

Buffer-Overflow Vulnerabilities and Attacks

1 Memory

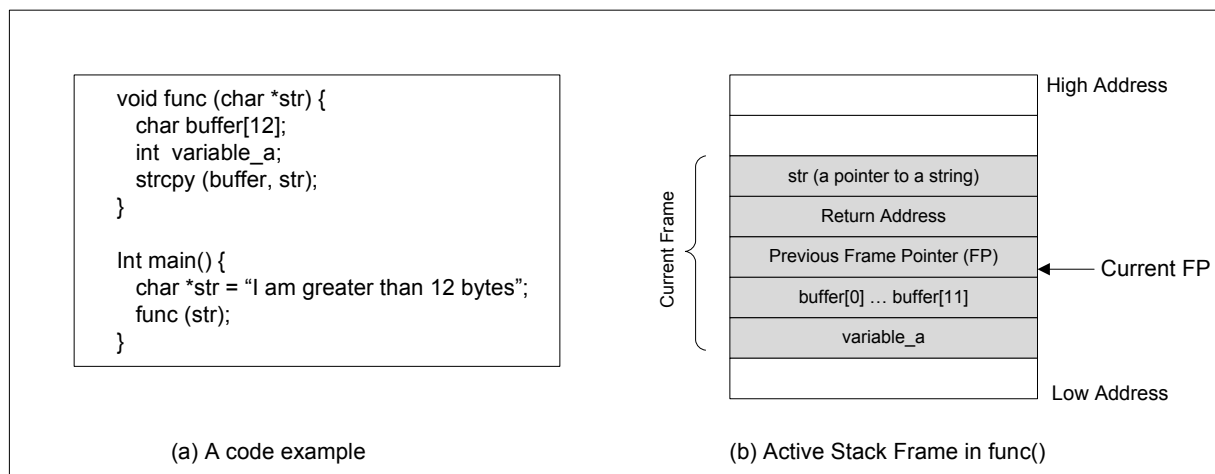
In the PC architecture there are four basic read-write memory regions in a program: Stack, Data, BSS (Block Started by Symbol), and Heap. The data, BSS, and heap areas are collectively referred to as the "data segment". In the tutorial titled "Memory Layout And The Stack" [1], Peter Jay Salzman described memory layout in a great detail.

- **Stack:** Stack typically located in the higher parts of memory. It usually "grows down": from high address to low address. Stack is used whenever a function call is made.
- **Data Segment**
 - **Data area:** contains global variables used by the program that are not initialized to zero. For instance the string "hello world" defined by `char s[] = "hello world"` in C would exist in the data part.
 - **BSS segment:** starts at the end of the data segment and contains all global variables that are initialized to zero. For instance a variable declared `static int i` would be contained in the BSS segment.
 - **Heap area:** begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by `malloc`, `realloc`, and `free`. The Heap area is shared by all shared libraries and dynamic load modules in a process.

2 Stack Buffer Overflow

2.1 Background about Stack

- Stack Layout: the following figure shows the stack layout after the execution has entered the function `func()`.



- **Stack Direction:** Stack grows from high address to low address (while buffer grows from low address to high address)

- Return Address: address to be executed after the function returns.
 - Before entering a function, the program needs to remember where to return to after return from the function. Namely the return address has to be remembered.
 - The return address is the address of the instruction right after the function call.
 - The return address will be stored on the stack. In Intel 80x86, the instruction `call func` will push the address of the next instruction that immediately follows the `call` statement into the stack (i.e. in the return address region), and then jumps to the code of function `func()`.
- Frame Pointer (FP): is used to reference the local variables and the function parameters. This pointer is stored in a register (e.g. in Intel 80x86, it is the `ebp` register). In the following, we use `$FP` to represent the value of the FP register.
 - `variable_a` will be referred to as `($FP-16)`.
 - `buffer` will be referred to as `($FP-12)`.
 - `str` will be referred to as `($FP+8)`.
- Buffer-Overflow Problem: The above program has a buffer-overflow problem.
 - The function `strcpy(buffer, str)` copies the contents from `str` to `buffer[]`.
 - The string pointed by `str` has more than 12 chars, while the size of `buffer[]` is only 12.
 - The function `strcpy()` does not check whether the boundary of `buffer[]` has reached. It only stops when seeing the end-of-string character `'\0'`.
 - Therefore, contents in the memory above `buffer[]` will be overwritten by the characters at the end of `str`.

2.2 A Vulnerable Program

Now, let us look at a more complicated program. Unlike the previous program, the string that is used to overflow the return address is not a static string; it is actually provided by users. In other words, users can decide what should be included in this string.

