The following paper was originally published in the
Proceedings of the 7th USENIX Security Symposium
San Antonio, Texas, January 26-29, 1998

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke,
Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang,
Oregon Graduate Institute of Science & Technology;
Heather Hinton, *Ryerson Polytechnic University*

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton†, Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

Abstract

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to `gcc`), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

*This research is partially supported by DARPA contracts F30602-96-1-0331 and F30602-96-1-0302.

†Ryerson Polytechnic University

1 Introduction

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attack gained notoriety in 1988 as part of the Morris Worm incident on the Internet [23]. Despite the fact that fixing individual buffer overflow vulnerabilities is fairly simple, buffer overflow attacks continue to this day, as reported in the SANS Network Security Digest:

> Buffer overflows appear to be the most common problems reported in May, with degradation-of-service problems a distant second. Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors, before releasing the software, if the vendors had performed elementary testing or code reviews along the way.[4]

The base problem is that, while individual buffer overflow vulnerabilities are simple to patch, the vulnerabilities are profligate. Thousands of lines of legacy code are still running as privileged daemons (`SUID root`) that contain numerous software errors. New programs are being developed with more care, but are often still developed using unsafe languages such as C, where simple errors can leave serious vulnerabilities.

The continued success of these attacks is also due to the "patchy" nature by which we protect against such attacks. The life cycle of a buffer overflow attack is simple: A (malicious) user finds the vulnerability in a highly priv-

ileged program and someone else implements a patch to *that particular attack, on that privileged program*. Fixes to buffer overflow attacks attempt to solve the problem at the source (the vulnerable program) instead of at the destination (the stack that is being overflowed).

This paper presents StackGuard, a systematic solution to the buffer overflow problem. StackGuard is a simple compiler extension that limits the amount of damage that a buffer overflow attack can inflict on a program. Programs compiled with StackGuard are safe from buffer overflow attack, regardless of the software engineering quality of the program.

Section 2 describes buffer overflow attacks in detail. Section 3 details how StackGuard defends against buffer overflow attacks. Section 4 presents performance and penetration testing of StackGuard-enhanced programs. Section 5 discusses some of the abstract ideas represented in StackGuard, and their implications. Section 6 describes related work in defending against buffer overflow attack. Finally, Section 7 presents our conclusions.

2 Buffer Overflow Attacks

Buffer overflow attacks exploit a lack of bounds checking on the size of input being stored in a buffer array. By writing data *past* the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. By far, the most common data structure to corrupt in this fashion is the stack, called a "stack smashing attack," which we briefly describe here, and is described at length elsewhere [15, 17, 21].

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible [14, 13]. For instance, many of the standard C library functions such as `gets` and `strcpy` do not do bounds checking by default.

The common form of buffer overflow exploitation is to attack buffers allocated on the stack. Stack smashing attacks strive to achieve two mutually dependent goals, illustrated in Figure 1:

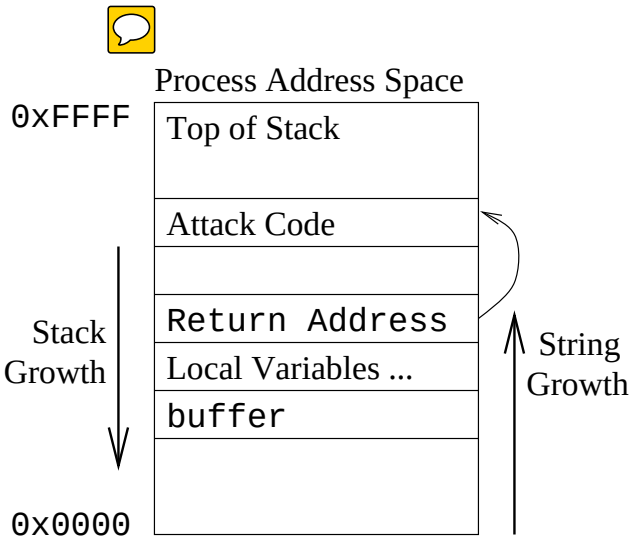

Figure 1: Stack Smashing Buffer Overflow Attack

Inject Attack Code The attacker provides an input string that is actually executable, binary code native to the machine being attacked. Typically this code is simple, and does something similar to `exec("sh")` to produce a `root` shell.

Change the Return Address There is a stack frame for a currently active function above the buffer being attacked on the stack. The buffer overflow changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps to the attack code.

The programs that are attacked using this technique are usually privileged daemons; programs that run under the user-ID of `root` to perform some service. The injected attack code is usually a short sequence of instructions that spawns a shell, also under the user-ID of `root`. The effect is to give the attacker a shell with `root`'s privileges.

If the input to the program is provided from a locally running process, then this class of vulnerability may allow any user with a local account to become `root`. More distressing, if the program input comes from a network connection, this class of vulnerability may allow any user anywhere on the network the ability to become root on the local host. Thus while new instances of this class of attack are not intellectually interesting, they are none the less critical to practical system security.

