Protecting Systems from Stack Smashing Attacks with StackGuard

Crispin Cowan, Steve Beattie, Ryan Finnin Day,
Calton Pu, Perry Wagle, and Erik Walthinsen
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

The StackGuard compiler provides robust automatic protection against the all-too-common problem of stack smashing vulnerabilities. However, this protection is only provided for programs and libraries that are re-compiled with StackGuard. Thus protecting an entire system requires that all potentially vulnerable programs be re-compiled to assure that an attacker cannot exploit a stack smashing vulnerability to gain privilege on the system. This paper describes securing a Linux distribution against stack smashing attacks by re-compiling all of the C software from source code using the StackGuard compiler. We present our experience re-compiling 526 packages from source code, and our experience deploying and using the resultant system.

1 Introduction

StackGuard is an extension to **`gcc`** that provides an integrity check for function call activation records, making programs largely immune to stack smashing attacks [5]. This paper describes our experiences protecting an entire host system by re-compiling all potentially vulnerable programs with StackGuard. This form of protection is necessary because of the broad proliferation of stack smashing vulnerabilities. Since the Morris Worm first illustrated stack smashing [25] literally thousands of buffer overflow vulnerabilities have been discovered in security-sensitive code, and continue to be discovered to this day:

> 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.[6]

This work supported by DARPA grant F30602-96-1-0331.

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

The continued success of these attacks is also due to the “patchy” way we protect against such attacks. In the life cycle of a buffer overflow attack a (malicious) user discovers the vulnerability in a highly privileged program and someone else implements a patch to defend against *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 being overflowed).

To escape this cycle, some groups have undertaken a systematic audit of their source code to look for potential security vulnerabilities, including stack smashing vulnerabilities [7]. However, such work is expensive, and rarely appreciated by customers [4], so few have followed this laudable path, as evidenced by the appearance of a new stack smashing vulnerability in the 2nd and 3rd releases of Microsoft’s Internet Explorer version 4. Code audits are also not necessarily totally effective, in that a subtle stack smashing vulnerability was discovered in the **`lprm`** utility *after* it had been audited for vulnerabilities [13].

StackGuard was developed to address these problems: programs compiled with StackGuard are largely immune to stack smashing attacks. However, effectively protecting a *system* requires that *all* security sensitive programs be compiled with StackGuard. This paper describes our experience doing a complete re-build of all the programs and all the shared libraries in a Linux distribution [12]. By re-compiling all of the programs in a distribution, we can ensure that all of the programs that run as **`root`** (either because they are **`setuid root`**, or because **`root`** runs them directly) are protected by StackGuard.

Section 2 describes stack smashing attacks. Section 3 reviews the StackGuard defense against stack

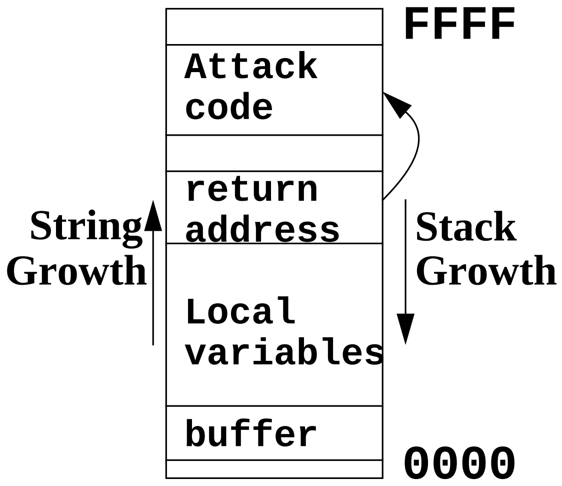

Figure 1: Stack Smashing Attack

smashing attacks. Section 4 presents the procedure for re-building the large set of packages in a Linux distribution. Section 5 describes the impact of StackGuard protection on the system's operation. Section 6 discusses the effective protection provided by these efforts. Section 7 covers related work. Section 8 discusses the availability of a StackGuard-protected Linux distribution. Finally, Section 9 presents our conclusions.

