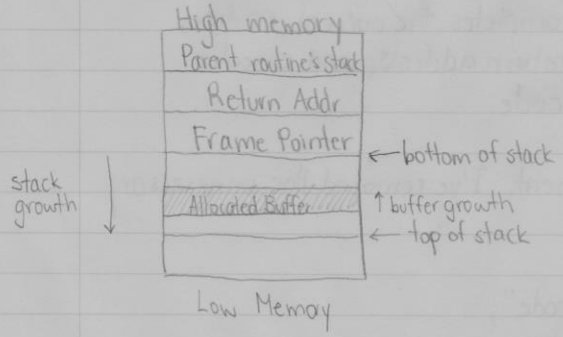
ECE458 Final Primer

Eliot Chan

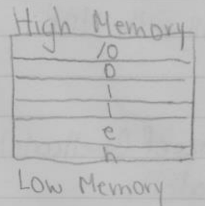
Stack Overflow / Stack Smashing

The core of stack smashing relies on providing input to a program that is beyond its expectation of the length of user input. In Dr. T's example, the memory looks like this, in the normal case.



The stack grows downward, and the buffer grows upward. Here, we've allocated a small buffer within the stack. Any user input will start at the lower memory address and be written upwards.

So for example, the input "Hello" would look like this:



So the question is: as the user's input grows in length, what happens?

First, the input extends past the end of its own buffer and into the rest of the stack, which may or may not have immediate consequences. Then, it'll overwrite the frame pointer, which will probably greatly mess up the execution of the program, as variables are referenced relative to the frame pointer.

The real security issues begin when we overwrite the return address - where the program is supposed to go after the current function completes.

Since the CPU will start executing instructions after it pops the return address out, the attack usually goes like this:

1) Insert malicious code into a buffer in memory
2) Give the program a huge input, overwriting the return address with the address of where the malicious code is located
3) The program completes the current routine, jumps to the return address, and executes the malicious code.

So let's go over the assignment. I've removed the unnecessary parts for brevity's sake.

exploit

```
char[] shellcode = "shellcode";
char [] bufferAddr = "address";
void main {
    Char buffer [517];
    memset (&buffer, 0x90, 517);  // fill with NOPs
    strcpy (buffer, buffAddr);

    memcpy (buffer + sizeof (buffer) - sizeof (shellcode),
            shellcode, sizeof (shellcode));
            // copy shell code to the end of the buffer
}
```

This is the exploit program. It'll write all 517 bytes to a file that will be read by a vulnerable program.

vulnerable program

```
int bof (char *str) {
     char buffer[24];
     strcpy (buffer, str);  // this is where the buffer overflow occurs
     return 1;
}
int main () {
     char str [517];
     File *badfile = fopen ("badfile", "r");
     fread (str, sizeof (char), 517, badfile);
     bof (str);
     return 1;
}
```

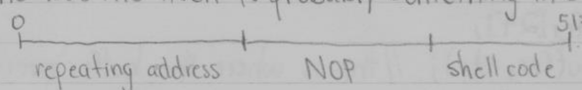
The problem in the vulnerable program is fairly obvious - we allocate a 24-character buffer, but copy 517 characters into it - an overflow of 493 characters.

So the question is, what do we need to put into it to overwrite the return address? We can find the address of our malicious code by subtracting the size of the shellcode from the end of the buffer (address of buffer + 517 - size of shellcode).

But what we don't know is where exactly the return address is. All we know is that it's somewhere past the 24th byte. So instead of trying to guess exactly where it might be, we're just going to spam the first few hundred bytes with the memory address.

So if the memory address of the shellcode is 0x3f402aff, we'd have \x3f \x40 \x2a \xff \x3f \x40 \x2a \xff, etc a few dozen times to pretty much ensure we overwrite the return address.

So the bad file itself is probably something like this



So how do we protect against stack smashing?

1) Checking user input, or using memory-safe copy functions

This one is pretty obvious, it prevents overflow of an allocated buffer.

2) Using the No-Execute bit

This bit marks certain portions of memory as non-executable. So if the malicious code is jumped to, it won't execute despite the stack smashing actually occurring.

3) Address Space Layout Randomization

Using ASLR jumbles up the positions of the stack, heap, libraries, etc. so we can no longer reliably determine where we've stored our malicious code, where the return address is located, etc.

4) Canaries

By dropping a known value between a given buffer and some control data, we can detect if the value is overwritten, thereby setting off an early warning that a buffer overflow attack is currently happening.

Setuid and Access-Based Attacks

Let's inspect the code Dr. T has provided to see if we can generate state machines from them

Setuid : I'll be using 3-tuples of (real, effective, eOld).