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int func (char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
}
```

```
        return 1;
    }

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    func (str);
    printf("Returned Properly\n");
    return 1;
}
```

It is not so difficult to see that the above program has a buffer overflow problem. The program first reads an input from a file called “badfile”, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 12 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. If this program is running as a set-root-uid program, a normal user can exploit this buffer overflow vulnerability and take over the root privileges.

2.3 Exploit the Buffer-Overflow Vulnerability

To fully exploit a stack buffer-overflow vulnerability, we need to solve several challenging problems.

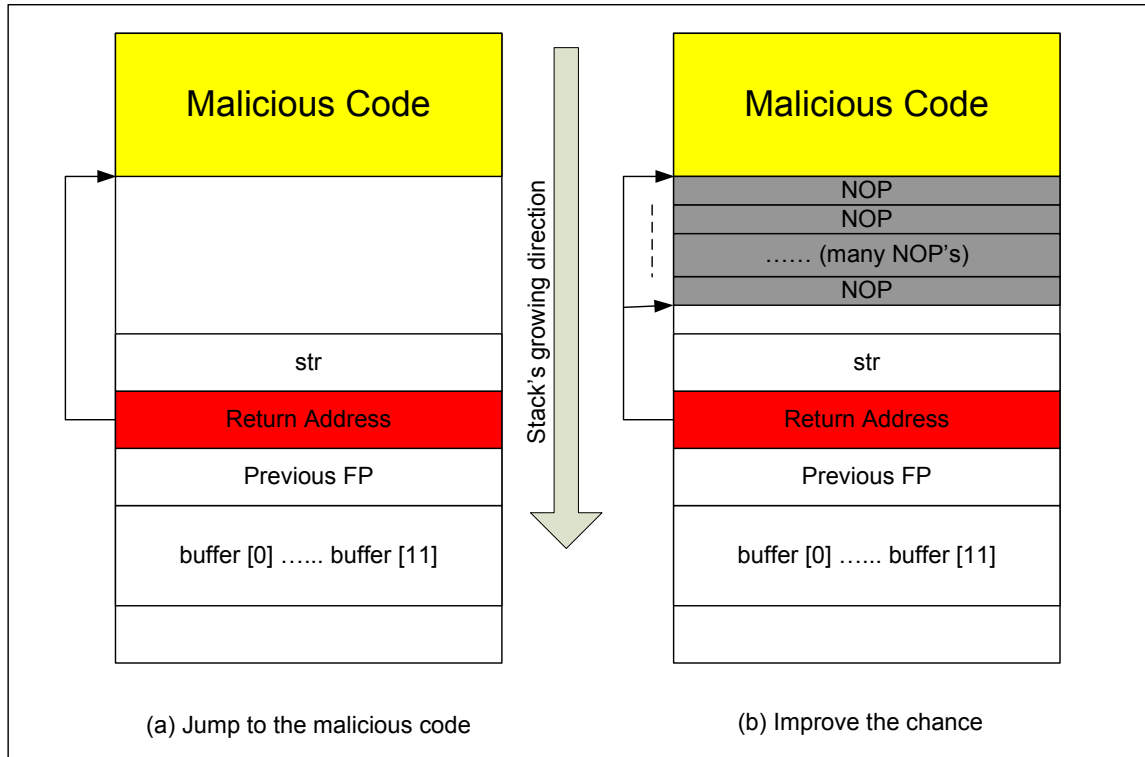
- **Injecting the malicious code:** We need to be able to inject the malicious code into the memory of the target process. This can be done if we can control the contents of the buffer in the targeted program. For example, in the above example, the program gets the input from a file. We can store the malicious code in that file, and it will be read into the memory of the targeted program.
- **Jumping to the malicious code:** With the malicious code already in the memory, if the targeted program can jump to the starting point of the malicious code, the attacker will be in control.
- **Writing malicious code:** Writing a malicious code is not trivial. We will show how a special type of malicious code, *shellcode*, can be written.

2.4 Injecting Malicious Code

With the buffer overflow vulnerability in the program, we can easily inject malicious code into the memory of the running program. Let us assume that the malicious code is already written (we will discuss how to write malicious code later).

In the above vulnerable program, the program reads the contents from the file “badfile”, and copy the contents to `buffer`. Therefore, we can simply store the malicious code (in binary form) in the “badfile”, the vulnerable program will copy the malicious code to the `buffer` on the stack (it will overflow the `buffer`).

2.5 Jumping to the Malicious Code



- To jump to the malicious code that we have injected into the target program's stack, we need to know the absolute address of the code. If we know the address before hand, when overflowing the buffer, we can use this address to overwrite the memory that holds the return address. Therefore, when the function returns, it will return to our malicious code.
- The challenge is to find where the malicious code starts.
- If the target program is a `Set-UID` program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a `Set-UID` program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of of your copy, but you should be quite close. You can try several values.
- If the target program is running remotely, and you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a quite feasible approach:
 - Stack usually starts at the same address.
 - Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
 - Therefore the range of addresses that we need to guess is actually quite small.
- **Improving the chance:** To improve the chance of success, we can add many NOP operations to the beginning of the malicious code. NOP is a special instruction that does nothing other than advancing

to the next instruction. Therefore, as long as the guessed address points to one of the NOPs, the attack will be successful. With NOPs, the chance of guessing the correct entry point to the malicious code is significantly improved.

2.6 Malicious Code: Shellcode

In the previous discussion, we assume that the malicious code is already available. In this subsection, we discuss how to write such malicious code.

If we can ask the privileged program to run our code, what code do we want it to run? The most powerful code that we want it to run is to invoke a shell, so we can run any command we want in that shell. A program whose only goal is to launch a shell is called a *shellcode*. To learn how to write a shellcode, let us see the following C program:

```
#include <stdio.h>

int main( ) {
    char *name[2];

    name[0] = ``/bin/sh``;
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

After we compile the above program into binary code, can we directly use the binary code as our shellcode in the buffer-overflow attack? Things are not that easy. There are several problems if we directly use the above code:

- First, to invoke the system call `execve()`, we need to know the address of the string “/bin/sh”. Where to store this string and how to derive the location of this string are not trivial problems.
- Second, there are several NULL (i.e., 0) in the code. This will cause `strcpy` to stop. If the vulnerability is caused by `strcpy`, we will have a problem.

To solve the first problem, we can push the string “/bin/sh” onto stack, and then use the stack pointer `esp` to get the location of the string. To solve the second problem, we can convert the instructions that contain 0 into another instructions that do not contain 0. For example, to store 0 to a register, we can use XOR operation, instead of directly assigning 0 to that register. The following is an example of shellcode in assembly code:

```
Line 1:  xorl    %eax,%eax
Line 2:  pushl   %eax           # push 0 into stack (end of string)
Line 3:  pushl   $0x68732f2f     # push "//sh" into stack
Line 4:  pushl   $0x6e69622f     # push "/bin" into stack
Line 5:  movl    %esp,%ebx       # %ebx = name[0]
Line 6:  pushl   %eax           # name[1]
Line 7:  pushl   %ebx           # name[0]
Line 8:  movl    %esp,%ecx       # %ecx = name
Line 9:  cdql                   # %edx = 0
```

```

Line 10: movb    $0x0b,%al
Line 11: int     $0x80          # invoke execve(name[0], name, 0)

```

A few places in this shellcode are worth mentioning:

- First, the third instruction pushes “//sh”, rather than “/sh” into the stack. This is because we need a 32-bit number here, and “/sh” has only 24 bits. Fortunately, “//” is equivalent to “/”, so we can get away with a double slash symbol.
- Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively.
 - Line 5 stores `name[0]` to `%ebx`;
 - Line 8 stores `name` to `%ecx`;
 - Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdql`) used here is simply a shorter instruction.
- Third, the system call `execve()` is called when we set `%al` to 11, and execute “`int $0x80`”.

If we convert the above shellcode into binary code, and store it in an array, we can call it from a C program:

```

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

const char code[] =
    "\x31\xc0"    /* Line 1:  xorl    %eax,%eax          */
    "\x50"        /* Line 2:  pushl   %eax              */
    "\x68" "//sh" /* Line 3:  pushl   $0x68732f2f       */
    "\x68" "/bin" /* Line 4:  pushl   $0x6e69622f       */
    "\x89\xe3"    /* Line 5:  movl   %esp,%ebx         */
    "\x50"        /* Line 6:  pushl   %eax              */
    "\x53"        /* Line 7:  pushl   %ebx              */
    "\x89\xe1"    /* Line 8:  movl   %esp,%ecx         */
    "\x99"        /* Line 9:  cdql                      */
    "\xb0\x0b"    /* Line 10: movb   $0x0b,%al         */
    "\xcd\x80"    /* Line 11: int    $0x80             */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}

```

The statement `((void(*)())buf)()` in the above `main` function will invoke a shell, because the shellcode is executed.