Engineering such an attack from scratch is non-trivial. Often, the attacks are based on reverse-engineering the attacked program, so as to determine the exact offset from the buffer to the return address in the stack frame, and the offset from the return address to the injected attack code. However, it is possible to soften these exacting requirements [17]:

- The location of the return address can be approximated by simply repeating the desired return address several times in the approximate region of the return address.
- The offset to the attack code can be approximated by prepending the attack code with an arbitrary number of `NOP` instructions. The overwritten return address need only jump into the middle of the field of `NOP`s to hit the target.

The cook-book descriptions of stack smashing attacks [15, 17, 21] have made construction of buffer-overflow exploits quite easy. The only remaining work for a would-be attacker to do is to find a poorly protected buffer in a privileged program, and construct an exploit. Hundreds of such exploits have been reported in recent years [4].

3 StackGuard: Making the Stack Safe for Network Access

StackGuard is a compiler extension that enhances the executable code produced by the compiler so that it detects and thwarts buffer-overflow attacks against the stack. The effect is transparent to the normal function of programs. The only way to notice that a program is StackGuard-enhanced is to cause it to execute C statements with undefined behavior: StackGuard-enhanced programs define the behavior of writing to the return address of a function *while* it is still active.

As described in Section 2, the common form of buffer-overflow attacks are stack smashers. They function by overflowing a buffer that is allocated on the stack, injecting code onto the stack, and changing the return address to point to the injected code. StackGuard thwarts this class of attack by effectively preventing changes to the return address while the function is still active. If the return address cannot be changed, then the attacker has no way of invoking the injected attack code, and the attack method is thwarted.

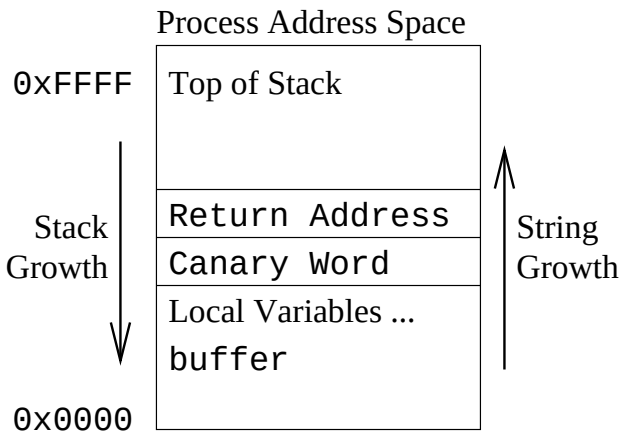

Figure 2: Canary Word Next to Return Address

StackGuard prevents changes to active return addresses by either detecting the change of the return address *before* the function returns, or by completely preventing the write to the return address. Detecting changes to the return address is a more efficient and portable technique, while preventing the change is more secure. StackGuard supports both techniques, as well as adaptively switching from one mode to the other.

Section 3.1 describes how StackGuard detects changes to the return address. Section 3.2 describes how StackGuard prevents changes to the return address. Section 3.3 discusses motives and methods for adaptively switching between techniques.

3.1 Detecting Return Address Change Before Return

To be effective, detecting that the return address has been altered must happen *before* a function returns. StackGuard does this by placing a "canary"[1] word next

[1] A direct descendent of the Welsh miner's canary.

to the return address on the stack, as shown in Figure 2. When the function returns, it first checks to see that the canary word is intact before jumping to the address pointed to by the return address word.

This approach assumes that the the return address is unaltered *IFF* the canary word is unaltered. While this assumption is not completely true in general (stray pointers can alter any word), it *is* true of buffer overflow attacks. The buffer overflow attack method exploits the fact that the return address word is located very close to a byte array with weak bounds checking, so the *only* tool the attacker has is a linear, sequential write of bytes to memory, usually in ascending order. Under these restricted circumstances, it is very difficult to over-write the return address word without disturbing the canary word.

The StackGuard implementation is a *very* simple patch to `gcc` 2.7.2.2. The `gcc function_prologue` and `function_epilogue` functions have been altered to emit code to place and check canary words. The changes are architecture-specific (in our case, `i386`), but since the total changes are under 100 lines of `gcc`, portability is not a major concern. All the changes in the gcc calling conventions are undertaken by the callee, so code compiled with the StackGuard-enhanced `gcc` is completely inter-operable with generic `gcc .o` files and libraries. The additional instructions added to the function prologue are shown in pseudo-assembly form in Figure 3, and the additional instructions added to the instruction epilogue are shown in Figure 4. Section 4 describes testing and performance of this patch.

3.1.1 Randomizing the Canary

The Canary defense is sufficient to stop most buffer overflow attacks that are oblivious to the canary. In fact, simply *changing* the compiler's calling conventions is sufficient to stop most buffer overflow attacks [8]. Most *current* buffer overflow attacks are quite brittle, making specific, static assumptions about the layout of the stack frame. However, it is not very hard for attackers to develop buffer overflows that are insensitive to minor changes in the stack frame layout [17]:

- To adapt to changes in the location of the return address relative to the buffer being overflowed, the attacker can repeat the new value several times in the input string.
- To adapt to imprecision in the offset of the injected code from the current program counter, the attacker can inject attack code consisting of many `NOP`s, and simply jump to somewhere in the middle of the `NOP` sequence. Control flow will then drop down to the attack code.
- To adapt to changes in alignment, the attacker need only guess 4 times at most to get the alignment correct.

