

SE16, Fall 2017
Lab Assignment L3: The Buffer Bomb
Assigned: Mon, Oct. 30
Due: Fri, Nov. 10, 11:59PM

Zinan Li (409146908@qq.com) is the lead person for this assignment.

1 Introduction

This assignment involves generating a total of three attacks on one program having security vulnerability. Outcomes you will gain from this lab include:

- You will learn different ways that attackers can exploit security vulnerabilities when programs do not safeguard themselves well enough against buffer overflows.
- Through this, you will get a better understanding of how to write programs that are more secure, as well as some of the features provided by compilers and operating systems to make programs less vulnerable.
- You will gain a deeper understanding of the stack and parameter-passing mechanisms of x86-64 machine code.
- You will gain a deeper understanding of how x86-64 instructions are encoded.
- You will gain more experience with debugging tools such as GDB and OBJDUMP.

Note: In this lab, you will gain firsthand experience with methods used to exploit security weaknesses in operating systems and network servers. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of these security weaknesses so that you can avoid them when you write system code. We do not condone the use of any other form of attack to gain unauthorized access to any system resources.

You will want to study Sections 3.10.3 and 3.10.4 of the CS:APP3e book as reference material for this lab.

2 Logistics

As usual, this is an individual project. You will generate attacks for target program that is custom generated for you.

2.1 Getting Files

You can obtain your files by pointing your Web browser at:

```
http://ipads.se.sjtu.edu.cn:15513/
```

The server will build your files and return them to your browser in a tar file called `target k .tar`, where k is the unique number of your target programs.

Note: It takes a few seconds to build and download your target, so please be patient.

Save the `target k .tar` file in a (protected) Linux directory in which you plan to do your work. Then give the command: `tar -xvf target k .tar`. This will extract a directory `target k` containing the files described below.

You should only download one set of files. If for some reason you download multiple targets, choose one target to work on and delete the rest.

Warning: If you expand your `target k .tar` on a PC, by using a utility such as Winzip, or letting your browser do the extraction, you'll risk resetting permission bits on the executable files.

The files in `target k` include:

`README.txt`: A file describing the contents of the directory

`ctarget`: An executable program vulnerable to *code-injection* attacks

`cookie.txt`: An 8-digit hex code that you will use as a unique identifier in your attacks.

`hex2raw`: A utility to generate attack strings.

In the following instructions, we will assume that you have copied the files to a protected local directory, and that you are executing the programs in that local directory.

2.2 Important Points

Here is a summary of some important rules regarding valid solutions for this lab. These points will not make much sense when you read this document for the first time. They are presented here as a central reference of rules once you get started.

- You must do the assignment on a machine that is similar to the one that generated your targets.

- Your solutions may not use attacks to circumvent the validation code in the programs. Specifically, any address you incorporate into an attack string for use by a `ret` instruction should be to one of the following destinations:
 - The addresses for functions `touch1`, `touch2`, or `touch3`.
 - The address of your injected code

3 Target Programs

CTARGET reads strings from standard input. They do so with the function `getbuf` defined below:

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

The function `Gets` is similar to the standard library function `gets`—it reads a string from standard input (terminated by ‘\n’ or end-of-file) and stores it (along with a null terminator) at the specified destination. In this code, you can see that the destination is an array `buf`, declared as having `BUFFER_SIZE` bytes. At the time your targets were generated, `BUFFER_SIZE` was a compile-time constant specific to your version of the programs.

Functions `Gets()` and `gets()` have no way to determine whether their destination buffers are large enough to store the string they read. They simply copy sequences of bytes, possibly overrunning the bounds of the storage allocated at the destinations.

