Buffer Overflows?|?d??|?d?Segmentation fault: 11

Learning Objectives

- Gain a detailed understanding of x86-64 stack organization
- Gain a better understanding of what decisions are made at compile time vs. what modifications/decisions can occur when the program runs
- Refine your skills examining x86-64 assembly in gdb
- Understand how several types of buffer overflow exploits can affect a program

Overview

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 called `bufbomb`.
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.

Acquiring the Code

Browser

You can download lab3.tar here from a browser.

Command Line (the CSE VM or your home machine)

Use wget to download from the command line. In a terminal, run:

```
wget https://courses.cs.washington.edu/courses/cse410/17wi/labs/lab3/lab3.tar
```

Extracting a tar file

Once you have a local copy, extract its contents using the following command:

```
tar xvf lab3.tar
```

This will cause a number of files to be unpacked in a directory called lab3:

File Name	Description
makecookie	Generates a "cookie" based on some string (which will be your username)
bufbomb	The executable you will attack
bufbomb.c	The important bits of C code used to compile bufbomb
sendstring	A utility to help convert between string formats
Makefile	For testing all your exploits prior to submission

All these programs are compiled to run on the 64-bit CSE VM, klaatu. The rest of the instructions assume that you will be performing your work on one of those platforms. You should be sure your solution works on one of those platforms before submitting it!

Instructions

Be sure to read this write-up in its entirety before beginning your work.

A Note about Line Endings

Linux (and UNIX machines in general) use a different line ending from Windows and traditional MacOS in text files. The reason for this difference is historical: early printers need more time to move the print head back to the beginning of the next line than to print a single character, so someone introduced the idea of separate line feed `\n` and carriage return `\r` characters. Windows and HTTP use the `\r\n` pairs, MacOS uses `\r`, and Linux uses `\n`. In this lab, it is important that your lines end with line feed (`\n`), not any of the alternative line endings. If you are working on the VM or klaatu or even another Linux system this will probably not be a problem, but if you working across systems, check your line endings. You can also use the Unix tool `dos2unix`to convert the line endings from your host OS (Windows or Mac) to Unix line endings.

Making Your Cookie

A cookie is a string of eight bytes (or 16 hexadecimal digits) that is (with high probability) unique to you. You can generate your cookie with the `makecookie` program giving your UWNetID as the argument (note, you must use your UWNetID):

```
$ ./makecookie your_UWNetID
0x5e57e63274f39587
```

As an example, if your UW email address is `thecookiemonster42@uw.edu`, you would run `./makecookie thecookiemonster42`.
While you are doing this, you might as well prepare the first file you need to turn in: UW_ID.txt

```
$ echo your_UWNetID > UW_ID.txt
```

Where you replaced `your_UWNetID` with your real username as above, this will generate a text file containing your UWNetID followed by a single new line. You could also use a text editor, but you have to be careful about line endings.
In most of the attacks in this lab, your objective will be to make your cookie show up in places where it ordinarily would not.

Using the bufbomb Program

The bufbomb program reads a string from standard input with the function getbuf():

```
unsigned long long getbuf() {
  char buf[36];
  volatile char* variable_length;
  int i;
  unsigned long long val = (unsigned long long)Gets(buf);
  variable_length = alloca((val % 40) < 36 ? 36 : val % 40);
  for(i = 0; i < 36; i++) {
    variable_length[i] = buf[i];
  }
  return val % 40;
}
```

Don't worry about what's going on with variable_length and val and alloca for now; all you need to know is that getbuf() calls the function Gets and returns some arbitrary value.

The function Gets is similar to the standard C library function gets—it reads a string from standard input (terminated by '\n') and stores it (along with a null terminator) at the specified destination. In the above code, the destination is an array buf having sufficient space for 36 characters.

Neither Gets nor gets have 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 36 characters long, it is clear that getbuf will return some value less than 0x28, as shown by the following execution example:

```
$ ./bufbomb
Type string: howdy doody
Dud: getbuf returned 0x20
```

It's possible that the value returned might differ for you, since the returned value is derived from the location on the stack that Gets is writing to. The returned value will also be different depending on whether you run the bomb inside gdb or run it outside of gdb for the same reason.

