Lecture Notes: Basics of Buffer Overflows

Related Challenges:

- stack0
- stack1
- stack2
- stack3

In this lecture, we are going to dive right into exploiting binaries, starting with the `stack0` challenge. We will begin with simple stack-based buffer overflows and work our way to the venerable stack-smashing example. We try to answer important questions, such as:

- What does it mean to exploit a binary?
- How are objects laid out in memory?
- How do we exploit basic buffer overflows to manipulate memory?
- What is a return address?
- How do we use the disassembly to determine the layout of variables on the stack?
- What are `setuid` binaries?
- What is privilege escalation?

Finding the Bug in `stack0`

The code box below shows the source code for the `stack0` challenge binary. Our objective is to figure out how to exploit this binary. What does it mean to “exploit” a binary? Well, loosely, exploiting a binary means manipulating how that binary executes to accomplish our goal. Here that goal is to read the contents of `flag.txt` . Presumably, we don’t have permission to read the flag file directly, but the challenge binary (on the course server) does.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  FILE *fp;

  modified = 0;
  gets(buffer);
  fp  = fopen("./flag.txt", "r");

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
      fgets(buffer, 64, (FILE*)fp);
      printf("flag: %s\n", buffer );
  } else {
      printf("Try again?\n");
  }
}
```

More specifically, a common goal for an attacker is gaining **arbitrary code execution** on a target machine. Arbitrary code execution means that the attacker can execute whatever code they want. Take a minute to consider what an attacker could do with such power and how that might be terrible for the machine's owner. Understanding how to exploit binaries to obtain arbitrary code execution is one of the primary goals of this course. Without understanding such attacks and their causes, we cannot design and implement appropriate defenses.

For the `stack0` challenge, the attacker's (i.e., our) goal is simpler than arbitrary code execution: we want to change the `modified` variable's value.

Broadly, our recipe for success calls for two ingredients. First, the program needs to contain a bug. Second, we need some way to supply input to the program to trigger that bug. Let's first focus on how to provide our malicious input.

Ingredient: Supplying malicious input

There are many different ways to supply input to a program. For many challenge binaries in this course, the most obvious way is to give input via `stdin` . We can also provide input in the form of command line arguments and environment variables. In real-world programs, external input might come from files on disk, network connections, external temperature sensors, etc. For `stack0` , `stdin` seems like a promising option due to the use of `gets` .

Ingredient: The bug

What is the bug in this program? Finding bugs often requires time and intuition—intuition, which comes from experience. A good place to start is by looking for some of the more common mistakes that programmers make. As programmers often make mistakes when manipulating arrays, let us first examine how `buffer` is used in the program.

The first time `buffer` is used in the program is as an argument to the function `gets()` . To understand how `gets()` works, we can start by reading the man page. It is a good idea to read the man page on a system that is similar to the target because different OSes may use slightly different versions of standard library functions. We can access the man page by using the command `man gets` in our local terminal.

From the man page, we learn that `gets()` reads a line from `stdin` into a buffer until either a terminating newline or EOF (end-of-file) character is encountered. After scrolling down to the `BUGS` section, we see the following line of text:

> Never use gets(). Because it is impossible to tell…how many characters gets() will read.

To understand what this warning means, we need to understand the memory layout of variables. You should have covered this already in an undergraduate course, but the following material will provide you with a quick review.

An array is essentially just a sequence of contiguous bytes thrown somewhere in the giant abstract block of bytes that we call **memory**. More generally, we can define a **buffer** as any contiguous region of memory associated with a variable. So an integer might be associated with a buffer of size 4 bytes.[1]

Using the information from the man page, we can conclude that `gets` in the `stack0` example will copy input from `stdin` into the `buffer` character array, aka the buffer named `buffer` . For example, if we run `stack0` and supply `aa` as input, then `gets()` will store three bytes into `buffer` : `a` , `a` , `\0` . But what happens if we supply an input that is larger than the size of `buffer` , e.g., a string of 100 `a` characters? The answer is that `gets()` will copy this entire string into memory (plus the null terminator). In other words, `gets()` will continue copying bytes even after it has blown past the end of `buffer` .

This type of bug is called a **buffer overflow**. Given that modern machines have a lot of memory, you might be wondering “what does it matter if a few bytes get overwritten?” Sometimes, it doesn’t matter! Even when it does matter, the symptoms may take a bit of time to manifest. For example, `stack0` will continue executing, at least for a little while, without any apparent issue. To understand the true danger of buffer overflows we have to consider what is *next to* `buffer` in memory. But first we need to review the concept of a virtual address space.

Review: Virtual Address Space

Let’s take a step back to review how processes use memory. These details should be familiar to you from other classes, e.g., memory is a major topic of CS3013: Operating Systems.

The operating system provides many wonderful abstractions to processes. One of the most important abstractions is the **virtual address space.** In particular, every **process** running on a machine (i.e., an executing program) has its own virtual address space. Due to this abstraction, from the perspective of the process, memory appears to be a contiguous array of bytes that the process can use however it wants. Each of those memory locations can store a byte and is associated with an address.

This memory abstraction is excellent for the program because everything is simple. For example, the process does not need to know the actual physical address of its memory. Further, virtual memory is the foundation of **process isolation**, i.e., each process is isolated from all other processes on the system. When someone says “memory” in the context of an executing process, they are typically referring to the abstracted illusion of memory provided by the OS.

By convention, the virtual address space is logically divided into different sections for different purposes. The layout usually includes regions for the stack, the heap, libraries, and program code. The **stack** is a per-thread data structure that keeps track of the thread’s execution state. It includes local variables, information about which functions have been called, etc. The stack for the main thread even includes the command line arguments to the program and environment variables. The **heap** is used for dynamically allocated memory. For example, calls to `malloc()` allocate space on the heap. The compiled program code is also stored memory; this area is often called the **text** section. The code for libraries is also included in memory, but that code is stored in a different location than the text section.

If we visualize this layout in a 32-bit virtual address space, we might see:

```
0xff ff ff ff
cmd env (set at process start, lives on main thread's stack)
stack (grows toward lower adddresses, bottom is fixed)
heap (managed by malloc)
libraries (code, dynamically linked)
text (code, read only)
0x00 00 00 00
```

Here I have represented memory with the 0x0 address at the bottom and highest address at the top. We can equivalently visualize memory in other orientations (up, down, and sideways). Different texts will use different orientations, but all of the representations are equivalent. It is also common to represent memory addresses using hex notation. In this course, we will be working with both 32-bit and 64-bit binaries. The former uses a 32-bit virtual address space, and each address can be represented in 4 bytes (or 8 hex digits) and the latter uses a 64-bit virtual address space which requires 8 bytes.

Side note: There are lots of ways we can determine if a binary targets a 32 or 64-bit architecture, but the easiest is probably the `file` command: `file stack0`.

Exploiting the Buffer Overflow in `stack0`

As discussed above, `gets()` will happily copy whatever input we provide to `stack0`, even if that input is larger than the buffer, allowing us to overwrite memory adjacent to `buffer`. So what is adjacent to `buffer`? In this challenge binary, `buffer` is a local variable and local variables are stored on the stack. Consequently, the memory adjacent to `buffer` includes other information stored on the stack. Hmmm, the `modified` variable—you know that variable we want to modify to solve the challenge—is also a local variable so it must also be on the stack. Let's see what happens when we try overflowing `buffer`.

```
python -c "print 'a'*100;" | ./stack0
```

The above command gives us the response that we want: "you have changed the 'modified' variable", so we can conclude that `modified` was indeed adjacent in memory to `buffer` and thus overwritten by the large input. However, there are some important caveats here. First, we assumed that the 'buffer' and 'modified' variables were adjacent to each other in memory because they are both local variables in the same function. Second, this input only worked because `modified` was stored at a higher address in memory than `buffer`, allowing the overflow to overwrite `modified`. Neither of these conditions are intrinsic to the C language, instead they arose because of decisions made by the compiler. For instance, our overflow input would not have given us the desired results if the compiler swapped the locations of buffer and modified in memory. To summarize visually:

Our assumption about the memory layout.

