CS 161 Computer Security
Fall 2008 Dawn Song

Notes 7

Common implementation flaws

We will discuss several classes of common implementation flaws.

1 Buffer overruns

C is essentially a portable kind of assembler: in many ways, the programmer is exposed to the bare machine. In particular, C does not provide any sort of automatic bounds-checking for array or pointer accesses. In the case of a *buffer overrun* vulnerability (sometimes also called a *buffer overflow*), out-of-bounds memory accesses are used to corrupt the intended behavior of the program and cause it to run amok.

Let us start with a simple example.

```
char buf[80];
void vulnerable() {
    gets(buf);
}
```

In this example, `gets()` reads as many bytes of input as are available on standard input, and stores them into `buf[]`. If the input contains more than 80 bytes of data, then `gets()` will write past the end of `buf`, overwriting some other part of memory. This is a bug. Obviously, this bug might cause the program to crash or core-dump if we are unlucky, but what might be less obvious is that the consequences can be far worse than that.

To illustrate some of the dangers, we modify the example slightly.

```
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

Imagine that elsewhere in the code is a login routine that sets the `authenticated` flag only if the user proves knowledge of a super-secret password, and other parts of the code test this flag to provide special access to such users. We can see the risk. An attacker who can control the input to this program can cause `buf` to be overrun, so that data is written after the end of `buf`. Assuming the compiler stores the `authenticated` variable in memory immediately after `buf`, then the `authenticated` flag will be overwritten by this data. Consequently, the attacker can arrange to make the `authenticated` flag become true by supplying, say, 81 bytes of input, where the 81st byte takes on any non-zero value. This would give the attacker special access even though the attacker doesn't know the secret password, a security breach.

We could conjecture a more serious version of this exploit, Suppose the code looked something like this:

```
char buf[80];
int (*fnptr)();
...
```

Here we have a function pointer `fnptr`, which we assume is invoked somewhere else in the program. Then the attacker could cause more serious harm: he could overwrite `fnptr` with any address of his choosing, and thereby cause program execution to be re-directed to any desired memory address. A crafty attacker might first arrange to introduce a malicious sequence of machine instructions somewhere in the machine's address space, and then use the buffer overrun to write into `fnptr` the address of the malicious code, so that when `fnptr` is next invoked, the flow of control is re-directed to the malicious code. Introduction of malicious code isn't very hard, if the attacker controls the inputs to the program: since those inputs are likely stored into buffers as they are read in, the attacker could send the malicious code as data in any part of the input, and then arrange for `fnptr` to point to the appropriate input buffer. This is a *malicious code injection* attack.

This demonstrates that, in some cases, an adversary may be able to take advantage of a buffer overrun bug to seize control of the program. This is very bad. For instance, consider a web server that receives requests from clients across the network and processes them. If the web server contains a buffer overrun in the code that processes such requests, a malicious client would be able to seize control of the web server process. If the web server is running as root, the attacker has gained access to a root account on the system, and the attacker can leave a backdoor to regain access later; the system has been "owned." It is no surprise that such buffer overrun vulnerabilities and malicious code injection attacks are a favorite method that worm writers like to use to spread infection.

This illustrates one way that buffer overrun bugs might be used to subvert system security. At this point, though, the skeptical might be inclined to suspect that the conditions required to exploit buffer overruns in this particular way are rare. This seems to be more or less true[1]. However, hackers have discovered much more effective methods of malicious code injection, which we shall illustrate next.

First, we have to back up and go over some background on how memory is layed out in a typical C program. Memory usually contains: (a) the text region, executable code of the program to be executed; (b) the heap, where dynamically allocated data is stored; (c) the stack, where local variables are stored. The heap grows and shrinks as objects are allocated and freed; the stack grows and shrinks as functions are called and return. To allow them to use memory as efficiently as possible, they are usually layed out like this:

text region	heap ···	stack
0x00..0		0xFF..F

Here the text regions starts at smaller-numbered memory addresses (e.g., 0x00..0), and the stack region ends at larger-numbered memory addresses (0xFF..F). When a function is called, a new stack frame is pushed onto the program stack. This stack frame contains space for all the local variables to be used by that function, and anything else the compiler wants to store assorted with this function execution. On Intel (x86) machines, pushing frames on the stack causes the stack to grow "down" (towards smaller memory addresses). Usually there is a special register, called the stack pointer (SP), that points to the beginning of the current stack frame. Thus, the stack extends from the address given in the SP until the end of memory.

It's also worth saying something about how the CPU executes instructions. There is a special register, called the instruction pointer (IP), that points to the next machine instruction to be executed. When executing

[1] However, it is worth noting that the famous Internet worm (the first of its kind) spread using several attacks, one of which used a buffer overrun to overwrite an authenticated flag in `in.fingerd`, the network finger daemon.

straight-line code, the machine reads the instruction pointed to by IP, executes that instruction, and then increments the IP. Jump and procedure-call instructions cause the IP to be updated differently. A jump instruction causes the IP to be updated to whatever value is specified by the jump instruction. A call instruction is special: it pushes the current value of IP onto the stack (this will be called the return address), and then branches to the beginning of the function to be called. Usually the compiler inserts a function prologue at the beginning of the function body, so it is the first thing to be executed when the function is called. The prologue pushes the current value of SP on the stack, and then allocates space on the stack for local variables by decrementing the SP by some appropriate amount. When the function returns, the old value of the SP is retrieved from the stack, then the return address is retrieved, the stack frame is popped from the stack, and execution continues starting from the return address.

After all that background, we're now ready to see how a *stack smashing* attack works. Suppose the code looks like this:

```
void vulnerable() {
    char buf[80];
    gets(buf);
}
```

When `vulnerable()` is called, a stack frame is pushed onto the stack. It will look something like this:

`buf`	saved SP	ret addr	caller's stack frame	···

Notice that if a too-long input is supplied, then the saved SP and subsequently the return address on the stack will be overwritten. The stack smashing attack now becomes apparent. First, the attacker stashes a malicious code sequence somewhere in the program's address space (e.g., using techniques previously mentioned). Next, the attacker provides a carefully-chosen 88-byte sequence, whose last four bytes are chosen to hold the address of this malicious code sequence. Of course, those last four bytes will overwrite the return address saved on the stack. When `vulnerable()` returns, the CPU will load the return address stored on the stack and transfer control to that address—thereby handing control over to the attacker's malicious code.

This barely begins to cover the full breadth of buffer overrun exploit techniques. The canonical reference on stack smashing exploits is the essay "Smashing the Stack for Fun and Profit", written by Aleph One in November, 1996. There are clever methods for constructing the malicious code sequence, for determining its address, and for making the exploit work even when you don't know the exact address where the malicious code is stored in the victim's address space. And that's just stack smashing. Since then, attackers have discovered a vast array of subtle and sneaky ways of exploiting buffer overrun bugs, even when the buffer is stored on the heap instead of on the stack, even when the program code only allows to overflow the buffer by one byte, even when the characters written to the buffer are limited (e.g., limited only to uppercase characters), and under all sorts of other restrictive conditions where one might initially think that the buffer overrun is harmless.

On paper, the subject of buffer overrun exploitation may appear mysterious, too tiring for anyone to bother with, or incredibly hard to exploit in the wild. In practice, it is none of the above. You will see in a future homework exercise that it's actually not all that hard to exploit such a bug. And worms exploit these bugs all the time; for instance, the Code Red II worm (mentioned in the introduction to have compromised a quarter million machines) exploited a buffer overrun in IIS (Microsoft's web server).

Historically, many security researchers have underestimated the opportunities for obscure and sophisticated attacks. It is an easy mistake to make. Over time, though, we have learned is that if your program has a buffer overrun bug, it is wisest to assume that the bug is exploitable and that an attacker could take control of the program. Buffer overruns are bad stuff—you don't want them in your programs.

2 Format string vulnerabilities

Here is another example of a C-related vulnerability:

```
void vulnerable() {
    char buf[80];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

Do you see the bug? The last line should have used `printf("%s", buf)`. Instead, it wrongly passed `buf` as the format string to `printf()`. Notice that if `buf` contains any `%` format-specifiers, `printf()` will look for arguments that aren't there, and may crash or core-dump the program when it follows an invalid pointer. But actually, it's worse than that. As you may have begun to suspect by now, any time that a program crashes or core-dumps, it is worth looking closely to see whether there is something more serious possible. Let's look.