If the string typed by the user and read by `getbuf` is sufficiently short, it is clear that `getbuf` will return 1, as shown by the following execution examples:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit. Getbuf returned 0x1
Normal return
```

Typically an error occurs if you type a long string:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

(Note that the value of the cookie shown will differ from yours.) As the error message indicates, overrunning the buffer typically causes the program state to be corrupted, leading to a memory access error. Your task is to be more clever with the strings you feed CTARGET so that it does more interesting things. These are called *exploit* strings.

CTARGET takes several different command line arguments:

- h: Print list of possible command line arguments
- q: Don't send results to the grading server
- i FILE: Supply input from a file, rather than from standard input

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program HEX2RAW will enable you to generate these *raw* strings. See Appendix A for more information on how to use HEX2RAW.

Important points:

- Your exploit string must not contain byte value 0x0a at any intermediate position, since this is the ASCII code for newline ('\n'). When Gets encounters this byte, it will assume you intended to terminate the string.
- HEX2RAW expects two-digit hex values separated by one or more white spaces. So if you want to create a byte with a hex value of 0, you need to write it as 00. To create the word 0xdeadbeef you should pass "ef be ad de" to HEX2RAW (note the reversal required for little-endian byte ordering).

When you have correctly solved one of the levels, your target program will automatically send a notification to the grading server. For example:

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

The server will test your exploit string to make sure it really works, and it will update the Attacklab scoreboard page indicating that your userid (listed by your target number for anonymity) has completed this phase.

You can view the scoreboard by pointing your Web browser at

<http://ipads.se.sjtu.edu.cn:15513/scoreboard>

Unlike the Bomb Lab, there is no penalty for making mistakes in this lab. Feel free to fire away at CTARGET with any strings you like.

Figure 1 summarizes the three phases of the lab.

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25

CI: Code injection

Figure 1: Summary of attack lab phases

4 Code Injection Attacks

For the three phases, your exploit strings will attack CTARGET. This program is set up in a way that the stack positions will be consistent from one run to the next and so that data on the stack can be treated as executable code. These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

4.1 Level 1

For Phase 1, you will not inject new code. Instead, your exploit string will redirect the program to execute an existing procedure.

Function `getbuf` is called within CTARGET by a function `test` having the following C code:

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within function `test` (at line 5 of this function). We want to change this behavior. Within the file `ctarget`, there is code for a function `touch1` having the following C representation:

```
1 void touch1()
2 {
3     vlevel = 1;          /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

Your task is to get CTARGET to execute the code for `touch1` when `getbuf` executes its return statement, rather than returning to `test`. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since `touch1` causes the program to exit directly.

Some Advice:

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of CTARGET. Use `objdump -d` to get this disassembled version.
- The idea is to position a byte representation of the starting address for `touch1` so that the `ret` instruction at the end of the code for `getbuf` will transfer control to `touch1`.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of `getbuf` to make sure it is doing the right thing.
- The placement of `buf` within the stack frame for `getbuf` depends on the value of compile-time constant `BUFFER_SIZE`, as well the allocation strategy used by GCC. You will need to examine the disassembled code to determine its position.

4.2 Level 2

Phase 2 involves injecting a small amount of code as part of your exploit string.

Within the file `ctarget` there is code for a function `touch2` having the following C representation:

```

1 void touch2(unsigned val)
2 {
3     vlevel = 2;          /* Part of validation protocol */
4     if (val == cookie) {
5         printf("Touch2!: You called touch2(0x%.8x)\n", val);
6         validate(2);
7     } else {
8         printf("Misfire: You called touch2(0x%.8x)\n", val);
9         fail(2);
10    }
11    exit(0);
12 }
```

Your task is to get CTARGET to execute the code for `touch2` rather than returning to `test`. In this case, however, you must make it appear to `touch2` as if you have passed your cookie as its argument.

Some Advice:

- You will want to position a byte representation of the address of your injected code in such a way that `ret` instruction at the end of the code for `getbuf` will transfer control to it.
- Recall that the first argument to a function is passed in register `%rdi`.
- Your injected code should set the register to your cookie, and then use a `ret` instruction to transfer control to the first instruction in `touch2`.
- Do not attempt to use `jmp` or `call` instructions in your exploit code. The encodings of destination addresses for these instructions are difficult to formulate. Use `ret` instructions for all transfers of control, even when you are not returning from a call.

- See the discussion in Appendix B on how to use tools to generate the byte-level representations of instruction sequences.

4.3 Level 3

Phase 3 also involves a code injection attack, but passing a string as argument.