```
←────────────  Lower Addrs              Higher Addrs  ───────────────→

┌─────────────────────────────────────────────────────────────────────┐
│          |               buffer                      | modified     │
└─────────────────────────────────────────────────────────────────────┘
```

What could have been the case:

```
←────────────  Lower Addrs              Higher Addrs  ───────────────→

┌─────────────────────────────────────────────────────────────────────┐
│  modified |               buffer                      |              │
└─────────────────────────────────────────────────────────────────────┘
```

Or even:

```
←────────────  Lower Addrs              Higher Addrs  ───────────────→

┌─────────────────────────────────────────────────────────────────────┐
│          buffer                | ???     |    ?????     | modified   │
└─────────────────────────────────────────────────────────────────────┘
```

Further, as there is no way to update the value of `modified` in any normal program control flow, the compiler might even remove the `modified` variable and instead use hard-coded zeroes where needed. To prevent this, `stack0` includes the `volatile` keyword, forcing the compiler not to optimize out the `modified` variable.

This discussion leads us to an important point: **if we want to know what code actually executes, we need to look at the disassembly of the binary.** In other words, the compiler is going to transform the c code and we don't know what the actual executed code will look like unless we examine the binary.

Setuid Binaries

The above text explains the mechanics of how we, as the attacker, can use a buffer overflow to modify adjacent variables on the stack. But the explanation doesn't answer an important question: Why is it that we don't have permission to read "flag.txt" from the command line, but the exploited binary *can* read and print out "flag.txt"? The answer is that the binary runs with different permissions than the user.

Let's take a look at the `stack0` binary using the `file` utility: `file ./stack0`. The output includes a curious bit of text calling stack0 a "setuid" ELF executable. What does "setuid" mean here? To find out, we start the same way we always do on Linux: reading the MAN page.

The `setuid` man page gives us more information—specifically, about the C library function. We can see that **setuid** stands for set user identity. It allows a process to run as if it were started by a different user and, consequently, run with that user's permissions. Setuid binaries can even run as `root`. If an attacker can exploit a setuid binary running as root, he effectively has root privileges. Thus, setuid binaries are useful for achieving **privilege escalation**, i.e., increasing the level of privilege that an attacker has on the system.

So why does the `setuid` functionality exist at all? It is often used to allow an unprivileged user to access hardware features or to *temporarily* give a user elevated privileges. For example, `ping` and `sudo` are both setuid binaries. Fortunately, a malicious user with basic user permissions cannot merely write a program and use setuid to make that program run as root. The OS has mechanisms and policies in place to prevent that, as we can infer from looking at the errors section of the `setuid` man page.

Looking at the Disassembly

Let's use `gdb` to take a closer look at the instructions and memory that comprise `stack0`. Note: this disassembly is for a simplified version of `stack0` that does not read from `flag.txt`.

The first thing we are going to do is change how `gdb` displays assembly. Specifically, we are telling `gdb` to use Intel syntax rather than AT&T syntax. We do this purely based on personal preference.

```
set disassembly-flavor intel
```

It is a good idea to first review intel syntax and the basics of the x86 ISA (http://www.cs.virginia.edu/~evans/cs216/guides/x86.html), but here are a few reminders:

- the format is `instruction destination, source`
- esp is the stack pointer register in 32-bit x86, and it points to the top of the stack
- `mov DWORD PTR [esp+0x5c], 0x0` means move the 32-bit representation of 0x0 (DWORD) into the address `esp+0x5c`.
- `lea` : loads the effective address into the register rather than the content at that address.

Now let's take a look at main:

```
(gdb) disassemble *main
0x080483f4 <main+0>:  push   ebp
0x080483f5 <main+1>:  mov    ebp,esp
0x080483f7 <main+3>:  and    esp,0xfffffff0           //done for aligment
0x080483fa <main+6>:  sub    esp,0x60                 //making space
0x080483fd <main+9>:  mov    DWORD PTR [esp+0x5c],0x0 //initialize 'modified'
0x08048405 <main+17>: lea    eax,[esp+0x1c]           //load addr of buffer
0x08048409 <main+21>: mov    DWORD PTR [esp],eax      //set up the params
0x0804840c <main+24>: call   0x804830c <gets@plt>     //our call to gets
0x08048411 <main+29>: mov    eax,DWORD PTR [esp+0x5c]
0x08048415 <main+33>: test   eax,eax
0x08048417 <main+35>: je     0x8048427 <main+51>
0x08048419 <main+37>: mov    DWORD PTR [esp],0x8048500
0x08048420 <main+44>: call   0x804832c <puts@plt>
0x08048425 <main+49>: jmp    0x8048433 <main+63>
0x08048427 <main+51>: mov    DWORD PTR [esp],0x8048529
0x0804842e <main+58>: call   0x804832c <puts@plt>
0x08048433 <main+63>: leave
0x08048434 <main+64>: ret
```

The first column here shows the memory location of each instruction in main. These addresses all fall within the text section of the virtual address space. The second column shows the byte offset for the instruction. For example, `<main+64>` means that the instruction is 64 bytes from the start of main. The remaining columns show the assembly mnemonic and operands for each instruction.

Again, our goal is to figure out where `modified` and `buffer` are relative to each other. But first we need to figure out where these variables are relative to the stack pointer `esp`.

We can figure out the memory location of the `modified` variable by looking for the instructions that might correspond to the C code `modified = 0;` and `if(modified != 0)`. Fortunately, this code is pretty simple so our candidates are limited. Specifically, the former is the `mov DWORD PTR [esp+0x5c],0x0` instruction at address 0x080483fd and the latter is sequence of instructions `mov,test,je` starting at 0x08048411. By looking at the operands for those instructions we can conclude that `modified` is at `esp + 0x5c`

Similarly, we can find the location of `buffer` by looking for the call to `gets()` that uses `buffer` as the argument. Here we need to use our knowledge of the 32-bit x86 **calling convention**, which requires that function arguments are passed via the stack. In the above disassembly, we see a `call` instruction targeting `gets()` at `<main+24>`. We can conclude that `esp + 0x1c` is the start of `buffer` because that address is passed to `gets()` via the `eax` register getting pushed on the stack.

Now that we suspect that `modified` is at `esp+0x5c` and `buffer` is at `esp+0x1c`, we can calculate the relative positions of `modified` and `buffer` in memory. Some quick subtraction (0x5c - 0x1c) tells us that the start of buffer and the start of modified are only 64 bytes apart. To avoid overwriting other important stack values (more on this later), we need to modify our malicious input such that it only overflows into `modified` and not any further:

```
python -c "print 'a'*64;" | stack0
```

This input will overwrite the first byte of `modified` as the python code `print 'a'*64` will produce a 65 character string consisting of 64 `a` characters and a `0x0a` newline character. Let's use `gdb` to see the impact on a running process.