2 Stack Smashing Attack

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 [20, 21, 23].

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the C programmers culture encourages a performance-oriented style that avoids error checking [18, 17]. For instance, many standard C library functions such as **`gets`** and **`strcpy`** do not do bounds checking.

The common form of buffer overflow exploitation is to attack buffers allocated on the stack. The attacker has two mutually dependent goals, illustrated in Figure 1 :

Inject Attack Code: The attacker provides an input string that contains executable, binary code for the machine being attacked. This code typically does something simple such as **`exec("sh")`** to produce a **`root`** shell.

Change the Return Address: There is a stack frame for a currently active function above the stack buffer being attacked. 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 run under the **`root`** user-ID. 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 program input is provided locally, then this class of vulnerability may allow any user with a local account to become **`root`**. 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.

Often the attacks are based on reverse-engineering the attacked program 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, the reverse-engineering has been reduced to a cook book [21]:

- 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 some **`NOP`** instructions. The overwritten return address need only jump into the middle of the field of **`NOP`**s to hit the target.

Cook-book descriptions of stack smashing attacks [20, 21, 23] have made building buffer-overflow exploits easy. The only remaining work for a would-be attacker 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 [6].

The implication for system administrators is that they *must* keep up with system patches. When a stack smashing vulnerability is announced, it is very shortly followed by "script kids" trying out their new toy on various hosts to see who has and has not deployed a patch. This has the effect of turning patches and upgrades from a casual "when it's convenient" task into an urgent task. StackGuard's goal is to make these patches less urgent.

3 StackGuard Defense

StackGuard is a compiler enhancement to protect programs against stack smashing attacks [5, 3]. The StackGuard code generator produces programs that defend themselves against stack smashing attack by doing integrity checks on the stack prior to returning from function calls. Section 3.1 briefly recounts the StackGuard integrity checking mechanism, and Section 3.2 describes some recent advances in the StackGuard mechanism.

3.1 StackGuard Integrity Check for Activation Records

StackGuard seeks not to prevent stack smashing attacks from occurring at all, but rather to prevent the victim program from executing the attacker's injected code. StackGuard does this by detecting that the return address has been altered *before* the function returns. The detection method is to place a "canary"[1] word next 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.

Inspecting for the canary word checks the integrity of the stack for stack smashing attacks. Because the attacker's tool in stack smashing is a sequential byte copy caused by a buffer overflow, inspecting the word immediately below the return address is sufficient to detect tampering with the return address. To prevent the attacker from guessing the canary value and placing it in the canary word's location during the buffer overflow, the canary value is a 32-bit random number chosen at the time the program starts.

The StackGuard implementation is a *very* simple patch to **`gcc`** 2.7.2.3 [26]. 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 to the **`gcc`** code generator are under 100 lines, 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 3.

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

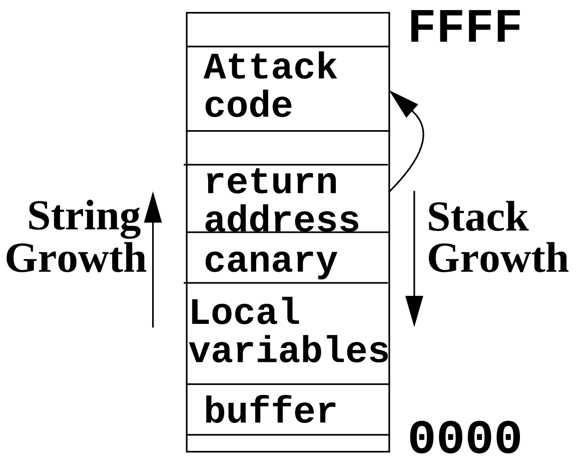

Figure 2: StackGuard Defense Against Stack Smashing Attack

3.2 Canary Integrity

The canary word provides an integrity check only so long as the attack cannot be performed without altering the canary word. If the attack can proceed *without* altering the canary value, either by carefully stepping

