the stack segment of the program's address space non-executable. Since virtually no legitimate programs have code in the stack segment, this causes few compatibility problems. There are two exceptional cases in Linux where executable code must be placed on the stack:

Signal Delivery: Linux delivers UNIX signals to processes by emitting code to deliver the signal onto the process's stack and then inducing an interrupt that jumps to the delivery code on the stack. The non-executable stack patch addresses this by making the stack executable during signal delivery.

GCC Trampolines: There are indications that gcc places executable code on the stack for "trampolines." However, in practice disabling trampolines has never been found to be a problem; that portion of gcc appears to have fallen into disuse.

The protection offered by non-executable stack segments is highly effective against attacks that depend on injecting attack code into automatic variables but provides no protection against other forms of attack (see Section 2.1). Attacks exist that bypass this form of defense [41] by pointing a code pointer at code already resident in the program. Other attacks could be constructed that inject attack code into buffers allocated in the heap or static data segments.

3.3 Array Bounds Checking

While injecting code is optional for a buffer overflow attack, the corruption of control flow is essential. Thus unlike non-executable buffers, array bounds checking *completely* stops buffer overflow vulnerabilities and attacks. If arrays cannot be overflowed at all, then array overflows cannot be used to corrupt adjacent program state.

To implement array bounds checking, then all reads and writes to arrays need to be checked to ensure that they are within range. The direct approach is to check *all* array references, but it is often possible to employ optimization techniques to eliminate many of these checks. There are several approaches to implementing array bounds checking, as exemplified by the following projects.

3.3.1 Compaq C Compiler. The Compaq C compiler for the Alpha CPU (cc on Tru64 UNIX, ccc on Alpha Linux [8]) supports a *limited* form of array bounds checking when the "-check_bounds" option is used. The bounds checks are limited in the following ways:

- only *explicit* array references are checked, i.e. "`a[3]`" is checked, while "`*(a+3)`" is not
- since all C arrays are converted to pointers when passed as arguments, no bounds checking is performed on accesses made by subroutines
- dangerous library functions (i.e. strcpy()) are not normally compiled with bounds checking, and remain dangerous even with bounds checking enabled

Because it is so common for C programs to use pointer arithmetic to access arrays, and to pass arrays as arguments to functions, these limitations are severe. The bounds checking feature is of limited use for program debugging, and no use at all in assuring that a program's buffer overflow vulnerabilities are not exploitable.

3.3.2 Jones & Kelly: Array Bounds Checking for C. Richard Jones and Paul Kelly developed a `gcc` patch [26] that does full array bounds checking for C programs. Compiled programs are compatible with other `gcc` modules, because they have not changed the representation of pointers. Rather, they derive a "base" pointer from each pointer expression, and check the

attributes of that pointer to determine whether the expression is within bounds.

The performance costs are substantial: a pointer-intensive program (ijk matrix multiply) experienced 30× slowdown, Since slowdown is proportionate to pointer usage, which is quite common in privileged programs, this performance penalty is particularly unfortunate.

The compiler did not appear to be mature; complex programs such as `elm` failed to execute when compiled with this compiler. However, an updated version of the compiler is being maintained [39], and it *can* compile and run at least portions of the SSH software encryption package. Throughput experiments with the updated compiler and software encryption using SSH showed a 12× slowdown [32] (see Section 3.4.2 for comparison).

3.3.3 Purify: Memory Access Checking. Purify [24] is a memory usage debugging tool for C programs. Purify uses "object code insertion" to instrument *all* memory accesses. After linking with the Purify linker and libraries, one gets a standard native executable program that checks all of its array references to ensure that they are legitimate. While Purify-protected programs run normally without any special environment, Purify is not actually intended as a production security tool: Purify protection imposes a 3 to 5 times slowdown. Purify also was laborious to construct, as evidenced by a purchase price of approximately $5000 per copy.

3.3.4 Type-Safe Languages. All buffer overflow vulnerabilities result from the lack of type safety in C. If only type-safe operations can be performed on a given variable, then it is not possible to use creative input applied to variable `foo` to make arbitrary changes to the variable `bar`. If new, security-sensitive code is to be written, it is recommended that the code be written in a type-safe language such as Java or ML.

Unfortunately, there are millions of lines of code invested in existing operating systems and security-sensitive applications, and the vast majority of that code is written in C. This paper is primarily concerned with methods to protect *existing* code from buffer overflow attacks.