```
$ python -c "print 'a'*64" > input.txt

$ gdb stack0
disass main // view the disassembly for the main function
break *0x08048411 // put a breakpoint right after the call to gets()
r < input.txt // run the binary with the long input
x/s $esp+0x1c // to see the input string in stack memory
x/68bx $esp+0x1c // to see input string bytes in hex
//Warning, everything is flipped! higher addresses are lower
//The 0x0a000000 right after the 'aaaa' is the memory for `modified`
```

Introducing Return Addresses

Another thing we may see when exploiting the buffer overflow in `stack0` is a `segmentation fault` error. This means that our actions have caused the program to attempt to use memory in a way that isn't allowed. In this case, our input overwrote the `modified` variable and then just kept on overwriting other adjacent values in memory. One of those other values was the **return address**. The return address specifies the location in memory that the function should return to when done executing. In normal operation, this value should be an address in the calling function. However, our overflow modified this value, causing the process to attempt to return to (and execute code at) a location in memory that (probably) does not contain valid code.

To understand the concept of a return address, we first need to remember that memory is also used to store code. Though, the code that is executed isn't the C source code that we showed earlier. Instead, that code is a sequence of machine instructions. When a process executes, the CPU will iterate through this sequence of instructions and perform the specified actions. We often visualize these machine instructions as assembly language, even though assembly is still an abstraction.

The x86 architecture uses a special register, called the **instruction pointer**, to keep track of what instruction to execute next, i.e., it stores the memory address of the next instruction to be executed. When a function call occurs, the instruction pointer must change to point to the that function's first instruction. As most functions eventually return to the caller, the current value of the instruction pointer must first be saved so that the CPU can return to where it was before (i.e., the calling function). We call this saved value, the **return address**. Given that the instruction pointer register can only store a single address, we have to find some place to save the old pointer. The answer is to put it in memory, but where in memory? That's another job for the stack.

With our new knowledge of return addresses, we can now understand what is causing the segmentation fault. We overflow the buffer, which overwrites `modified` and eventually overwrites the saved instruction pointer (i.e., the return address). So when the process tries to return from the function, by loading the old instruction pointer from the stack into our IP register, the address is now just a bunch of 'aaaa' bytes. The address 'aaaa' (i.e., 0x61616161 in hexidecimal) probably does not contain any valid code so hardware notifies the OS of the problem and the latter kills the process.

We can avoid the segmentation fault in `stack0` by adjusting our exploit code to only overwrite the 'modified' variable, leaving the return address untouched. This changes is easy to make if we know exactly where `buffer` and `modified` are relative to each other in memory. Fortunately, we can figure out their relative positions by looking at the disassembly of `stack0` .

PIE Binaries

Some binaries are compiled to be position-independent, meaning that the code section can be positioned at an arbitrary memory location when the process is loaded. We refer to such binaries as position independent executables or PIE binaries.

PIE can complicate the generation of exploits because we, as the attacker, often need to know the location of particular functions (or code sequence) in memory. PIE makes such addresses a little harder to determine. PIE is also an essential enabler of defenses like ASLR—we will save the discussion of ASLR for a future lecture.

It is possible to check if a binary is compiled with PIE-support using the `checksec` utility, which comes preinstalled with the EpicTreasure docker image.

```
$ checksec ./stack2-64
[*] '/root/host-share/stack2-64'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Another clue that PIE is enabled is if the function addresses are at low addresses. For example, the following shows the `win()` function at address `0x7aa`, which far from where the function will be placed at runtime.

```
$ gdb ./stack2-64
+pwndbg> p win
$1 = {void ()} 0x7aa <win>
```

Fortunately, we can use GDB to determine where a function is loaded at runtime without much trouble by using GDB and breakpoints, as shown in the following example.

```
$ gdb ./stack2-64
+pwndbg> b main
Breakpoint 1 at 0x824: file stack2.c, line 24.
+pwndbg> r

...Omitted for clarity...

+pwndbg> p win
$2 = {void ()} 0x5555555547aa <w

```

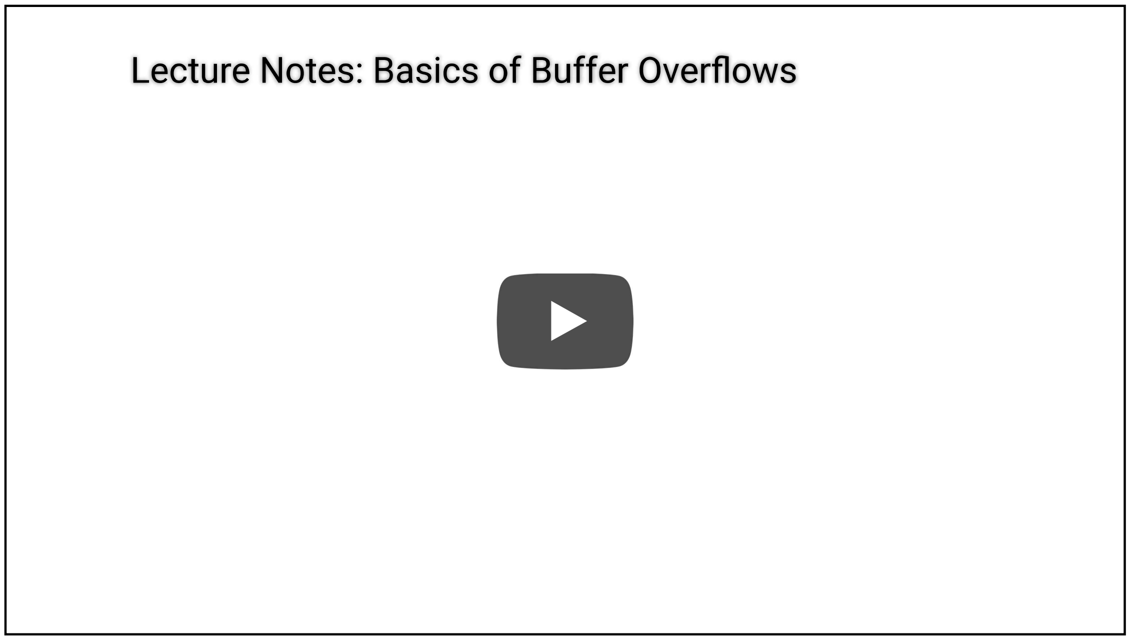
An astute reader might notice that the last three nibbles of the actual address, `0x7aa`, are the same as what we saw in our first attempt to find the address of `win()`. That is not a coincidence. Instead, the `0x7aa` is an offset from the start of the text section. We can find out where the text section starts (and the location and size of other regions) using `info proc map` in GDB.

