中国科学技术大学 软件学院	***University of Science and Technology of China*** ***School of Software Engineering of USTC***

Lab3: The Buffer Bomb

G430113385: Computer Systems A Programmer’s Perspectives

2015-12-20

Junmin Wu (jmwu@ustc.edu.cn) is the lead person for this assignment.

TA: Xiaodong Zhu(xdzhu001@mail.ustc.edu.cn)

1 Introduction

This assignment helps you develop a detailed understanding of the calling stack organization on an IA32 processor. 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 these or any other form of attack to gain unauthorized access to any system resources. There are criminal statutes governing such activities.

2 Logistics

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be an automated logging of your successful attacks. Any clarifications and revisions to the assignment will be posted on the web server

3 Hand Out Instructions

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 cause a number of files to be unpacked in the directory:

MAKECOOKIE: Generates a "cookie" based on your team name.

BUFBOMB: The code you will attack.

SENDSTRING: A utility to help convert between string formats.

All of these programs are compiled to run on Linux machines.

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.

4 Team Name and Cookie

You should create a team name for the one or two people in your group of the following form:

- "ID1+ID2" where ID1 is the Student Number of the first team member and ID2 is the Student Number of the second team member.

You should choose a consistent ordering of the IDs in the second form of team name. Teams "SA08225155+ SG08225120" and "SG08225120 + SA08225155" are considered distinct. You must follow this scheme for generating your team name. Our grading program will only give credit to those people whose Student Number can be extracted from the team names.

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

```
unix> ./makecookie SA08225155+ SG08225120
0x451a3cdd
```

In four of your five buffer attacks, your objective will be to make your cookie show up in places where it ordinarily would not.

5 The *BUFBOMB* Program

The BUFBOMB program reads a string from standard input with a function getbuf having the following C code:

```
int getbuf()
{
  char buf[12];
  Gets(buf);
  return 1;
}
```

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, the destination is an array buf having sufficient space for 12 characters. Neither Gets nor gets has any way to determine whether there is enough space at the destination to store the entire string. Instead, they simply copy the entire string, possibly overrunning the bounds of the storage allocated at the destination.

If the string typed by the user to getbuf is no more than 11 characters long, it is clear that getbuf will return 1, as shown by the following execution example:

```
unix> ./bufbomb -t SA08225155+SG08225120
Type string: howdy doody
Dud: getbuf returned 0x1
Better luck next time
```

Typically an error occurs if we type a longer string:

```
unix> ./bufbomb -t SA08225155+SG08225120
Type string: This string is too long
Ouch!: You caused a segmentation fault!
Better luck next time
```

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:

-t TEAM: Operate the bomb for the indicated team. You should always provide this argument for several reasons:

- BUFBOMB determines the cookie you will be using based on your team name, just as does the program MAKECOOKIE.
- We have built features into BUFBOMB so that some of the key stack addresses you will need to use depend on your team's cookie.

-h: Print list of possible command line arguments

-n: Operate in “Nitro” mode, as is used in Level 4 below.

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program SENDSTRING can help you generate these raw strings. It takes as input a hexformatted 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 is 0x3x.) Non-hex digit characters are ignored, including the blanks in the example shown.

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 SENDSTRING.

```
unix> cat exploit.txt | ./sendstring | ./bufbomb -t SA08225155+SG08225120
```

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

```
unix> ./sendstring < exploit.txt > exploit-raw.txt
unix> ./bufbomb -t SA08225155+SG08225120 < exploit-raw.txt
```

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

```
unix> gdb bufbomb
(gdb) run -t SA08225155+SG08225120 < exploit-raw.txt
```

One important point: 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. SENDSTRING will warn you if it encounters this byte value.

