

CS 304, Fall 2007

Lab Assignment L3: Buffer Overflow Exploits

Assigned: October 5, 2007

Due: Sunday, October 19, 8:50 am

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* (or *exploits*) 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.

The most widely publicized computer hacking episode based on a buffer overflow attack is the *Internet Worm* unleashed in November 1988 by Robert T. Morris, then a computer science graduate student at Cornell University. The worm used buffer overflow attacks to subvert the finger daemon and sendmail program on a UNIX machine, then used the opportunity provided by this lapse to break into the machine and send copies of itself to other UNIX machines. Because of a bug in the code, the program replicated and reinfected other machines at a faster rate than Morris intended, and as a result, many locations around the country either crashed or became “catatonic”, unable to deal with the increased workload. The buffer overflow techniques that Morris used are similar to those that you will be using here. Eugene Spafford of the Purdue Department of Computer Science wrote a detailed analysis of the worm describing exactly how it worked. Because of its relationship to buffer overflow attacks, you may find the report helpful and informative. You can access a copy of the report through a link on the web page for this project.

Since this was the first widely known case of its kind, there was considerable debate at the time about what penalty should be imposed on Robert Morris. After all, his worm didn’t have any malevolent intent (other than reproducing itself), and no harm was done (other than tying up UNIX systems—and their systems administrators—all over the country for a couple of days). In the end, the court system expressed a dim view of Morris’s computer hacking. Robert Morris was convicted of violating the computer Fraud and Abuse Act (Title 18), and sentenced to three years of probation, 400 hours of community service, a fine of \$10,050, and the costs of his supervision. His appeal, filed in December, 1990, was rejected the following March.

Hand Out Instructions

In the directory `/home/f85/cs304lab/304lab3`, you will see two executable files:

BUFBOMB: The code you will attack.

SENDSTRING: A utility to help convert between string formats.

These two programs are compiled to run on the machines in M-S 121.

In the following, we will assume that you have defined the lab directory to be on your execution path. You can do this by executing the following command:

```
prompt% setenv PATH /home/f85/cs304lab/304lab3:$PATH
```

Your Cookie

Your *cookie* is a unique (within the class) string of eight hexadecimal digits. In four of your five buffer attacks, your objective will be to make your cookie show up in places where it ordinarily would not. You will receive your cookie by email sometime during the day on October 5, 2007. Watch for it. Your progress on this project (and several subsequent projects) will be posted by cookie. You can access the current results of the lab by a link on the lab's web page.

The BUFBOMB Program

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

```
1 int getbuf()
2 {
3     char buf[20];
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 C 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 (allowing one character for the null terminator), it is clear that `getbuf` will return 1, as shown by the following execution example:

```
prompt% bufbomb
Type string: howdy doody
Dud: getbuf returned 0x1
```

Typically an error occurs if we type a longer string:

```
prompt% bufbomb
Type string: This string is too long
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. What will happen when the buffer is overrun cannot be determined from the C source code. To do this, you have to look carefully at the assembly language code for `getbuf` procedure.

Your task is to be more clever with the strings you feed BUFBOMB so that it does more interesting things than segmentation faults. These strings are called *exploit* strings. We have built features into BUFBOMB so that some of the key stack addresses you will need to use depend on your cookie.

BUFBOMB takes two different command line arguments:

- 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. Whatever you type at the terminal is in ASCII, so you will not be able to enter your exploit strings to the BUFBOMB program directly from the terminal. You can use the program SENDSTRING to convert the ASCII that you type at the terminal or enter into a text file into the *raw* strings that BUFBOMB requires.

The SENDSTRING program takes as input a *hex-formatted* ASCII string. In this format, each byte value is represented by two hex digits in ASCII. For example, if you wanted to construct an exploit string consisting of the hex bytes “21 38 99 6a a6 44”, then you could create a text file, say “exploit.txt”, consisting of the single line

```
21 38 99 6a a6 44
```

Note that these values are stored in the file in ASCII, as the following hex dump of the file shows:

```
prompt% hd exploit.txt
Hex Dump Utility, v. 1.1, 15 Nov 95
File: exploit.txt    Wed Sep 29 07:38:24 2004
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
000000  32 31 20 33 38 20 39 39 20 36 61 20 61 36 20 34  21 38 99 6a a6 4
000010  34 0A                                     4.
000012 (18) bytes read from file:  exploit.txt
```

The ASCII values of the bytes on the left are shown on the right, for those bytes that have a corresponding ASCII value; otherwise, a period (.) is shown on the right. Recall that the ASCII code for decimal digit x is $0x3x$. Thus, the first two characters of the file, 21, are represented in ASCII as 32 31. The next character, a blank, is represented as 20, the ASCII value for a blank (see “man ascii” for a full list of the ASCII character sequence). The rest of the ASCII characters of `exploit.txt` can be determined similarly.

