1-1-2002

Study of buffer overflow attacks and microarchitectural defenses

Jingfeng Xu

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Xu, Jingfeng, "Study of buffer overflow attacks and microarchitectural defenses" (2002). Retrospective Theses and Dissertations. 21366.
https://lib.dr.iastate.edu/rtd/21366

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Study of buffer overflow attacks and microarchitectural defenses

by

Jingfeng Xu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Simanta Mitra
Doug Jacobson

Iowa State University

Ames, Iowa

2002
This is to certify that the master's thesis of

Jingfeng Xu

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
Table of Contents

Chapter 1. Introduction 1
   Section 1. What is a buffer overflow attack 1
   Section 2. History and popularity 1

Chapter 2. How does it work 3
   Section 1. Basic framework 3
   Section 2. Buffer overflow attacks 7
   Section 3. Advanced techniques 11

Chapter 3. The defenses 14
   Section 1. Static method 14
   Section 2. Dynamic methods 16

Chapter 4. Our method 19
   Section 1. Program counter encoding 19
   Section 2. Implementation 23
   Section 3. Performance overhead 25

Chapter 5. Exploit experiments 30
   Section 1. Exploits 30
   Section 2. Limitations and future work 33

Chapter 6. Conclusions 35

References 36
Chapter 1. Introduction

Section 1. What Is a Buffer Overflow Attack?

Buffer overflow means filling a buffer with input which has larger size than the size of the buffer. In C programming language, buffer overflow is possible since there is no array boundary checking when copying strings. When a program takes input string and copies the string into an internal buffer without checking the length of the input string, and the length of the string is larger than the size of the internal buffer, then a buffer overflow happens and this program can be potentially exploited. The input string could come from different sources, such as command line, environment variable or network if the vulnerable program is running as a daemon. The internal buffers in the vulnerable program which are suitable for this attack are normally located near some important memory blocks like activation records of procedure calls and function pointers. By overflowing such internal buffers, the adjacent memory blocks can be overwritten by some memory address which points to some malicious instructions located somewhere in the memory or in the overflowed buffer. Then attackers can gain the control flow of the vulnerable program and obtain partial or full privilege of the host machine. Under UNIX/Linux system, the unrestricted root privilege can be obtained if the vulnerable program is owned by the root.

Section 2. History and Popularity

Buffer overflow attack is one of the most common network attacks for the last 10 years. The first buffer overflow attack was caused by Robert Tappan Morris's Internet Worm in
November 1988 which brought down more than 6000 sites. After that there were not many new buffer overflow attacks until some popular introductory articles came out in between 1996-1998, including [1, 2, 3, 4]. Aleph One's paper [1] is the first one which focuses on buffer overflow attacks on Intel x86 Linux platform and many of the early buffer overflow exploit codes for vulnerable programs were based on the technique illustrated in this paper.

To see the frequency of buffer overflow attacks, one can scan some popular security web sites. For CERT/CC [5], the Computer Emergency Response Team Coordination Center at the Software Engineering Institute of Carnegie Mellon University, in 1998, 7 out of 13 advisories were due to buffer overflows, in 1999, 9 out of 17 were due to buffer overflows, in 2000, 3 out of 22 were due to buffer overflows.
Chapter 2. How Does It Work?

The internal buffer could be located in stack, which is the most common case, or static storage, or heap. The control flow of the vulnerable program can be obtained by overwriting some important memory blocks adjacent to the internal buffer. The case of overwriting activation records is considered here.

Section 1. Basic Framework

In Unix/Linux system, when a process starts, its memory space is organized as follows. The virtual memory space goes from 0x00000000 to 0xFFFFFFFF. Program text resides in the lowest memory block, then initialized static data area, then uninitialized static data area (BSS). From the highest memory comes the kernel stack area, then the user stack area. Between the user stack area and the data segment is the heap where dynamic allocated storages reside. In Unix/Linux, stack grows downwards, i.e., from high memory address to low memory address, whereas heap grows upwards, i.e., from low memory to high memory. In the user stack area, starting from the high memory address, the first is the environment variable strings if the vulnerable program declares to use them, then the argument strings if the vulnerable program declares to use them, then the envp pointer, then the argv pointer, then the argc, then the stack frame for the main function, then the stack frames for other procedure calls, so on and so forth.

In Intel x86/Linux system, the register ESP is used as the stack pointer which points to the top of the stack, register EBP is used as the frame pointer which points to the base
memory address of the current activation record. The stack pointer changes with any push and pop operations, while the frame pointer stays unchanged during a procedure call. So all of the local variables and temporaries can be conveniently addressed by offsets from the frame pointer. The call sequence when a procedure call is made and the return sequence when a procedure returns vary with different platforms and compilers. In the following, Intel x86/Linux platform and GCC compiler is considered.

**Call Sequence**

First, the caller evaluates the actual parameters to the callee procedure and pushes these actuals on the stack in reverse order, i.e., for a function call of `foo(a, b, c)`, `c` is pushed into the stack, then `b`, then `a`. The reverse order is important for those procedures with variable number of parameters like C run-time library function `printf`. The first parameter of `printf` is the string format which contains the information of the following parameters. By scanning this string, the callee can easily determine the total number of parameters. Since this string is the last actual being pushed on the stack, so it can be addressed by some offset from the frame pointer. But if the actuals are pushed on stack in sequence, then the first string can not be located by the callee, and therefore the number of total parameters can not be determined.

Second, the caller pushes the return address, i.e., the current value of register EIP, on the stack and loads the callee's starting address into EIP. Register EIP is the instruction pointer for Intel x86, which points to the address of the next instruction to be executed, so the control flow switches to the callee. These activities are initiated with one instruction, `call foo`. In Intel x86, the return value of a procedure (if there is any) is passed through registers, generally EAX and other general registers like EDX (for gcc, this is the case) if more than
one register is needed. So there is no need to allocate storage for the return value in the stack before the caller transfers control to the callee.