Typically an error occurs if we type a longer string:

```
$ ./bufbomb
Type string: This string is too long and it starts overwriting things.
Ouch!: You caused a segmentation fault!
```

As the error message indicates, overrunning the buffer typically causes the program state (e.g., the return addresses and other data structures that were stored on the stack) 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` must be run with the `-u your_UWNetID` flag, which operates the bomb for the indicated UWNetID. (We will feed bufbomb your UWNetID with the `-u` flag when grading your solutions.) `bufbomb` determines the cookie you will be using based on this flag value, just as the program `makecookie` does. Some of the key stack addresses you will need to use depend on your cookie.

Formatting Your Exploit Strings

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. `sendstring` takes as input a hex-formatted string and prints the raw string to standard output. In a hex-formatted string, each byte value is represented by two hex digits. Byte values are separated by spaces. For example, the string `"012345"` could be entered in hex format as `30 31 32 33 34 35`. (The ASCII code for decimal digit `Z` is `0x3Z`. Run `man ascii` for a full table.) Non-hex digit characters are ignored, including the blanks in the example shown.

If you generate a hex-formatted exploit string in a file named `exploit.txt`, you can send it to `bufbomb` through a couple of pipes (see CSE390A course notes if you are unfamiliar with Unix pipes that take the output of one program and direct it as input to another program):

```
$ cat exploit.txt | ./sendstring | ./bufbomb -u your_UWNetID
```

Or you can store the raw bytes in a file and use I/O redirection to supply it to `bufbomb`:

```
$ ./sendstring < exploit.txt > exploit.bytes
$ ./bufbomb -u your_UWNetID < exploit.bytes
```

With the above method, when running bufbomb from within gdb, you can pass in the exploit string as follows:

```
$ gdb ./bufbomb
(gdb) run -u your_UWNetID < exploit.bytes
```

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 using gdb, you may find it useful to save a series of gdb commands to a text file and then use the -x commands.txt flag. This saves you the trouble of retyping the commands every time you run gdb. You can read more about the -x flag in gdb's manpage.

Generating Byte Codes

(You may wish to come back and read this section later after looking at the problems.)
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
movq $0x1234abcd,%rax     # Move 0x1234abcd to %rax
pushq $0x401080           # Push 0x401080 on to the stack
retq                      # Return
```

The code can contain a mixture of instructions and data. Anything to the right of a '#' character is a comment.
We can now assemble and disassemble this file:

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

The generated file example.d contains the following lines

```
   0:    48 c7 c0 cd ab 34 12         mov    $0x1234abcd,%rax
   7:    68 80 10 40 00          pushq  $0x401080
   c:    c3                      retq
```

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 $0x401080` has a hex-formatted byte code of `68 80 10 40 00`.
If we read off the 4 bytes starting at address 8 we get: `80 10 40 00`. This is a byte-reversed version of the data word `0x00401080`. This byte reversal represents the proper way to supply the bytes as a string, since a little-endian machine lists the least significant byte first.
Finally, we can read off the byte sequence for our code (omitting the final `0`'s) as:
`48 c7 c0 cd ab 34 12 68 80 10 40 00 c3`

The Exploits

There are three functions that you must exploit for this lab. The exploits increase in difficulty. For those of you looking for a challenge, there is a fourth function you can exploit for extra credit.

Level 0: Candle

The function `getbuf` is called within `bufbomb` by a function `test`:

```
void test()
{
   volatile unsigned long long val;
   volatile unsigned long long local = 0xdeadbeef;
   char* variable_length;
   entry_check(3);  /* Make sure entered this function properly */
   val = getbuf();
   if (val <= 40) {
      variable_length = alloca(val);
   }
   entry_check(3);
   /* Check for corrupted stack */
   if (local != 0xdeadbeef) {
      printf("Sabotaged!: the stack has been corrupted\n");
   }
   else if (val == cookie) {
      printf("Boom!: getbuf returned 0x%llx\n", val);
      if (local != 0xdeadbeef) {
          printf("Sabotaged!: the stack has been corrupted\n");
      }
      if (val != cookie) {
          printf("Sabotaged!: control flow has been disrupted\n");
      }
      validate(3);
   }
   else {
      printf("Dud: getbuf returned 0x%llx\n", val);
   }
}
```

When getbuf executes its return statement, the program ordinarily resumes execution within function test. Within the file bufbomb, there is a function smoke:

```
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, because 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.

- Check the line endings in your smoke.txt with `od -c smoke.txt` or `hexdump -C smoke.txt`.