There are several different ways to use SENDSTRING to convert an ASCII hex-formatted exploit string in the file `exploit.txt` into a “raw” string to feed to BUFBOMB:

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

```
prompt% cat exploit.txt | sendstring | bufbomb
```

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

```
prompt% sendstring < exploit.txt > exploit.raw
prompt% bufbomb < exploit.raw
```

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

```
prompt% gdb bufbomb
(gdb)run < exploit.raw
```

For the `exploit.txt` file shown above, here is the hexadecimal dump of the `exploit.raw` file that `SENDSTRING` produces from it:

```
prompt% hd exploit.raw
Hex Dump Utility, v. 1.1, 15 Nov 95
File: exploit.raw    Wed Sep 29 07:53:36 2004
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
000000  21 38 99 6A A6 44 0A                                !8.j.D.
000007 (7) bytes read from file:  exploit.raw
```

Note that the `exploit.raw` file contains the string of “raw” hex bytes that was desired.

`SENDSTRING` ignores non-hex digit characters, including the blanks in the example shown.

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.

When you correctly solve one of the levels, `BUFBOMB` will automatically notify the lab grading server. The server will test your exploit string to make sure it really works, and it will update the lab web page indicating that you (listed by cookie) have completed this level.

Unlike the bomb lab, there is no penalty for making mistakes in this lab. Feel free to fire away at `BUFBOMB` with any string you like.

Level 0: Candle (8 pts)

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

```
1 void test(int nitro)
2 {
3     int val;
4     volatile int local = 0xdeadbeef;
5     val = getbuf();
6     /* Check for corrupted stack */
7     if (local != 0xdeadbeef) {
8         printf("Sabotaged!: the stack has been corrupted\n");
9     }
10    else if (val == cookie) {
11        if (nitro) {
12            printf("INVALID!!! Attempting to validate nitro phase \
13 with test(), not testn()\n");
14            exit(0);
15        }
16        printf("Boom!: getbuf returned 0x%x\n", val);
17        validate(3, val);
18    }
19    else {
20        printf("Dud: getbuf returned 0x%x\n", val);
21    }
22 }
```

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

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0,0);
    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 address 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 disassembled 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.

Level 1: Sparkler (16 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);
        validate(1, val);
    } 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.

Level 2: Firecracker (24 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)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2, global_value);
    } 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 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 writeup.)
- Keep in mind that your exploit string depends on your machine, your compiler, and even your cookie. You must do all of your work on a machine in M-S 121.
- 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.
- Do not attempt to use a `jmp` instruction to jump to the code for `bang`. This instruction uses 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 (32 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.
- Since `getbuf` returns to its caller `test`, you will have great difficulty with this phase unless you realize what effect your exploit code will have on the stackframes of **both** `getbuf` and `test`.
- 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 cookie. You must do all of your work on a machine in M-S 121.

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 (8 pts)

If you have completed the first four levels, you have earned 80 points. You have mastered the principles of the runtime stack operation, and you have gained firsthand experience with buffer overflow attacks. We consider this a satisfactory mastery of the material. You are welcome to stop right now.

The next level is for those who want to push themselves beyond our baseline expectations for the course, and who want to face a challenge in designing buffer overflow attacks that arise in real life. This part of the

assignment only counts 8 points, even though it requires a fair amount of work to do, so don't do it just for the points.

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` and the exact saved value of `%ebp`. 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`:

```
int getbufn()
{
    char buf[512];
    Gets(buf);
    return 1;
}
```

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 (using library function `alloca`) that ranges between 0 and 127 bytes. Thus, if you were to sample the value of `%ebp` during two successive executions of `getbufn`, you would find they differ by as much as ± 127 .

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.

Some Advice:

- You can use the program `SENDSTRING` 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:

```
prompt% cat exploit.txt | sendstring -n 5 | bufbomb -n
```

You must use the same string for all 5 executions of `getbufn`. Otherwise it will fail the testing code used by our grading server.

- The trick is to make use of the `nop` instruction. It is encoded with a single byte (code `0x90`). You can place a long sequence of these at the beginning of your exploit code so that your code will work correctly if the initial jump lands anywhere within the sequence.

- You will need to restore the saved value of `%ebp` in a way that is insensitive to variations in stack positions.

Logistical Notes

Hand in occurs automatically whenever you correctly solve a level. The program notifies our grading server of your cookie and your exploit string. You will be informed of this by `BUFBOMB`. Upon receiving the notification, the server will validate your string and update the lab web page. You should check this page a few minutes after your submission to make sure your string has been validated. [If you really solved the level, your string *should* be valid.]

Note that each level is graded individually. You do not need to do them in the specified order, but you will get credit only for the levels for which the server receives a valid message.

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:

```
prompt% gcc -c example.s
prompt%> 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 indicate 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
```

Some of the exploit strings consist of a (long) sequence of “pad” characters, like `0x90` (the `nop` instruction) or `0x00` (the null byte), followed by a few instructions or addresses that you determine through the procedure outlined above. In this case, it is easier to generate the ASCII version of the exploit string with a Perl script. The web page for the project contains a link to the sample source for such a Perl script.