However, it is also the case that the Java Virtual Machine (JVM) is a C program, and one of the ways to attack a JVM is to apply buffer overflow attacks to the JVM itself [17, 33]. Because of this, applying buffer overflow defensive techniques to the systems that *enforce* type safety for type-safe languages may yield beneficial results.

3.4 Code Pointer Integrity Checking

The goal of *code pointer integrity checking* is subtly different from bounds checking. Instead of trying to prevent corruption of code pointers (as described in Section 2.2) code pointer integrity checking seeks to detect that a code pointer has been corrupted *before* it is dereferenced. Thus while the attacker succeeds in corrupting a code pointer, the corrupted code pointer will never be used because the corruption is detected before each use.

Code pointer integrity checking has the disadvantage relative to bounds checking that it does not perfectly solve the buffer overflow problem; overflows that affect program state components *other* than code pointers will still succeed (see Table 3 in Section 4 for details). However, it has substantial advantages in terms of performance, compatibility with existing code, and implementation effort, which we detail in Section 3.5.

Code pointer integrity checking has been studied at three distinct levels of generality. Snarskii developed a custom implementation of `libc` for FreeBSD [36] that introspects the CPU stack to detect buffer overflows, described in Section 3.4.1. Our own StackGuard project [14, 9] produced a compiler that automatically generates code to perform integrity checking on function activation records, described in Section 3.4.2. Finally, we are in the process of developing PointGuard, a compiler that generalizes the StackGuard-style of integrity checking to all code pointers, described in Section 3.4.3.

3.4.1 Hand-coded Stack Introspection. Snarskii developed a custom implementation of `libc` for FreeBSD [36] that introspects the CPU stack to detect buffer overflows. This implementation was hand-coded in assembler, and only protects the activation records for the functions within the `libc` library. Snarskii's implementation is effective as far as it goes, and protects programs that use `libc` from vulnerabilities within `libc`, but does not extend protection to vulnerabilities in any other code.

3.4.2 StackGuard: Compiler-generated Activation Record Integrity Checking.

StackGuard is a compiler technique for providing code pointer integrity checking to the return address in function activation records [14]. StackGuard is implemented as a small patch to gcc that enhances the code generator for emitting code to set up and tear down functions. The enhanced setup code places a "canary"[4] word next to the return address on the stack, as shown in Figure 2. The enhanced function tear down code first checks to see that the canary word is intact before jumping to the address pointed to by the return address word. Thus if an attacker attempts a "stack smashing" attack as shown in Figure 1, the attack will be detected before the program ever attempts to dereference the corrupted activation record.

Critical to the StackGuard "canary" approach is that the attacker is prevented from *forging* a canary by

4.A direct descendent of the Welsh miner's canary.

Table 1: StackGuard Penetration Resistance

Vulnerable Program	Result Without StackGuard	Result with StackGuard
dip 3.3.7n	**root** shell	program halts
elm 2.4 PL25	**root** shell	program halts
Perl 5.003	**root** shell	program halts irregularly
Samba	**root** shell	program halts
SuperProbe	**root** shell	program halts irregularly
umount 2.5K/libc 5.3.12	**root** shell	program halts
wwwcount v2.3	**httpd** shell	program halts
zgv 2.7	**root** shell	program halts

embedding the canary word in the overflow string. StackGuard employs two alternative methods to prevent such a forgery:

Terminator Canary: The terminator canary is comprised of the common termination symbols for C standard string library functions; 0 (null), CR, LF, and -1 (EOF). The attacker cannot use common C string libraries and idioms to embed these symbols in an overflow string, because the copying functions will terminate when they hit these symbols.

Random Canary: The canary is simply a 32-bit random number chosen at the time the program starts. The random canary is a secret that is easy to keep and hard to guess, because it is never disclosed to anyone, and it is chosen anew each time the program starts.

StackGuard's notion of integrity checking the stack in this way is derived from the Synthetix [31, 38] notion of using *quasi-invariants* to assure the correctness of *incremental specializations*. A specialization is a *deliberate* change to the program, which is only valid if certain conditions hold. We call such a condition a *quasi-invariant*, because it changes, but only occasionally. To assure correctness, Synthetix developed a variety of tools to *guard* the state of quasi-invariants [10].