Within the file `ctarget` there is code for functions `hexmatch` and `touch3` having the following C representations:

```

1 /* Compare string to hex representation of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13     vlevel = 3;          /* Part of validation protocol */
14     if (hexmatch(cookie, sval)) {
15         printf("Touch3!: You called touch3(\"%s\")\n", sval);
16         validate(3);
17     } else {
18         printf("Misfire: You called touch3(\"%s\")\n", sval);
19         fail(3);
20     }
21     exit(0);
22 }
```

Your task is to get `CTARGET` to execute the code for `touch3` rather than returning to `test`. You must make it appear to `touch3` as if you have passed a string representation of your cookie as its argument.

Some Advice:

- You will need to include a string representation of your cookie in your exploit string. The string should consist of the eight hexadecimal digits (ordered from most to least significant) without a leading “0x.”
- Recall that a string is represented in C as a sequence of bytes followed by a byte with value 0. Type “`man ascii`” on any Linux machine to see the byte representations of the characters you need.
- Your injected code should set register `%rdi` to the address of this string.
- When functions `hexmatch` and `strncmp` are called, they push data onto the stack, overwriting portions of memory that held the buffer used by `getbuf`. As a result, you will need to be careful where you place the string representation of your cookie.

Good luck and have fun!

A Using HEX2RAW

HEX2RAW takes as input a *hex-formatted* string. In this format, each byte value is represented by two hex digits. For example, the string “012345” could be entered in hex format as “30 31 32 33 34 35 00.” (Recall that the ASCII code for decimal digit x is $0 \times 3x$, and that the end of a string is indicated by a null byte.)

The hex characters you pass to HEX2RAW should be separated by whitespace (blanks or newlines). We recommend separating different parts of your exploit string with newlines while you’re working on it. HEX2RAW supports C-style block comments, so you can mark off sections of your exploit string. For example:

```
48 c7 c1 f0 11 40 00 /* mov      $0x40011f0,%rcx */
```

Be sure to leave space around both the starting and ending comment strings (“/*”, “*/”), so that the comments will be properly ignored.

If you generate a hex-formatted exploit string in the file `exploit.txt`, you can apply the raw string to CTARGET in several different ways:

1. You can set up a series of pipes to pass the string through HEX2RAW.

```
unix> cat exploit.txt | ./hex2raw | ./ctarget
```

2. You can store the raw string in a file and use I/O redirection:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget < exploit-raw.txt
```

This approach can also be used when running from within GDB:

```
unix> gdb ctarget
(gdb) run < exploit-raw.txt
```

3. You can store the raw string in a file and provide the file name as a command-line argument:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -i exploit-raw.txt
```

This approach also can be used when running from within GDB.

B Generating Byte Codes

Using GCC as an assembler and OBJDUMP as a disassembler makes it convenient to generate the byte codes for instruction sequences. For example, suppose you write a file `example.s` containing the following assembly code:

```
# Example of hand-generated assembly code
    pushq    $0xabcdef          # Push value onto stack
    addq     $17,%rax           # Add 17 to %rax
    movl     %eax,%edx          # Copy lower 32 bits to %edx
```

The code can contain a mixture of instructions and data. Anything to the right of a '#' character is a comment.

You can now assemble and disassemble this file:

```
unix> gcc -c example.s
unix> objdump -d example.o > example.d
```

The generated file `example.d` contains the following:

```
example.o:      file format elf64-x86-64
```

Disassembly of section `.text`:

```
0000000000000000 <.text>:
    0: 68 ef cd ab 00      pushq  $0xabcdef
    5: 48 83 c0 11         add    $0x11,%rax
    9: 89 c2              mov    %eax,%edx
```

The lines at the bottom show the machine code generated from the assembly language instructions. Each line has a hexadecimal number on the left indicating the instruction's starting address (starting with 0), while the hex digits after the ':' character indicate the byte codes for the instruction. Thus, we can see that the instruction `push $0xABCDEF` has hex-formatted byte code `68 ef cd ab 00`.

From this file, you can get the byte sequence for the code:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

This string can then be passed through `HEX2RAW` to generate an input string for the target programs.. Alternatively, you can edit `example.d` to omit extraneous values and to contain C-style comments for readability, yielding:

```
68 ef cd ab 00    /* pushq  $0xabcdef */
48 83 c0 11        /* add    $0x11,%rax */
89 c2              /* mov    %eax,%edx */
```

This is also a valid input you can pass through HEX2RAW before sending to one of the target programs.