6 Level 0: Candle (10 pts)

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

```
void test()
{
  int val;
  volatile int local = 0xdeadbeef;
  entry_check(3); /* Make sure entered this function properly */
  val = getbuf();
  /* Check for corrupted stack */
  if (local != 0xdeadbeef) {
  printf("Sabotaged!: the stack has been corrupted\n");
}
  else if (val == cookie) {
  printf("Boom!: getbuf returned 0x%x\n", val);
  validate(3);
}
  else {
  printf("Dud: getbuf returned 0x%x\n", val);
}
}
```

When getbuf executes its return statement (line 5 of getbuf), the program ordinarily resumes execution within function test (at line 8 of this function). Within the file bufbomb, there is a function smoke having the following C code:

```
void smoke()
{
    entry_check(0); /* Make sure entered this function properly */
    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. You can do this by supplying an exploit string that overwrites the stored return pointer in the stack frame for getbuf with the address of the first instruction in smoke. Note that your exploit string may also corrupt other parts of the stack state, 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 diassembled version of BUFBOMB.
- 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 which version of GCC was used to compile bufbomb. You will need to pad the beginning of your exploit string with the proper number of bytes to overwrite the return pointer. The values of these bytes can be arbitrary.

7 Level 1: Sparkler (20 pts)

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

```
void fizz(int val)
{
    entry_check(1); /* Make sure entered this function properly */
    if (val == cookie) {
    printf("Fizz!: You called fizz(0x%x)\n", val);
    validate(1);
} else
    printf("Misfire: You called fizz(0x%x)\n", val);
    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. You can do this by encoding your cookie in the appropriate place within your exploit string.

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.

8 Level 2: Firecracker (30 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. 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)
{
    entry_check(2); /* Make sure entered this function properly */
    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 team’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 breakpoint 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 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 document.)
- Keep in mind that your exploit string depends on your machine, your compiler, and even your team’s cookie. Do all of your work on a Fish machine, and make sure you include the proper team name on the command line to BUFBOMB.
- Our solution requires 16 bytes of exploit code. Fortunately, there is sufficient space on the stack, because we can overwrite the stored value of %ebp. This stack corruption will not cause any problems, since bang causes the program to exit

directly.

- 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.

9 Level 3: Dynamite (40 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 the saved value of register %ebp and the return pointer.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that patches up the stack and 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 the 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:

- In order to overwrite the return pointer, you must also overwrite the saved value of %ebp. However, it is important that this value is correctly restored before you return to test. You can do this by either 1) making sure that your exploit string contains the correct value of the saved %ebp in the correct position, so that it never gets corrupted, or 2) restore the correct value as part of your exploit code. You’ll see that the code for test has some explicit tests to check for a corrupted stack.
- 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 and the saved value of %ebp.
- Again, let tools such as GCC and OBJDUMP do all of the work of generating a byte encoding of the instructions.
- Keep in mind that your exploit string depends on your machine, your compiler, and even your team’s cookie. Do all of your work on a Fish machine, and make sure you include the proper team name on the command line to BUFBOMB.

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 a miss.

10 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
    pushl $0x89abcdef # Push value onto stack
    addl $17,%eax # Add 17 to %eax
    .align 4 # Following will be aligned on multiple of 4
    .long 0xfedcba98 # A 4-byte constant
    .long 0x00000000 # Padding
```

The code can contain a mixture of instructions and data. Anything to the right of a '#' character is a comment. We have added an extra word of all 0s to work around a shortcoming in OBJDUMP to be described shortly.

We 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 lines

```
0: 68 ef cd ab 89  push $0x89abcdef
5: 83 c0 11        add $0x11,%eax
8: 98              cwtl                       Objdump tries to interpret
9: ba dc fe 00 00  mov $0xfedc,%edx           these as instructions
```

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 indicates the byte codes for the instruction. Thus, we can see that the instruction `pushl $0x89ABCDEF` has hex-formatted byte code `68 ef cd ab 89`.

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. Note also that it only generated two of the four bytes at the end with value `00`. Had we not added this padding, OBJDUMP gets even more confused and does not emit all of the bytes we want.

Finally, we can read off the byte sequence for our code (omitting the final 0's) as:

```
68 ef cd ab 89 83 c0 11 98 ba dc fe
```

11 Hand In Instructions

- Record your exploit strings for each level in a .txt file named ID1+ID2.txt
- Send your txt file to the TA (sse.ustc.edu.cn), the attachment is your own txt document, and the subject of your mail is as follows:
  - "Lab1 [ID1+ID2]" where ID1 is the Student Number of the first team member and ID2 is the Student Number of the second team member.

- Please check out the list of the teams who have turned in the codes on the ftp server.
- If you have turned in the codes, but your name doesn’t appear in the list, please contact to the TA.