Transparent Run-Time Defense Against Stack Smashing Attacks

Arash Baratloo and Navjot Singh
{arash,singh}@research.bell-labs.com
Bell Labs Research, Lucent Technologies
600 Mountain Ave
Murray Hill, NJ 07974 USA

Timothy Tsai*
ttsai@rstcorp.com
Reliable Software Technologies
21351 Ridgetop Circle, Suite 400
Dulles, VA 20166 USA

Abstract

The exploitation of buffer overflow vulnerabilities in process stacks constitutes a significant portion of security attacks. We present two new methods to detect and handle such attacks. In contrast to previous work, the new methods work with any existing pre-compiled executable and can be used transparently per-process as well as on a system-wide basis. The first method intercepts all calls to library functions known to be vulnerable. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. The second method uses binary modification of the process memory to force verification of critical elements of stacks before use. We have implemented both methods on Linux as dynamically loadable libraries and shown that both libraries detect several known attacks. The performance overhead of these libraries range from negligible to 15%.

1 Introduction

As the Internet has grown, the opportunities for attempts to access remote systems improperly have increased. Several security attacks, such as the 1988 Internet Worm [7, 18, 19], have even become entrenched in Internet history. Some attacks merely annoy or occupy system resources. However, other attacks are more insidious because they seize root privileges and modify, corrupt, or steal data.

*This work was performed while the author was with Lucent Technologies, Bell Labs, Murray Hill, NJ USA.

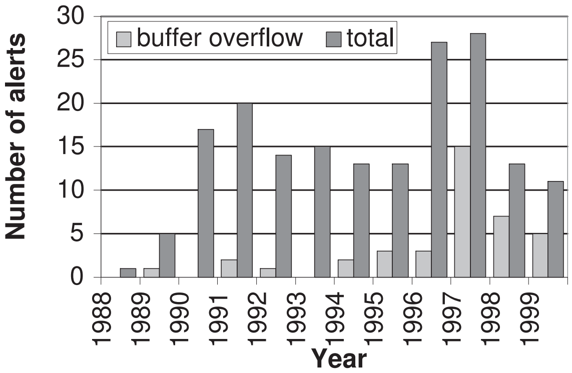


Figure 1: Number of Reported CERT Security Advisories and the Number Attributable to Buffer Overflow (Data from [24])

Figure 1 shows the increase in the number of reported CERT [3] security advisories that are based on buffer overflow. In recent years, attacks that exploit buffer overflow bugs have accounted for approximately half of all reported CERT advisories. The buffer overflow bug may be due to errors in specifying function prototypes or in implementing functions. In either case, an inordinately large amount of data is written to the buffer, thus overflowing it and overwriting the memory immediately following the end of the buffer. The overflow injects additional code into an unsuspecting process and then hijacks control of that process to execute the injected code. The hijacking of control is usually accomplished by overwriting return addresses on the process stack or by overwriting function pointers in the process memory. In either case, an instruction that alters the control flow (such as a call, return, or

Function prototype	Potential problem
`strcpy(char *dest, const char *src)`	May overflow the `dest` buffer.
`strcat(char *dest, const char *src)`	May overflow the `dest` buffer.
`getwd(char *buf)`	May overflow the `buf` buffer.
`gets(char *s)`	May overflow the `s` buffer.
`fscanf(FILE *stream, const char *format, ...)`	May overflow its arguments.
`scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the `path` buffer.
`sprintf(char *str, const char *format, ...)`	May overflow the `str` buffer.

Table 1: Partial List of Unsafe Functions in the Standard C Library

jump instruction) may inadvertently transfer execution to the wrong address that points at the injected code instead of the intended code.

Programs written in C have always been plagued with buffer overflows. Two reasons contribute to this problem. First, the C programming language does not automatically bounds-check array and pointer references. Second, and more importantly, many of the functions provided by the standard C library are unsafe, such as those listed in Table 1. Therefore, it is up to the programmers to check explicitly that the use of these functions cannot overflow buffers. However, programmers often omit these checks. Consequently, many programs are plagued with buffer overflows and are therefore vulnerable to security attacks.

Preventing buffer overflows is clearly desirable. If one did not have access to a C program's source code, the general problem of automatically bounds-checking array and pointer references is very difficult, if not impossible. So at first, it might seem natural to dismiss any attempts to perform automatic bounds checking at runtime when one does not have access to the source code. One of the contributions of this paper is to demonstrate that by leveraging some information that is available only at runtime, together with context-specific security knowledge, one can automatically foil security attacks that exploit unsafe functions to overflow stack buffers.

2 Buffer Overflow Exploit

The most general form of security attack achieves two goals:

1. Inject the attack code, which is typically a small sequence of instructions that spawns a shell, into a running process.
2. Change the execution path of the running process to execute the attack code.