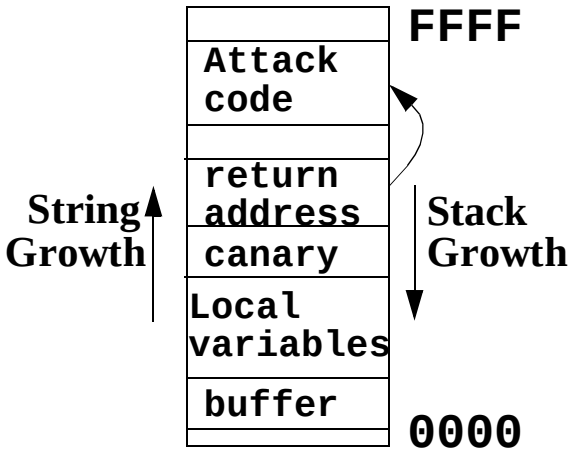

Figure 2: StackGuard Defense Against Stack Smashing Attack

The changes imposed by attackers employing buffer overflow techniques can be viewed as *invalid* specializations. In particular, buffer overflow attacks violate the quasi-invariant that the return address of an active function should *not change* while the function is active. StackGuard's integrity checks enforce this quasi-invariant.

Experimental results have shown that StackGuard provides effective protection against stack smashing attacks, while preserving virtually all of system compatibility and performance. Previously [14] we reported StackGuard's penetration resistance when exploits were applied to various vulnerable programs, reproduced here in Table 1. Subsequently we built an entire Linux distribution (Red Hat Linux 5.1) using StackGuard [9]. When attacks were released against vulnerabilities in `XFree86-3.3.2-5` [3] and `lsof` [43] we tested them as well, and found that StackGuard had successfully detected and rejected these attacks. This penetration analysis demonstrates that StackGuard is highly effective in detecting and preventing both current and *future* stack smashing attacks.

We have had the StackGuarded version of Red Hat Linux 5.1 in production on various machines for over one year. This StackGuarded Linux runs on both Crispin Cowan's personal laptop computer, and on our group's shared file server. This Linux distribution has been downloaded from our web site hundreds of times, and there are 55 people on the StackGuard user's mailing list. With only a single exception, StackGuard has functioned identically to the corresponding original Red Hat Linux 5.1. This demonstrates that StackGuard

Table 2: Apache Web Server Performance With and Without StackGuard Protection

StackGuard Protection	# of Clients	Connections per Second	Average Latency in Seconds	Average Throughput in MBits/Second
No	2	34.44	0.0578	5.63
No	16	43.53	0.3583	6.46
No	30	47.2	0.6030	6.46
Yes	2	34.92	0.0570	5.53
Yes	16	53.57	0.2949	6.44
Yes	30	50.89	0.5612	6.48

protection does not materially affect system compatibility.

We have done a variety of performance tests to measure the overhead imposed by StackGuard protection. Microbenchmarks showed substantial increases in the cost of a single function call [14]. However, subsequent macrobenchmarks on network services (the kinds of programs that *need* StackGuard protection) showed very low aggregate overheads.

Our first macrobenchmark used SSH [42] which provides strongly authenticated and encrypted replacements for the Berkeley `r*` commands, i.e. `rcp` becomes `scp`. SSH uses software encryption, and so performance overheads will show up in lowered bandwidth. We measured the bandwidth impact by using scp to copy a large file via the network loopback interface as follows:

```
scp bigsource localhost:bigdest
```

The results showed that StackGuard presents virtually no cost to SSH throughput. Averaged over five runs, the generic **`scp`** ran for 14.5 seconds (+/- 0.3), and achieved an average throughput of 754.9 kB/s (+/- 0). The StackGuard-protected **`scp`** ran for 13.8 seconds (+/- 0.5), and achieved an average throughput of 803.8 kB/s (+/- 48.9).[5]

Our second macorbenchmark measured performance overhead in the Apache web server [4], which is also clearly a candidate for StackGuard protection. If Apache can be stack smashed, the attacker can seize control of the web server, allowing the attacker to read confidential web content, as well as change or delete web content without authorization. The web server is also a performance-critical component, determining the amount of traffic a given server machine can support.

We measure the cost of StackGuard protection by measuring Apache's performance using the WebStone benchmark [27], with and without StackGuard protection. The WebStone benchmark measures various aspects of a web server's performance, simulating a load generated from various numbers of clients. The results with and without StackGuard protection are shown in Table 2.

