

SOFT130056.01, Fall 2020

Lab Assignment L3: The Buffer Bomb

Assigned: Nov. 17, 2020, Due: Nov. 30, 2020 23:59

Shao Kaiqiang(kqshao18@fudan.edu.cn) is the lead people for this assignment.

Introduction

This assignment will help you develop a detailed understanding of IA-32 calling conventions and stack organization. It involves applying a series of *buffer overflow attacks* on an executable file `bufbomb` in the lab directory.

Note:

In this lab, you will gain firsthand experience with one of the methods commonly 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 this form of security weakness so that you can avoid it when you write system code. We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources. There are criminal statutes governing such activities.

Logistics

As usual, this is an individual project.

We generated the lab using gcc's `-m32` flag, so all code produced by the compiler follows IA-32 rules, even if the host is an x86-64 system. This should be enough to convince you that the compiler can use any calling convention it wants, so long as it's consistent.

Hand Out Instructions

You can download the lab at [elearning](#).

Start by copying `buflab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the command `"tar xvf buflab-handout.tar"`. This will create a directory called `buflab-handout` containing the following three executable files:

bufbomb: The buffer bomb program you will attack.

makecookie: Generates a "cookie" based on your `userid`.

hex2raw: A utility to help convert between string formats.

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

(You should install 32 bit runtime environment if you run the binary failed!)

Userids and Cookies

Phases of this lab will require a slightly different solution from each student. The correct solution will be based on your userid.

A *cookie* is a string of eight hexadecimal digits that is (with high probability) unique to your userid. You can generate your cookie with the `makecookie` program giving your userid as the argument. For example:

```
unix> ./makecookie bovik
0x1005b2b7
```

You should use your school number as your username.

For example:

```
unix> ./makecookie 18212010069
0x7a3ea7e9
```

The BUFBOMB Program

The BUFBOMB program reads a string from standard input. It does so with the function `getbuf` defined

below:

```
1 /* Buffer size for getbuf */
2 #define NORMAL_BUFFER_SIZE 32
3
4 int getbuf()
5 {
6     char buf[NORMAL_BUFFER_SIZE];
7     Gets(buf);
8     return 1;
9 }
```

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` having sufficient space for 32 characters.

`Gets` (and `gets`) grabs a string off the input stream and stores it into its destination address (in this case `buf`). However, `Gets()` has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

The \x00 shouldn't in your input string, system may truncate your input string!

If the string typed by the user to `getbuf` is no more than 31 characters long, it is clear that `getbuf` will return 1, as shown by the following execution example:

```
unix> ./bufbomb -u bovik
```

Type string: *I love 15-213.*

Dud: `getbuf` returned 0x1

Typically an error occurs if we type a longer string:

```
unix> ./bufbomb -u bovik
```

Type string: *It is easier to love this class when you are a TA.* Ouch!: You caused a segmentation fault!

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 `BUFBOMB` so that it does more interesting things. These are called *exploit* strings.

`BUFBOMB` takes several different command line arguments:

`-u userid`: Operate the bomb for the indicated `userid`. You should always provide this argument for several reasons:

- It is required to submit your successful attacks to the grading server.
- `BUFBOMB` determines the cookie you will be using based on your `userid`, as does the program
- We have built features into `BUFBOMB` so that some of the key stack addresses you will need to use depend on your `userid`'s cookie.

`-h`: Print list of possible command line arguments.

`-n`: Operate in "Nitro" mode, as is used in Level 4 below.

At this point, you should think about the x86 stack structure a bit and figure out what entries of the stack you will be targeting. You may also want to think about *exactly* why the last example created a segmentation fault, although this is less clear.

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program `HEX2RAW` can help you generate these *raw* strings. It 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." (Recall that the ASCII code for decimal digit `x` is `0x3x`.)

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

```
bf 66 7b 32 78 /* mov $0x78327b66,%edi */
```

Be sure to leave space around both the starting and ending comment strings (`/*`, `*/`) so they will be properly ignored. If you generate a hex-formatted exploit string in the file `exploit.txt`, you can apply the raw string to `BUFBOMB` in several different ways:

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

```
unix> cat exploit.txt | ./hex2raw | ./bufbomb -u bovik
```

2. You can store the raw string in a file and use I/O redirection to supply it to `BUFBOMB`:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt unix> ./bufbomb -u bovik < exploit-raw.txt
```

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

```
unix> gdb bufbomb
```

```
(gdb) run -u bovik < exploit-raw.txt
```

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 a whitespace. So if you want to create a byte with a hex value of `0`, you need to specify `00`. To create the word `0xDEADBEEF` you should pass `DE AD BE EF` to `HEX2RAW`.

When you have correctly solved one of the levels, say level 0:

```
./hex2raw < smoke-bovik.txt | ./bufbomb -u bovik
Userid: bovik
Cookie: 0x1005b2b7
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

Level 0: Candle (10 pts)

The function `getbuf` is called within `BUFBOMB` by a function test having the following C code:

```
1 void test()
2 {
3     int val;
4     /* Put canary on stack to detect possible corruption */
5     volatile int local = uniqueval();
```

```

6
7  val = getbuf();
8
9  /* Check for corrupted stack */
10 if (local != uniqueval()) {
11     printf("Sabotaged!: the stack has been corrupted\n");
12 }
13 else if (val == cookie) {
14     printf("Boom!: getbuf returned 0x%x\n", val);
15     validate(3);
16 } else {
17     printf("Dud: getbuf returned 0x%x\n", val);
18 }
19 }

```

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within function `test` (at line 7 of this function). We want to change this behavior. Within the file `bufbomb`, there is a function `smoke` having the following C code:

```

void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}

```

Your task is to get `BUFBOMB` to execute the code for `smoke` 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 `smoke` 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 `BUFBOMB`. Use `objdump` to get this disassembled version. (if you know how to use `ida`, you can use it)
- **Be careful about byte ordering.**

Level 1: Sparkler (10 pts)

Within the file `bufbomb` there is also a function `fizz` having the following C code:

```

void fizz(int val)
{

```

```

    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
    } else
        exit(0);
}

```

Similar to Level 0, your task is to get BUFBOMB to execute the code for fizz rather than returning to test. In this case, however, you must make it appear to fizz as if you have passed your cookie as its argument. How can you do this?

Some Advice:

- Note that the program won't really call fizz—it will simply execute its code. This has important implications for where on the stack you want to place your cookie.

Level 2: Firecracker (15 pts)

A much more sophisticated form of buffer attack involves supplying a string that encodes actual machine instructions. The exploit string then overwrites the return pointer with the starting address of these instructions on the stack. When the calling function (in this case getbuf) executes its ret instruction, the program will start executing the instructions on the stack rather than returning. With this form of attack, you can get the program to do almost anything. The code you place on the stack is called the *exploit* code. This style of attack is tricky, though, because you must get machine code onto the stack and set the return pointer to the start of this code.

Within the file bufbomb there is a function bang having the following C code:

```

int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}

```

Similar to Levels 0 and 1, your task is to get BUFBOMB to execute the code for bang rather than returning to test. Before this, however, you must set global variable global_value to your userid's cookie. Your exploit code should set global_value, push the address of bang on the stack, and then execute a ret instruction to cause a jump to the code for bang.

Some Advice:

- You can use GDB to get the information you need to construct your exploit string. Set a break- point within getbuf and run to this breakpoint. Determine parameters such as the address of global_value and the location of the buffer.
- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with gcc -m32 -c and disassemble it with objdump -d. You should be able to get the exact byte sequence that you will type at the prompt. (A brief example of how to do this is included at the end of this writeup.)
- Keep in mind that your exploit string depends on your machine, your compiler, and even your user's cookie. Do all of your work on one of the machines assigned by your instructor, and make sure you include the proper userid on the command line to BUFBOMB.
- Watch your use of address modes when writing assembly code. Note that movl \$0x4, %eax moves the *value* 0x00000004 into register %eax; whereas movl 0x4, %eax moves the value *at* memory location 0x00000004 into %eax. Since that memory location is usually undefined, the second instruction will cause a segfault!
- Do not attempt to use either a jmp or a call instruction to jump to the code for bang. These instructions use PC-relative addressing, which is very tricky to set up correctly. Instead, push an address on the stack and use the ret instruction.

Level 3: Dynamite (20 pts)

Our preceding attacks have all caused the program to jump to the code for some other function, which then causes the program to exit. As a result, it was acceptable to use exploit strings that corrupt the stack, overwriting saved values.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that changes the program's register/memory state, but makes the program return to the original calling function (test in this case). The calling function is oblivious to the attack. This style of attack is tricky, though, since you must:

- 1) get machine code onto the stack,
- 2) set the return pointer to the start of this code, and
- 3) undo any corruptions made to the stack state.

Your job for this level is to supply an exploit string that will cause getbuf to return your cookie back to test, rather than the value 1. You can see in the code for test that this will cause the program to go "Boom!." Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a ret instruction to really return to test.

Some Advice:

You can use GDB to get the information you need to construct your exploit string. Set a breakpoint within `getbuf` and run to this breakpoint. Determine parameters such as the saved return address.

Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with `GCC` and disassemble it with `OBJDUMP`. You should be able to get the exact byte sequence that you will type at the prompt. (A brief example of how to do this is included at the end of this writeup.)