It is also possible to write attacks *specifically* designed to overcome StackGuard.[2] There are two ways to overcome the Canary method of detecting buffer overflows:

1. Skip over the canary word. If the attacker can locate a poorly checked copy of an array of structs, which have alignment requirements, and are not big enough to fulfill the alignment requirements while densely packing the array, then it is possible that the copy could occur such that the canary word is in one of the holes left in the array. We expect this form of vulnerability to be rare, and difficult to exploit.
2. Simulate the canary word. If the attacker can easily guess the canary value, then the attack string can include the canary word in the correct place, and the check at the end of the function. If the canary word is completely static, then it is *very* easy to guess. This form of attack is problematic.

To deal with easily-guessed canaries, we use randomly chosen canary values. Our current implementation enhances the `crt0` library to choose a set of random canary words at the time the program starts. These random words are then used as distinct random canary words, one per function in the object code. While it is not *impossible* to guess such a canary value, it is difficult: the attacker must be able to examine the memory image of the running process to get the randomly selected word. Even so, a determined attacker could break such a defense eventually; we discuss our approach to this problem in Section 3.3.

[2] Naturally, none have been found to date :-)

```
    move canary-index-constant into register[5]
    push canary-vector[register[5]]
```

Figure 3: Function Prologue Code: Laying Down a Canary

```
    move canary-index-constant into register[4]
    move canary-vector[register[4]] into register[4]
    exclusive-or register[4] with top-of-stack
    jump-if-not-zero to constant address .canary-death-handler
    add 4 to stack-pointer
    < normal return instructions here>
.canary-death-handler:
    ...
```

Figure 4: Function Epilogue Code: Checking a Canary

3.2 Preventing Return Address Changes With MemGuard

The Synthetix project [18, 1, 2, 24] introduced a notion called "*quasi*-invariants." Quasi-invariants are state properties that hold true for a while, but may change without notice. Quasi-invariants are used to specify *optimistic* specializations: code optimizations that are valid *only* while the quasi-invariants hold. We have extended this work to treat return addresses on the stack as quasi-invariant during the activation lifetime of the function. The return address is read-only (invariant) while the function is active, thus preventing effective buffer overflow against the stack.

MemGuard [3] is a tool developed to help debug optimistic specializations by locating code statements that change quasi-invariant values. MemGuard provides fine-grained memory protection: individual words of memory (quasi-invariant terms) can be designated as read-only, except when explicitly written via the MemGuard API. We have used MemGuard to produce a more secure, if less performant, version of the StackGuard compiler.

MemGuard is used to prevent buffer overflow attacks by protecting a return address when a function is called, and un-protecting the return address when the function returns. The protection and un-protection occur in precisely the same places as the canary placement and checks described in Section 3.1: the `function_prologue` and `function_epilogue` functions. Figure 5 shows the prologue code sequence for MemGuard. The epilogue code sequence is identical, but uses system call 165 instead of 164.

```
push a
push b
move 164 into a
move arg[0] into b
trap 0x80
pop b
pop a
```

Figure 5: Function Prologue Code: Protecting the Return Address With MemGuard

MemGuard is implemented by marking virtual memory pages containing quasi-invariant terms as read-only, and installing a trap handler that catches writes to protected pages, and emulates the writes to non-protected words on protected pages. The cost of a write to a non-protected word on a protected page is approximately 1800 times the cost of an ordinary write. This is an acceptable cost when quasi-invariant terms are in quiet portions of the kernel's address space, and when MemGuard is primarily used for debugging.

This cost is not acceptable when the protected words are located near the top of the stack, next to some of the most frequently written words in the program. MemGuard was originally designed to protect variables within the kernel. To protect the stack, MemGuard had to be extended in several ways:

- Extend VM model to protect user pages.
- Deal with the performance penalties due to "false sharing" caused by frequent writes to words near the return address.
- Provide a light-weight system-call interface to MemGuard. Loading virtual memory hardware is a privileged operation, and so the application process must trap to kernel mode to protect a word.

Most of these extensions are simple software development, but the performance problems are challenging. Fortunately, the Pentium processor has four "debug" registers. These registers can be configured to watch for read, write, and execute access to the virtual address loaded into each register, and generate an exception when such access occurs.

We use these registers as a cache of the most recently protected return addresses. The goal is to eliminate the need for the top-most page of the stack to be read-only, to eliminate page faults resulting from writes to variables at the top of the stack. Because of the locality behavior of stack variables, restoring write privileges to the top of the stack should handle most of the writes to stack variables.