3 Countermeasures

3.1 Apply Secure Engineering Principles

- Use strong type language, e.g. java, C#, etc. With these languages, buffer overflows will be detected.
- Use safe library functions.
 - Functions that could have buffer overflow problem: `gets`, `strcpy`, `strcat`, `sprintf`, `scanf`, etc.
 - These functions are safer: `fgets`, `strncpy`, `strncat`, and `snprintf`.

3.2 Systemic Code Modification

- **StackShield**: separate control (return address) from data.
 - It is a GNU C compiler extension that protects the return address.
 - When a function is called, StackShield copies away the return address to a non-overflowable area.
 - Upon returning from a function, the return address is stored. Therefore, even if the return address on the stack is altered, it has no effect since the original return address will be copied back before the returned address is used to jump back.
- **StackGuard**: mark the boundary of buffer
 - Observation: one needs to overwrite the memory before the return address in order to overwrite the return address. In other words, it is difficult for attackers to only modify the return address without overwriting the stack memory in front of the return address.
 - A canary word can be placed next to the return address whenever a function is called.
 - If the canary word has been altered when the function returns, then some attempt has been made on the overflow buffers.
 - StackGuard is also built into GNU C compiler.
 - We can understand how StackGuard work through the following program (we emulate the compiler, and manually add the protection code to the function). For the sake of simplicity, we only use an integer for the canary word in the following example; this is not strong enough. We can use several integers for the canary word.

```
/* This program has a buffer overflow vulnerability. */
/* However, it is protected by StackGuard */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int func (char *str)
{
    int canaryWord = secret;
    char buffer[12];
```

```
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);

if (canaryWord == secret) // Return address is not modified
    return 1;
else // Return address is potentially modified
    { ... error handling ... }
}

static int secret; // a global variable

int main(int argc, char **argv)
{
    // getRandomNumber will return a random number
    secret = getRandomNumber();

    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    func (str);
    printf("Returned Properly\n");
    return 1;
}
```

3.3 Operating System Approach

- **Address Space Randomization:** Guessing the addresses of the malicious code is one of the critical steps of buffer-overflow attacks. If we can make the address of the malicious code difficult to predict, the attack can be more difficult. Several Linux distributions have already used *address space randomization* to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult. The following commands (can only run by root) enable or disable the address space randomization.

```
# sysctl -w kernel.randomize_va_space=2 // Enable Randomization
# sysctl -w kernel.randomize_va_space=0 // Disable Randomization
```

Unfortunately, in 32-bit machines, even if the the addresses are randomized, the entropy is not large enough against random guesses. In practice, if you try many times, your chance of success is quite high. Our experience has shown that a few minutes of tries are enough to succeed in a Intel 2GHz machine.

- **Non-executable stack:** From the attack, we can observe that the attackers put the malicious code in the stack, and jump to it. Since the stack is a place for data, not for code, we can configure the stack to be non-executable, and thus preventing the malicious code from being executed.

This protection scheme is called *ExecShield*. Several Linux distributions have already implemented this protection mechanism. ExecShield essentially disallows executing any code that is stored in the stack. The following commands (can only run by root) enable or disable ExecShield.

```
# sysctl -w kernel.exec-shield=1 // Enable ExecShield
# sysctl -w kernel.exec-shield=0 // Disable ExecShield
```

In the next section, we can see that such a protection scheme does not solve the buffer-overflow problem, because another type of attack, called *Return-to-libc* attack does not need the stack to be executable.

4 Non-Executable Stack and Return-to-libc Attack

To exploit the stack-based buffer overflow vulnerability, adversaries need to inject a piece of code into the user stack, and then execute the code from the stack. If we can make the memory segment used for stack non-executable, even if the code is injected into the stack, the code will not be able to execute. This way, we can prevent the buffer-overflow attacks. Technically this is easy to achieve because modern CPU architectures (such as Intel 386) do allow operating systems to turn a block of memory into non-executable memory. However, things are not so easy: many operating systems, such as Linux, do save code into stacks, and thus need the stack to be executable. For example, in Linux, to handle signals, a small sequence of code is put on the user stack; this sequence will be executed when handling signals.

Newer versions of Linux have since made the stack for data only. Therefore, stacks can be configured to be non-executable. In Fedora Linux, we can run the following commands to make stacks non-executable:

```
# /sbin/sysctl -w kernel.exec-shield=1
```

Unfortunately, making stacks non-executable cannot totally defeat the buffer-overflow attacks. It makes running malicious code from the stack infeasible, but there are other ways to exploit buffer-overflow vulnerabilities, without running any code from the stack. *Return-to-libc attack* is such an attack.

