The Hong Kong Polytechnic University

COMP 3334 – Computer Systems Security (Semester 2, Winter 2024)

Lab 3 – Buffer Overflow Vulnerability

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. The objective of this lab is for students to gain practical insights into this type of vulnerability, and learn how to exploit the vulnerability in attacks. 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, first to crash the program, then to change the flow of execution and let parts of the code run that was not intended to run at that moment.

1 Environment Setup

Follow these steps:

1. First, open VirtualBox (look on the Windows desktop), and start the *COMP3334-SEED-Lab* virtual machine.
2. Next, we want to allow drag-and-drop between the Windows host and the Linux VM. In VirtualBox, on the started VM, go to `Devices` menu, then *Drag and Drop*, and select *Bidirectional*.
3. Similarly, to synchronize the clipboard in and out of the VM, so you can easily copy-paste instructions and commands, return to the *Devices* menu, then *Shared Clipboard*, and select *Bidirectional*.
4. Finally, the following normal user is logged in automatically, here are credentials if needed:

   ```
   Username: seed
   Password: dees
   ```

2 Exercises

Exercise 1: Get started with the assembly language [50min]

The x86 instruction set architecture is at the heart of CPUs that power our home computers and remote servers for over 40+ years. Being able to read and write code in low-level assembly language is a valuable skill to have, and is required to understand buffer overflow vulnerabilities in C programs. In this exercise, you will familiarize yourself with the 64-bit variant of the x86 instruction set, also called x86-64 or x64 for short.

Registers

The following list shows the description of main *registers* available to a program. A register is a tiny memory inside the CPU. Each of these registers are 64 bits, i.e., 8 bytes long:

1. `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, and `r8` to `r15`: These registers are "general purpose", and can be used to store any data. `rax` specifically contains the return value of a function.
2. `rsp`, `rbp`: These important registers delimit parts of the stack, a memory region inside a process memory. `rbp` is the stack base pointer, while `rsp` is the stack pointer for top address of the stack.
3. `rip`: This special register stores the address of the next instruction to run, and cannot be directly accessed. It evolves automatically as the program runs.

Basic instructions

For the purpose of this lab, you need to understand the meaning and behavior of ten simple instructions. We will follow the AT&T assembly convention, which is the same as what we covered in class. Warning: Depending on the assembly convention you use outside of this lab, or with different tools, operands (parameters) of the following instructions may be swapped!

mov. Copies the content of one register into another, or from one register to a memory address. If the destination register is surrounded by parenthesis, the destination will actually be the address pointed by the register, rather than the register itself.
Example: `mov %rax, %rbx` will copy the value stored in `rax` into `rbx`. Notice the % before the register name, this is a simple convention we use to denote a register in an instruction.
Example 2: `mov %rax, (%rbx)` will copy the value stored in `rax` to the memory location pointed by `rbx`.
Example 3: `mov %rax, -0x10(%rbp)` will first take the value in `rbp`, subtract 0x10 to it, then copy the value stored in `rax` to that memory address.

push. Writes the content of a given register at the top of the stack. We say that we *push* a value onto the stack. The address of the top of the stack is tracked by `rsp`, so this is where the data will be written. Next, when the stack grows due to a push operation, the stack pointer `rsp` is updated so that the *new* top of the stack is above the data that has just been written. The register `rsp` moves to lower memory addresses, because the stack grows downward in most modern operating systems (cf. Lecture 8). Since a register in our case is 8 bytes, pushing a register onto the stack will grow the stack by 8 bytes, and therefore move the address stored in `rsp` down by 8 bytes.
Example: `push %rax` writes the content of `rax` at the address pointed by `rsp`, and updates `rsp`=`rsp`−8.

pop. The counterpart to `push` is `pop`. It reads the value on the top of the stack, copies it to the given register, then shrinks the stack by lowering the top of the stack, which in turn translates into increasing the value of `rsp` by 8.
Example: `push %rax` followed by `pop %rax` will return you back to where you started.