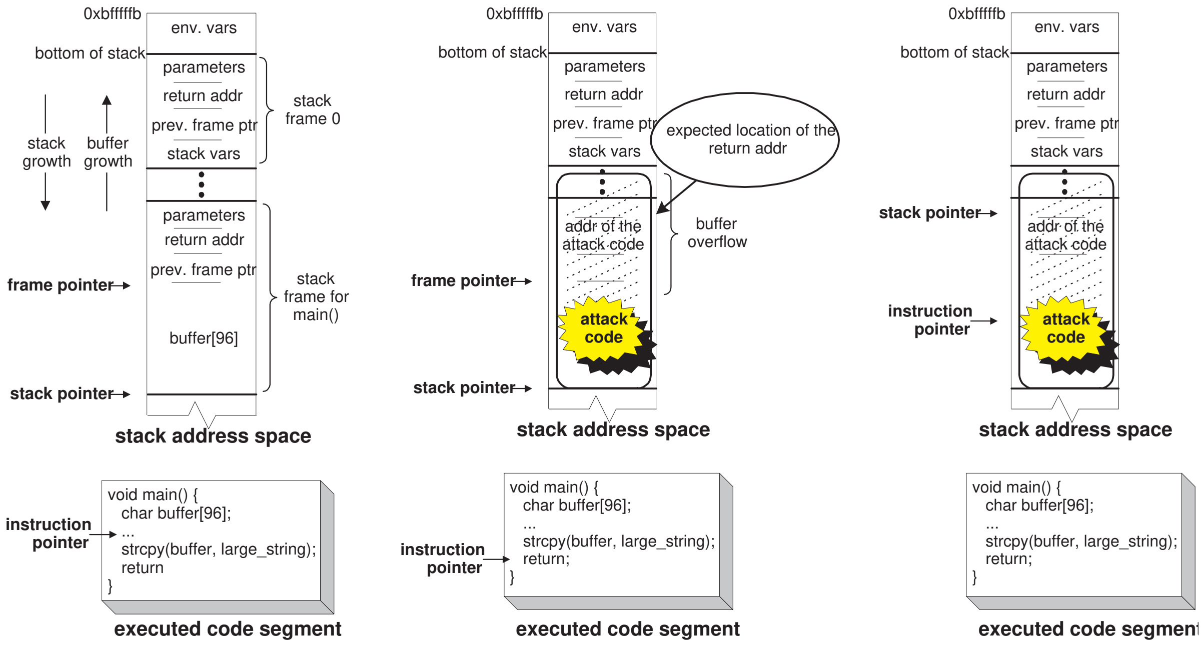
It is important to note that these two goals are mutually dependent on each other: injecting attack code without the ability to execute it is not necessarily a security vulnerability.

By far, the most popular form of buffer overflow exploitation is to attack buffers on the stack, referred to as the *stack smashing attack*. As is discussed below, the reason for this popularity is because overflowing stack buffers can achieve *both goals simultaneously*. Another form of buffer overflow attack known as the *heap smashing attack*, is to attack buffers residing on the heap (a similar attack involves buffers residing in data space). Heap smashing attacks are much harder to exploit, simply because it is difficult to change the execution path of a running process by overflowing heap buffers. For this reason, heap smashing attacks are far less prevalent.

A complete C program to demonstrate the stack smashing attack is shown in Figure 2. Figure 3 illustrates the address space of a process undergoing this attack. The process stack after executing the initialization code and entering the `main()` function (but before executing any of the instructions) is illustrated in Figure 3(a). Notice the structure of the top stack frame (i.e., the stack frame for `main()`). This stack frame contains, in order, the function parameters, the return address of the calling function, the previous frame pointer, and finally the stack variable `buffer`. Looking at the sample program in Figure 2, a sequence of instructions for spawning a shell is stored in a string variable called `shellcode` (lines 3-6). The `shellcode` is equivalent to execut-

```
#include <stdio.h>

char shellcode[] =
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
int i;
long *long_ptr;

int main() {
  char buffer[96];

  long_ptr = (long *)large_string;
  for (i=0; i<32; i++)
    *(long_ptr+i) = (int)buffer;
  for (i=0; i<(int)strlen(shellcode); i++)
    large_string[i] = shellcode[i];
  strcpy(buffer, large_string);
  return 0;
}
```

Figure 2: A Sample Program to Demonstrate a Stack Smashing Attack



(a) before the attack (b) after injecting the attack code (c) executing the attack code

Figure 3: A Process Undergoing a Stack Smashing Attack

ing `exec(''/bin/sh'')`. The two `for` loops in the `main` function prepare the attack code by writing two sequences of bytes to `large_string`: the `for` loop starting on line 16 writes the (future) starting address of the attack code; then the `for` loop starting on line 18 copies the attack code (excluding the terminating null character). The stack is smashed on line 20 by the `strcpy()` function. Figure 3(b) depicts the process' stack space after executing the `strcpy()` call. Notice how the unsafe use of `strcpy()` simultaneously achieves both requirements of the stack smashing attack: (1) it injects the attack code by writing it on the process' stack space, and (2) by overwriting the return address with the address of the attack code, it instruments the stack to alter the execution path. The attack completes once the `return` statement on line 21 is executed: the instruction pointer "jumps" and starts executing the attack code. This step is illustrated in Figure 3(c).