To understand this new attack, let us recall the main purpose of running the malicious shellcode from the stack. We know it is to invoke a shell. The question is whether we can invoke a shell without using injected code. This is actually doable: we can use the code in the operating system itself to invoke a shell. More specifically, we can use the library functions of operating systems to achieve our goal. In Unix-like operating systems, the shared library called `libc` provides the C runtime on UNIX style systems. This library is essential to most C programs, because it defines the “system calls” and other basic facilities such as `open`, `malloc`, `printf`, `system`, etc. The code of `libc` is already in the memory as a shared runtime library, and it can be accessed by all applications.

Function `system` is one of the functions in `libc`. If we can call this function with the argument “`/bin/sh`”, we can invoke a shell. This is the basic idea of the Return-to-libc attack. The first part of Return-to-libc attack is similar to the attack using shellcode, i.e., it overflows the buffer, and modify the return address on the stack. The second part is different. Unlike the shellcode approach, the return address is not pointed to any injected code; it points to the entry point of the function `system` in `libc`. If we do it correctly, we can force the target program to run `system("/bin/sh")`, which basically launches a shell.

Challenges. To succeed in the Return-to-libc attack, we need to overcome the following challenges:

- How to find the location of the function `system`?
- How to find the address of the string `"/bin/sh"`?
- How to pass the address of the string `"/bin/sh"` to the `system` function?

4.1 Finding the location of the `system` function.

In most Unix operating systems, the `libc` library is always loaded into a fixed memory address. To find out the address of any `libc` function, we can use the following `gdb` commands (let `a.out` is an arbitrary program):

```
$ gdb a.out
(gdb) b main
(gdb) r
(gdb) p system
      $1 = {<text variable, no debug info>} 0x9b4550 <system>
(gdb) p exit
      $2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

From the above `gdb` commands, we can find out that the address for the `system()` function is `0x9b4550`, and the address for the `exit()` function is `0x9a9b70`. The actual addresses in your system might be different from these numbers.

We can also use functions `dlopen` and `dlsym` to find out the address location of a `libc` function:

```
#include <dlfcn.h>

#define LIBCPATH "/lib/libc.so.6" /* on Fedora */

void *libh, *sys;

if ((libh = dlopen(LIBCPATH, RTLD_NOW)) == NULL) {
    // report error
}

if ((sys = dlsym(libh, "system")) == NULL) {
    // report error
}

printf("system @ %p\n", sys);
```

4.2 Finding the address of `"/bin/sh"`.

There are many ways to find the address of such a string:

- Insert the string directly into the stack using the buffer overflow problem, and then guess its address.
- Before running the vulnerable program, create an environment variable with value `"/bin/sh"`. When a C program is executed from a shell, it inherits all the environment variables from the shell. In the following, we define a new shell variable `MY_SHELL` and let its value be `/bin/sh`:

```
$ export MYSHELL=/bin/sh
```

We will use the address of this variable as an argument to `system()` call. The location of this variable in the memory can be found out easily using the following program:

```
void main()
{ char* shell = getenv("MYSHELL");
  if (shell)
    printf("%x\n", shell);
}
```

If the stack address is not randomized, we will find out that the same address is printed out. However, when we run another program, the address of the environment variable might not be exactly the same as the one that you get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes difference). The good news is, the address of the shell will be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed.

- We also know that the function `system` uses `"/bin/sh"` in its own code. Therefore, this string must exist in `libc`. If we can find out the location of the string, we can use directly use this string. You can search the `libc` library file (`/lib/libc.so.6`) for the string `"rodata"`:

```
$ readelf -S /lib/libc.so.6 | egrep 'rodata'
[15] .rodata PROGBITS 009320e0 124030 .....
```

The result of the above command indicates that the `".rodata"` section starts from `0x009320e0`. The `".rodata"` section is used to store constant data, and the constant string `"/bin/sh"` should be stored in this section. You can write a program to search for the string in the memory starting from `0x00932030`.

4.3 Passing the address of `"/bin/sh"` to `system`.

In order to let `system` run the command `"/bin/sh"`, we need to pass the address of this command string as an argument to `system`. Just like invoking any function, we need to pass the argument via the stack. Therefore, we need to put the argument in a correct place on the stack. To do this correctly, we need to clearly understand how the stack frame for a function is constructed when invoking a function. We use a small C program to understand the effects of a function invocation on the stack.

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
  printf("Hello world: %d\n", x);
}

int main()
{
  foo(1);
}
```

```
    return 0;
}
```