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

Figure 3: Function Prolog 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
    return from function
canary-death-handler:
    ...
```

Figure 4: Function Epilog Code: Checking a Canary

```
#!/usr/bin/perl
open(RPMDB,"rpm -qa --queryformat '[%{filemodes:perms} %{fileusername} %{=name} %{filenames} \n]' |");
while(<RPMDB>) {
    # print;
    ($perms, $root, $package, $file) = split;
    if (($perms =~ "s") || ($perms =~ "S")) {
    next if ($perms =~ "d");
    next if ($root !~ "root");
    print(" $perms, $root, $package, $file \n");
    }
}
```

Figure 5: Perl Script to Search for `setuid root` RPM Packages

over the canary word, or by including the canary word in the attack string, then the StackGuard integrity check produced by **`function_epilogue`** will fail to detect the attack.

The initial release of StackGuard protected the canary value by choosing a 32-bit random number as a canary value at program **`exec()`** time, making it intractable for the attacker to guess the canary value [5]. A subsequent version of StackGuard used a null as a canary value (an idea proposed by "der Mouse" [9]) since most string copying functions (the stack smasher's major tool) terminate when they encounter a null, it was reasoned that the attacker could not simultaneously deposit a null in the canary's location, *and* move on to alter the return address above the canary.

However, not all string functions terminate when they encounter a null. For instance **`gets()`** only terminates on a newline or an EOF. We have extended the null-canary mechanism to produce the "terminator" canary: a 32-bit word comprised of a null byte, a carriage return (**`0x0D`**), a line-feed (**`0x0A`**), and an "EOF" in the **`libc`** representation (**`0xFF`**). Most string copying functions will halt when they encounter one of these bytes.

The random canary is impervious to all string operations, and not just those that terminate on the "usual" termination symbols, and thus is more secure than the terminator canary. Conversely, the terminator canary is faster than the random canary check because it does not have to look up the current canary value. The terminator canary can be used to produce relocatable code, and thus protect shared libraries.

The StackGuard compiler thus offers command line options to select either the terminator or random canary mechanism. If complete random canary protection is desired, then all libraries must be compiled with random canaries, and the program must be statically linked. However, if terminator canaries are acceptable, then shared libraries with terminator canary protection can be used, and the program itself can be compiled with either option.

4 Securing a Host with StackGuard

Section 3 describes how StackGuard protects a single program from being hijacked by the attacker. Our general goal is to prevent the attacker from raising their privileges on the host, which requires that *all* potentially vulnerable programs be protected from stack smashing. This section describes our efforts to protect all potentially vulnerable programs in a Red Hat 5.1 Linux distribution. Section 4.1 discusses which programs should be protected. Section 4.2 describes how we re-built these programs.

4.1 Which Programs are Vulnerable

Our goal is to ensure that any program an attacker might try to stack smash to gain privileges is protected by StackGuard. In principle, this is the set of programs that have more privilege than the attacker and are exposed to attacker input. In practice, that set is so large and ill-defined that the safest practice is to StackGuard *everything* on the system, lest a program gets overlooked and the attacker can exploit a vulnerability.

To illustrate why it is difficult to select the programs to be protected, consider the problem of finding all programs that run as "**`root`**."[1] One can quickly find all the programs that are installed **`setuid root`** with the following command:

```
find / -perm +4000 -user 0 -ls
```

With some more effort, one can find the set of *potential* **`setuid root`** programs by inspecting a repository of RPM files, as shown in Figure 5. Caveat:

1.More generally, we are interested in processes running as *any* user-ID that has priviliges of interst to the attacker, e.g. **`httpd`** can change web content, and any user-ID is more priviliged than an anonymous remote attacker.

pre- and post-install scripts in the RPM may **`chmod`** executables, which will not be found by the script in Figure 5. No such scripts have been found.

However, these techniques only capture the programs that are explicitly marked in the file system as being **`setuid root`**, and miss those programs that root may run via other means, such as the **`/etc/rc.d*`** scripts, **`/etc/crontab`**, **`/etc/inetd.conf`**, and anything an administrator may run directly from a **`root`** shell. These programs are also vulnerable, especially if they are daemons that run constantly taking input from arbitrary users.

There is no reliable way to detect *all* of the programs that **`root`** may run, because there is no marker in the file system that indicates that **`root`** is permitted to run the program. On the contrary, **`root`** is explicitly permitted to run *all* executable files. At best, one can use heuristic techniques such as inspection of the **`/etc/*`** files, and instrumenting a kernel to report all the programs that **`root`** runs over a substantial period of time. Thus we concluded that StackGuarding everything was the only truly secure option. Section 5 describes the impact of this decision on system compatibility and performance.

4.2 Re-compiling Programs and Libraries

Re-compiling all of the software included in a full Linux distribution was an interesting exercise, because it required two software technologies that we did not have:

- emit code suitable for shared libraries
- a complete re-build environment

Producing shared library code required a compiler enhancement so that StackGuard could emit PIC (Position Independent Code). PIC is problematic for StackGuard using the random canary method (see Section 3.2) because the shared library code must reference the *per process* random canary table, which it cannot do in PIC mode because it lacks global offsets. The "terminator canary" enhancement eliminated the dependence on the per process canary table, allowing StackGuard to emit PIC code, and thus support shared libraries.

Creating a complete build environment was also challenging. All of the source files, header files, other files, and utilities must be installed in the same location as that assumed by the program author. To minimize the problems of configuring the system the way a source package expects, we selected the Red Hat 5.1 Linux distribution for its RPM feature.

RPM (Red Hat Package Manager) goes a long way towards automating system configuration. An RPM package includes all of the files a program needs (much like a **`tar`** ball) but also includes a database of what packages have been installed, provides additional scripts to be run when packages are to be installed and uninstalled, and explicitly declares dependencies on other packages and files.

A special SRPM (Source RPM) is used to package source code. SRPMs, unfortunately, do *not* provide the any *build-time* dependency information. Thus one has to install *all* binary RPM packages to ensure that all required files will be present.

Installing everything in turn creates problems. RPM packages typically turn on services at boot time by editing a file in **`/etc/rc.d/*`**. Many of these services present security vulnerabilities. To prevent intrusion and corruption *while* constructing a secure system, the host should be disconnected from the network until the vulnerable services have been turned off.