Now the control is in the callee function. The callee needs to do some housekeeping work encapsulated in function prologue before anything else. Generally, function prologue does the following work.

1. Save the value of the frame pointer EBP onto the stack. Sometimes from efficiency considerations, the stack pointer, instead of the frame pointer is used to address local variables, then there is no need to save the frame pointer and EBP can also be used as a general register. In GNU C compiler, this can be done with the switch \texttt{-fomit-frame-pointer}. With this switch, the compiler won't generate frame pointer related instructions.

2. If the frame pointer is used, then the frame pointer is adjusted by loading the value of the stack pointer in it.

3. Allocate the storage for the local variables and possible temporaries inside the callee function and initialize the locals if needed. This can be done by simply decrementing the stack pointer with the number of bytes needed for the local variables. The locals and temporaries then can be easily addressed by some (negative) offset from the frame pointer.

4. If other general registers are used inside the callee function, the callee also needs to save the values of these registers on the stack and restore these values before it returns.

At this point, the callee begins execution.

For an Intel x86 and GCC compiler, the function prologue looks like the following. The following assembly uses AT&T style instead of the Intel x86 native style.
pushl %ebp  /* push EBP onto stack */
movl %esp,%ebp  /* move ESP to EBP */
subl $12,%esp /* decrement ESP to allocate storage */
pushl %edx  /* if EDX needs to be saved */

Return Sequence

Before the callee function returns, it executes its function epilogue. The epilogue is responsible for restoring the saved registers and stack pointer to their values at the entry into the function, and returning control to the caller. In the function epilogue, the following work is performed.

1. If there are any general registers that were saved in the function prologue stage, then restore them.

2. Release the current stack frame. If the frame pointer is involved, first move EBP to the ESP, so now the top of the stack is the saved frame pointer, i.e., the base address of the caller's stack frame. Then pop the stack to EBP, so that the old frame pointer is restored. These two steps can be done with one instruction, leave. If no frame pointer is involved, then simply increment the stack pointer by the size of the current stack frame.

3. Transfer control to the caller. This is done by popping the stack to EIP with instruction ret. For an Intel x86, the function epilogue looks as the follows.

popl %edx  /* if EDX needs to be restored */
movl %ebp,%esp /* release callee's stack frame*/
popl %ebp  /* restore the caller's frame pointer */
ret  /* return to the caller */
When the caller has the control, it needs to pop the actual parameters (if any) provided to the callee.

**Section 2. Buffer Overflow Attacks**

Buffer overflow attack code tries to overwrite the activation records of the vulnerable program, in particular, the return address part of the activation records.

![Figure 1. Stack layout of a running program.](image)

Typical simple buffer overflow attacks overflow the local buffer on the stack of the vulnerable programs. As an example, consider the following case. The attacking program first forms a string and provides it to the vulnerable program as an input. Common places to put this string include the command line argument to the vulnerable program and
environment variables. The vulnerable program copies this input string into its local buffer without boundary checking. In C language, the run-time function strcpy copies all of the source string to the destination until the null terminator of the source string is found. The length of the input string is greater than the size of the local buffer in the vulnerable program and is large enough to overwrite the return address part of the activation record adjacent to the local buffer when being copied to the buffer. The attacking program puts a memory address or a sequence of the same memory address in somewhere in this input string so that the return address part of the activation record of vulnerable program in which the local buffer is declared is overwritten by this memory address after the buffer copy. When the vulnerable program returns from the function in which the local buffer is declared, the corrupted return address is loaded into the program counter and the control will be transferred to the location specified by the memory address in the malicious input string. The memory address in the input string typically points to some malicious instructions the attacker wants to execute. And these instructions can also be put into somewhere in the input string.

The most common malicious instructions are to spawn a shell, so these type instructions are also commonly called shell code. One simple way to get a shell is to make an exec system call as follows.

```c
foo () {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    exec1 (name[0], name, NULL);
}
```
With the help of a debugger, the corresponding machine code can be easily found. The following Intel x86 instructions do the same work as the code above. First set EAX to 0xb, which is the system call code corresponding to exec, then store the three parameters in EBX, ECX, EDX, respectively and followed by an software interrupt, int 80.

```
mov $addr %esi /* here addr is the address to "/bin/sh" */
mov %esi,0x8(%esi)
xor %eax,%eax /* set EAX to 0 */
mov %al,0x7(%esi)
mov %eax,0xc(%esi)
mov $0xb,%al /* set EAX to Oxb */
mov %esi,%ebx
lea 0x8(%esi),%ecx
lea 0xc(%esi),%edx
int $0x80 /* make the system call */
```

In the assembly code above, the address of the string "/bin/sh" is still unknown before run time. There is a trick here to fill in the address dynamically at run time. It takes advantage of the fact that before control is transferred to the callee when a direct call is made, the PC (value of EIP) is pushed into the stack. So if the address of the string "/bin/sh" is put right behind a direct call instruction, then this address is dynamically pushed into the stack before control transfers and is known to the attacking code. The trick works as follows.

```
jmp $offset_to_call /* jump to the call instruction */
pop %esi /* pop the address of "/bin/sh" to %esi */
mov %esi,0x8(%esi)
xor %eax,%eax /* set EAX to 0 */
mov %al,0x7(%esi)
```
\begin{verbatim}
mov %eax,0xc(%esi)
mov $0xb,%al /* set EAX to 0xb */
mov %esi,%ebx
lea 0x8(%esi),%ecx
lea 0xc(%esi),%edx
int $0x80 /* make the system call */
call $offset_to_pop /* call the pop instruction */
"/bin/sh"
\end{verbatim}