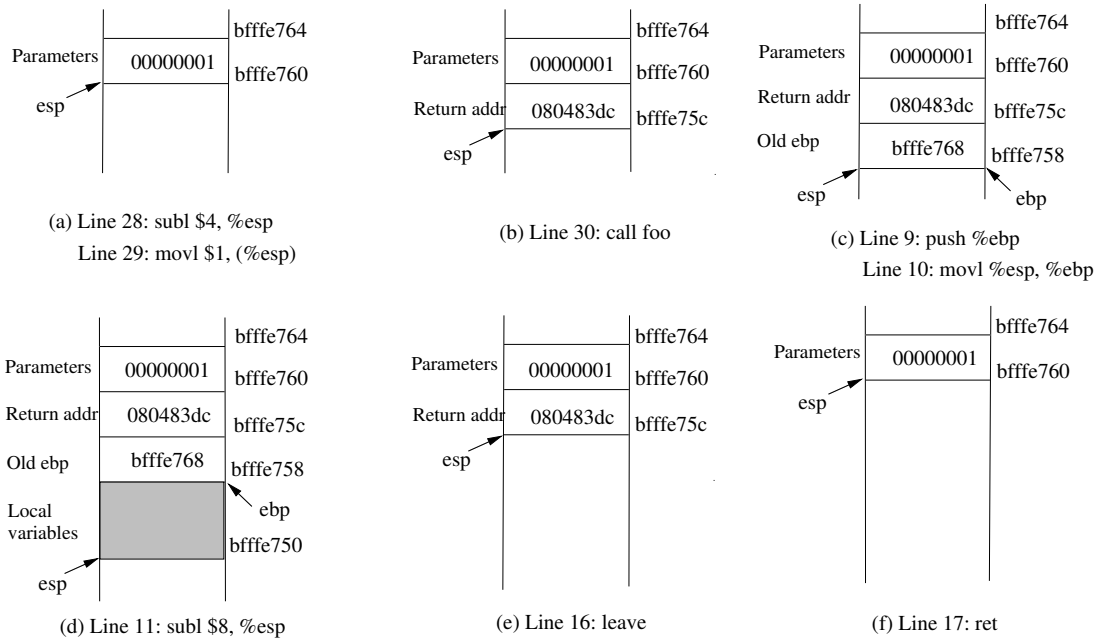
We can use "gcc -S foobar.c" to compile this program to the assembly code. The resulting file foobar.s will look like the following:

```
    .....
 8 foo:
 9     pushl   %ebp
10     movl   %esp, %ebp
11     subl   $8, %esp
12     movl   8(%ebp), %eax
13     movl   %eax, 4(%esp)
14     movl   $.LC0, (%esp) : string "Hello world: %d\n"
15     call   printf
16     leave
17     ret

    .....
21 main:
22     leal   4(%esp), %ecx
23     andl   $-16, %esp
24     pushl  -4(%ecx)
25     pushl  %ebp
26     movl   %esp, %ebp
27     pushl  %ecx
28     subl   $4, %esp
29     movl   $1, (%esp)
30     call   foo
31     movl   $0, %eax
32     addl   $4, %esp
33     popl   %ecx
34     popl   %ebp
35     leal  -4(%ecx), %esp
36     ret
```

Calling and Entering `foo()`. Let us concentrate on the stack while calling `foo()`. We can ignore the stack before that. Please note that line numbers instead of instruction addresses are used in this explanation.

- **Line 28-29:** These two statements push the value 1, i.e. the argument to the `foo()`, into the stack. This operation increments `%esp` by four. The stack after these two statements is depicted in Figure 1(a).
- **Line 30: `call foo`:** The statement pushes the address of the next instruction that immediately follows the `call` statement into the stack (i.e the return address), and then jumps to the code of `foo()`. The current stack is depicted in Figure 1(b).
- **Line 9-10:** The first line of the function `foo()` pushes `%ebp` into the stack, to save the previous frame pointer. The second line lets `%ebp` point to the current frame. The current stack is depicted in Figure 1(c).

Figure 1: Entering and Leaving `foo()`

- **Line 11: `subl $8, %esp`:** The stack pointer is modified to allocate space (8 bytes) for local variables and the two arguments passed to `printf`. Since there is no local variable in function `foo`, the 8 bytes are for arguments only. See Figure 1(d).

Leaving `foo()`. Now the control has passed to the function `foo()`. Let us see what happens to the stack when the function returns.

- **Line 16: `leave`:** This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

```
mov  %ebp, %esp
pop  %ebp
```

The first statement release the stack space allocated for the function; the second statement recover the previous frame pointer. The current stack is depicted in Figure 1(e).