In a real security attack, the attack code would normally come from an environment variable, user input, or even worse, from a network connection. A successful attack on a privileged process would give the attacker an interactive shell with the user-ID of root, referred to as a root shell.

3 Related Work

The Internet Worm that infected tens of thousands of hosts in 1988 was one of the first well-known buffer overflow attacks, although there are some anecdotal evidence that buffer overflow attacks date back to the 1960's [4]. The proportion of attacks based on buffer overflows is increasing each year—in recent years, buffer overflow attacks have become the most widely used type of security attack [24]. Among such attacks, the stack smashing attack is the most popular form [10, 22].

The majority of buffer overflow attacks, including the one exploited by the Internet Worm is based on the stack smashing attack. Detailed descriptions of stack smashing attacks are presented in [20, 22], and cook-book-like recipes are presented in [6, 15, 16].

Researchers in the areas of operating systems, static code analyzers and compilers, and run-time middleware systems have proposed solutions to circumvent stack smashing type of attacks. In most operating systems the stack region is marked as executable, which means that code located in the stack memory can be executed. Because this "feature" is used by stack smashing attacks, making the stack non-executable is a commonly proposed method for thwarting overflow attacks. A kernel patch removing the stack execution permission has been made available [17]. This approach, however, has some drawbacks. First, patching and recompiling the kernel is not feasible for everyone. Second, *nested function calls* or *trampoline functions*, which are used extensively by LISP interpreters and Objective C compilers, and the most common implementation of signal handler returns on Unix (as well as Linux), rely on an executable stack to work properly. And finally, an alternative attack on stacks known as *return-into-libc*, which directs the program control into code located in shared libraries, cannot be defeated by making the stack non-executable [25]. Because of those reasons, Linus Torvalds has consistently refused to incorporate this change into the Linux kernel [23].

Snarskii has developed a custom implementation of the standard C library for FreeBSD [21]. This library targets the set of unsafe functions, and inspects the process stack to detect buffer overflows that write across frame pointers. In contrast to our work, this is a custom implementation and replaces the standard C library.

Several commonly used tools, such as Lint [11], and those proposed in [8] use compile-time analysis to detect common programming errors. Existing compilers have also been augmented to perform bounds-checking [13]. These projects have demonstrated limited success in preventing the general buffer overflow problem. Wagner *et al.* have recently proposed the use of compile-time range analysis to ensure the "safe" use of C library functions [24]. This project specifically concentrates on the set of unsafe library functions. Unlike our approach, this method requires access to a program's source code, which is not always available. Moreover, preliminary results indicate that this method may produce false positives: a correct program may produce warning or error messages.

StackGuard [5] is another compiler extension that instruments the generated code with stack-bounds checks. Specifically, on function entry, a *canary* is placed near the caller's return address on the stack. Before the function returns to the caller, the validity of this canary is checked and the program is terminated if a discrepancy is detected. This approach works on the assumption that if the return address is tampered with (due to buffer overflows),

Program Name	Version	Description	Result of Attack	Result with libsafe or libverify
xlockmore	3.10	Lock an X Window display	root shell	terminated
amd	6.0	Automatic remote file system mount daemon	root shell	terminated
imapd	3.6	IMAP mail server	root shell	terminated
elm	2.5 PL0pre8	ELM mail user agent	root shell	terminated
SuperProbe	2.11	Probes and identifies video hardware	root shell	terminated

Table 2: List of Some Known Exploits That Are Detected

the canary will also be modified, thus causing validation of the canary to fail. With the exception of a few programs, this approach has shown to be effective. StackGuard introduces a noticeable run-time overhead. Furthermore, StackGuard requires source code access, and there are some programs, such as Netscape Navigator, Adobe Acrobat Reader, and Star Office, that it does not currently support.

Janus [9] is a run-time sand-boxing environment that confines each application to a set of predefined operations. It works on the principle that "an application can do little harm if its access to the underlying operating system is appropriately restricted." It relies on the operating system's debugging features, such as `trace` and `strace`, to observe and to confine a process to a sand-box. Similar to our work, this approach works with existing binary applications and does not require access to application's source code. However, unlike our approach, Janus does not work with applications that legitimately need high privileges. For example, the Unix `login` process requires a high level of privilege to execute, but Janus is unable to selectively allow legitimate privileges while denying unauthorized privileges. This inherent limitation prevents Janus from being applied to high privileged applications, where secure execution is most critical.

4 Overview of Techniques

This paper presents two novel methods for performing detection and handling of buffer overflow attacks. In contrast to previous methods and without requiring access to a program's source code, our novel methods can transparently protect processes against stack smashing attacks, even on a system-wide basis. The first method intercepts all calls to library functions that are known to be vulnerable. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. This method has been implemented as a dynamically loadable library called *libsafe*. The second method uses binary re-writing of the process memory to force verification of critical elements of stacks before use. This method has also been implemented as a dynamically loadable library called *libverify*.