When this shell code starts to execute, it jumps to the call instruction, the call instruction transfer control to the pop instruction, but before the control transfer happens, the PC, which is the memory address right after the call instruction, i.e., the address of the string "/bin/sh", is pushed on the stack. Then this address is popped into ESI register by the pop instruction so is known to the attacking code. The memory address offsets which are used in jmp and call instructions can be easily calculated using a debugger by counting the number of bytes those instructions take. These assembly instructions are converted into hexadecimal numbers to put into the input string. There can not be any zero hex numbers appeared in the shell code, otherwise the string copy functions in C will stop copying when it hits a null byte. To eliminate zero numbers in an instruction, some equivalent alternative instructions have to be used. For example, instruction "mov $0 %eax" can be substituted by another instruction "xor %eax, %eax", which does not contain any null bytes.

In the input string given to the vulnerable program, there is also a memory address which is used to overwrite the saved PC of the activation record. Normally, these address bytes are put in the end part of the input string and this address points to the beginning of the
buffer which is being overflowed. When the function returns, the bytes at the very beginning of the buffer are fetched and interpreted as a machine instruction to execute. If the buffer is in the stack, as long as the stack is executable, which is true for Intel x86 architecture, the control will transfer to the malicious code contained in the post-overflowed buffer. The probability of overwriting the saved PC on the stack can be greatly increased by putting adjacent copies of such return address and by trying different memory address alignments.

The best case for attack is that the address used to overflow the saved PC points to the start of the shell code in the overflowed buffer. But it requires many experiments to get the exact memory address. Much of the effort can be saved by putting some null instructions right in the front part of the shell code. Null instruction is a machine instruction which does nothing, it's encoded as 0x90 in x86 instruction set. So even if the address does not point to the shell code, as long as it points to one of the null instructions, the control will ultimately transfer to the shell code. If a shell is spawned by the shell code and the vulnerable program is owned by root and has SUID bit set, then root privilege can be gained.

**Section 3. Advanced Techniques**

The basic buffer overflow attack technique is discussed in last section. There are some advanced techniques which are useful in attacking those less vulnerable programs.

**Filter Passing**

There are many vulnerable programs having potential buffer overflow problems, but some of them are hard to exploit because they filter the input string. For example, a filter could convert all input characters into upper case or it could filter out some special characters like ‘#’, ‘!’, etc. For an even worse case, if a vulnerable program filters out all printable
characters, then it is really hard to exploit the vulnerability. Here a simpler case is considered. Assume that a program converts all characters in the input string to uppercase before copying them into the buffer. Then the shellcode cannot contain any lower case characters. The characters in the string "/bin/sh" can be easily converted to lower case characters after the buffer is overflowed by subtracting 56 from them, i.e., the offset of lower case and upper case characters. For the machine instructions with lower case characters, alternative instructions with no lower case characters have to be used and these substitute instructions generally are not too hard to find.

**Retrieve Root UID**

For local buffer overflow attacks, getting a root shell is the final goal. To obtain such a root privilege, the vulnerable program has to be owned by the root, can be executed from non-root user, and the SUID bit has to be set. Otherwise, even if the program could be exploited and a shell spawned, it still is a user shell and worth nothing to a local user. Many programmers think that by using `setuid(getuid())` function calls the root suid can be dropped. So they make such calls at the very beginning of the program and only call `setuid(0)` when root uid is required and drop it immediately after that. But that is not as safe as it might look. The attacker can retrieve the root uid by adding `setuid(0)` at the beginning of the shellcode.

**Remote Buffer Overflow Attacks**

Remote buffer overflow can be much more harmful since it does not require a local user account. The attack is mounted through the network, so it is harder to trace the attacker. Remote buffer overflows are also harder to exploit than the local buffer overflow. They generally need more work and much longer shellcode to open sockets. Since the vulnerable
programs run on remote machines, even if the attacker successfully overflows the buffer with shellcode, the shell spawned is on the remote machine. So to take advantage of the spawned shell, the network sockets need to be connected to the standard I/O ports of the attacker machine. To attack a TCP-based daemon, the general idea is to first open a connection to the vulnerable daemon, then send the shellcode via the network to overflow the buffer in the vulnerable daemon. After the buffer is successfully overflowed, the shellcode opens a backdoor port which can accept connection requests as a server and then spawn a shell. The shellcode then connects this port to stdin, stdout, stderr of the shell just spawned via dup2 system calls. On the client side, the attacking program then tries to connect to that backdoor port on the vulnerable machine. If it succeeds, then it also connects its stdin, stdout, stderr to the local socket connected to the remote machine. By using this approach, the attacker can take advantage of the spawned shell on the remote machine. Of course, the length of the shellcode is much larger given the complexity of those system calls for opening and binding sockets, listening and accepting connection request and changing I/O ports, but it is still possible to restrict the size of shellcode to about 200 bytes. For many vulnerable daemons, the input buffer is larger than this size.
Chapter 3. The Defenses

There are many ways of preventing buffer overflow that have been proposed and implemented. Generally, these methods fall into two wide categories, static methods and dynamic (run-time) methods. Static methods try to detect and eliminate the potential vulnerabilities before the deployment of program. Dynamic methods focus on invalidating any attack attempts at run time.

The disadvantage of runtime methods is that they increase the runtime overhead. Also when the attacks are detected, the running program typically will be shutdown, so the buffer overflow attacks are effectively converted to a denial-of-service attack for those server programs. The overhead of static method happens only before the software is deployed, they do not impose runtime overhead. But one drawback of the static methods is they normally cannot eliminate all of the possible vulnerabilities, and they can also incorrectly predict vulnerabilities, so all of these require further human auditing.