- **Line 17: `ret`:** This instruction simply pops the return address out of the stack, and then jump to the return address. The current stack is depicted in Figure 1(f).
- **Line 32: `addl $4, %esp`:** Further restore the stack by releasing more memories allocated for `foo`. As you can clearly see that the stack is now in exactly the same state as it was before entering the function `foo` (i.e., before line 28).

Setting up the frame for `system()`. From Lines 9 and 10, we can see that the first thing that a function does is to push the current `%ebp` value to stack, and then set the register `%ebp` to the top of the stack. Although we see this from our example function `foo()`, other functions behave the same, including those

functions in `libc`. Therefore, within each function, after executing the first two instructions, `%ebp` points to the the frame pointer of the previous frame, `(%ebp + 4)` points to the return address, and the location above the return address should be where the arguments are stored. For function `system()`, `(%ebp + 8)` should be the address of the string passed to the function.

Therefore, if we can figure out what the stack pointer `%esp` points to after returning from `foo()`, we can put the the address of the string `"/bin/sh"` to the correct place, which is `(%esp + 4)`. For example, in Figure 1(d), if we want the function `foo` to return to `system`, we should put the starting address of function `system` at `%esp - 4` (`0xbfffe756`), a return address at `%esp` (`0xbfffe760`) and the address of the string `"/bin/sh"` at `(%esp + 4)` (`0xbfffe764`).

If we want the function `system()` to return to another function, such as `exit(0)`, we can use the starting address of function `exit()` as the return address of `system`, and put it in `0xbfffe760`.

Note: Details of how to setup the frame for `system()` are intentionally left out. Students are asked to work on a lab, in which they need to figure out all the details of the return-to-libc attack. We do not want this lecture note to give students all the details.

4.4 Protection in `/bin/bash`

If the `"/bin/sh"` is pointed to `"/bin/bash"`, even if we can invoke a shell within a `Set-UID` program that is running with the root privilege, we will not get the root privilege. This is because `bash` automatically downgrades its privilege if it is executed in the `Set-UID` root context;

However, there are ways to get around this protection scheme. Although `/bin/bash` has restriction on running `Set-UID` programs, it does allow the real root to run shells. Therefore, if we can turn the current `Set-UID` process into a real root process, before invoking `/bin/bash`, we can bypass that restriction of `bash`. The `setuid(0)` system call can help you achieve that. Therefore, we need to first invoke `setuid(0)`, and then invoke `system("/bin/sh");` all of these can be done using the Return-to-libc mechanism.

Basically, we need to “return to libc” twice. We first let the target program to return to the `setuid` function in `libc`. When this function returns, it will fetch the return address from the stack, and jump to that address. If we can let this return address point to `system`, we can force the function `setuid` to return to the entry point of `system`. We have to be very careful when conducting this process, because we have to put the appropriate arguments in the right place of the stack.

5 Heap/BSS Buffer Overflow

- Contents in Heap/BSS
 - Constant strings
 - Global variables
 - Static variables
 - Dynamic allocated memory.
- Example: Overwriting File Pointers

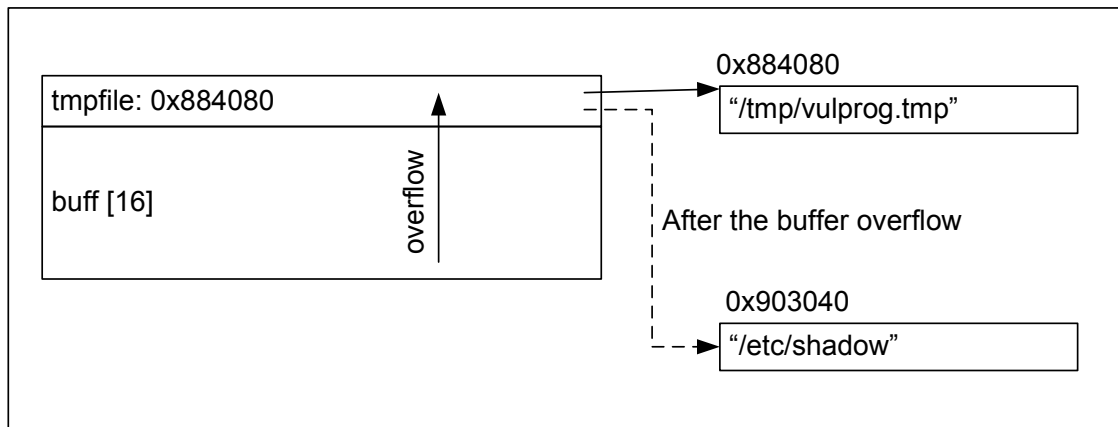
```
/* The following variables are stored in the BSS region */
static char buf[BUFSIZE], *tmpfile;
```

```

tmpfile = "/tmp/vulprog.tmp";
gets(buf); /* buffer overflow can happen here */

... Open tmpfile, and write to it ...