The key idea behind libsafe is the ability to estimate a safe upper limit on the size of buffers automatically. This estimation cannot be performed at compile time because the size of the buffer may not be known at that time. Thus, the calculation of the buffer size must be made after the start of the function in which the buffer is accessed. Our method is able to determine the maximum buffer size by realizing that such local buffers cannot extend beyond the end of the current stack frame. This realization allows the substitute version of the function to limit buffer writes within the estimated buffer size. Thus, the return address from that function, which is located on the stack, cannot be overwritten, and control of the process cannot be commandeered.

The libverify library relies on verification of a function's return address before use, a scheme similar to that found in StackGuard. The difference is the manner of implementation. Whereas StackGuard introduces the verification code during compilation, libverify injects the verification code at the start of the process execution via a binary re-write of the process memory. Furthermore, libverify uses the actual return address for verification instead of a "canary" value representing the return address. Thus, in contrast to StackGuard, libverify can protect pre-compiled executables.

We have implemented the previously described methods as dynamically loadable libraries on Linux

	Instrumentation Techniques					
	None	libsafe	libverify	StackGuard	Janus	Non-Executable Stack
Effectiveness (what types of errors are handled?)						
Kernel Errors	No	No	Yes	Yes	No	Yes
Specification Errors	No	Yes	Yes[a]	Yes[a]	Maybe[b]	Maybe[c]
Implementation Errors	No	Maybe[d]	Yes[a]	Yes[a]	Maybe[b]	Maybe[c]
User Code Errors	No	No	Yes	Yes	Maybe[b]	Maybe[c]
Other characteristics						
Performance Overhead	None	Very low	Medium	Medium	Medium	None
Disk Usage Overhead	None	Very low	Very low	Low	Very low	None
Source Code Needed	No	No	No	Yes	No	No
Ease of Use	—	Very easy	Very easy	Medium[e]	Easy-Medium[f]	Easy-Medium[g]

[a] If libraries are instrumented.
[b] Cannot catch hijacked privileges that are similar to legitimate privileges.
[c] For certain types of exploits (see Section 3).
[d] If we know which functions have errors.
[e] Source code must be recompiled, and the compiler may also needed to be recompiled.
[f] Policies need to be written.
[g] Kernel may need to be patched and recompiled.

Table 3: Summary of Detection Technique Characteristics

and tested them against several security attacks. Table 2 lists several commonly used applications and the result of running publicly available exploits against the applications with and without our libraries.[1] As the table indicates, libsafe and libverify were able to detect the exploits and terminate the programs before any serious harm was done.

The characteristics of libsafe and libverify are shown in Table 3 along with the corresponding characteristics of alternative methods: StackGuard, Janus, and kernel patches for non-executable stack, which were described earlier in Section 3. The first instrumentation technique labeled "None" is presented as a point of comparison and represents the original program with no modifications. The upper half of Table 3 describes the types of errors that each method is able to handle. Specification and implementation errors refer to errors in standard library functions. In particular, by specification errors we mean the set of functions known to be unsafe as described in Section 1; implementation errors refer to the set of functions that are unsafe due to implementation errors. Kernel errors and user code errors refer to implementation errors in kernel code and user code, respectively. The bottom half of the table describes other characteristics. The performance overhead includes only the run-time overhead. Time spent during configuration and compilation are not included. The disk usage overhead is the extra disk space required due to additional shared libraries, increased executable binary file sizes, and configuration files. The next to last row indicates whether access to source code of the defective program is needed. The ease of use considers the complexity and time requirement of human efforts needed for configuration and compilation.

[1] The security attacks are available from Crv's Security Bugware Page (`http://oliver.efri.hr/~crv/`).

5 Libsafe

The fundamental observations forming the basis of the libsafe library are the following:

- Overflowing a stack variable—that is, injecting the attack code into a running process—does not necessarily lead to a successful stack smashing attack. The attack must also divert