(real, eff, eOld)	setuid()	new result
(0, 0, 0)	0	(0, 0, 0)
(0, 0, 0)	1	(1, 1, 0)
(0, 0, 1)	0	(0, 0, 0)
(0, 0, 1)	1	(1, 1, 0)
(0, 1, 0)	0	(0, 0, 1)
(0, 1, 0)	1	(0, 1, 1)
(0, 1, 1)	0	(0, 0, 1)
(0, 1, 1)	1	(0, 1, 1)
(1, 0, 0)	0	(0, 0, 0)
(1, 0, 0)	1	(1, 0, 0)
(1, 0, 1)	0	(0, 0, 0)
(1, 0, 1)	1	(1, 0, 0)
(1, 1, 0)	0	(1, 1, 1)
(1, 1, 0)	1	(1, 1, 1)
(1, 1, 1)	0	(1, 1, 1)
(1, 1, 1)	1	(1, 1, 1)

We can see that since the previous value of eOld is never used, it's not actually necessary in the state machine

$$\begin{tikzcd}
(0,1) \arrow[r, "0"] \arrow[loop below, "1"] & (0,0) \arrow[loop above, "0"] \arrow[d, "1"] & (1,0) \arrow[l, "0"'] \arrow[loop below, "1"] \\
 & (1,1) &
\end{tikzcd}$$

As we can see, this setuid is unable to create the result (0,1) or (1,0) unless they started at those values.

Setresuid:

(real, eff)	setuid (r, e)	result
(0,0)	(-1,-1)	(0,0)
	(-1,0)	(0,0)
	(-1,1)	(0,1)
	(0,-1)	(0,0)
	(0,0)	(0,0)
	(0,1)	(0,1)
	(1,-1)	(1,0)
	(1,0)	(1,0)
	(1,1)	(1,1)
(0,1)	(-1,-1)	(0,1)
	(-1,0)	(0,0)
	(-1,1)	(0,1)
	(0,-1)	(0,1)
	(0,0)	(0,0)
	(0,1)	(0,1)
	(1,-1)	(1,1)
	(1,0)	(1,0)
	(1,1)	(1,1)
(1,0)	(-1,-1)	(1,0)
	(-1,0)	(1,0)
	(-1,1)	(1,1)
	(0,-1)	(0,0)
	(0,0)	(0,0)
	(0,1)	(0,1)
	(1,-1)	(1,1)
	(1,0)	(1,0)
	(1,1)	(1,1)

Continued on the next page

(real, eff)	Setresuid (r, e)	result
(1, 1)	(-1, -1)	(1, 1)
	(-1, 0)	(1, 1)
	(-1, 1)	(1, 1)
	(0, -1)	(1, 1)
	(0, 0)	(1, 1)
	(0, 1)	(1, 1)
	(1, -1)	(1, 1)
	(1, 0)	(1, 1)
	(1, 1)	(1, 1)

This yields the following state machine (transitions to self are omitted):

$$\begin{tikzcd}
(1,0) \arrow[dr, bend left, "{(0,-1)\ (0,0)}"] \arrow[ddr, "{(1,1)\ (-1,1)}"'] \arrow[rr, bend right=100, "{(0,1)}"'] & & (0,1) \arrow[dl, bend right, "{(-1,0)\ (0,0)}"'] \arrow[ddl, "{(1,1)\ (1,-1)}"] \arrow[ll, bend left=80, "{(1,0)}"] \\
& (0,0) \arrow[ul, "{(1,-1)\ (1,0)}"] \arrow[ur, "{(-1,1)\ (0,1)}"'] \arrow[d, "{(1,1)}"] & \\
& (1,1) &
\end{tikzcd}$$

So as long as the process is not unprivileged, it is able to make a transition to any other state. Once a process is unprivileged, though, it will stay there.

Whether this is an issue or not, it isn't immediately clear.

So what about access-based attacks in general?

One common vulnerability is the access-open race (sometimes called time-of-check to time-of-use). Let's say we have a check that looks like this:

```
if (r == 0 || e == 0)
      open (file)
```

This seems rather innocuous, but there's a vulnerability here. After we complete the access check, if an attacker can modify the 'file' we want to open, we will unintentionally open a file that the process may not have access to.

This is because the open command itself is run with root permissions. Generally, this is done as such:

Attacker Alice wants to open file B, but she only can open file A.

1) Alice runs program, saying she wants to open A
2) Access check is passed
3) Alice deletes A, and makes A a symlink to B
4) The program opens A, which actually opens B.

There's a few solutions to this.

1) Changing how open works.

By allowing a user to specify which id to use with the open command itself, it would prevent opening files not currently owned by the process

```
open ("fileName", id, "r")
```

2) Activating kernel protection for sticky symlinks

This prevents the usage of symlinks if the person trying to follow the symlink AND the directory owner don't match the symlink owner

So in Alice's case, even though she owns the symlink, she won't own the directory of B, so this wouldn't be allowed

3) Temporarily downgrade process to the real user's id, read the file, then upgrade.

4) If we assume an attacker only has some probability of executing the attack (as opposed to being able to do it every time), we can ask it to win the race $2k+1$ times (where k is an arbitrary strengthening parameter) instead.

Choosing k correctly can greatly reduce the possibility of Alice's attack succeeding

That's about it for setuid attacks.