add, sub. These instructions simply add of subtract a value to a given register. The value to add/sub can be a constant or can be stored in a register.
Example: `add $0x10, %rsp` is equivalent to `rsp`=`rsp`+0x10 (which is 16 in decimal). Note the dollar sign ($) preceding the numerical value is a prefix for constants.
Example 2: `sub $0x10, %rsp` is equivalent to `rsp`=`rsp`−0x10.
Example 3: `sub %rax, %rbx` is equivalent to `rbx`=`rbx`−`rax`.

lea. This instruction takes two operands: an address, and a register. It calculates the effective address of the first operand and puts it into the destination register without actually loading the data from the memory address. Often, the first operand includes an addition or subtraction to the value in a register.
Example: `lea -0x10(%rbp),%rax` will take the value in `rbp`, subtract 0x10 to it and put the result into `rax`. So, if `rbp` is 0x2010, `rax` will be 0x2000.

call. This instruction is used to call a function. It will let the execution of the program continue at that function's location. Prior to starting execution of the called function, the address of the next instruction to run after we return from that function will be pushed onto the stack. This will be the address of the instruction right after the `call` instruction.

leave. This is a shortcut for two instructions:

```
mov %rbp, %rsp
pop %rbp
```

ret. At the end of a function, to return to where the program should continue (somewhere in the calling function), this instruction takes the value on the top of the stack and loads it into `rip`. It would be an equivalent of "`pop %rip`" if accessing `rip` directly was allowed.

nop. No-Operation! This instruction does nothing! Useless, but useful at the same time.

To finish, you may see instructions with a `q` at the end, such as `retq`, `leaveq`, `callq`. These are the same as the base instructions for our purpose.

Put it together

Consider the following lines of assembly of a function. What is the function's return value? Pay attention that the values 10, 20, and 5 are written in decimal, not hexadecimal (they are not prefixed with 0x).

```
mov $10, %rbx
mov $20, %rcx

mov %rbx, %rax
add %rcx, %rax

push %rax
mov $5, %rdi
mov %rdi, %rsi
pop %rax

sub %rdi, %rax

ret
```

⇒ Submit your answer on Blackboard.

Exercise 2: Crash a Program via a Buffer Overflow Vulnerability [80min]

A buffer overflow vulnerability is, unfortunately, a common computer security vulnerability that occurs when a program misuses a buffer. When writing more data to a buffer than it can accommodate, the out-of-bounds data may overwrite adjacent memory areas, resulting in abnormal program behavior or potential attacks. An attacker can exploit a buffer overflow vulnerability to corrupt the intended program's flow and effectively crash the program.

Turning Off Countermeasures

Many countermeasures have been developed to make buffer overflow vulnerabilities more difficult to exploit nowadays. Some of them are implemented in the operating system, while others are integrated by the compiler at compile time. To simplify our attacks, we need to disable them first.

StackGuard. The compiler `gcc` will turn this feature on by default. It can be disabled using the `-fno-stack-protector` parameter:

```
$ gcc -fno-stack-protector -o example example.c
```

Position Independent Executable. This type of executable is designed to have its code loaded at random memory locations, enhancing security and making it more difficult for attackers to predict the process memory layout. We turn off PIE using the `-no-pie` parameter:

```
$ gcc -no-pie -o example example.c
```

CET. Finally, Intel Control-Flow Enforcement Technology (CET) protection helps to defend a program from certain attacks such as buffer overflow vulnerabilities, and is partially supported by the CPU. The compiler support for this feature can be disabled using the `-fcf-protection=none` parameter:

```
$ gcc -fcf-protection=none -o example example.c
```

All-in-one. Overall, to turn off all three features/countermeasures, a program can be compiled once using a combination of all parameters:

```
$ gcc -fno-stack-protector -no-pie -fcf-protection=none -o example example.c
```

Task 1: Copy Lab Files into the VM

1. Download the lab files from Blackboard (from the Windows host, as it would be more convenient).
2. In the Linux VM, open the *seed* folder on the desktop.
3. Drag-and-drop or copy-paste the ZIP file from the Windows host into the *seed* folder in the virtual machine.
4. Right click on the ZIP file, and select Extract Here.

Task 2: Compile the Vulnerable Program

You will work on the following program, given as an attachment to the lab instructions (`bf.c`). This program has been discussed in Lecture 8. A buffer overflow vulnerability lies in the `do_job` function when copying strings with `strcpy`. For the sake of this lab, the size of the buffer will depend on your student ID, see after.