Level 1: Sparkler

Within the file `bufbomb` there is also a function `fizz`:

```
void fizz(int arg1, char arg2, long arg3,
    char* arg4, short arg5, short arg6, unsigned long long val)
{
    entry_check(1); /* Make sure entered this function properly */
    if (val == cookie)
    {
        printf("Fizz!: You called fizz(0x%llx)\n", val);
        validate(1);
    }
    else
    {
        printf("Misfire: You called fizz(0x%llx)\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.

Advice:

- Note that in x86-64, the first six arguments are passed into registers and additional arguments are passed through the stack. Your exploit code needs to write to the appropriate place within the 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 address of the buffer `buf`.

Level 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.

For level 2, you will need to run your exploit within `gdb` for it to succeed. (the CSE VM and `klaatu` have special memory protection that prevents execution of memory locations in the stack. Since `gdb` works a little differently, it will allow the exploit to succeed.)

Within the file `bufbomb` there is a function `bang`:

```
unsigned long long global_value = 0;

void bang(unsigned long long val)
{
    entry_check(2); /* Make sure entered this function properly */
    if (global_value == cookie)
    {
       printf("Bang!: You set global_value to 0x%llx\n", global_value);
       validate(2);
    }
    else
    {
        printf("Misfire: global_value = 0x%llx\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 `retq` 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 address of the buffer buf.
- 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 in the Generating Byte Codes section above.)
- Keep in mind that your exploit string depends on your machine, your compiler, and even your cookie. Make sure your exploit string works on klaatu or your VM, and make sure you include your UWNetID on the command line to bufbomb.
- Watch your use of address modes when writing assembly code. Note that movq $0x4, %rax moves the value 0x0000000000000004 into register %rax; whereas movq 0x4, %rax moves the value *at* memory location 0x0000000000000004into %rax. Because that memory location is usually undefined, the second instruction will cause a segmentation fault!
- 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 retq instruction.

Extra Credit – Level 3: Dynamite

For level 3, you will need to run your exploit within gdb for it to succeed.
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 %rbpand 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`.

Advice:

- In order to overwrite the return pointer, you must also overwrite the saved value of `%rbp`. 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 `%rbp` 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 `%rbp`.
- 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. Again, again make sure your exploit string works on `klaatu` or the VM, and make sure you include your UWNetID on the command line to `bufbomb`.

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.

`execve` is system call that replaces the currently running program with another program inheriting all the open file descriptors. What are the limitations of the exploits you have preformed so far? How could calling `execve` allow you to circumvent this limitation? If you have time, try writing an additional exploit that uses execve and another program to print a message.

Submitting Your Work

You should submit the following files:

- smoke.txt
- fizz.txt
- bang.txt
- dynamite.txt (if you did the extra credit)
- UW_ID.txt(as per below)

The first four files correspond to the different exploits. Each of these files should *only* contain the hex-formatted exploit string. The UW_ID.txt file should contain your UWNetID (*without* the @uw.edu part) followed by an empty line. *Please follow the formatting specified here. Our grading scripts won't be nice if you don't name the files like we've asked or if you include additional text in any of the files.* Note in particular that the text files should have the data that is sent to sendstring, *not* the data produced by sendstring.

Before submitting your exploits, you can check them by placing them in the same directory as bufbomb and running make test. This will output a summary of your exploits (the Makefile looks for all the files ending with .txt and sends the contents of each to bufbomb, one by one) and whether they succeed. You need to create UW_ID.txt before using the Makefile.

Once you're satisfied with your solutions, submit them through the Canvas. There will not be partial credit within a level.

GDB (Gnu DeBugger) Resources

Overview

GDB is an immensely useful tool to help you debug your C programs. It lets you insert *breakpoints* into your programs so that you can stop execution and examine the contents of memory and registers. It also supports single-stepping your program one line of source code at a time. It leads to much more productive debugging than just using `printf`. The time you spend getting familiar with GDB will be an excellent investment.

GDB cheat sheet

A handy dandy guide to the most commonly used GDB commands. Useful to have open while you watch the video and work on the examples below.

Watch the video

This very useful video shows you how to get started with GDB.

An extensive tutorial

Looking for more details? Take a look at this extensive tutorial thanks to Norman Matloff at UC Davis.

Review with two examples

Here are two examples for playing with GDB with some example commands you can try:

- Array example and typical GDB commands
- Linked list example and typical GDB commands