It is only probabilistically true that protecting the four most recent return addresses will capture all protection needs for the top of the stack. However, if the compiler is adjusted to emit stack frames with a minimum size of 1/4 of a page, then it is always true that 4 registers will cover the top page. The time/space trade-off implied by this approach can be continuously adjusted, reducing the minimum size of stack frames to reduce space consumption, and also increasing the probability that the top page of the stack actually will require MemGuard protection, with its associated costs.

3.3 Adaptive Defense Strategies

StackGuard is a product of the Immunix project [11], whose focus is adaptive responses to security threats. Thus we provide an adaptive response to intrusions, switching between the more performant Canary version, and the more robust MemGuard versions of StackGuard.

The basic model of operation for StackGuard is that when a buffer overflow is detected, either by the Canary or by MemGuard, the process is terminated. The process must exit, because an unknown amount of state has already been corrupted at the time the attack is detected, and so it is impossible to safely recover the state of the process. Thus the process exits, using only static data and code, so as to avoid any possible corruption from the attacker.

Replacing the dead process is context-dependent. In many cases, it suffices to just let `inetd` re-start the daemon when a connection requests service. However, if the daemon is not managed by `inetd`, then it may be necessary for a watch-dog process to re-start the daemon, most especially in the case of `inetd` itself.

It is also possible for these re-start mechanisms to adaptively select which form of protection to use *next*. The Canary and MemGuard variants of StackGuard offer different points in the trade-off between security and performance. The Canary version is more performant, while the MemGuard version is more secure (see Section 4). More specifically, the important security vulnerability in the Canary variant is that it is potentially subject to guessing of the canary value. The Canary variant can defend itself against guessing by exiting, and replacing the attacked Canary-guarded daemon with a MemGuard-guarded daemon.

This adaptive response allows systems to run in a relatively high-performance state most of the time, and adaptively switch to a lower-performance, higher-security state when under attack. At worst, the attacker can carry out a degradation-of-service attack by periodically attacking daemons, forcing them to run in the lower-performance MemGuard mode most of the time. However, service is not totally denied, because the daemons continue to function, and the attacker no longer is able to obtain illegitimate privilege via buffer overflow attack.

4 Experimental Results

This section describes experimental evaluation of StackGuard. Subsection 4.1 describes penetration experiments, to show StackGuard's effectiveness in deterring past and future attacks. of Subsection 4.2 describes the performance cost of StackGuard under various circumstances.

4.1 StackGuard Effectiveness

Here we illustrate StackGuard's effectiveness in thwarting stack smashing buffer overflow attacks. StackGuard is intended to thwart generic stack smashing attacks, even those that have not yet appeared. To simulate that, we sought out buffer overflow exploits, and tried them against their intended software targets, with and without protection from StackGuard. Table 1 summarizes these results.

The programs listed in Table 1 are conventionally installed as `SUID root`. If the attacker can get one of these programs to start a shell, then the attacker gets a `root` shell.

In each case, the experiment is to install the vulnerable program `SUID root` (`SUID httpd` for `wwwcount`) and attack it with the exploit. We then re-compile the program with the Canary variant of StackGuard, re-install the StackGuard-enhanced program as `SUID root`, and attack it again with the exploit. We did *not* alter the source code of any of the vulnerable programs at all, and StackGuard has no specific knowledge of any of these attacks. Thus this experiment *simulates* the effect of StackGuard defending against unknown attacks.

In all cases we have studied, both the Canary and the MemGuard variants of StackGuard stopped what would have been an attack that obtains a `root` shell. Several cases deserve special discussion:

`umount 2.5k/libc 5.3.12`: The buffer overflow vulnerability is actually in `libc`, and not in `umount`. Simply re-compiling `umount` with either variant of StackGuard does not suffice to stop the attack. However, when `libc` is also compiled using StackGuard (either variant) then the attack is defeated. Thus for full protection, either the system shared libraries must be protected with StackGuard, or the privileged programs must be statically linked with libraries that are protected with StackGuard.

`SuperProbe`: This attack does not actually attack the function return address. Rather, it over-writes a function pointers in the program that is allocated on the stack. The Canary variant stopped the attack by perturbing the layout of the stack, but an adjusted attack produced a `root` shell even with Canary protection. The MemGuard variant stopped the attack because a return address was in the way of the buffer overflow. Proper treatment of this kind of attack requires an extension to StackGuard, as described in Section 5.4.

`Perl`: Like `SuperProbe`, the `Perl` attack does not attack the function return address. This attack overwrites data structures in the global data area, and thus is not properly a "stack smashing" attack. Permutations in the alignment of the global data area induced by the StackGuard's vector of canary values prevented the attack from working, but a modified form of the attack produced a `root` shell despite Canary protection. MemGuard had no effect on the attack.

`Samba`, `wwwcount`: These buffer overflow vulnerabilities were announced *after* the StackGuard compiler was developed, yet the StackGuard-enhanced versions of these programs were *not* vulnerable to the attacks. This illustrates the point that StackGuard can effectively prevent attacks even against unknown vulnerabilities.

