



[Skip to page content](#)



Lab 3: Buffer Overflows?

Segmentation fault: 11

Assigned: Friday, April 26, 2024

Due Date: Wednesday, May 8, 2024 at 11:59 pm

Video(s): Watch this  video on Phase 0  (with captions) before you begin!

You may also find this  video on endianness  (with captions) helpful as you work with GDB throughout the lab.

Overview

Learning Objectives:

- Label and describe the utility of different parts of a x86-64 stack frame.
- Understand and explain what decisions are made at compile time and what modifications/decisions can occur when the program runs.
- Given a program, students are able to examine and execute x86-64 assembly instructions and use `gdb` commands (e.g., set and use breakpoints, print register values).
- Provide examples of several types of buffer overflow exploits and explain how they can affect a program.

This assignment involves applying a series of buffer overflow attacks on an executable file called `bufbomb` (for some reason, the textbook authors have a penchant for pyrotechnics).

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.*

Code for this lab

Browser:

 [Download here](#)

Terminal:

wget

`https://courses.cs.washington.edu/courses/cse351/24sp/labs/lab3.tar.gz`

Unzip:

Running `tar xzvf lab3.tar.gz` from the terminal will extract the lab files to a directory called `lab3` with the following files:

- `bufbomb` - The executable you will attack
- `bufbomb.c` - The C code used to compile `bufbomb` (You don't need to compile it)
- `lab3synthesis.txt` - For your synthesis questions responses
- `Makefile` - For testing your exploits prior to submission
- `makecookie` - Generates a "cookie" based on some string (which will be your username)
- `sendstring` - A utility to help convert between string formats

Lab 3 Instructions

Lab Format

You have been provided a vulnerable executable called `bufbomb` that we will perform a few variants of buffer overflow attacks on. The vulnerability comes from reading 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` , `val` , and `alloca()` ; 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 character) at the specified destination. In the above code, the destination is an array `buf` that has 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, no buffer overflow occurs and the program proceeds as written, with `getbuf()` returning a value less than `40 = 0x28` because of the modulus. For example:

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

The actual value returned might differ for you, since `Gets()` returns its argument, which is the stack address of `buf` . This value will also differ depending on whether you run `bufbomb` inside `gdb` or not.

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 stored on the stack) to be corrupted, leading to a memory access error. Your tasks for this lab is to be more clever with the strings you feed `bufbomb` so that it does more interesting things. These are called *exploit strings*.

Your Cookie

In order to personalize your experiences, you will have to use a unique "cookie" based on your UWNetID. Substitute "netid" wherever you see it with your UWNetID (in all lowercase). We are omitting the usual brackets (`<>`) around this because these characters are used in some of the commands for this lab.

Your cookie is a string of eight bytes (= 16 hex digits) that is generated with the `makecookie` program, passing in your UWNetID as the argument. An example output might be:

```
$ ./makecookie jhsia
42
5b
7d
72
40
b9
c3
1d
0x1dc3b940727d5b42
```


You will need to submit your UWNetID in a file called `UW_ID.txt` and this is also used by `make` when you `test your exploits`. You should generate this file using the following command:

```
$ echo netid > UW_ID.txt
```

Remember to replace "netid" with your lowercase UWNetID. Our autograder scripts will verify that the contents match either your or your partner's UWNetID.

When running `bufbomb`, you must supply it with the `-u netid` flag. Most of the levels in this lab will compare values against the generated cookie for the provided "netid", so it is crucial that this flag value matches what is stored in `UW_ID.txt`.

Your Exploit Strings

The following commands use Unix pipes to redirect the output of one program to the input of another program. See the  [CSE391 slides](#) on piping and I/O redirection if you want to learn more or would like a refresher.

Formatting requirements and `sendstring`


Linux uses 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 CSE Linux environment (or even another Linux system), this will probably not be a problem, but if you are 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.

Your exploit strings will typically contain byte values outside of the printable ASCII character range. The program `sendstring` will help you generate these raw strings by taking a hex-formatted text string and printing the converted binary string to standard output. In a hex-formatted text string:

- Each byte value is represented by two ASCII hex digits (e.g., "5e" – letters can be upper or lowercase).
- Byte values are separated by spaces (e.g., "30 48 5e").
- Non-hex digits (i.e., anything outside of the ranges 0–9, A–F, and a–f) are ignored.
- Do not ever use the byte value `0x0A`, since this is the ASCII code for newline (`'\n'`). When `Gets()` encounters this byte, it will assume you intended to terminate the string and won't use your full exploit string.