Section 1. Static Methods

Writing Correct Code

As we discussed in the previous chapter, the most common buffer overflows come from those dangerous function calls like `sprintf`, `gets`, etc., since these functions do not do buffer boundary checking for array and pointer references. These functions should be avoided as much as possible.

Unfortunately, programs that use just the “safe” subset of the C string API are not necessarily safe, because the “safe” string primitives have their own pitfalls. Function
strncpy may leave the target buffer unterminated. In comparison, strcat and
snprintf always append a terminating '\0' byte. Also using strncpy has
performance implication since it zero-fills all the available space in the buffer after the null
terminator. Furthermore, if these "safe" version functions are used with an elementary off-
by-one error, then there could be potential buffer overflow vulnerabilities.

Fault Injection

Fault injection technique is mainly used in software assurance for analyzing the
behavior of programs under those anomalous conditions such as unexpected input from
users. Anup Ghosh [6] applied this technique in FIST (The Fault Injection Security Tool) to
analyze potential buffer overflows in programs. FIST inserts malicious strings (instructions)
into potentially vulnerable buffers identified by the analyst during program execution and
observes the effect on system security. FIST employs several different fault injection
functions that simulate the effects of program errors and malicious threats against programs.
FIST can read the stack frame in which the vulnerable buffer resides and locate the return
address and overwrite this address to point to the buffer, by doing this, it can simulate a
potential buffer overflow attack. If the malicious code (such as modifying local file system)
gets executed, then a real attack can also happen.

Source Code Lexical / Syntax/ Semantics Analysis

There are many methods that fall in this category. Generally, these methods are used to
detect wide range of security holes, not just buffer overflows. We do not discuss them
extensively here.

Lexical analyzer can be as simple as using "grep" to catch dangerous function calls, or
as complicated as IST4 [7], which scan the source code to identify potential vulnerabilities
based on a database of dangerous constructs.

Syntax/semantic analyzers include [8], and LCLint (now called Splint) [9], as some examples.

Section 2. Dynamic Methods

Buffer Bounds Checking

The buffer overflow attacks happen only because of C’s feature of not checking bounds of array and pointer references. So if we have a C language extension that does bounds checking, it will automatically prevent any buffer overflow attacks.

In an extension to the GNU C compiler, Richard Jones and Paul Kelly [10] have developed a new method to enforce array bounds and pointers checking in the C language. They make the check without changing the representation of pointers, so checked code is compatible with unchecked code. But the performance penalties are substantial. For an $ijk$ matrix multiplication, there is a 30x slowdown reported.

Stack Integrity Check of libc Functions

Alexandre Snarskii [11] has written a patch to libc to make FreeBSD unexploitable with standard stack overflow attacks. This patch checks the integrity of stack before calls of those dangerous libc functions like strcpy, gets return. Because the patch is only for C library functions, vulnerabilities in user-defined functions still exist and continue to threaten the security of computer systems.

StackGuard

StackGuard [12] [13] is a compiler enhancement to provide integrity checking to the
return address in function activation records. When generating code for function entry, the compiler puts a canary word next to every return address in the stack. It generates code in the function epilogue to check the integrity of the canary before jumping to the return address. The canary is either a randomly generated value or a terminator (such as 0, which will effectively terminate those functions like `strcpy`), so StackGuard can either detect the compromise of the return address or prevent the modification of return address. But if attackers can guess the canary value, this method will not work. Also under certain conditions attackers could overwrite the pre-computed canary values with their own values to turn off the protection. Furthermore, due to alignment requirement, it is possible to overwrite a return address but skip over the canary word.

The major limitation of StackGuard is that it changes the format of an activation record. It will not be compatible with those programs that are introspective with respect to the format of data on the stack. For instance, GDB inspects other program's stack frames, and thus will fail to produce correct stack traces when applied to StackGuard-protected programs. The Linux kernel also does not compile under StackGuard.

**Non-Executable User Stack**

Injecting code into stack frames and overwriting the return address to point to the injected code is the most common form of buffer overflow attacks. So a natural solution would be making a stack non-executable. With a non-executable stack, even if an attacker can successfully load some malicious instructions on the victim program's stack and also successfully has control flow transfer to the malicious instructions, he still can not have the malicious instruction executed. Solar Designer [14] implemented such a solution by providing a patch for Linux kernel to make the stack non-executable. But this method only
works for traditional and standard stack attacks. Exploit code injected into data segment still can hijack the attacked program.

A big problem with this method is that sometimes the stack is required to be executable. For example, Linux signal handler returns need an executable stack. Nested function calls and trampoline functions also need an executable stack to work properly. And functional languages, e.g. LISP, also need an executable stack. In order to solve these problems, the patch needs to temporarily set the stack as executable when these events occur. But this also creates a window for attackers to launch a buffer overflow attack.

Another type of common buffer overflow attacks is to point the return address to a function in libc, usually system. To prevent such kind of attacks, the same patch also changes the default address that shared libraries are mmap()'ed at. But simply changing the default address that shared libraries are mmap()'ed is not enough. The fact is that exploit codes do not have to call libc functions like system() directly. If a vulnerable program calls libc function, the text segment of this program will contain a PLT (procedure linkage table) entry which can be used for exploiting purpose. There are already some exploits which can get around non-executable stack patch [15].
Chapter 4. Our Method

Most of the proposed protection or detection schemes focus on data. We, instead, focus on program counter integrity. Until the attack takes over the control by updating program counter for its advantage, the system secure is considered. Only program counter is necessary to be protected. For example, we encode the program counter to handle buffer overflow attacks.