We would like the list of programs studied to be larger. Two factors limit this kind of experimentation:

Obtaining the Exploit: It is difficult to obtain the exploit code for attacking programs. Security organizations such as CERT are reluctant to release exploits, and thus most of these exploits were obtained either from searching the web, or from the bugtraq mailing list [16].

Obtaining Vulnerable Source Code: Buffer overflow attacks exploit specific, *simple* vulnerabilities in popular software. Because of the severe security

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

Table 1: Protecting Vulnerable Programs with StackGuard

risks posed, and the ease of patching the individual vulnerability, new releases appear soon after the vulnerability is publicized. Moreover, the vulnerability is often *not* publicized until it can be announced with a patch in hand. The older vulnerable source code is often not easily available. We have begun archiving source code versions, so that we will be able to add experiments as new vulnerabilities appear.

4.2 StackGuard Overhead

This section describes experiments to study the performance overhead imposed by StackGuard. Note that StackGuard need only be used on programs that are `SUID root`, and such programs are not usually consumers of large amounts of CPU time. Thus it is only necessary that the overhead be sufficiently low that the privileged administrative daemons do not impose a noticeable compute load. The MemGuard and Canary variants of StackGuard impose different kinds of overhead, and so we microbenchmark them separately in Sections 4.2.1 and 4.2.2. Section 4.2.3 presents macrobenchmark performance data.

4.2.1 Canary StackGuard Overhead

The Canary mechanism imposes additional cost at two points in program execution:

- function prologue: there is a small cost in pushing the canary word onto the stack.
- function epilogue: there is a moderate cost in checking that the canary word is intact before performing the function return.

We model this cost as a % overhead per function call. The % overhead is a function of the base cost of a function call, which varies depending on the number of arguments and the return type, so we studied a range of function types.

The experiments seek to discover the % overhead of a function call imposed by StackGuard. We did this by writing a C program that increments a statically allocated integer 500,000,000 times. The base case is just "`i++`", and the experiments use various functions to increment the counter. The results are shown in Table 2. All experiments were performed on a 200 MHz Pentium-S with 512K of level 2 cache, and 128M of main memory.

The "`i++`" is the base case, and thus has no % overhead. The "`void inc()`" entry is a function that does `i++` where `i` is a global variable; this shows the overhead of a zero-argument `void` function, and is the worst-possible case, showing a 125% overhead on function calls. The "`void inc(int *)`" entry is a function that takes an `int *` argument and increments it as a side-effect; this shows that there is 69% overhead on a one-argument `void` function. The "`int inc(int)`" entry is an applicative function that takes an `int` argument, and returns that value + 1; this shows that the overhead of a one-argument function returning an `int` is 80%.

Increment Method	Standard Run-Time	Canary Run-Time	% Overhead
`i++`	15.1	15.1	NA
`void inc()`	35.1	60.2	125%
`void inc(int *)`	47.7	70.2	69%
`int inc(int)`	40.1	60.2	80%

Table 2: Microbenchmark: Canary Function Call Overhead

Numerous other experiments are possible, but they all increase the base cost of function calls, while the cost of the Canary mechanism remains fixed at 7 instructions (see Figures 3 and 4), decreasing the Canary % overhead. Thus these overhead microbenchmarks can be considered an upper-bound on the cost of the Canary compiler.

4.2.2 MemGuard StackGuard Overhead

The MemGuard variant of StackGuard suffers substantial performance penalties compared to the Canary variant, for reasons described in Section 3.2. Section 4.1 showed that the MemGuard variant provides better security protection for stack attacks than the Canary variant (specifically, MemGuard stopped the `SuperProbe` attack, and guessing canary values will not help get past MemGuard). This section measures the cost of that added protection.

The MemGuard variant of StackGuard is still under development, but as of this writing, we have some preliminary results. We have measured the performance of two versions of MemGuard StackGuard:

MemGuard Register This version uses *only* the Pentium's debugging registers for protection, so only the four most recent function calls' return addresses are protected. This version pays no penalty for page protection faults induced by protecting the stack with virtual memory protection. **NOTE**: this version stopped **all** of the stack smashing attacks that we tested[3].

MemGuard VM This version uses the virtual memory page protection scheme described in Section 3.2. It has not fully exploited the optimization of using the debugging registers as a cache, to keep the top page of the stack writable. Thus this version suffers *substantial* performance penalties due to a large number of page protection faults.

Table 3 shows the overhead costs for the MemGuard variant of StackGuard. Because of the use of a heavyweight system call to access privileged hardware for protection, function calls slow down by $70\times$ for the MemGuard Register protection. The additional penalty of page protection fault handling for false sharing of the page on the top of the stack raises the cost of function calls by $160\times$. Proper use of the debugging registers as a cache for the VM mechanism should bring the costs in line with the MemGuard Register costs.

4.2.3 StackGuard Macrobenchmarks

Sections 4.2.1 and 4.2.2 present microbenchmark results on the additional cost of function calls in programs protected by StackGuard. However, these measurements are *upper bounds* on the real costs of running programs under StackGuard; the true penalty of running StackGuard-enhanced programs is the overall cost, not the microbenchmark cost. We have benchmarked two programs: `ctags`, and the StackGuard-enhanced `gcc` compiler itself.