`sendstring` will read each byte and convert it into the corresponding ASCII character (see

 [this website](#) or run `man ascii` for a full table). For example, if we stored the text "30 48 5e" in a file called `ex.txt`, then running it through `sendstring` would produce the following output:

```
$ ./sendstring < ex.txt
0H^
```

because `0x30` → `'0'`, `0x48` → `'H'`, and `0x5e` → `'^'` according to the ASCII table.

Generating binary files and `bufbomb`

We will need to store the output of `sendstring` to a file so we can use it with `bufbomb` within `gdb`. Assuming that you used your text editor of choice to save your hex-formatted text string in `exploit.txt` (we'll rename these based on the name of the level you're working on), the following command will store the output of `sendstring` to the file `exploit.bytes`:

```
$ ./sendstring < exploit.txt > exploit.bytes
```

The choice of file extension `.bytes` is arbitrary but is intended to remind you that this is a binary file (as opposed to a text file). It doesn't really make sense to open `*.bytes` files in a text editor, as non-printable characters will show up in weird ways. **You will only be submitting the `*.txt` files.**

Now you can pass the binary exploit string to `bufbomb` from within `gdb` as follows:

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

When using `gdb`, you may find it useful to save a series of 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 `man` page.

Testing

To test each exploit individually, follow the commands above to generate your `.bytes` file and pass it into `bufbomb` within `gdb`. Each exploit uses a different name for the files.

For example, Level 0 is called "smoke" so you would write your hex-formatted text string in a text editor into the file `smoke.txt` and then run the following commands:

```
$ ./sendstring < smoke.txt > smoke.bytes
$ gdb bufbomb
(gdb) run -u netid < smoke.bytes
```

The individual levels are explained below in the [Exploits section](#), including what the expected output text is on success. You should also make sure that you do not encounter a segfault.

Makefile

You can also test all your exploits at once by running `make` from within the `lab3` directory, which will output a summary of their success.

- This relies on you having your UWNNetID in `UW_ID.txt`.
- Naming is very important here: this will specifically look for `smoke.txt`, `fizz.txt`, `bang.txt`, and `dynamite.txt` and, if found, run them through `sendstring` and `gdb`.
- You do NOT need to have all of the levels completed before using `make`; it will skip any of the levels that are not found.

We strongly recommend that you use `make` before submitting your lab, as this will catch issues in file naming and contents.

Generating Byte Codes

You won't write assembly code for Levels 0 and 1. You can read this section later after finishing these levels.

Using `gcc` as an assembler and `objdump` as a disassembler makes it convenient to generate the byte codes for instruction sequences.

1. **Write a file containing your desired assembly code.** For example, suppose we have the following `example.s` (recall that anything to the right of a `#` character is a comment):

```
# Example of hand-generated assembly code
movq $0x1234abcd,%rax    # Move 0x1234abcd to %rax
pushq $0x401080          # Push 0x401080 onto the stack
retq                    # Return
```

Refer back to course material for how to construct valid assembly instructions and how to distinguish between different types of operands. Lots of students make mistakes here that can be difficult to debug, as the assembler does not always produce an error when you might expect it to.

2. **Assemble and disassemble this file.**

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

3. **Open/view the generated file** `example.d` to see 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).
- The hex digits after the `:` character indicate the byte codes for the instruction, with byte positions *increasing* from left-to-right (e.g., the byte `c7` is at "address" 1, the byte `12` is at "address" 6).
- The right column is the *interpreted* assembly instruction from the bytes. If this does not match your hand-written assembly, this likely indicates a syntax error.

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.

4. **Construct the byte sequence for the code from the disassembly:**

```
48 c7 c0 cd ab 34 12 68 80 10 40 00 c3
```

The Exploits

Many different functions and line numbers within `bufbomb` are mentioned below.

If you want to **view the source (C) code** of the functions in order to get a sense for what the code is intended to do, there are a few recommended ways to do so:

- Open `bufbomb.c` in a text editor and navigate to the line number or search for the function definition.
- Within `gdb bufbomb`, use `list <#>`, where `<#>` is a line number, to display 10 lines of code centered around `<#>`.
- Within `gdb bufbomb`, use `list <func>`, where `<func>` is a name of a function, to display 10 lines of code centered around the beginning of that function's definition. Pressing [Enter] again (repeat command) will display the next 10 lines and you can repeat this until you've read through the whole function definition.