As with SSH, performance with and without StackGuard protection is virtually indistinguishable. The StackGuard-protected web server shows a very slight advantage for a small number of clients, while the unprotected version shows a slight advantage for a large number of clients. In the worst case, the unprotected Apache has a 8% advantage in connections per second, even though the protected web server has a slight advantage in average latency on the same test. As before, we attribute these variances to noise, and conclude that StackGuard protection has no significant impact on web server performance.

3.4.3 PointGuard: Compiler-generated Code Pointer Integrity Checking.

At the time StackGuard was built, the "stack smashing" variety formed a gross preponderance of buffer overflow attacks. It is conjectured that this resulted from some "cook book" templates for stack smashing attacks released in late 1996 [25]. Since that time, most of the "easy" stack smashing vulnerabilities have been exploited or otherwise discovered and patched, and the attackers have moved on to explore the more general form of buffer overflow attacks as described in Section 2.

PointGuard is a generalization of the StackGuard approach designed to deal with this phenomena. PointGuard generalizes the StackGuard defense to place "canaries" next to all code pointers (function pointers

5. We do not actually believe that StackGuard enhanced SSH's performance. Rather, the test showed considerable variance, with latency ranging from 13.31 seconds to 14.8 seconds, and throughput ranging from 748 kB/s to 817 kB/s, on an otherwise quiescent machine. Since the two averages are within the range of observed values, we simply conclude that StackGuard protection did not significantly impact SSH's performance.

and `longjmp` buffers) and to check for the validity of these canaries when ever a code pointer is dereferenced. If the canary has been trampled, then the code pointer is corrupt and the program should issue an intrusion alert and exit, as it does under StackGuard protection. There are two issues involved in providing code pointers with canary protection:

Allocating the Canary: Space for the canary word has to be allocated when the variable to be protected is allocated, and the canary has to be initialized when the variable is initialized. This is problematic; to maintain compatibility with existing programs, we do *not* want to change the size of the protected variable, so we cannot simply add the canary word to the definition of the data structure. Rather, the space allocation for the canary word must be "special cased" into *each* of the kinds of allocation for variables, i.e. stack, heap, and static data areas, standalone vs. within structures and arrays, etc.

Checking the Canary: The integrity of the canary word needs to be verified every time the protected variable is loaded from memory into a register, or otherwise is read. This too is problematic, because the action "read from memory" is not well defined in the compiler's semantics; the compiler is more concerned with when the variable is actually used, and various optimization algorithms feel free to load the variable from memory into registers whenever it is convenient. Again, the loading operation needs to be "special cased" for all of the circumstances that cause the value to be read from memory.

We have built an initial prototype of PointGuard (again, a `gcc` enhancement) that provides canary protection to function pointers that are statically allocated and are not members of some other aggregate (i.e. a `struct` or an array). This implementation is far from complete. When PointGuard is complete, the combination of StackGuard and PointGuard protection should create executable programs that are virtually immune to buffer overflow attacks.

Only the relatively obscure form of buffer overflow attack that corrupts a non-pointer variable to affect the program's logic will escape PointGuard's attention. To address this problem, the PointGuard compiler will include a special "canary" storage class that forces canary protection onto *arbitrary* variables. Thus the programmer could manually add PointGuard protection to any variable deemed to be security-sensitive.

3.5 Compatibility and Performance Considerations

Code pointer integrity checking has the disadvantage relative to bounds checking that it does not perfectly solve the buffer overflow problem. However, it has substantial advantages in terms of performance, compatibility with existing code, and implementation effort, as follows:

Performance: Bounds checking must (in principle) perform a check every time an array element is read or written to. In contrast, code pointer integrity checking must perform a check every time a code pointer is dereferenced, i.e. every time a function returns or an indirect function pointer is called. In C code, code pointer dereferencing happens a *great* deal less often than array references, imposing substantially lower overhead. Even C++ code, where virtual methods make indirect function calls common place, still may access arrays more often than it calls virtual methods, depending on the application.

Implementation Effort: The major difficulty with bounds checking for C code is that C semantics make it difficult to determine the bounds of an array. C mixes the concept of an array with the concept of a generic pointer to an object, so that a reference into an array of elements of type `foo` is indistinguishable from a pointer to an object of type `foo`. Since a pointer to an *individual* object does not normally have bounds associated with it, it is only one machine word in size, and there is no where to store bounds information. Thus bounds checking implementations for C need to resort to exotic methods to recover bounds information; array references are no longer simple pointers, but rather become pointers to buffer descriptors.