The `ctags` program constructs an index of C source code. It is 3000 lines of C source code, comprising 68 separate functions. When run over a small set of source files (78 files, 37,000 lines of code) with a hot buffer cache, `ctags` is completely compute-bound. When run over a large set of files (1163 files, 567,000 lines of code) `ctags` it is still compute-bound, because of the large

[3] Except `Perl`, which is not really a stack smashing attack.

Increment Method	Standard Run-Time	MemGuard Register Run-Time	% Overhead	MemGuard VM Run-Time	% Overhead
`i++`	15.1	15.1	NA	NA	NA
`void inc()`	35.1	1808	8800%	34,900	174,300%
`void inc(int *)`	47.7	1820	5400%	40,420	123,800%
`int inc(int)`	40.1	1815	7000%	41,610	166,200%

Table 3: Microbenchmark: MemGuard Function Call Overhead

amount of RAM in our test machine.

On a smaller machine, the test becomes I/O bound, consuming 50% of the CPU's time, so it is approximately balanced. While the Canary variant still consumes more CPU time than the generic program, it is overlapped with disk I/O, and the program completes in the same amount of real time. The MemGuard variants consume so much CPU time that the program's real time is dramatically impacted.

Table 4 shows `ctag`'s run-time in these two cases. The Canary variant's performance penalties are moderate, at 80% for the small case, and 42% for the large case. The MemGuard Register penalties are substantial, at 1100% for the small case, and 1000% for the large case. The MemGuard VM performance penalties are prohibitive, at 46,000% for the small case, and 36,000% for the large case.

Table 5 shows a similar experiment for the run-time of a StackGuard-protected `gcc` compiler. We thus use a StackGuard-protected `gcc` to measure the performance cost of StackGuard for a large and complex program. To be clear, the experiment measures the cost of running `gcc` protected by StackGuard, and only incidentally measures the cost of adding StackGuard protection to the compiled program.

Table 5 shows the time to compile `ctags` using `gcc` enhanced with StackGuard. Because there is more computation per function call for `gcc` than `ctags`, this time the costs are lower. The Canary version consumes only 6% more CPU time, and only 7% more real time. The MemGuard variants benefited as well; the Register version's additional real time cost is 214%, and the VM version's additional cost is 5100%.

Recall that the StackGuard protective mechanism is only necessary on privileged administrative programs. Such programs present only a minor portion of the compute load on a system, and so the StackGuard overhead will have only a modest impact on the total system load. Thus the overhead measured here could be considered within reason for heightened security, without a significant change in the administrative complexity of the system. We discuss administration of StackGuard in Section 5.

5 Discussion

This section discusses some of the abstract ideas represented in StackGuard, and their implications. Section 5.1 describes how StackGuard can help defend against *future* attacks. Section 5.2 describes potential administration and configuration techniques for systems using StackGuard. Section 5.3 describes some possible performance optimizations. Section 5.4 describes future enhancements to StackGuard.

5.1 Defending Against Future Attacks

Fundamentally, the attacks that StackGuard prevents are not very interesting. They are serious security faults that result from minor programming errors. Once discovered, fixing each error is easy. The significant contribution that StackGuard makes is not only that it patches a broad collection of existing faults, but rather that it patches a broad collection of *future* faults that have yet to be discovered. That StackGuard defeats the attacks against `Samba` and `wwwcount` discovered *after* StackGuard was produced is testament to this effect.

Input	Version	User Time	System Time	Real Time
37,000 lines	Generic	0.41	0.14	0.55
	Canary	0.68	0.13	0.99
	MemGuard Register	1.30	5.45	6.84
	MemGuard VM	16.5	238.0	255.1
586,000 lines	Generic	7.74	2.08	10.2
	Canary	11.9	2.07	14.5
	MemGuard Register	21.1	91.5	115.0
	MemGuard VM	236	3482	3728

Table 4: Macrobenchmark: `ctags`

Version	User Time	System Time	Real Time
Generic	1.70	0.12	1.83
Canary	1.79	0.16	1.96
MemGuard Register	2.22	3.35	5.76
MemGuard VM	8.17	87.7	96.2

Table 5: Macrobenchmark: `gcc` of the `ctags` program

Using StackGuard does not eliminate the need to fix buffer overflow vulnerabilities, but by converting `root` vulnerabilities into mild degradation-of-service attacks, it does eliminate the urgency to fix them. This gives software developers the breathing room to fix buffer overflows when it is convenient (i.e. when the next release is ready) rather than having to rush to create and distribute a patch. More importantly, StackGuard eases security administration by relieving the system administrators of the need to apply these patches as soon as they are released, often several times a month.

5.2 Administration and Configuration

The adaptive response described in Section 3.3 requires management: StackGuard causes programs to give notice that they need to be replaced because they have been (unsuccessfully) attacked, but does not make policy about what version, if any, to replace it with.