Section 1. Program Counter Encoding

The basic idea is to encode every instances of program counter saved on the stack, which are vulnerable to buffer overflow attacks. During a procedure call, instead of storing the value of the program counter to the stack as the return address directly, we encode it by doing \( \text{MEM[} %sp \text{]} \leftarrow e(\text{PC}) \). On the return from a procedure call, when the return address in the activation record is restored into the program counter, it is decoded by a decoding function \( d \) such that \( d(e(\nu)) = \nu \).

This scheme works as long as every PC value goes through both the encoding and decoding processes. In a buffer overflow, the attacker’s intended return address will only go through the decoding stage and when its decoded value is loaded to the program counter, control flow will be transferred to somewhere other than the attacker’s intended location, and eventually it will lead to an exception. So this method suffices to prevent the most typical buffer overflow attacks.

Microarchitecture / Compiler Support

The program counter encoding/decoding can be implemented either on
microarchitecture level or on compiler level. In this paper, we will focus on the compiler support of PC encoding.

**Encoder/Decoder**

The encoding and decoding functions can have variable complexity, from as simple as bitwise exclusive-or to any arbitrary complex. Here we consider the simplest case, with a compiler support of bitwise exclusive-or encoder/decoder. To support encoding/decoding the PC value in the activation record, the compiler only needs to add some extra instructions in the function prologue and epilogue section. Some simple choices of the encoder/decoder key include the frame pointer and the stack pointer.

If the frame pointer FP is used as the encoding/decoding key, the function prologue will look like the following,

```c
xorl (%esp), %ebp, (%esp) /* use FP to encode the return address, which is the top of the stack at this time */
pushl %ebp /* save the last frame pointer */
movl %esp, %ebp /* update the frame pointer to the base address of this stack frame (of this function) */
```

and the function epilogue looks like,

```c
pop any local variables /* at this time, the top of the stack is the saved frame pointer of the last stack frame */
movl %ebp, %esp /* pop stack frame */
popl %ebp /* restore the last frame pointer, after this pop, the top of stack is the return address */
xorl (%esp), %ebp, (%esp) /* decoding the return address using the last frame pointer */
ret /* pop to PC */
```
The encoding/decoding can be implemented with only one extra instruction in the function prologue and epilogue, respectively. Obviously, the overhead of this security feature will be the least. One disadvantage of using frame pointer as the encoding/decoding key is that for efficiency consideration some program could be compiled with an option of not generating frame-pointer related code, gcc has such a compiling switch. In this case, the instruction 2 and 3 in the prologue, and the instruction 2 and 3 in epilogue will not be generated. Then this framework will not work unless the FP has the same value in the entry and exit points of the function call. Otherwise, the encoding and decoding keys are different and the result is unpredictable. In a typical buffer overflow, one same value (return address) is used to fill the very end part of the long string which is used to overflow a local buffer. That means both the saved frame pointer and the return address (on the stack) typically contain the same value since these two memory locations are adjacent to each other. Then in the decoding stage, the instruction

\[ \text{xorl} \ (\%\text{esp}), \%\text{ebp}, (\%\text{esp}), \]

which does

\[ \text{return address} \leftarrow \text{return address} \oplus \text{saved frame pointer} \]

will produce 0. After this 0 value being loaded to PC, an interrupt (exception) will be generated on most of the operating systems. So by using the frame pointer as the encoding/decoding key, it will provide a significant check of any potential buffer overflow attacks if a 0 address is loaded in PC.

Another choice of simple encoding/decoding key is the stack pointer. The function prologue looks like

\[ \text{xorl} \ (\%\text{esp}), \%\text{esp}, (\%\text{esp}) \ /* \text{use SP to encode the return} \]
address, which is the top of the stack at this time */
pushl %ebp /* save the last frame pointer */
movl %esp, %ebp /* update the frame pointer */

and the function epilogue looks like,

    pop any local variables /* at this time, the top of the stack is the saved frame pointer of the last stack frame */
movl %ebp, %esp /* pop off callee’s stack frame */
popl %ebp /* restore the last frame pointer, after this pop, the top of stack is the return address of the caller */
xorl (%esp),%esp,(%esp) /* decoding the return address using the stack pointer */
ret /* pop to PC */

From the function prologue and epilogue, we can see that the value of the keys for encoding and decoding are the same. This allows the correctness of the normal program execution in function calls. Also since the value of the stack pointer changes dynamically in program execution, so the encoding/decoding key also updates dynamically, which significantly improves the security of this PC encoding mechanism. The advantage of using stack pointer as the key is that it does not depend on the FP register, so that even if non-frame-pointer compilation option is chosen, the framework will still work. After a typical buffer overflow which tends to overwrite the return address and the saved frame pointer with the same value, the recovered frame pointer will be changed (points to the entry point of the malicious instructions typically). But since the SP value is not recovered from the stack, so the key used for decoding does not change, it will always be the same as the encoding key. Then the value which is loaded to PC will be the result of xor’ing the address intended by the
attacker and the stack pointer, and it will not be 0 except that the address happens to be the same as the stack pointer. Under Linux, every process has a virtual memory space of 4G bytes in size. The upper 1G is reserved for kernel usage, the lower 3G is used for text segment, data segment, heap and the user stack. The user stack starts from 0xBFFFFFFF and grows downward. For most of the programs that do not have a very deep function call trace, the stack pointer tends to be like 0xBFFFx.xxx. If the attacker injects the malicious code into stack, then the intended return address will also be like 0xBFFFx.xxx, then the result PC value will be 0x0000xxxx, this address does not fall into the text segment of a process under Linux since in Linux the text segment of a running process starting at 0x8048000. So an interrupt will also be generated for such a case.