We found that autoconf scripts ("**`./configure`**") could be brittle, causing problems by making odd decisions that are hard to detect. For instance, one such script scanned assembly output for keywords that happen to show up in the comments produced by the StackGuard code generator.

In another example, the **`glibc-2.0.7-13`** package tests for the existence of a C compiler option by compiling a trivial program with all default libraries suppressed, and looking for error output. This procedure causes the implicit dependence on StackGuard's **`canary_death_handler()`** to fail, producing an error.[1] Amending the option detection procedure for **`glibc`** fixed this problem.

Once the complete set of binary and source RPMs are installed, a for loop performing "**`rpm -ba $(package}.spec`**" (build binary and source packages after doing the prep, build, and install stages) is used to build the packages. Since the SRPMs do not have build dependency information, we iterate the build loop until the binary packages cease changing.

Unfortunately, some packages were not assembled correctly. In one trivial instance, an SRPM package was missing an icon image, and thus failed to build correctly. In a more serious instance, the XFree86 package moves **`/usr/X11R6`** to **`/usr/X11R6.$$`** and replaces it, but fails to put the original **`/usr/X11R6`** directory back, despite the fact that the package does not own the **`/usr/X11R6`** directory. This is problematic, because the original **`/usr/X11R6`** directory contained files that were *not*

1.Standard gcc requires 3 symbols to compile the null program: **`_start()`**, **`__eh_pc`**, and **`__throw()`**. StackGuard adds a 4th requirement: **`canary_death_handler()`**.

owned by the XFree86 package, so this build procedure breaks *other* packages, including those that have already been built.[1]

5 System Impact

A StackGuard-protected system is not noticeably different from an un-protected system. It functions identically to an un-protected system, unless you try to stack-smash it. This paper was written on a StackGuard-protected system, and the talk will be presented on a StackGuard-protected notebook computer. It also does not “feel” slower, however we would like a more qualitative analysis of the performance cost of StackGuard protection. Section 5.1 presents performance measures of StackGuard protection for the SSH encryption system [27] and in Section 5.2 we measure the StackGuard performance impact on the Apache web server [2], both of which are broadly used, vulnerable to attack, have available source, and are quantifiably performance-intensive. Section 5.3 describes the trace incompatibilities introduced by StackGuard.

5.1 StackGuard Cost in SSH Bandwidth

SSH [27] provides strongly authenticated and encrypted replacements for the Berkeley **`r*`** commands. For instance, while **`rcp`** copies a file from one machine to another as clear text, **`scp`** copies the file by encrypting it on the source machine, and decrypting it on the destination machine.

SSH is a primary candidate for StackGuard protection because it is a crucial part of the security perimeter. The **`sshd`** daemon mediates requests for login, execution, and copying. If **`sshd`** can be stack-smashed, then the attacker can completely bypass all of SSH’s strong authentication and encryption features.

Encryption and decrypting are done in software using one of several ciphers, IDEA being the default. Software encryption is performance-intensive, and may limit the bandwidth of data communications done via SSH. We measure the performance costs of StackGuard protection by measuring the bandwidth of a large-file copy operation via **`scp`** using the following command:

```
scp bigsource localhost:bigdest
```

`bigsource` file is a 10 MB test file. The “**`locahost:`**” clause forces the copy to go via the SSH encrypted interface, but also bypasses the cost of using the LAN hardware, allowing us to focus on the StackGuard-induced slowdown of encryption as much as possible. The test was run on a Pentium 200 MHz machine running Red Hat 5.1 Linux, with 64 MB of RAM, using **`ssh 1.2.25`**.

We measured the performance impact both by measuring the wall-clock elapsed time of the copy operation, and by observing the bandwidth of the copy as reported by **`scp`**. The compute time of the scp command is not useful, because most of the encrypt/decrypt work is done behind the scenes by other processes. 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).

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.

5.2 StackGuard Cost in the Apache Web Server

The Apache web server [2] is also clearly a candidate for StackGuard protection. While it is not normally run as **`root`**, Apache is the primary mediator for access to all content on a web server, and also must accept highly complex input from both the client machines and the web content it manages. 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 impact of StackGuard protection for Apache by measuring Apache’s performance using the WebStone benchmark [19], 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 1.

As in Section 5.1, 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

1. That the failure induced by XFree86 occurs very *late* in the entire build procedure just adds to frustration :-) This problem is fixed in the current XFree release.

Table 1: 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

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.

5.3 Incompatibilities

The first StackGuard incompatibility is that it is not capable of compiling the Linux kernel. StackGuard changes the format of a stack activation record, and introduces a dependency on the **`canary_death_handler()`** function. It is thus necessary to switch to a standard compiler to build kernels.

The only other incompatibility found to date is that there is an unknown bug in manifested by the StackGuarded **`ld.so`** (dynamic linker). When the StackGuarded **`ld.so`** is present, then certain popular binary applications (Netscape, Acroread, Star Office 4.0) that depend on **`libc5`** will fail. The precise nature of the bug is difficult to determine, since it is only manifested by binary-only applications. Replacing the StackGuarded **`ld.so`** with a generic **`ld.so`** completely solves this problem, and no other compatibility problems have been found. StackGuard is in production on the author's laptop, our group's file server, and has been downloaded by at least 1000 separate individuals. Other than the **`ld.so`** problem, no incompatibilities have been found.

6 Protection

An exhaustive demonstration that a system is completely invulnerable, even to the limited scope of attacks represented by stack smashing, is difficult and beyond the scope of this paper. Such a demonstration requires us to show that *all* possible entries are protected, while the attacker need only find *one* unprotected program. Rather, we will show the degree of protection offered by StackGuard by examining the behavior of programs known to be vulnerable to stack smashing attacks.

Previously [5], we reported StackGuard's penetration resistance when exploits were applied to various vulnerable programs, reproduced here in Table 2. To test the effectiveness of the protections applied in this paper, we applied new exploits against vulnerabilities found in **`XFree86-3.3.2-5`** [1] and **`lsof`** [28]. When applied to an un-protected Red Hat 5.1 Linux machine, the exploits yielded root shells. When applied to the identical system protected as described in this paper, the exploit just produced a StackGuard intrusion alert in **`syslog`**, but no penetration was achieved.

7 Related Work

This section describes related work in protecting systems against stack smashing attacks. Section 7.1 and Section 7.2 describe techniques specifically designed to protect a system against stack smashing attacks, while the remaining sections describe programming techniques to reduce or eliminate the vulnerabilities that enable stack smashing attacks.

7.1 Non-Executable Stack

Casper Dik and "Solar Designer" have developed Solaris and Linux patches, respectively, that make the stack non-executable [10, 11], precisely to address the stack smashing problem. These patches simply make the stack portion of a user process's virtual address space non-executable, so that attack code injected

Table 2: 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

onto the stack cannot be executed. They offer the advantages of *zero* performance penalty, and that programs work and are protected without re-compilation. However, they do necessitate running a specially-patched kernel, unless this approach is adopted as a standard; Solaris 2.6 incorporates a non-executable stack as a configuration option in **`/etc/system`**.

This technique is non-trivial and non-obvious, for the following reasons:

- **`gcc`** uses executable stacks for function trampolines for nested functions.
- The Linux kernel (for example) uses executable user stacks for signal delivery.
- Functional programming languages, and some other programs, rely on executable stacks for run-time code generation.

Solar Designer's Linux patch addresses the trampoline problem and other 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.

Non-executable stack protection intersects with that offered by StackGuard; there are attacks that can bypass each technique that are caught by the other. For instance, an attacker can bypass a non-executable stack by injecting the attack code into a separate heap buffer, and then just use a stack buffer to re-point a return address to the attack code on the heap. StackGuard would stop this form of attack.

Conversely, an attacker can bypass StackGuard protection using buffer overflows to alter *other* pointers in the program besides the return address, such as function pointers and **`longjmp`** buffers, which need not even be on the stack. If the attack code were injected into a buffer that is on the stack, then a non-executable stack would stop it. Such attacks are relatively rare, but have been constructed, as is the case for exploits against **`Perl 5.003`** and **`SuperProbe`** shown in Table 2.

Non-executable stacks are entirely compatible with StackGuard-protected programs. For protection-in-depth, one may well choose to use both techniques. Also, an extension to StackGuard to defend function pointers and **`longjmp`** buffers is under development.

7.2 FreeBSD Stack Integrity Check