Different policy decisions will have different implications; switching to a higher level of protection will drastically reduce performance, yet failure to switch can lead to successful penetration via guessing. The decision to revert to the more performant, less secure mode is even more difficult, because the attacker may try to *induce* such a switch. Making the right choice, automatically, is challenging. We propose to create a small, domain-specific language [19] for specifying these policy choices.

StackGuard comes with a performance price, and can be viewed as an insurance policy. If one is *very* sure that a program is correct, i.e. contains no buffer overflow vulnerabilities because it has been verified using formal methods, or a validation tool [9], then the program can be re-compiled and installed without benefit of StackGuard.

StackGuard offers powerful protection of any program compiled with the StackGuard compiler, but does nothing for programs that have not been thus compiled. However, tools such as COPS [7], which search for programs that should not be `SUID root`, can be configured to look for programs that are `SUID root`, and have not been compiled using StackGuard or some other security verification tool [9]. If COPS reports that all `SUID root` programs on a machine have been protected, then one can have some degree of assurance that the machine is not vulnerable to buffer overflow attacks.

5.3 Performance Optimizations

Section 4.2.2 mentions that a light-weight trap to kernel mode can reduce the overhead of the MemGuard mechanism. However, it is also possible for the compiler to optimize StackGuard performance, both for the MemGuard and Canary variants.

If it is the case that no statement takes the address of any stack variable in the function `foo`, then `foo` does not need StackGuard protection. This is because any buffer overflow must attack an array, which is always a pointer. If an attack seeks to alter a variable in a function above `foo` on the stack, then it must come from below `foo`. But to get to the variable above `foo` it would have to go through the StackGuard protection that necessarily exists on the function below `foo` because of the array being overflowed.

The information regarding whether any variable has been aliased is already available in `gcc`, so it should be a simple matter to turn StackGuard protection off for functions that do not need it. We are working on this optimization, and expect to have it available in a future release of StackGuard.

5.4 Future Work

StackGuard defends against stack smashing buffer overflow attacks that over-write the return address and inject attack code. While this is the most common form of buffer overflow attack, it is not the only form, as illustrated by `SuperProbe` in Section 4.1.

In the general case, buffer overflow attacks can write arbitrary data to arbitrary pieces of process state, with arbitrary results limited only by the opportunities offered by buggy programs. However, some data structures are far easier to exploit than others. Notably, function pointers are highly susceptible to buffer overflow attack. An attacker could conceivably use a buffer overflow to overwrite a function pointer that is on the heap, pointing it to attack code injected into some other buffer on the heap. The attack code need not even overflow its buffer.

We propose to treat this problem by extending StackGuard to protect other data sensitive structures in addition to function return addresses. "Sensitive data structures" would include function pointers, as well as other structures as indicated by the programmer, or clues in the source code itself.

This extension highlights a property of StackGuard, which is that it is "destination oriented." Rather than trying to prevent buffer overflow attacks at the source, StackGuard strives to defend that which the attacker wants to alter. Following the notion that a TCB should be small to be verifiable (and thus secure) we conjecture that the set of data structures needing defending is smaller than the set of data structures exposed to attackers. Thus it should be easier to defend critical data structures than to find all poorly defended interfaces.

6 Related Work

There have been several other efforts pertinent to the problem of buffer overflow attacks. Some are explicitly directed at the security problem, while others are more generally concerned with software correctness. This section reviews some of these projects, and compares them against StackGuard. The result is not a conclusion of which approach is better, but rather a description of the different trade-offs that each approach provides.

6.1 Non-Executable Stack

"Solar Designer" has developed a Linux patch that makes the stack non-executable [6], precisely to address the stack smashing problem. This patch simply makes the stack portion of a user process's virtual address space non-executable, so that attack code injected onto the stack cannot be executed. This patch offers the advantages of *zero* performance penalty, and that programs work and are protected without re-compilation. However, it does necessitate running a specially-patched kernel, unless this extension is adopted as standard.

This patch was non-trivial and non-obvious, for the following reasons:

- `gcc` uses executable stacks for function trampolines

for nested functions.

- Linux uses executable user stacks for signal handling.

- Functional programming languages, and some other programs, rely on executable stacks for run-time code generation.

The patch addresses the problem of trampolines and other application use of executable stacks by detecting such usage, and permanently enabling an executable stack for that process. The patch deals with signal handlers by dynamically enabling an executable stack only for the duration of the signal handler. Both of these compromises offer potential opportunities for intrusion, e.g. a buffer overflow vulnerability in a signal handler.

In addition to the above vulnerabilities, making the stack non-executable fails to address the problem of buffer overflow attacks that do not place attack code on the stack. The attacker may inject the attack code into a heap-allocated or statically allocated buffer, and simply re-point a function return address or function pointer to point to the attack code. This is exactly the kind of attack brought against `Perl` as described in Section 4.1, and a non-executable stack is no more effective than the current StackGuard in stopping it.

The attacker may not even need to inject attack code at all, if the right code fragment can be found within the body of the program itself. Thus additional protection for critical data structures such as function pointers and function return addresses, as described in Section 5.4.