Compatibility with Existing Code: Some of the bounds checking methods such as Jones and Kelly [26] seek to preserve compatibility with existing programs, and go to extraordinary lengths to retain the property that "`sizeof(int) == sizeof(void *)`", which increases the performance penalty for bounds checking. Other implementations resort to making a pointer into a tuple ("base and bound", "current and end", or some variation there of). This breaks the usual C convention of "`sizeof(int) == sizeof(void *)`", producing a kind-of C compiler that can compile a limited subset of C programs; specifically those that either don't use pointers, or those crafted to work with such a compiler.

Many of our claims of the advantages of code pointer integrity checking vs. bounds checking are speculative. However, this is because of the distinct lack of an effective bounds checking compiler for C code. There does not exist any bounds checking compiler capable of approaching the compatibility and performance abilities of the StackGuard compiler. While this makes for unsatisfying science with regard to our performance claims, it supports our claims of compatibility and ease of implementation. To test our performance claims, someone would have to invest the effort to build a fully compatible bounds checking enhancement to a C compiler that, unlike Purify [24] is not intended for debugging.

4 Effective Combinations

Here we compare the varieties of vulnerabilities and attacks described in Section 2 with the defensive measures described in Section 3 to determine which *combinations* of techniques offer the potential to completely eliminate the buffer overflow problem, and at what cost. Table 3 shows the cross-bar of buffer overflow attacks and defenses. Across the top is the set of places where the attack code is located (Section 2.1) and down the side is the set of methods for corrupting the program's control flow (Section 2.2). In each cell is the set of defensive measures that is effective against that particular combination. We omit the bounds checking defense (Section 3.3) from Table 3. While bounds checking is effective in preventing all forms of buffer overflow attack, the costs are also prohibitive in many cases.

The most common form of buffer overflow attack is the attack against an activation record that injects code into a stack-allocated buffer. This form follows from the recipes published in late 1996 [30, 28, 35]. Not surprisingly, both of the early defenses (the Non-executable stack [19, 18] and StackGuard [14]) both are effective against this cell. The non-executable stack expands up the column to cover all attacks that inject code into stack allocated buffers, and the StackGuard defense expands to cover all attacks that corrupt activation records. These defenses are completely compatible with each other, and so using *both* provides substantial coverage of the field of possible attacks.

Of the remaining attacks not covered by the combination of the non-executable stack and the StackGuard defense, many can be automatically prevented by the code pointer integrity checking proposed by PointGuard. The remaining attacks that corrupt *arbitrary* program variables can be nominally addressed by PointGuard, but require significant manual intervention. Fully automatic PointGuard defense would require canary integrity checking on *all* variables, at which point bounds checking begins to become competitive with integrity checking.

It is interesting to note that the first popular buffer overflow attack (the Morris Worm [21, 37]) used this last category of buffer overflow to corrupt a file name, and yet virtually no contemporary buffer overflow attacks uses this method, despite the fact that none of the current or proposed defenses is strongly effective against this form of attack. It is unclear whether the present dearth of logic-based buffer overflow attacks is because such vulnerabilities are highly unusual, or simply because attacks are easier to construct when code pointers are involved.

5 Conclusions

We have presented a detailed categorization and analysis of buffer overflow vulnerabilities, attacks, and defenses. Buffer overflows are worthy of this degree of analysis because they constitute a majority of security vulnerability issues, and a substantial majority of remote penetration security vulnerability issues. The results of this analysis show that a combination of the StackGuard [14, 9] defense and the non-executable stack defense [19, 18] serve to defeat many contemporary buffer overflow attacks, and that the proposed PointGuard defense will address most of the remaining contemporary buffer overflow attacks. Of note is the fact that the particular form of buffer overflow attack used by Morris in 1987 to "popularize" the buffer overflow technique is both uncommon in contemporary attacks, and not easily defended against using existing methods.

Table 3: Buffer Overflow Attacks and Defenses

		Attack Code Location			
		Resident	**Stack Buffer**	**Heap Buffer**	**Static Buffer**
Code Pointer types	**Activation Record**	StackGuard	StackGuard, Non-executable stack	StackGuard	StackGuard
	Function Pointer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Longjmp Buffer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Other Variables	Manual PointGuard	Manual PointGuard, Non-executable stack	Manual PointGuard	Manual PointGuard

References

[1] Alfred V. Aho, R. Hopcroft, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] Anon. Linux Security Audit Project. `http://lsap.org/`.