Section 2. Implementation

Program counter encoding by compilation enhancement is selected for implementation here. To protect the return address in activation records, most of the compilation enhancement methods, such as StackGuard, need to add some runtime return address protection mechanisms in the function prologue and epilogue. Since every function call will have to run these extra instructions, the overall overhead increases rapidly. Obviously, there is a trade-off between security and performance. If the protection mechanism requires too many instructions in these sections, then the runtime overhead tends to be very large and the performance penalty will defeat the benefit of security provided by such a protection and makes the method practically impossible. Bearing this consideration in mind, we choose the simplest exclusive-or encoding method, with either the frame pointer or the stack pointer as the encoder/decoder key.
To implement this mechanism in a compiler, we only need to modify the code generation portion of function prologue and epilogue. The layout of the function stack frame stays the same, which will maintain the compatibility of compiled programs with other existing libraries. This is another advantage of this method, compared with other compiler enhancement approaches, which generally involve modifications of stack frame layout. For example, in StackGuard, the modified compiler needs to find place to fit a canary into the function stack frame. These features together reduce the implementation effort significantly.

With this security aware compiler in hand, we still need to recompile those critical libraries like C run time library and the kernel itself. In library routines, if local buffers (or buffers in the heap) are used and unsafe functions like strcpy are called to manipulate these buffers, then attacker can still seek to overrun these buffers indirectly. There is no fundamental difference between the buffers in user program and the buffers in library routines. Even though there are very few, if not any, buffer overflow attack can overrun buffers in shared libraries, but it is conceivable that future attacks might be able to do that. For the kernel, the same argument applies.

When the critical libraries like C run time library and kernel have security enhancement, the extra instructions added to function prologue and epilogue will be executed not only for the user routines, but also every C run time function call and every system call. These will significantly increase the overhead associated with the program counter encoding. For example, every strcpy call needs to execute at least two more instructions, one for encoding, one for decoding. So this puts even tighter constraints for the performance efficiency of a particular protection mechanism. For the StackGuard, it is not desirable at all to insert a canary into the activation record, check the integrity of the canary, and report the
compromise of the return address if non-integrity of the canary is found for a function call as simple as strcpy.

We use the GNU CC compiler version 2.95.3 and modify the code generation portion of function prologue and epilogue. Then we recompile the C run-time library libc version 2.1 from GNU and Linux kernel 2.4.3 with the secure aware compiler on an Intel x86 platform.

With our implementation, the change of function prologue and epilogue is shown in Figure 2.

Figure 2. Function prologue and epilogue with program counter encoding.

Function Prologue

```
pushl %ebp
movl esp,%ebp
subl $12,%esp
```

```
xorl(%esp) ,%ebp, (%esp)
pushl %ebp
movl %esp,%ebp
```

Function Epilogue

```
movl %ebp,%esp
popl %ebp
ret
```

```
movl %ebp,%esp
popl %ebp
xorl(%esp),%ebp,(%esp)
ret
```

Section 3. Performance Overhead

For most of the compiler enhancements of return address protection, the majority work, if not all, is done in function entry and exit sections. Assume that all of the enhancements are
done in these sections, the run time overhead of a program that is built with such a compiler enhancement is given by

\[
\text{Overhead} = n \times F / IC
\]

Where

\[
n = \text{number of extra instructions added into the function prologue and epilogue}
\]

\[
F = \text{number of function calls which have the extra instructions added in the prologue and epilogue}
\]

\[
IC = \text{instruction count (without the extra instructions added by the compiler enhancement)}
\]

So the theoretical overhead associated with a compiler enhancement method depends on the ratio of the number of function calls to the instruction count, of course, also on the number of extra instructions needed per function call. This ratio will vary across different type of applications and also depends on the style of source code, i.e., whether there are a lot of function calls. We can get a rough estimate of this ratio using the benchmark.

SPEC95 INT benchmarks were used for a rough estimation of the ratio of number of function calls verses the instruction count. Inside the SPEC95 INT package, there are 7 benchmarks, they are

1. "Go" application, an artificial intelligence application
2. "GCC", the well-known GCC compiler
3. "Perl", the perl interpreter
4. "Lisp", the Lisp interpreter
5. "Vortex", a database application
6. "Compress", a data compression application which is I/O intensive

7. "m88ksim", a Motorola 88k architecture simulator

To get full set of statistics of these benchmarks, we run them in a simulator, the SimpleSim, and use the sim-profiler to collect the statistics. The following table contains these statistics for some of the benchmarks. For some of the benchmark programs, such as “Go”, “Perl”, and “Lisp”, multiple data sets were used. The ratios of function calls over total instructions vary from 1.07% to 3.33%, with the most ratios falling around 2%. Suppose this ratio is about 3%, for our implementation, with either the frame pointer or the stack pointer as the encoding key, the number of extra instructions will be 2, and the predicted typical overhead will be 6%, which is almost ignorable.

Table 1. Ratio of function calls versus total instructions for SPEC95 CINT benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>Call Direct</th>
<th>Call Indirect</th>
<th>Total Function Calls</th>
<th>Ratio of Calls/Instrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go, Null data</td>
<td>35839364540</td>
<td>382303056</td>
<td>19</td>
<td>382303075</td>
<td>1.07%</td>
</tr>
<tr>
<td>Go, Train data</td>
<td>548130806</td>
<td>6028418</td>
<td>19</td>
<td>6028437</td>
<td>1.10%</td>
</tr>
<tr>
<td>GCC, Train data</td>
<td>253113281</td>
<td>2938332</td>
<td>156322</td>
<td>3094654</td>
<td>1.22%</td>
</tr>
<tr>
<td>Perl, Jumble data</td>
<td>2391507167</td>
<td>39824778</td>
<td>39824778</td>
<td>79649556</td>
<td>3.33%</td>
</tr>
<tr>
<td>Perl, Prism data</td>
<td>10513027</td>
<td>170619</td>
<td>34</td>
<td>170653</td>
<td>1.62%</td>
</tr>
<tr>
<td>Perl, Scrible data</td>
<td>40485598</td>
<td>732355</td>
<td>100</td>
<td>732455</td>
<td>1.81%</td>
</tr>
<tr>
<td>Lisp, Full Refer data</td>
<td>55438296151</td>
<td>1549371314</td>
<td>28951797</td>
<td>1578323111</td>
<td>2.85%</td>
</tr>
<tr>
<td>Lisp, Train Data</td>
<td>183285616</td>
<td>5075317</td>
<td>114729</td>
<td>5190046</td>
<td>2.83%</td>
</tr>
<tr>
<td>Vortex, Train Data</td>
<td>2520154652</td>
<td>53565291</td>
<td>14879</td>
<td>53580170</td>
<td>2.13%</td>
</tr>
</tbody>
</table>