Alexander Snarskii developed a FreeBSD patch [24] that does similar integrity checks to StackGuard's. However, these integrity checks were non-portable, hard-coded in assembler, and embedded in **`libc`**. This protects against stack smashing attacks inside **`libc`**, but is not as general as StackGuard.

7.3 Array Bounds Checking for C

Richard Jones and Paul Kelly developed a **`gcc`** patch [16] 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 is also not mature; complex programs such as **`elm`** fail to execute when compiled with this compiler.

However, this method is strictly more secure than StackGuard, preventing all buffer overflow attacks, not just those that attempt to alter function activation records. If the bounds-checking compiler were mature, we could use programs compiled with the bounds-checking compiler the to provide a "fall back position" if StackGuard-protected programs start detecting a determined attempt to break in [5].

7.4 Memory Access Checking

Purify [15] is a memory usage debugging tool for C programs. 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 call return. As a result, Purify is both more general slower 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.

7.5 Type-Safe Languages

All of the vulnerabilities described here 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`**.

Type-safety is one of the foundations of the Java security model. Consequently, *errors* in the Java type checking system are one of the ways that Java programs and Java virtual machines can be attacked [8, 22]. If the correctness of the type checking system is in question, then programs depending on that type checking system for security get the same benefit as type-unsafe programs. Applying StackGuard techniques to Java programs and JVMs may yield beneficial results.

7.6 Debugging Tools

Various tools have been developed to minimize the number of buffer overflow vulnerabilities. The simplest is to **`grep`** the source code for highly vulnerable library calls such as **`strcpy`** and **`sprintf`**. Versions of the C standard library have also been developed that complain when a program links to vulnerable functions like **`strcpy`** and **`sprintf`**.

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 [13], 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 [14]. The idea is to inject deliberate buffer overflow faults at random to search for vulnerable program components.

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 as StackGuard should be employed unless one is *very* sure that all potential buffer overflow vulnerabilities have been eliminated.

8 Availability

This paper describes our efforts to create an entire Linux system that is protected by StackGuard everywhere that it is potentially vulnerable to stack smashing attacks. The StackGuard compiler has been available for some time under the GPL [5]. The protected Linux distribution has also been available since August 1998, and includes:

- the StackGuard compiler
- StackGuard-protected binary RPM files for all C programs
- the scripts for re-building Red Hat 5.1 from source code
- SSH binary distributions[1]

This paper describes the protection of Red Hat Linux 5.1. A protected version of Red Hat 5.2 is being constructed, but is not complete at press time. Being a research group, we do *not* want to be in the business of distributing and supporting a Linux version. This distribution is intended to demonstrate the viability of

1.Due to export restrictions, SSH is available only to residents of the United States and Canada. Foreign users are encouraged to get the StackGuard compiler from us, and the source code for SSH from it's home site in Finland, and build their own protected SSH.

StackGuard protection for security in production systems, by putting it into production ourselves. It is our hope that Linux distributors will adopt the use of StackGuard to enhance the security of their products.

9 Conclusions

StackGuard is an effectively transparent replacement for **`gcc`** that offers protection from the pervasive problem of stack smashing vulnerabilities. In previous work [5] we demonstrated the penetration resistance of StackGuard, along with some quantitative measurements of StackGuard's performance costs. This paper presents a qualitative view of the practical and performance implications of using StackGuard in a production environment. If StackGuard is used thoroughly to protect all of the potentially vulnerable programs in a system, then the system is effectively protected against stack smashing attacks.

System administrators wishing to protect their systems can use the tools and procedures presented in this paper to protect themselves. System and application vendors wishing to offer more secure products can use the StackGuard compiler to deliver executable codes that are resistant to stack smashing attacks. Finally, users wishing to obtain a system protected from stack smashing can use the Linux distribution that we presented here.

References

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

[2] 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`.

[3] 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.

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

[5] 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*, San Antonio, TX, January 1998.

[6] 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.

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

[8] 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`.

[9] "der Mouse". Defeating Solar Designer non-executable stack patch. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, February 4 1998.

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

[11] 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.

[12] Linus Torvalds et al. Linux Operating System. `http://www.linux.org/`.

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

[14] 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.

[15] 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/`.

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

[17] 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.

[18] 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.

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

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

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

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

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

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

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

[26] Richard M. Stallman. *Using and Porting GNU C*. Free Software Foundation, Inc., Cambridge, MA.

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

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