[3] Andrea Arcangeli. `xterm` Exploit. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, May 8 1998.

[4] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson. Apache `HTTP` Server Project. `http://www.apache.org`.

[5] Steve Bellovin. Buffer Overflows and Remote Root Exploits. Personal Communications, October 1999.

[6] M. Bishop. How to Write a Setuid Program. *:login;*, 12(1), Jan/Feb 1986. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[7] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[8] Compaq. `ccc` C Compiler for Linux. `http://www.unix.digital.com/linux/compaq_c/`, 1999.

[9] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[10] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, and Jonathan Walpole. Automated Guarding Tools for Adaptive Operating Systems. Work in progress, December 1996.

[11] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998. Brief presentation and poster session.

[12] Crispin Cowan and Calton Pu. Survivability From a Sow's Ear: The Retrofit Security Requirement. In *Proceedings of the 1998 Information Survivability Workshop*, Orlando, FL, October 1998. `http://www.cert.org/research/isw98.html`

[13] Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, September 1998.

[14] 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 *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[15] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, *The SANS Network Security Digest*. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard.

[16] Theo de Raadt and et al. OpenBSD Operating System. `http://www.openbsd.org/`.

[17] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996. `http://www.cs.princeton.edu/sip/pub/secure96.html`.

[18] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[19] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, `http://x10.dejanews.com/getdoc.xp?AN=207344316&CONTEXT=890082637.1567359211&% hitnum=69&AH=1`, January 2 1997.

[20] "DilDog". The Tao of Windows Buffer Overflow. `http://www.cultdeadcow.com/cDc_files/cDc-351/`, April 1998.

[21] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, September 1990.

[22] Chris Evans. Nasty security hole in `lprm`. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, April 19 1998.

[23] Anup K Ghosh, Tom O'Connor, and Gary McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[24] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at `http://www.rational.com/support/techpapers/fast_detection/`.

[25] Alfred Huger. Historical Bugtraq Question. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, September 30 1999.

[26] Richard Jones and Paul Kelly. Bounds Checking for C. `http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html`, July 1995.

[27] Mindcraft. WebStone Standard Web Server Benchmark. `http://www.mindcraft.com/webstone/`.

[28] "Mudge". How to Write Buffer Overflows. `http://l0pht.com/advisories/bufero.html`, 1997.

[29] "Aleph One". Bugtraq Mailing List. `http://geek-girl.com/bugtraq/`.

[30] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[31] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[32] Kurt Roeckx. Bounds Checking Overhead in SSH. Personal Communications, October 1999.

[33] Jim Roskind. Panel: Security of Downloadable

Executable Content. NDSS (Network and Distributed System Security), February 1997.

[34] Fred B. Schneider, Steven M. Bellovin, Martha Branstad, J. Randall Catoe, Stephen D. Crocker, Charlie Kaufman, Stephen T. Kent, John C. Knight, Steven McGeady, Ruth R. Nelson, Allan M. Schiffman, George A. Spix, and Doug Tygar. *Trust in Cyberspace*. National Acadamy Press, 1999. Committee on Information Systems Trustworthiness, National Research Council.

[35] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. `http://millcomm.com/ nate/machines/security/stack-smashing/nate-buffer.ps`, 1997.

[36] Alexander Snarskii. FreeBSD Stack Integrity Patch. `ftp://ftp.lucky.net/pub/unix/local/libc-letter`, 1997.

[37] E. Spafford. The Internet Worm Program: Analysis. *Computer Communication Review*, January 1989.

[38] Synthetix: Tools for Adapting Systems Software. World-wide web page available at `http://www.cse.ogi.edu/DISC/projects/synthetix`.

[39] Herman ten Brugge. Bounds Checking C Compiler. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge/`, 1998.

[40] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 2000.

[41] Rafel Wojtczuk. Defeating Solar Designer Non-Executable Stack Patch. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, January 30 1998.

[42] Tatu Ylonen. SSH (Secure Shell) Remote Login Program. `http://www.cs.hut.fi/ssh`

[43] Anthony C. Zboralski. [HERT] Advisory #002 Buffer overflow in lsof. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, February 18 1999.