But for other methods of return address protection with compiler enhancement, the extra instructions per function call can easily be more than 10 or 20 instructions. Then the overhead will be easily more than 30% or 60%.
Performance Experiments

The benchmark can give us a rough estimate of the performance overhead of our method. We also did a macro benchmark test for the overhead of our method. We measured the performance of Apache Web server using the WebStone benchmark [16], built with both normal GCC and our security aware compiler. The test was also done by Cowan [12] for the StackGuard. The WebStone benchmark tests performance of web server by sending simulated requests from different clients to the web server and measuring different statistics like response time to clients, server throughput, etc. This test was run on a Pentium 233MHz machine with 32M memory. The result of this test is shown in Table 2.

Table 2. Apache Web server performance with and without PC Encoding

<table>
<thead>
<tr>
<th># of Clients</th>
<th>Connection Rate (Connections/sec)</th>
<th>Average Latency (sec.)</th>
<th>Average Throughput (Mbit/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No PC Encoding</td>
<td>With PC Encoding</td>
<td>Overhead (%)</td>
</tr>
<tr>
<td>4</td>
<td>165.15</td>
<td>160.27</td>
<td>2.95%</td>
</tr>
<tr>
<td>8</td>
<td>168.18</td>
<td>159.18</td>
<td>5.35%</td>
</tr>
<tr>
<td>12</td>
<td>184.00</td>
<td>173.87</td>
<td>5.51%</td>
</tr>
<tr>
<td>16</td>
<td>184.33</td>
<td>184.40</td>
<td>-0.04%</td>
</tr>
<tr>
<td>20</td>
<td>192.62</td>
<td>191.53</td>
<td>0.57%</td>
</tr>
<tr>
<td>24</td>
<td>187.77</td>
<td>183.77</td>
<td>2.13%</td>
</tr>
<tr>
<td>28</td>
<td>193.20</td>
<td>192.53</td>
<td>0.35%</td>
</tr>
<tr>
<td>32</td>
<td>199.20</td>
<td>204.27</td>
<td>-2.55%</td>
</tr>
</tbody>
</table>

From Table 2, for the connection rate, the biggest overhead of PC encoding is 5.51% with 12 clients, and the Apache server even had a performance gain from PC encoding of up to 2.55%, when 32 clients were connected. For the average latency, the biggest overhead is 6.52% with 8 clients, but there was also a performance gain of 9.0% with 20 clients. For the average throughput of the web server, the biggest overhead is 9.98% with 8 clients, and there is also a performance gain of 1.3% with 16 clients. Theoretically, there should not be any
performance gains of the security aware web server, the performance gains we observed could be simply coming from noise. So generally speaking, even though there is some performance overhead, this overhead is so small that it is even not statistically significant.
Chapter 5. Exploit Experiments

To prove the effectiveness of the proposed method, few experiments of some exploits were conducted.

Section 1. Exploits

elm2.5.3

First this package is compiled with GCC 2.95.3. The original exploit written by _kiss, was modified to cope with wide range of memory addresses which can successfully exploit the elm program. In Red Hat 7.0, there is a pre-built elm with version 2.5.3 installed. The source package of the same version is obtained to enable the compilation with the security aware compiler. The exploit code is then used to exploit the original elm2.5.3. With an exploit buffer of size 300 bytes, one can successfully smash the internal buffer of elm with a wide range of offset. The shellcode is put in the middle of the exploit buffer, before that is NOPs, the memory after that are all filled with the return addresses. The exploit takes advantage of the "-f" option of the elm, which is used for forwarding a mail. Then the same source was compiled with GCC 2.95.3 and the same exploit was conducted, the results were similar to the results obtained from the pre-built version. Finally, the elm package were built with PC encoding, either with the frame pointer as the key or the stack pointer as the key. For the frame pointer as the PC encoding key, all of the exploits which work with elm built with non-security GCC failed, and the program halted with segmentation fault with the PC values as zeroes. For the case of exploiting the elm built with the security aware
compiler with stack pointer as the encoding key, all of the exploits failed too, with segmentation faults, and the PC values all had the pattern of "0x0000xxxx". These results matched the expectation.

**inn-2.2.3**

INN (InterNetNews), originally written by Rich Salz, is an extremely flexible and configurable Usenet / netnews news server. The following vulnerability exists in the "innfeed/misc.c" file.

```c
vsprintf (buffer,fmt,ap) ; /* line 159 */
```

The exploit code used an attack string of size of 600 bytes, filled with normal shellcode for spawning a shell in the middle, NOP's before the shellcode, and return addresses after the shellcode. With an INN that was compiled with standard GCC running, the exploit code could successfully overwrite the return address in the stack frame which contained the code shown above and obtained a shell that has a user id of "news", group id of "news". The return address that was used to overwrite the stack location of the original return address can be selected from a wide range, which means that this vulnerability can be easily exploited. With an INN compiled with the secured compiler with the frame pointer as the encoding/decoding key, none of the exploits that worked before worked. The program halted with segmentation fault and the program counter contained a 0. With an INN compiled with the secured compiler using the stack pointer as the encoding/decoding key, none of the exploits that worked before worked either. The program halted with a segmentation fault. The program counter contained values that had a pattern of "0x0000xxxx".

