Homework 3

1 Logistics

Homework 3 is a one-person assignment worth 100 points.

The assignment is due on **Thursday, March 5** at **9:00 PM**. Hand-in is automatic.

2 Grading

Homework 3 has the following graded components:

- Level 0: Candle (10 points)
- Level 1: Sparkler (20 points)
- Level 2: Firecracker (30 points)
- Level 3: Dynamite (40 points)
- Bonus Level 4: Nitroglycerin (10 points)

3 Overview

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 run-time 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.

4 Assignment

4.1 Setup

You should complete this lab on osage.unl.edu. Because there are additional security features on `osage`, the bomb has been wrapped with a stack derandomizer. Be sure to read the last section of this document, which explains how this affects you.

Start by downloading 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 cse login.
- `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.

4.2 Cookie

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

```
linux> ./makecookie raik284h
0x316fd3c0
```

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

4.3 The Bufbomb Program

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

Listing 1: getbuf()

```
1 int getbuf(){
2   char buf[12];
3   Gets(buf);
4   return 1;
5 }
```

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:

```
linux> ./bufbomb
Type string: howdy doody
Dud:  getbuf returned 0x1
```

Typically an error occurs if we type a longer string:

```
linux> ./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. 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 login`: Operate the bomb for the indicated login, where `login` is your cse login. You should always provide this argument for several reasons:
  - It is required to log your successful attacks.
  - `bufbomb` determines the cookie you will be using based on your login, 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 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 hexadecimal-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 `0x3`x.) 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:

- You can set up a series of pipes to pass the string through `sendstring`:

```
linux> cat exploit.txt | ./sendstring | ./bufbomb -t raik284h
```

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

```
linux> ./sendstring < exploit.txt > exploit-raw.txt
linux> ./bufbomb -t raik284h < exploit-raw.txt
```

This approach can also be used when running `bufbomb` from within `gdb`:

```
linux> gdb bufbomb
(gdb) run -t raik284h < 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.

When you correctly solve one of the levels, `bufbomb` will automatically send an e-mail notification to our 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.

4.4 Attacks

4.4.0 Candle

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

Listing 2: test()

```
1  void test(){
2    int val;
3    volatile int local = 0xdeadbeef;
4    entry_check(3);  /* Make sure we entered this function properly */
5    val = getbuf();
6    /* Check for corrupted stack */
7    if(local != 0xdeadbeef){
8      printf("Sabotaged!: the stack has been corrupted\n");
9    }else if(val == cookie){
10     printf("Boom!: getbuf returned 0x%x\n", val);
11     validate(3);
12   }else{
13     printf("Dud: getbuf returned 0x%x\n", val);
14   }
15 }
```

When `getbuf` executes its return statement (line 4 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:

Listing 3: smoke()

```
1  void smoke(){
2    entry_check(0);  /* Make sure we entered this function properly */
3    printf("Smoke!: You called smoke()\n");
4    validate(0);
5    exit(0);
6  }
```

Your task is to get `bufbomb` to go to 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.

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.
- 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.

4.4.1 Sparkler

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

Listing 4: fizz()

```
void fizz(int val){
  entry_check(1);  /* Make sure we 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 Candle, 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.

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.

4.4.2 Firecracker

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:

Listing 5: bang()

```
int global_value = 0;

void bang(int val){
  entry_check(2);  /* Make sure we 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 Candle and Sparkler, 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`.

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. Do all of your work on osage, and make sure you include the proper login 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. If your solution is just a tad too big, you are probably using an immediate value as a memory address. 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 uses PC-relative addressing, which is tricky to set up correctly. Instead, push an address on the stack and use the `ret` instruction.

4.4.3 Dynamite

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
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`.

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: 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 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 cookie. Do all of your work on osage, and make sure you include the proper login 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 amiss.

