CS 105
The Buffer Bomb

See calendar for lab due dates

Introduction

This assignment helps you develop a detailed understanding of the calling stack organization on an x86-64 processor. It involves applying a series of *buffer overflow attacks* on an executable file, `bufbomb`.

Important: 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, and people *have* gone to jail.

Submission

Once again you will work in pairs in solving the problems for this assignment. The only "hand-in" will be automated logging of your successful attacks. Any clarifications and revisions to the assignment will be posted on the course web page or announced on the mailing list.

Handout Instructions

First, download the file `buflab-handout.tar` from the course web page. Copy that file 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 Wilkes. 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.

Team Name and Cookie

You should create a team name for your group of the form "ID_1+ID_2" where ID_1 is the username of the first team member and ID_2 is the username of the second team member. For a solo team, just use that person's username with no plus sign.

Choose a consistent ordering of the IDs in the team name; teams "`jdoe+bsmith`" and "`bsmith+jdoe`" are considered distinct. **You must follow this scheme for generating your team name. Our grading program will only give credit to those people whose usernames 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 jdoe+bsmith
0x69d04aa0
```

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

The `bufbomb` Program

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

```
1 int getbuf()
2 {
3     char buf[12];
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, the destination is an array, `buf`, which has enough 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
Type string:howdy doody
Dud: getbuf returned 0x1
Better luck next time
```

Typically an error occurs if we type a longer string:

```
unix>./bufbomb
Type string:This string is far 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:

- It is required to log your successful attacks.
- `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 *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 `0x3`x.) Non-hex-digit characters are ignored, including the blanks in the example shown. **IMPORTANT NOTE:** `sendstring` will only pay attention to the first line of its input file.

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

1. You can use a pipe to pass the string through `sendstring` and feed it directly to `bufbomb`:

   ```
   unix>./sendstring < exploit.txt | ./bufbomb -t bovik
   ```

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 bovik < exploit-raw.txt
   ```

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

   ```
   unix>gdb bufbomb
   (gdb)run -t bovik < 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.

Because the exploit string contains non-ASCII characters, you should not try to use programs like `cat` to look at it, because it will mess up your terminal and you will have to log out of Wilkes and close your terminal window to clean things up. It's safe to use `less`, but it won't be very informative. The safe way to look at a binary file is with "`od -x file`".

When you correctly solve one of the levels, `bufbomb` will automatically send an email 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 your team (listed by cookie) has 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 (10 pts)

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

```
1  void test()
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         printf("Boom!: getbuf returned 0x%x\n", val);
12         validate(3);
13     }
14     else {
15         printf("Dud: getbuf returned 0x%x\n", val);
16     }
17 }
```

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`, which has 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 having `getbuf` return to `test` as it normally would. 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 (except for 0x0A). We are found of counting (e.g. 01 02 03...).

Level 1: Sparkler (20 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);
    } 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. That's problematic, because the argument is passed in register `%rdi`, and you can't change that register simply by writing your cookie into an appropriate place on the stack.

However, there is a clever technique called *Return-Oriented Programming* (ROP), which takes advantage of preexisting code. The idea is to find a few instructions somewhere in the program that do what you want and are followed by a `ret` or `retq` instruction. These sequences are called *gadgets*. By appropriately clobbering the stack, you can cause one or more gadgets to execute and do what you want, and then cause `fizz` to be called.

In general, gadgets can be a bit hard to find, so there are automated tools that can help you out. For this lab, we have provided two functions named `nsq` and `clobbercookie`. Neither is useful to you as-is, but you might find one or more useful gadgets inside.

Some Advice

- You will have to arrange that more than one `ret` instruction is executed, and that they execute useful code in the proper sequence.

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)
{
    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 writeup.)
- 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 Wilkes, and make sure you include the proper team name on the command line to `bufbomb`.
- Our solution requires 17 bytes of exploit code. Fortunately, there is enough room on the stack because `gcc` allocates a bit of extra space to alilgn the stack.
- 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 *contents of* memory location `0x00000004` into `%eax`. Since that memory location is normally 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.

A Side Note on Defenses

The attack in Firecracker requires executing code from the stack. Since no sane program has instructions on the stack, modern operating systems use special hardware settings to prevent the machine from accepting instructions that are in the part of memory devoted to the stack. We have specially disabled that feature for the purposes of this lab.

In fact, Return-Oriented Programming (Used in Level 1) was originally invented to work around this particular defense. Since ROP uses gadgets that are already in the text of a program, it doesn't need to execute from the stack. And it turns out that any large program is likely to contain enough random gadgets to implement a Turing machine—i.e., to do any chosen computation. Ouch!

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 corrupted the stack in various ways, including overwriting 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 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.
- 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 Wilkes, 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 amiss.

Level 4: Nitroglycerin (10 pts)

If you have completed the first four levels, you have earned 100 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, and 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 for 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. Stack positions may also differ when running a program under `gdb`, since `gdb` uses stack space for some of its own state. And finally, the operating system uses a technique called *Address Space Layout Randomization* (ALSR) to make addresses harder to predict so that buffer-overflow attacks won't work as well. (We've disabled ALSR on Wilkes for the duration of this lab.)

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 technique made it possible for you to write an exploit string that knows the exact starting address of `buf`. If you tried to use such an exploit on a normal program, you would find that it works sometimes, 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 the library function `alloca`) that ranges between 0 and 127 bytes. Thus, if you were to sample the value of `%rsp` during two successive executions of `getbufn`, you would find they differ by as much as ± 128.

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 `testn`, rather than the value 1. You can see in the code for `testn` 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 `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:

  ```
  unix>./sendstring -n 5 < exploit.txt | ./bufbomb -n -t bovik
  ```

  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. (This technique is often referred to as a "nop slide".)

Logistical Notes

Hand-in occurs automatically whenever you correctly solve a level. The program sends email to our grading server containing your team name (be sure to set the "`-t`" command-line flag properly) and your exploit string. `Bufbomb` will inform you when the message is sent. Upon receiving the email, 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!

Cheating the Lab

On certain levels, you may discover that you can outsmart the lab and come up with a simpler solution that still causes the grading server to accept what you did. You are welcome to experiment to discover these tricks. However, for three reasons we ask that you do not use such tricks as your final solution for that level:

1. You won't learn as much as you will by doing it the "right" way.
2. Later levels aren't susceptible to the same trick, so you'll just wind up having to come up with the proper solution later, in a more difficult setting.
3. It's an honor-code violation.

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
        pushq $0x01abcdef       # Push value onto stack
        addq $17,%rax           # Add 17 to %rax
        .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 (among others):

```
   0: 68 ef cd ab 01        pushq  $0x1abcdef     Leading 0 is missing
   5: 48 83 c0 11           add    $0x11,%rax     Constant is given in hex
   9: 0f 1f 00              nopl   (%rax)         Inserted by .align
   c: 98                    cwtl                  Objdump tries to interpret
   d: 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 `pushq $0x01abcdef` has hex-formatted byte code `68 ef cd ab 01`.

Starting at address c, 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 the file only generated two of the four bytes at the end with value `00`. Had we not added this padding, `objdump` would get even more confused and would 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 01 48 83 c0 11 0f 1f 00 98 ba dc fe 00 00
```

It can be convenient to save the output of `objdump` into a file and then edit it by removing the address at the front, deleting the textual representation, and joining the lines together. The `emacs` editor is very good at this sort of task.