Oh, by the way, here is the flag for the `Back to Basics` challenge: `DoTheRequiredReading`.

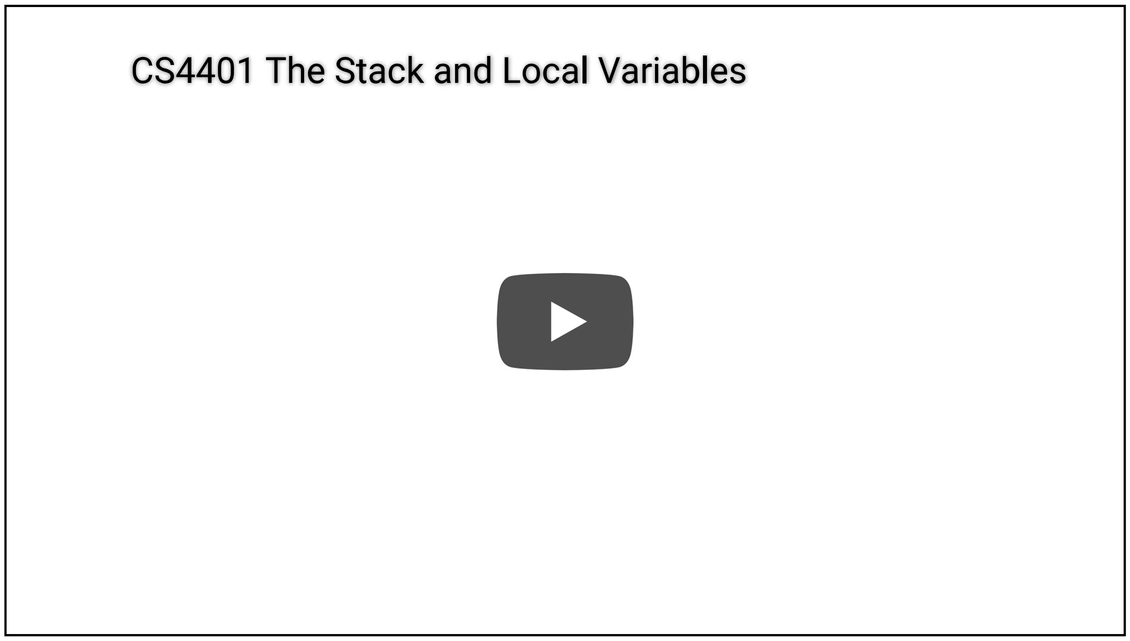
```
$ gdb ./stack2-64
+pwndbg> b main
Breakpoint 1 at 0x824: file stack2.c, line 24.
+pwndbg> r

...Omitted for clarity...

+pwndbg> info proc map
process 2155
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /root/host-share/stack2-64
...Omitted for clarity...
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
...Omitted for clarity...
```

Video

The video below and these written notes cover much of the same material, but they are not identical in content. It is worth taking a look at both.



The following video shows how to examine the stack and local variables for a process using GDB.



1. Some texts will define buffer slightly differently than the definition we use here by only considering variable-length data to have buffers. ↩