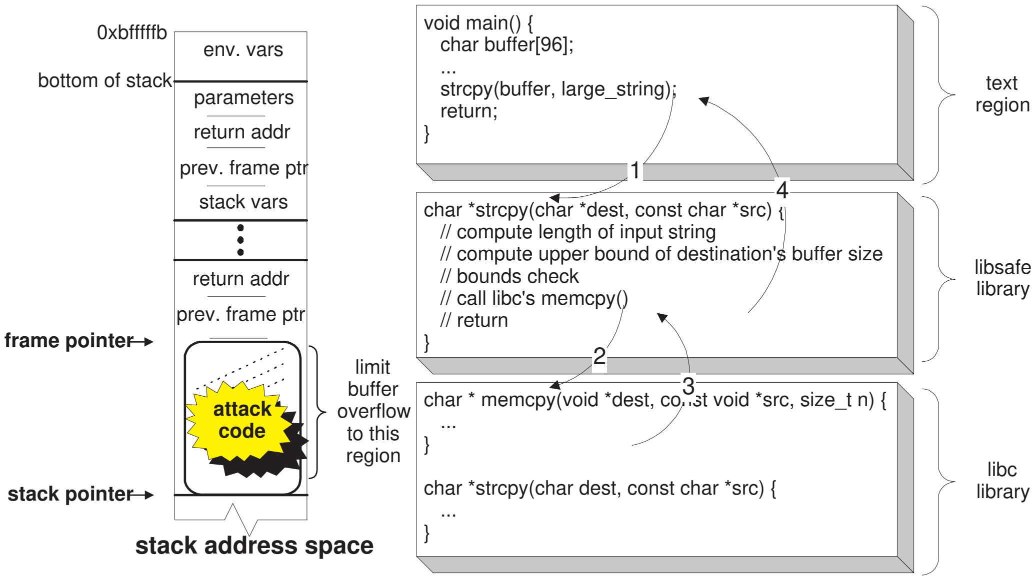


Figure 4: Libsafe Containment of Buffer Overflow

the execution sequence of a process to run the attack code.

- Although buffer overflows cannot be stopped in general, automatic and transparent run-time mechanisms can prevent the overflow from corrupting a return address and altering the control flow of a process.

Refer to Figure 3(a) for an example. At the time `strcpy()` is called, the frame pointer (i.e., the `ebp` register in the Intel Architecture) will be pointing to a memory location containing the previous frame's frame pointer. Furthermore, the frame pointer separates the stack variables (local to the current function) from the parameters passed to the function. Continuing with the example of Figure 3(a), the size of `buffer` and all other stack variables residing on the top frame cannot extend beyond the frame pointer—this is a safe upper limit. A correct C program should never explicitly modify any stored frame pointers, nor should it explicitly modify any return addresses (located next to the frame pointers). We use this knowledge to detect and limit stack buffer overflows. As a result, the attack executed by calling the `strcpy()` can be detected and terminated before the return address is corrupted (as in Figure 3(b)). In the case that a local buffer on one of the previous stack frames is accessed, then frame pointers are traversed up the stack until the the right stack frame is found, and then libsafe computes the upper bound.

Libsafe implements the above technique. It is implemented as a dynamically loadable library that is preloaded with every process it needs to protect. The preloading injects the libsafe library between the program code and the dynamically loadable standard C library functions. The library can then intercept and bounds-check the arguments before allowing the standard C library functions to execute. In particular, it intercepts the unsafe functions listed in Table 1 to provide the following guarantees:

- Correct programs will execute correctly, i.e., no false positives.
- The frame pointers, and more importantly return addresses, can never be overwritten by an intercepted function—an overflow that would lead to overwriting the return address is always detected.

Figure 4 illustrates the memory of a process that has been linked with the libsafe library, and in particular, it shows the new implementation of `strcpy()` in the libsafe library. Once the program invokes `strcpy()`, the version implemented in the libsafe library gets executed—this is due to the order in which the libraries were loaded. The libsafe implementation of the `strcpy()` function first computes the length of the source string and the upper bound on the size of the destination buffer (as explained above). It then verifies that the length of the source string is less than the bound on the destination buffer. If the verification succeeds, then the strcpy() calls memcpy() (implemented in the standard C library) to perform the operation. However, if the verification fails, strcpy() creates a syslog entry and terminates the program. A similar approach is applied to the other unsafe functions in the standard C library.

The libsafe library has been implemented on Linux. It uses the preload feature of dynamically loadable ELF libraries to automatically and transparently load with processes it needs to protect. In essence, it can be used in one of two ways: (1) by defining the environment variable LD_PRELOAD, or (2) by listing the library in /etc/ld.so.preload. The former approach allows per-process control, where as the latter approach automatically loads the libsafe library machine-wide.

The libsafe library does not use any Linux specific feature of ELF; these ELF features are available for many other versions of Unix such as Solaris, and have been used for other purposes [1, 14]. Furthermore, an alternative technique with a similar feature can be used for Windows NT [2, 12].

We have installed the libsafe library on a Linux machine. The library is automatically loaded with every process and transparently protects each process from stack smashing attacks. The protected applications include daemon processes such as the Apache HTTP server, sendmail, and an NFS server, as well as those started by users such as the XFree86 server, the Enlightenment window manager, GNU Emacs, Netscape Navigator, and Adobe Acrobat Reader. We have used this machine for several months and found the machine to be stable and running without a noticeable performance hit.

6 Libverify

The libverify library implements a return address verification scheme similar to that used in StackGuard.

Both methods protect return addresses on the process stack by saving canary values at the start of a function and verifying the canary value at the end of the function to determine if any buffer overflow occurred. However, in contrast to StackGuard, libverify requires no recompilation of source code and is therefore applicable to legacy programs. Instead, all code for saving and verifying canaries is contained in a special library. This library also contains instrumentation code to link the canary code with the program. As with libsafe, the library is activated by specifying it as part of the LD_PRELOAD environment variable or the /etc/ld.so.preload file.