4.4.4 Nitroglycerin

If you have completed the first four levels, you have earned 100 points. You have mastered the principles of the run-time 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 arises in real life. This part of the assignment only counts 10 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. More significantly though, modern operating systems intentionally add variation to the stack to hamper buffer overflow attacks.

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 only rarely and otherwise causes segmentation faults. 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 far less stable (but still stable enough to be vulnerable). 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`:

Listing 6: 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.

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`.

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 use the following command:

```
linux> cat exploit.txt | ./sendstring -n 5 | ./bufbomb -n -t raik284h
```

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.

4.5 Status

Hand in occurs automatically whenever you correctly solve a level. The program sends e-mail to our grading server containing your login (be sure to set the "`-t`" command line flag properly) and your exploit string to the grading server. You will be informed of this by `bufbomb`. Upon receiving the e-mail, 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. Have fun!

4.6 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:

Listing 7: example.s

```
# 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 zeros to work around a property of `objdump` to be described shortly.

We can now assemble and disassemble this file:

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

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

Listing 8: example.d

```
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 'code:' 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 zeros) as: `68 ef cd ab 89 83 c0 11 98 ba dc fe`.

5 The Derandomizer

To make buffer overflow attacks more difficult, most current operating systems randomize the stack. In such an operating system, it is as if the bomb is always running in Nitro mode, except that the variations in addresses can be larger than the maximum size of the stack! Even with the extra stabilizing code originally included in this assignment, exploding the bomb with a randomized stack would require many tries. With a recent upgrade, osage now has a randomized stack.

Therefore, we've wrapped a derandomizer around the bomb to eliminate these variations. Unfortunately, the de-randomization process is also good at confusing `gdb`. To get around this you need to start the program, wait for the stack to derandomize, and then attach the debugger.

The `shell` command will start a separate process from within `gdb`; you should also add a trailing ampersand (`\&`) so that `gdb` doesn't wait for the program to terminate before returning you to the debug prompt. The ampersand also means that you won't be able to type input directly, but this should be okay if you're using the `sendstring` utility.

```
(gdb) shell ./bufbomb -t raik284h &
[1] 5505
(gdb)  Stack successfully derandomized.
It is now safe to attach your debugger to process 5505.
Send an interrupt from the debugger to continue,
or directly if you aren't debugging.
```

The `attach` command will attach `gdb` to the separate process; it takes a process id as its argument. When you attach, `gdb` will break and wait for your input.

You can ignore error messages about "failing to read a valid object file." If you step into a system call, you will see that there is some wrapping code in its own object file responsible for getting the CPU into kernel mode so that the OS can process the request. `gdb` is trying to read this page of code to give you better information during debugging. Unfortunately, the kernel isn't letting `gdb` do that, so you'll have to get by with what `gdb` does know.

When you're ready for the derandomizer to invoke the bomb, ask `gdb` to send it a `SIGINT`. The program will give you a warning about the `next` command, but you shouldn't see any problems on osage, nor should you need the `next` command to complete this homework.

```
(gdb) signal SIGINT
Continuing with signal SIGINT.
Interrupt received; continuing.
If you are using a debugger, note that its 'next' command may no longer work;
the debugger might not have permission to put maintenance breakpoints on
siglongjmp().  This should be okay; just use 'step,' 'step by instruction,'
or place another breakpoint and use 'continue' if you need similar behavior.
Team:  raik284h
Cookie:  0x316fd3c0
Type string:Dud:  getbuf returned 0x1
Better luck next time

Program exited normally.
```

If you try to invoke the program directly from the debugger, it will give you an error message.

```
(gdb) r -t raik284h
Starting program:  ./bufbomb -t raik284h
Failed to read a valid object file image from memory.

It appears that you have started the program directly from a debugger.

This is probably a bad idea because the stack derandomization could confuse it.
Therefore, for your own safety (and your grade), the program will now exit.
If you want to debug this program, start it separately and attach your debugger
when prompted to do so.
```

```
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
```