```



- The (Set-UID) program's file pointer points to /tmp/vulprog.tmp.
- The program needs to write to this file during execution using the user's inputs.
- If we can cause the file pointer to point to /etc/shadow, we can cause the program to write to /etc/shadow.
- We can use the buffer overflow to change the content of the variable tmpfile. Originally, it points to the "/tmp/vulprog.tmp" string. Using the buffer overflow vulnerability, we can change the content of tmpfile to 0x903040, which is the address of the string "/etc/shadow". After that, when the program use tmpfile variable to open the file to write, it actually opens the shadow file.
- How to find the address of /etc/shadow?
 - * We can pass the string as the argument to the program, this way the string /etc/shadow is stored in the memory. We now need to guess where it is.

- Example: Overwriting Function Pointers

```

int main(int argc, char **argv)
{ static char buf[16]; /* in BSS */
  static int (*funcptr)(const char *str); /* in BSS */

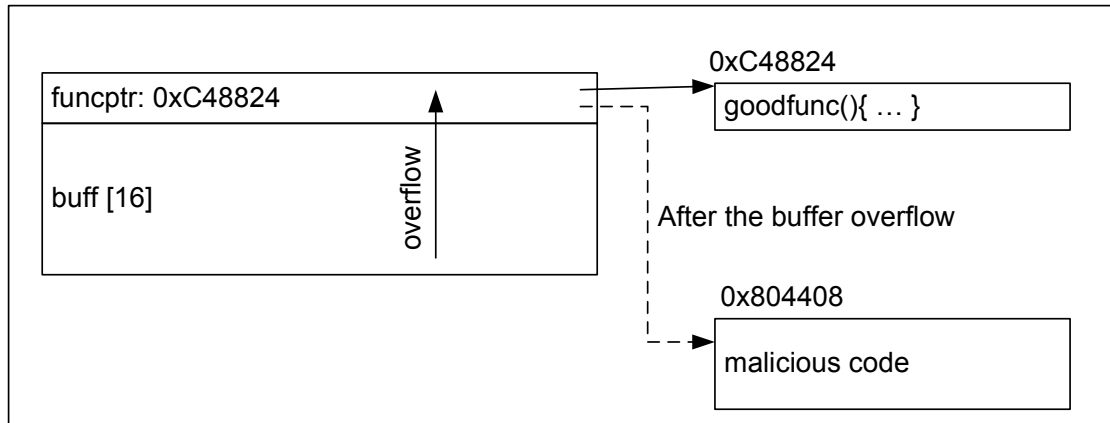
  funcptr = (int (*)(const char *str))goodfunc;

  /* We can cause buffer overflow here */
  strncpy(buf, argv[1], strlen(argv[1]));

  (void) (*funcptr) (argv[2]);
  return 0;
}

```

```
/* This is what funcptr would point to if we didn't overflow it */
int goodfunc(const char *str) { ... ... }
```



- A function pointer (i.e., "int (*funcptr) (char *str) ") allows a programmer to dynamically modify a function to be called. We can overwrite a function pointer by overwriting its address, so that when it's executed, it calls the function we point it to instead.
- argv[] method: store the shellcode in an argument to the program. This causes the shellcode to be stored in the stack. Then we need to guess the address of the shellcode (just like what we did in the stack-buffer overflow). This method requires an executable stack.
- Heap method: store the shellcode in the heap/BSS (by using the overflow). Then we need to guess the address of the shellcode, and assign this estimated address to the function pointer. This method requires an executable heap (which is more likely than an executable stack).

- Function Pointers

- Function pointers can be stored in heap/BSS through many different means. They do not need to be defined by the programmer.
- If a program calls `atexit()`, a function pointer will be stored in the heap by `atexit()`, and will be invoked when the program terminates.
- The `svc/rpc` registration functions (`librpc`, `libnsl`, etc.) keep callback functions stored on the heap.

- Other Examples

- The BSDI `crontab` heap-based overflow: Passing a long filename will overflow a static buffer. Above that buffer in memory, we have a `pwd` structure, which stores a user name, password, uid, gid, etc. By overwriting the uid/gid field of the `pwd`, we can modify the privileges that `crond` will run our `crontab` with (as soon as it tries to run our `crontab`). This script could then put out a `suid root shell`, because our script will be running with uid/gid 0.

References

- [1] P. J. Salzman. *Memory Layout And The Stack*. In Book Using GNU's GDB Debugger. URL: http://dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php.