Figure 5 shows the memory of a process that has been linked with libverify. Before the process commences execution, the library is linked with the user code. As part of the link procedure, the _init() function in the library is executed. The _init() function contains code to instrument the process such that the canary verification code in the library will be called for all functions in the user code. The instrumentation includes the following steps:

1. Determine the location and size of the user code.
2. Determine the starting addresses of all functions in the user code.
3. For each function
   (a) Copy the function to heap memory.
   (b) Overwrite the first instruction of the original function with a jump to the wrapper_entry function.
   (c) Overwrite the return instruction of the copied function with a jump to the wrapper_exit function.

The wrapper_entry function saves a copy of the canary value on a canary stack and then jumps to the copied function. The wrapper_exit function verifies the current canary value with the canary stack. A canary stack is needed to save canary values for nested function calls. If the canary value is not found on the canary stack, then the function determines that a buffer overflow has occurred. In that case, the wrapper_exit function then calls the die() function, which creates a syslog entry, prints an error message to the standard error device, and terminates. The die() function can also perform additional notification and handling, such as sending an email message or shutting down the entire system.

In contrast to StackGuard, which generates random numbers for use as canaries, libverify uses the actual return address as the canary value for each function. This simplifies the binary instrumentation procedure because no additional data is pushed onto the stack, which means that the relative offsets to all data within each stack frame remain the same. Although the return address can sometimes be guessed

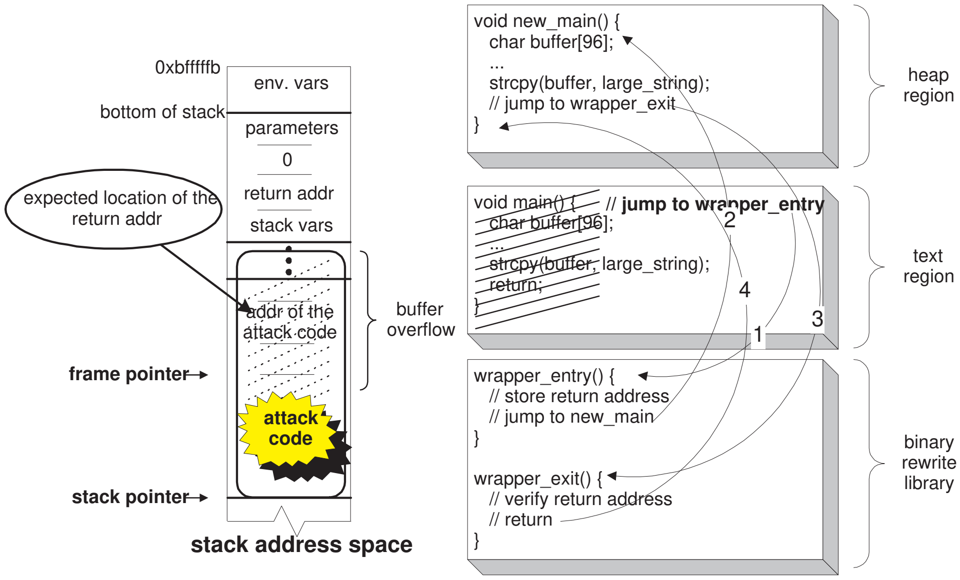


Figure 5: Memory Usage for libverify

by an attacker, control flow is still protected because the actual value of any return address is explicitly verified before execution of that return instruction. The canary stack resides in heap memory. The size is dynamically extended to accommodate a large number of simultaneous canaries. The canary stack itself is not protected against overflow attacks in the current libverify implementation. However, such protection can be easily added by using the `mprotect()` function to designate the page immediately preceding the canary stack as non-writable.

A difficulty does arise when a function performs an absolute jump to an address within the same function. As an example, this situation might occur for some `switch()` statements. Because we copy the original function to heap memory and execute that function from the copied version, an absolute jump in the copied function would force control flow to the original function. To handle this situation, we overwrite the original function with trap instructions. If control is forced to the original function, the trap is activated, and a trap handler returns control flow back to the copied function.

7 Experiments

The libsafe and libverify libraries are effective in detecting and defeating stack smashing attacks. Extra code is needed to perform this detection, and that extra code incurs a performance overhead. In this section we quantify the performance overhead associated with use of these libraries. Section 7.1 describes the overheads associated with micro benchmarks to illustrate the range of possible overheads. Section 7.2 gives performance data for a selected set of actual applications.

Application	Size (Bytes)	Initialization time (μs)
`quicksort`	27330	13032
`imapd`	1305379	67491
`tar`	418283	40334
`xv`	1242686	195205

Table 4: The Initialization Elapsed Times for libverify Library

All experiments were conducted on a 400 MHz Pentium II machine with 128 MB of memory running RedHat Linux version 6.0. Our libraries and all programs in Sections 7.1 and 7.2 were compiled (and optimized using -O2) with GCC compiler version 2.91.66.