Once you complete this level, pause to reflect on what you have accomplished. You caused a program to execute machine code of your own design. You have done so in a sufficiently stealthy way that the program did not realize that anything was amiss.

Level 4: Nitroglycerin (15 pts)

Please note: You'll need to use the `"-n,"` command-line flag in order to run this stage.

From one run to another, especially by different users, the exact stack positions used by a given procedure will vary. One reason for this variation is that the values of all environment variables are placed near the base of the stack when a program starts executing. Environment variables are stored as strings, requiring different amounts of storage depending on their values. Thus, the stack space allocated for a given user depends on the settings of his or her environment variables. Stack positions also differ when running a program under GDB, since GDB uses stack space for some of its own state.

In the code that calls `getbuf`, we have incorporated features that stabilize the stack, so that the position of `getbuf`'s stack frame will be consistent between runs. This made it possible for you to write an exploit string knowing the exact starting address of `buf`. If you tried to use such an exploit on a normal program, you would find that it works some times, but it causes segmentation faults at other times. Hence the name "dynamite"—an explosive developed by Alfred Nobel that contains stabilizing elements to make it less prone to unexpected explosions.

For this level, we have gone the opposite direction, making the stack positions even less stable than they normally are. Hence the name "nitroglycerin"—an explosive that is notoriously unstable.

When you run `BUFBOMB` with the command line flag `"-n,"` it will run in "Nitro" mode. Rather than calling the function `getbuf`, the program calls a slightly different function `getbufn`:

```
/* Buffer size for getbufn */  
#define KABOOM_BUFFER_SIZE 512
```


This function is similar to `getbuf`, except that it has a buffer of 512 characters. You will need this additional space to create a reliable exploit. The code that calls `getbufn` first allocates a random amount of storage on the stack, such that if you sample the value of `%ebp` during two successive executions of `getbufn`, you would find they differ by as much as ± 240 .

In addition, when run in Nitro mode, `BUFBOMB` requires you to supply your string 5 times, and it will execute `getbufn` 5 times, each with a different stack offset. Your exploit string must make it return your cookie each of these times.

Your task is identical to the task for the Dynamite level. Once again, your job for this level is to supply an exploit string that will cause `getbufn` to return your cookie back to test, rather than the value 1. You can see in the code for test that this will cause the program to go "KABOOM!." Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `testn`.

Some Advice:

- You can use the program `HEX2RAW` to send multiple copies of your exploit string. If you have a single copy in the file `exploit.txt`, then you can use the following command:

```
unix> cat exploit.txt | ./hex2raw -n | ./bufbomb -n -u bovik
```

- The trick is to make use of the `nop` instruction. It is encoded with a single byte (code `0x90`). It may be useful to read about "nop sleds" on page 262 of the CS:APP2e textbook.
- Important Note

You also need to hand in a report `buflab-your school num.pdf` (e.g. `buflab-192xxxxxxxx.pdf`) which explains your solutions.

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 we write a file `example.S` containing the following assembly code:

```
# Example of hand-generated assembly code
    push $0xabcdef          # Push value onto stack
    add  $17,%eax           # Add 17 to %eax
    .align 4                # Following will be aligned on multiple of 4
    .long 0xfedcba98        # A 4-byte constant
```

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

We can now assemble and disassemble this file:

```
unix> gcc -m32 -c example.S
unix> objdump -d example.o > example.d
```

The generated file example.d contains the following lines

```
0: 68 efc dab 00          push    $0xabcdef
5: 83 c0 11              add     $0x11,%eax
8: 98                   cwtl
9: ba                   .byte 0xba
a: dc fe                   fdivr   %st,%st(6)
```

Each line shows a single instruction. The number on the left indicates the 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.

Starting at address 8, the disassembler gets confused. It tries to interpret the bytes in the file example.o as instructions, but these bytes actually correspond to data. Note, however, that if we read off the 4 bytes starting at address 8 we get: 98 ba dc fe. This is a byte-reversed version of the data word 0xFEDCBA98. This byte reversal represents the proper way to supply the bytes as a string, since a little endian machine lists the least significant byte first.

Finally, we can read off the byte sequence for our code as:

```
68 ef cd ab 00 83 c0 11 98 ba dc fe
```

This string can then be passed through HEX2RAW to generate a proper input string we can give to BUFBOMB. Alternatively, we can edit example.d to look like this:

```
68 ef cd ab 00      /* push    $0xabcdef */
83 c0 11            /* add     $0x11,%eax */
98
ba dc fe
```

which is also a valid input we can pass through HEX2RAW before sending to BUFBOMB.