If you want to **find the address of a function**, which is also the address of the first instruction of the function, you can either:

- Within `gdb bufbomb`, use `print <func>`, where `<func>` is a name of a function, to print out its address.
- Within `gdb bufbomb`, use `disas <func>`, where `<func>` is a name of a function, to disassemble the beginning of the function. The address associated with the `<+0>` instruction is the function's address.

Level 0: Smoke

The function `getbuf()` is called by a function `test()` on **Line 108**. When this call to `getbuf()` executes its return statement, the program ordinarily resumes execution within `test()`.

Your task is to get `bufbomb` to return to a different function, `smoke()` (**Line 62**), from `getbuf()` instead of `test()`. You will supply an exploit string that overwrites the stored return address in `getbuf()`'s stack frame with the address of the first instruction in `smoke()`. When supplied with a correct exploit string, you should see the following output:

```
Smoke!: You called smoke()
```

Smoke Hints:

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of `bufbomb`.

- The placement of `buf` within the stack frame for `getbuf()` depends on which version of `gcc` was used to compile the executable. 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.
- Be careful with byte ordering (*i.e.*, endianness) so that the corrupted return address is read correctly.
- You can use `gdb` to step the program through the last few instructions of `getbuf()` to make sure that it is doing the right thing. You can also print out the data in the stack (the `x` command) to see the change.

Level 1: Fizz

There is another function called `fizz()`, which compares one of its arguments (`val`) against your cookie. Similar to Level 0, your task is to get `bufbomb` to return to `fizz()` from `getbuf()` instead of `test()`. However, you must also get your exploit string to change the value of `val` to your cookie by encoding it in the appropriate place. When supplied with a correct exploit string, you should see the following output:

```
Fizz!: You called fizz(<your cookie value>)
```

Fizz Hints:

- Which argument number/position is `val`? How/where is this argument passed in x86-64 and how can you use your exploit string to change this argument?
- Note that the output from `fizz()` prints out `val`, if you are having issues with byte ordering or byte shifts. We do highly recommend using `gdb` to step through the return from `getbuf()` and print out the data in the stack (the `x` command) to verify behavior instead of brute forcing things.

Level 2: Bang

A much more sophisticated form of buffer attack involves supplying a string that encodes actual machine instructions (*i.e.*, a "code injection" attack). The exploit string then overwrites the return address 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. For this style of attack, you must get machine code onto the stack and set the return pointer to the start of this code.

The bang exploit will ONLY work within `gdb` because the Linux shell has memory protections that prevents instruction execution from the stack (which will result in a segfault). So the

exploit will always fail if you run `./bufbomb -u netid < bang.bytes` , but may succeed if you use `run -u netid < bang.bytes` from within `gdb bufbomb` .

There is another function called `bang()` , which compares a global variable (`global_value`) against your cookie. Your task is to get `bufbomb` to execute the code for `bang()` . However, you must set `global_value` to your cookie *before* reaching `bang()` . To do this, 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()` . When supplied with a correct exploit string, you should see the following output:

Bang!: You set `global_value` to <your cookie value>

Note that even if you see the expected output above, your solution won't be considered correct if `bufbomb` doesn't exit "normally" (e.g., segfaults).

Bang Hints:

- You will need the addresses of `global_value` and `buf` , which can be determined using the `print` command within `gdb`.
- Refer to the [Generating Byte Codes](#) section for how to get the bytes that correspond to your desired assembly instructions to put into your exploit string.
 - Watch your use of memory address modes when writing assembly. For example, `movq $0x4, %rax` moves the value `0x0000000000000004` into register `%rax`, whereas `movq 0x4, %rax` moves the value *at* memory location `0x0000000000000004` into `%rax`, i.e., `0x4` is being interpreted as the `D` field in memory addressing mode `D(Rb, Ri, S)`.
 - The `movq` instruction cannot directly move an 8-byte immediate (e.g., `$0x0123456789ABCDEF`) to a memory location. There are multiple ways to achieve this desired behavior, such as first moving it to a register before moving to the memory address.
 - 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.
 - Note that the generated byte code depends on your hand-written assembly code as well as the compiler you used. To verify that they are being interpreted properly, you can use the command `x /<#>i <addr>` in `gdb` to print out the contents of memory starting at the address expression `<addr>` until it interprets `<#>` assembly instructions. Any discrepancies between the interpretation and your original code might indicate a syntax error in the assembly or a mismatch in the compiler used to generate the byte code.