7.1 Micro Benchmarks

As the part of the link procedure, libverify executes its initialization section, the `_init()` function), as described in Section 6. This initialization section first reads, then copies and modifies the entire instruction sequence of the application. Table 4 presents the initialization times of libverify with four commonly used applications: `quicksort` (a fast sorting program), `imapd` (an Internet Message Access Protocol server), `tar` (an archiving utility), and `xv` (an interactive image displayer for the X Window System). The numbers in Table 4 represent the start-up overhead associated with libverify. This overhead depends on the size and complexity of the program libverify is instrumenting. As the numbers indicate, the start-up overhead takes approximately $50 - 160$ milliseconds per Megabyte.

Libsafe does not require an initialization section. However, the first time each libsafe function is activated, the initialization of that particular function makes a `dlsym()` call for each libc function that is called from that libsafe function. Because the libsafe function has the same name as the corresponding libc version, the `dlsym()` call is needed to obtain a pointer to the libc function. Each `dlsym()` call requires 1.26 μs. The interception and redirection of a C library function consists of an additional user-level function call, which approximately adds 0.04 μs of overhead.

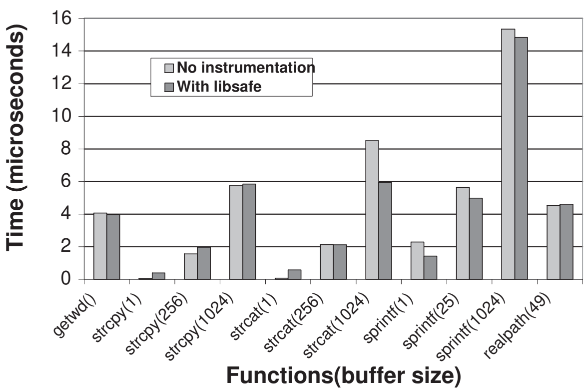


Figure 6: Performance of Libsafe Functions

To quantify the performance overhead of the libsafe library we measured the execution times of five unsafe C library functions and compared the results with our "safe" versions. The results are depicted in Figure 6. Reported times are "wall clock" elapsed times as reported by `gettimeofday()`. An interesting observation is that the libsafe versions of several functions outperform the original versions. This is a repeatable behavior, and we have observed consistent findings on different machines and operating system versions. This effect is due both to low-level optimizations and the fact that libsafe's implementation of most functions is different than those of C library. For example, consider the performance of the `getwd()` and `sprintf()` functions. Our libsafe library replaces these functions with equivalent safe versions. In particular, `getwd()` is replaced with `getcwd()` and `sprintf()` is replaced with `snprintf()`; on Linux, the safe versions execute faster.

The figure also shows that the libsafe library can slow down the string operations `strcpy()` and `strcat()` by as much as 0.5 μs per function call. However, as the string size increases, the absolute overhead decreases because the execution time of the safe versions increases more slowly than that for the unsafe versions. In fact, the safe version of `strcat()` used with strings longer than 256 bytes is actually faster than the unsafe version! This is an example of how using a different implementation (e.g., using `memcpy()` to copy a string) can outperform the standard implementation for certain cases.

The slowdown effect of `strcpy()` is observed in the `realpath()` experiment. When a program calls `realpath()`, the libsafe library calls `realpath()` but stores the result in a buffer in its own memory region. It then uses `strcpy()` to copy the result to the final destination. As Figure 6 shows the slowdown effect of `strcpy()` on `realpath()` is less than 0.05 μs.

7.2 Application Benchmarks

Since we propose that the libraries are best used on a machine-wide bases to protect against yet unknown attacks, their performance impact is important for all commonly used application. We used four real-world applications to illustrate the performance overhead of our libraries. The applications are `quicksort` (a CPU-bound program) ordering 1,000,000 integers, `imapd` (a network-bound program) transmitting 100 email messages of size 2 kilobyte each, `tar` (an I/O-bound program) archiving 5 Megabytes of data, and `xv` (a CPU and video-bound program) displaying a 1.2 Megabyte image. Figure 7 shows the execution time for each of these applications (1) unmodified and without any security measure, (2) using the libsafe library, (3) using the libverify library, and (4) compiled with StackGuard.

The execution times are based on 100 runs and are given in seconds, with associated 95% confidence intervals. Reported times are elapsed times as reported by `/bin/time`, and include the extra initialization time required by libverify.

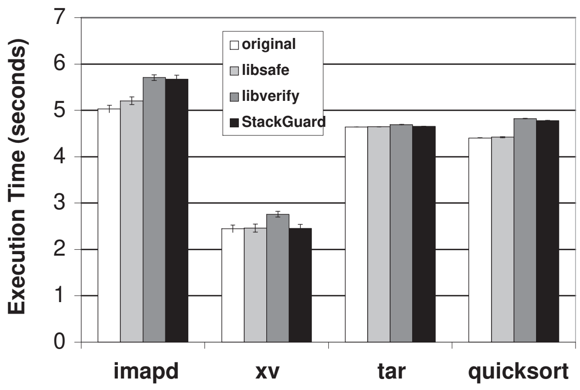


Figure 7: Mean Execution Times (With 95% confidence intervals) of Sample Applications

Figure 7 shows that the overheads associated with all detection methods are reasonable (i.e., less than 15% for these applications). Libsafe is the most efficient method because only the unsafe library functions are intercepted.