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

// Assumes TWO_DIGITS is defined during compilation
#define TOTAL_SIZE (100 + TWO_DIGITS)

void unreachable_code(char *str)
{
  printf("There is no way this message is displayed!?\n");
}

void do_job(char *input)
{
  char buffer[TOTAL_SIZE];
  strcpy((char*) buffer, input);
  printf("%s\n", (char*) buffer);
}

int main(int argc, char **argv)
{
  do_job(argv[1]);
  printf("Finished!\n");
  return 0;
}
```

To compile the vulnerable program, you will use of the `Makefile` we created for you, through the `make` command. Notice that the vulnerable program is compiled with all three `gcc` parameters discussed above.

Open a Terminal, and go into the program's folder:

```
$ cd lab3
```

Before compiling the program, always run the following line:

```
$ make clean
```

Then, compile your program in the following way:

```
$ make TWO_DIGITS=00
```

IMPORTANT: Set the variable `TWO_DIGITS` to the last two digits (letter not included) of your student ID! If your student ID is `12345678d`, set `TWO_DIGITS=78`. This will make the buffer size dependent on your student ID, and therefore, your lab experiments and results will be unique to you :) While assessing your lab test, we will take your student ID to re-generate the same program and verify your answers. Also note that the answers could vary significantly depending on your environment. In this lab, you are in a controlled environment. For the sake of the lab test, DO NOT WORK ON THIS LAB FROM YOUR OWN LAPTOP.

Go ahead and test your program! Try running it with a simple input.

```
$ ./bf TEST1234
```

What do you see?

Task 3: Disassemble the Program

To understand what the CPU is actually executing, and find out what input value will create a problem, you need to look at the code generated by the compiler. This step involves disassembling the generated executable file using the `objdump` tool. Run:

```
$ objdump -d bf
```

This command will generate a lot of output. These are all the CPU instructions in your program. We will skip most of them to focus on the `do_job` function. Scroll up until you see the line:

```
0000000000401151 <do_job>:
```

The 15-or-so instructions following this line represent the function `do_job`. The format of the output is composed as follows. Note the code below is not the same as yours:

```
401136:       55                          push   %rbp
401137:       48 89 e5                    mov    %rsp,%rbp
40113a:       48 83 ec 10                 sub    $0x10,%rsp
```

The first column lists the addresses of the instructions in the TEXT section of the process memory. The second column lists the op-codes of the corresponding assembly instructions, which is the actual data being executed. On the right, at last, you can see the assembly instructions, as you are now familiar.

Task 4: Study the Program

Complete the tasks below.

1. Read the assembly instructions of the function `do_job` to find out how much stack space above `rbp` is reserved for the buffer. You will need to identify which memory location corresponds to the buffer.

   Several important items to help you read the code:

   (a) As a reminder, the buffer is the destination of the `strcpy` call, and this function works as `strcpy(destination, source)`. Refer back to the C program.

   (b) When a function is called, in the x64 calling convention, simple arguments are passed through certain registers. On Linux, the registers used for function arguments differ from those seen in class on Windows. Here, the first function argument will be stored in `rdi` and the second one in `rsi` before the call `strcpy`.

   (c) If you see an addition with a very large number such as:

   ```
   add    $0xffffffffffffff80,%rsp
   ```

   This addition is equivalent to the following subtraction:

   ```
   sub    $0x80,%rsp
   ```

   The compiler somehow overflows the register with an addition instead of making a small subtraction. Compilers have their own reason (efficiency, space, etc.) for doing it.

   (d) Tip: You can convert a hexadecimal number to a decimal number on the Terminal. For instance, here is what 0x10 represents:

   ```
   $ echo $((0x10))            # outputs 16
   ```

2. You should now think about a plan to crash the program with the *smallest working input.* To help you with this task, here are a few guidelines:

   (a) Recall that below `rbp`, we expect to have on the stack: 1) the saved `rbp` register, corresponding to the caller function, and 2) the return address where execution should continue at the end of this function. If any of these values get corrupted, they will result in the crach of the program.

   (b) The size of the input you need to give is not easy to write manually (more than 100 characters). Instead, you should make use of a this simple Perl command:

   ```
   $ echo $(perl -e 'print "A"x10')
   ```

   This will output AAAAAAAAAA. You can run the program on this input too:

   ```
   $ ./bf $(perl -e 'print "A"x10')
   ```

   (c) Strings in C are always terminated with a NULL byte (`0x00`). If you provide a string of 8 characters such as `"TEST1234"`, it will effectively be stored as `"TEST1234\x00"`. This additional null byte is unavoidable. Make sure you account for it in your calculations.

   (d) The program is considered crashed if it shows `Segmentation fault` or `Bus error`. If it doesn't crash, verify the size of your input, and go back to study the assembly code to make sure you know how much data you need.

   (e) Verify by removing one byte from your input that the program does NOT crash. If it crashes, you did not provide the smallest input possible.

3. ⇒ Once you succeed, answer questions on Blackboard!

Exercise 3: Hijack the Execution Flow [30min]

Congratulations on crashing the program. Did you notice the `unreachable_code` function? Is it called anywhere? No? Let's "call" it ourselves!

For this exercise, you need to exploit the buffer overflow vulnerability to overwrite the return address on the stack. But with which value?

The address of the `unreachable_code` is given to you in the output of `objdump`. This will be the same address during the execution of the process since we disabled PIE. However, there are two challenges you need to overcome.

Little endianness. First, since an address in 64-bit systems is 64-bit long, it takes 8 bytes to store an address in memory. The sequence of bytes to represent the address follows the little endian ordering. That is, the least significant byte of the address is written first to memory, up until the most significant.

Say we want to store this address in memory: 0x00123456abcdefff. This is how it is actually stored:

Address	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Value	0xFF	0xEF	0xCD	0xAB	0x56	0x34	0x12	0x00

Take the full 64-bit address of `unreachable_code`. You see it contains many zeros. Try to write each byte of the address in little endian ordering like above.

Null-terminated strings. Second, another consequence of null-terminated strings is that a string cannot include any null bytes. At the first null byte, the `strcpy` function will stop copying data from the input. You must provide a string without null bytes. But how are you going to write the full address of the function if you cannot write null bytes? Let's rethink whether you really need to write the full address. During normal execution, the return address that will be used at the end of the `do_job` function is the address of the next instruction after the call to `do_job` in the `main` function. What is this address in your case? How much difference there is with the address you are targeting?

Task: 1 Writing your payload

You should realize that it is sufficient to overwrite only four bytes of the return address on the stack. Devise an input to the program that overwrites the return address to change it to the address of the `unreachable_code` function. In doing so, you are still corrupting the saved `rbp`, so the program will still crash, but not before it executes the function you want.

You will be able to use Perl again for this task, note that you can use it in the following way to concatenate repeated letters to bytes written in hexadecimal:

```
echo $(perl -e 'print "A"x10,"\x01\x02\x03"')
```

This will output ten As followed by the three bytes 0x01, 0x02 and 0x03.

If your program input is correct, the `unreachable_code` function will run and the message "*There is no way this message is displayed!?*" will appear on the screen. Return to Blackboard to answer related questions.

Task 2: Using GDB to debug your payload

If your program input does not work, you can quickly see a preview of the return address on the stack by using the GDB debugger and a few commands.

1. Run:

   ```
   $ gdb bf
   ```

2. Disassemble the `do_job` function to look for the last instruction:

   ```
   gdb-peda$ disas do_job
   [...]
      0x000000000040117e <+45>: ret
   ```

3. Look for the offset between < and > on the line of the `ret` instruction. In the example above, this is 45.

4. Place a breakpoint at this offset:

   ```
   gdb-peda$ b *do_job+45
   ```

5. Run your program with the same input you were trying before, but within the debugger:

   ```
   gdb-peda$ run $(perl -e 'print ...')
   ```

   This should stop at the breakpoint.

6. Check the address on top of the stack:

   ```
   gdb-peda$ x/gx $rsp
   ```

   Does the right side look like the return address you want? If not, you should be able to understand how to fix your payload.