**Fancylogin-0.99.7**

This is a login manager program. This program has a local buffer of 128 bytes long...
which is used to store the host name for remote login. It uses a `strcpy` to copy from a command line argument to this buffer. The exploit code uses a string with length of 400 bytes filled with shellcode to overrun the buffer. When the login manager is compiled by a normal GCC compiler, the exploit code can obtain a shell. With the login manager built with the secure enhanced compiler, none of the exploits worked. The login manager halted with PC value of 0 when the encoding key is the frame pointer. If the login manager is built with the stack pointer as the encoding key, all of the PC values when the program halted had a pattern of “0x0000xxxx”.

**LPRng-3.6.22-5**

This is the default `lpr` print server installed on RedHat 7.0 system. It runs as a daemon with root privilege when system starts up. This version of LPRng has a buffer overflow vulnerability when it forms `syslog` strings. So by exploiting this server, attackers can get a full control of the target system. This is a remote buffer overflow exploit. The shellcode of this exploit contains the normal shellcode used in a typical local buffer overflow attack to spawn a shell. Beyond this, the shellcode also opens a TCP port (other than those well-know ports) and connects the `stdin/stdout` of the spawned shell to this port. By doing this, if the return address on the activation record which contains the buffer being successfully overwritten and the shellcode got executed inside the server, then the attacker can get an interactive shell with root `uid/gid`. Using a brute force method, an interactive root shell can be spawned inside one worker process of the `lpd`. Actually, it's not really a shell, it does not have a shell prompt, only the standard in/out of the shell are connected to network socket, so it accepts commands and dump the results of execution of the command. After recompile the source RPM with our PC encoding (with either the frame pointer or the stack pointer as
the encoding/decoding key) compiler, the worker processes of lpd exit when they get the malicious exploit string and so no shells can be spawned and the main lpd daemon is still alive taking further requests.

Section 2. Limitations and Future Work

From the different exploit experiments conducted, one can see that the proposed method is very effective in preventing the traditional “stack smashing” type of buffer overflow attacks. It can successfully halt the program execution with segmentation fault when the control flow is transferred to malicious instructions which are injected into the memory space of the running program by the attacking code. However, there are also some limitations in the method.

Detection of Attacks

At this time, the implementation does not support detection of compromised PC values, i.e., detection of attacks, which is desired under some circumstances. Generally, there are two ways to achieve this.

The first method is by checking the status of some objects other than the return address which is guaranteed to be modified when attackers break in. For example, StackGuard detects buffer overflow attacks by checking the integrity of the canary placed before the return address in the activation record. For the current implementation, there are no such objects available at this time. But this method will fail if attackers can overwrite the return address without modifying the indication object. For instance, future smarter attacks can potentially overwrite just the return address, leaving the memory locations around the return address untouched. Under such circumstances, methods like StackGuard will fail to prevent
the attack. But the secured compiler implementation will not, since it relies on return address encoding/decoding instead of checking the status of other indicators. In terms of this, the implementation is more robust.

Another way to detect attack is by keeping a copy of every PC values written to stack in some places that are safe enough and then compare these copies to the return addresses popped from the stack when functions return. On microarchitecture level, those places could be a parallel register stack for saving the trace of PC values, and be hidden to programs. But for a compiler support only, the only available place is the memory. And the memory region of saving these PC copies needs to be protected from possible reading/modifying by attackers.

So exploiting these and new methods to support compromised PC detection would be one feasible direction for future work.

**Preventing More Complicated Attacks**

At this time, the implementation does not support preventing non-stack-smashing type of buffer overflows, such as heap buffer overflow, function pointer related buffer overflow attacks, etc. This should be some of the possible areas for future research. Also for the traditional stack smashing attacks, smarter attacks might be able to guess the encoding/decoding function and the key. Making this more secure from being defeated is also an important research problem.
Chapter 6. Conclusions

Buffer overflow will continue to be one of the biggest sources of security vulnerabilities. This work examined the working mechanism of buffer overflow attacks, mainly the traditional attack format, i.e. "stack smashing". Different methods proposed to prevent buffer overflow attacks were also reviewed and our method for preventing buffer overflow attacks in run-time was presented. By encoding/decoding every PC values saved on stack, the proposed method can effectively prevent the traditional stack smashing type of buffer overflow attacks. The experiments conducted showed that the vulnerable software compiled with the enhanced GCC compiler halted in the case of a modified PC value as a result of a stack smashing attack. Performance experiments also showed that the method is quite competitive in terms of run-time performance. Furthermore, it almost does not impose any significant run-time overhead, which is another big advantage of the method compared with other similar methods in this class. In addition, it also has partial ability of detecting compromised PC values on stack, i.e., detecting of possible attacks. This approach is very successful in providing an efficient and effective protection to programs with potential buffer overflow flaws. On the other hand, the method has its own limitations. It does not provide protections from other advanced types of buffer overflow attacks like heap buffer overflow attacks. It does not prevent buffer overflow attacks via function pointer. But all of these limitations cannot keep this method from being a practically useful one to prevent the most common buffer overflow attacks.
References

[1]. Aleph One. “Smashing The Stack For Fun and Profit”. Phrack, 7(49), November of 1996.

[2]. Mudge. “How to Write Buffer Overflows”.

[3]. Nathan P. Smith. “Stack Smashing Vulnerabilities in the UNIX Operating System”.