Libverify incurs a greater overhead than the libsafe because all user functions are verified. For most of the applications, the overhead is similar to that for StackGuard because the same number of functions is verified. For `xv`, the need to handle a large number of traps (as described in Section 6) increases the overhead. The overall application test results are encouraging, particularly with libsafe. We have installed and used libsafe on one of our own machines, and have found that the overhead is not noticeable in practice.

8 Conclusions

We have described two complementary methods for foiling stack smashing attacks that rely on corrupting the return address, and implemented these methods as dynamically loaded libraries called libsafe and libverify.

An interesting finding is the performance of libsafe. We anticipated a low performance overhead at the onset of the project. We were happily surprised to find how little this overhead is in practice. Because of low-level optimizations and because libsafe's implementation of most functions is different than those of C library, for some applications we actually observed a speedup. This is encouraging since it indicates the viability of this approach. Furthermore, the elegance and simplicity of instrumenting the standard C library led to a stable implementation.

The implementation of libverify gave us quite a challenge. Our initial goal in re-writing binary instruction streams was to insert the minimum amount of code at beginning of each function to divert the execution control to the `wrapper_entry`, and similarly, to insert the minimal code at the end of the function to execute `wrapper_exit` before returning to the caller. However on the Intel Architecture, we could not fit the required instructions at the end of each function. Hence, we settled with copying the entire function to the heap where space was not a limitation. Relocating functions from the text region to the heap gave rise to the problems we encountered with absolute jumps (as discussed in Section 6). Furthermore, it doubled the code space required for each process. We believe this approach to verifying return addresses is well suited for RISC architectures such as the Alpha or SPARC where the instructions are all the same size.

We believe that the stability, minimal performance overhead, and ease of use (i.e., no modification or recompilation of source code) of the two libraries makes them an attractive first line of defense against stack smashing attacks. It is generally accepted that the best solution to buffer overflow attacks is to fix the original defects in the programs. However, fixing the defects requires knowing that a particular program is defective. The true benefit of using libsafe and libverify is protection against attacks on programs that are not yet known to be vulnerable.

9 Acknowledgments

We are thankful to Scott Alexander for his insightful comments and to Vandoorselaere Yoann for his assistance with the software.

10 Availability

The libsafe library is available under the GNU Library General Public License. Further information is available from `http://www.bell-labs.com/org/11356/libsafe.html`.

References

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user-level: the Ufo global file system. In *Proceedings of the 7th USENIX Annual Technical Conference*, 1997.

[2] Robert Balzer and Neil Goldman. Mediating connectors. In *Proceedings the 19th IEEE International Conference on Distributed Computing Systems Workshop*, 1999.

[3] CERT coordination center. `http://www.cert.org`.

[4] Crispin Cowan. `http://geek-girl.com/bugtraq/1999_1/0481.html`, 1999. Posting to Bugtraq Mailing List.

[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 *Proceedings of the 7th USENIX Security Conference*, 1998.

[6] dark spyrit aka Barnaby Jack. Win32 buffer overflows (location, exploitation and prevention). `http://www.insecure.org`.

[7] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, 1989.

[8] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[9] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[10] Shawn Instenes. Stack smashing: What to do? *;login: the USENIX Association newsletter*, April 1997.

[11] Stephen C. Johnson. *Lint, a C program checker*. Bell Laboratories, Murray Hill, New Jersey, USA, December 1977. Computer Science Technical Report 65.

[12] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[13] Richard Jones. Bounds checking patches for gcc. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge`.

[14] Alain Knaff. Zlibc - transparent access to compressed file. `http://zlibc.linux.lu`.

[15] Mudge. How to write buffer overflows. `http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html`, 1995.

[16] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.

[17] Openwall Project. Linux kernel patch from the openwall project. `http://www.openwall.com/linux`.

[18] Jon A. Rochlis and Mark W. Eichin. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, June 1989.

[19] Donn Seeley. A tour of the worm. In *Proceedings 1989 Winter USENIX Technical Conference*, January 30 - February 3 1989.

[20] Nathan Smith. Stack smashing vulnerabilities in the UNIX operating system. `http://millcomm.com/~nate/machines/security/stack-smashing/nate-buffer.%ps`, 1997.

[21] Alexandre Snarskii. Increasing overall security.... `ftp://ftp.lucky.net/pub/unix/local/libc-letter` and `http://www.lexa.ru:8100/snar/libparanoia`, 1997.

[22] Evan Thomas. Attack class: Buffer overflows. *Hello World!*, 1999.

[23] Linus Torvalds. Posting to linux kernel mailing list. `http://www.lwn.net/980806/a/linus-noexec.html`, 1998.

[24] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings 7th Network and Distributed System Security Symposium*, February 2000.

[25] Rafel Wojtczuk. Defeating solar designer non-executable stack patch. `http://geek-girl.com/bugtraq`, January 1998.