6.2 FreeBSD Stack Integrity Check

Alexander Snarskii developed a FreeBSD patch [22] that does similar integrity checks to those used by the Canary variant of StackGuard. However, these integrity checks were non-portable, hard-coded in assembler, and embedded in `libc`. This method protects against stack smashing attacks inside `libc`, but is not as general as StackGuard.

6.3 Array Bounds Checking for C

Richard Jones and Paul Kelly have developed a `gcc` patch [12] that does full array bounds checking for C programs. Programs compiled with this patch are compatible with ordinary `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 $30\times$ slowdown. Since the slowdown is proportionate to pointer usage, which is quite common in privileged programs, this performance penalty is particularly unfortunate.

However, this method is strictly more secure than StackGuard, because it will prevent all buffer overflow attacks, not just those that attempt to alter return addresses, or other data structures that are perceived to be sensitive (see Section 5.4). Thus we propose that programs compiled with the bounds-checking compiler be treated as the "backing store" for MemGuard-protected programs, just as MemGuard-protected programs are the back-up plan for Canary-protected programs (see Section 3.3).

6.4 Memory Access Checking

Purify [10] is a debugging tool for C programs with memory access errors. Purify uses "object code insertion" to instrument *all* memory accesses. The approach is similar to StackGuard, in that it does integrity checking of memory, but it does so on each memory access, rather than on each function return. As a result, Purify is both more general and more expensive than StackGuard, imposing a slowdown of 2 to 5 times the execution time of optimized code, making Purify more suitable for debugging software. StackGuard, in contrast, is intended to be left on for production use of the compiled code.

6.5 Type-Safe Languages

All of the vulnerabilities described here result from the lack of type safety in C. If the only operations that can be performed on a variable are those described by the type, then it is not possible to use creative input applied to variable `foo` to make arbitrary changes to the variable `bar`.

Type-safety is one of the foundations of the Java security model. Unfortunately, *errors* in the Java type checking system are one of the ways that Java programs and Java virtual machines can be attacked [5, 20]. If the correctness of the type checking system is in question, then programs depending on that type checking system for security benefit from these techniques in similar ways to the benefit provided to type-unsafe programs. Applying StackGuard techniques to Java programs and Java virtual machines may yield beneficial results.

7 Conclusions

We have presented StackGuard, a systematic compiler tool that prevents a broad class of buffer overflow security attacks from succeeding. We presented both security and performance analysis of the tool. Because the tool is oblivious to the specific attack and vulnerability being exploited, it is expected that this tool will also be able to stop buffer overflow attacks that have yet to be discovered, reducing the need for constant, rapid patching of software to stay secure.

In its most basic form, the tool requires only recompilation to make a program largely secure against buffer overflow attacks. In more elaborate forms, it provides an adaptive response to buffer overflow attacks, allowing systems to be configured to trade performance for survivability. We concluded with discussion on how to generalize these techniques to other areas of security vulnerability.

8 Availability

StackGuard is a small set of patches to `gcc`. We are releasing StackGuard under the Gnu Public License, while retaining copyright to OGI. StackGuard is available both as a patch to `gcc 2.7.2.2`, and as a complete `tar` file, at this location: `http://www/cse.ogi.edu/DISC/projects/immunix/StackGuard/`.

References

[1] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[2] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization Classes: An Object Framework for Specialization. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, Seattle, WA, October 27-28 1996.

[3] Crispin Cowan, Dylan McNamee, Andrew Black, Calton Pu, Jonathan Walpole, Charles Krasic, Renaud Marlet, and Qian Zhang. A Toolkit for Specializing Production Operating System Code. Technical Report CSE-97-004, Dept. of Computer Science and Engineering, Oregon Graduate Institute, March 1997.

[4] 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.

[5] 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.

[6] "Solar Designer". Non-Executable User Stack. http://www.false.com/security/linux-stack/.

[7] D. Farmer. The COPS Security Checker System. In *Summer 1990 USENIX Conference*, page 165, Anaheim, CA, June 1990. http://www.trouble.org/cops/.

[8] Stephanie Forrest, Anil Somayaji, and David. H. Ackley. Building Diverse Computer Systems . In *HotOS-VI*, May 1997.

[9] Virgil Gligor, Serban Gavrila, and Sabari Gupta. Penetration Analysis Tools. Personal Communications, July 1997.

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

[11] Immunix. Adaptive System Survivability. http://www.cse.ogi.edu/DISC/projects/immunix, 1997.

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

[13] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A re-examination of the Reliability of UNIX Utilities and Services. Report, University of Wisconsin, 1995.

[14] B.P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):33–44, December 1990.

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

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

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

[18] 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.

[19] Calton Pu, Andrew Black, Crispin Cowan, Jonathan Walpole, and Charles Consel. Microlanguages for Operating System Specialization. In *SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[20] Jim Roskind. Panel: Security of Downloadable Executable Content. NDSS (Network and Distributed System Security), February 1997.

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

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

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

[24] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, Atlanta, GA, October 1997.