- To verify that the exploit code does what you expect/intend it to, you should use `gdb` to step through it line-by-line after the `ret` in `getbuf()`.
- Make sure that your exploit string works (1) on the CSE Linux environment, (2) within `gdb`, and (3) with your *correctly spelled* (and lowercase) UWNetID.

Level 3: Dynamite [Extra Credit]

Similar to Level 2, the `dynamite` exploit will ONLY work within `gdb`.

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 `%rbp` and the return address. 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), meaning that the calling function is oblivious to the attack! For this style of attack, you must: (1) get machine code onto the stack, (2) set the return address to the start of this code, and (3) undo the corruptions made to the stack state.

Look at the `test()` function, the one that calls `getbuf()`. Your task 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, all while *not corrupting important stack values*. To do this, your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location onto the stack, and execute a `ret` instruction to really return to `test`. When supplied with the correct exploit string, you should see the following output:

```
Boom!: getbuf returned <your cookie value>
```

Note that even if you see the expected output above, your solution won't be considered correct if `bufbomb` doesn't exit "normally" (e.g., segfaults).

Dynamite Hints:

- All of the hints from Level 2 still apply, except you will need to find the saved value of `%rbp` on the stack instead of `global_value`.
- In order to overwrite the return address, 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.

- Remember that `test()` is "still executing" while your exploit string does its thing. What does it mean for a function to still be executing in regards to the stack memory? Remember that `test()` needs to find everything as it left it when it resumes.

Food for Thought

Reflect on what you have accomplished – you caused a program to execute machine code of your own design! You may have even 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 performed 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.

Lab 3 Synthesis Questions

Start with a *fresh* copy of `lab0.c` again and go to `part_2()` to change the second argument to the first call to `fill_array` so that you see the message "Segmentation fault" when you run part 2:

```
$ wget https://courses.cs.washington.edu/courses/cse351/24sp/labs/lab0.c
$ gcc -g -std=c18 -fomit-frame-pointer -o lab0 lab0.c
$ ./lab0 2
*** LAB 0 PART 2 ***
...
Segmentation fault
```

Examine the contents of memory in GDB to figure out what happened and answer the following questions:

- In your own words, *explain* the cause of this specific segmentation fault.
 - What value gets corrupted and why does it cause the segmentation fault?
 - Which assembly instruction causes the segmentation fault to occur at the moment it is executed? (Be specific: give the name of the instruction as well as the name of the function where it is found.) [3 pt]
- It turns out that you can figure out when you will get a segfault in `part_2` just by looking at the assembly code! There are a few instructions that contribute to determining the limit on the second argument to `fill_array`. What is the minimum length needed to cause a

segmentation fault? (Please briefly explain the calculation you did to find the minimum length.) [2 pt]

⚠ The following questions are open-ended! They are less about "right" or "wrong" and more about you giving us your personal reflection.

3. What point in the lab made you "feel like a hacker" the most? Did it feel exciting, concerning, or something else and why? [2 pt]
4. How does the realization of your ability to exploit these vulnerabilities affect the way you view computer security? [2 pt]


Submission

You will submit: `UW_ID.txt` , `smoke.txt` , `fizz.txt` , `bang.txt` , and `lab3synthesis.txt` .

It is your responsibility to make sure that your submission files are named correctly and have the proper contents. Common issues to watch out for include:

- Spelling and capitalization issues in file names (especially `UW_ID.txt`).
- `UW_ID.txt` should contain the UW netid (not CSE netid, if different) of either you or your partner in *all lowercase*.
- The exploit files should contain the human-readable characters (*i.e.*, text files that you would pass `INTO sendstring`) and not the converted binary data (*i.e.*, output of `sendstring`).

After submitting, please wait until the autograder is done running and double-check that you passed the "File Check" and "Compilation and Execution Issues" tests. If either test returns a score of -1, be sure to read the output and fix any problems before resubmitting. Failure to do so will result in a programming score of ZERO for the lab.

Submit your files to the "Lab 3" assignment on  Gradescope . Don't forget to add your partner, if you have one.

If you completed the extra credit, also submit `dynamite.txt` and `UW_ID.txt` to the "Lab 3 Extra Credit" assignment.

It is fine to submit multiple times up until the deadline; we will only grade your last submission. If you do re-submit, you must re-submit ALL files again AND add your partner again to the new submission.



UW Site Use Agreement ([//www.washington.edu/online/terms/](https://www.washington.edu/online/terms/))