First, we can see that it may be possible for an attacker to learn information about the contents of the function's stack frame. By specifying a string like `"%x:%x"`, an attacker who can see what is printed has the chance to view the first two words of stack memory.

In fact, we can refine this a bit. Suppose we supply a string like `"%x:%x:%s"`. What does this do? It prints out the first two words of stack memory, then treats the next word of stack memory as a memory address and prints out everything at that address until the first `'\0'` byte. Where does that last word of stack memory come from? Well, it comes from somewhere in the stack frame for `printf()`, or, if we supply enough `%x` specifiers to walk past the end of `printf()`'s stack frame, then it comes from somewhere in `vulnerable()`'s stack frame.

Ah-hah! Now we are on to something. Notice that `buf` is stored in `vulnerable()`'s stack frame, and the attacker can control the contents of `buf`, so this means the attacker can control part of the contents of `vulnerable()`'s stack frame—but that's exactly where the `%s` specifier is getting its memory address from. So, the attacker can store a memory address somewhere in `buf`, then arrange that when the `%s` specifier reads a word from the stack to get a memory address, it will receive the memory address he thoughtfully put there for it. The exploit will involve supplying a string like `"\x04\x03\x02\x01:%x:%x:%x:%x:%s"`. If the attacker has arranged to have just the right number of `%x` specifiers, then the address that is read will be read from the first word of `buf`, which he has arranged to contain the value `0x01020304` (it looks like it has been reversed, but that is because values are stored in little-endian format on x86 machines). We can see that in this way, the attacker can pick any desired memory address, and read out the contents of memory starting at that address. Thus, an attacker can exploit a format string vulnerability to learn any secrets stored in the victim's address space—cryptographic keys, passwords, they're all potential targets.

It gets worse than that. Thanks to an obscure format specifier (`%n`), it is possible for an attacker to write any desired value to any desired address in the victim's memory, if the victim has a format string bug. I won't bother you with the details of how it is done; suffice it to say that it is possible. This is now enough to allow attackers to mount malicious code injection attacks. For instance, the attacker can introduce a malicious code sequence anywhere into the victim's memory, then use a format string bug to overwrite a return address on the stack (or a function pointer) so that it now points to the beginning of the malicious code.

Consequently, any program that contains a format string bug can usually be exploited by the attacker to take control of the victim program and all privileges it has been granted on the target system. Format string bugs are, like buffer overruns, nasty business.

3 Integer overflow and implicit cast vulnerabilities

What's wrong with this code?

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

Here's a hint. I'll show you the prototype for `memcpy()`:

```
void *memcpy(void *dest, const void *src, size_t n);
```

If that isn't enough, perhaps I should show the definition of `size_t`:

```
typedef unsigned int size_t;
```

Do you see it now? Elementary, my dear Watson! If the attacker provides a negative value for `len`, then the `if` statement won't notice anything wrong, and we'll execute `memcpy()` with a negative third argument—but then the third argument will be implicitly cast to an `unsigned int`, whence it becomes a very large positive integer. This means that the `memcpy()` copies a huge amount of memory into `buf`, far more than there is space for, a buffer overrun.

This is sometimes known as a signed/unsigned or an implicit casting bug. This kind of bug is nasty, because it can be hard to spot unless you know the language and libraries well. The C compiler doesn't usually give any warnings when there is a type mismatch between `signed int` and `unsigned int`; it silently inserts an implicit cast, with the programmer none the wiser.

Here is another example. What's wrong with this code?

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
...
```

Looks ok from a buffer overrun standpoint, since we've allocated more than enough space to hold what we read in (with 5 bytes to spare). No signed/unsigned problems here, since every integer in sight is unsigned.

If you look more closely, though, you will see that the expression `len+5` can overflow if `len` is large enough. For instance, if `len = 0xFFFFFFFF`, then the value of `len+5` is `4` on most platforms. In other words, we might allocate a 4-byte buffer and then proceed to read a heckuva lot more than 4 bytes into the buffer. That's a buffer overrun for you right there.

This illustrates that you've got to know the semantics of your programming language very well, because there are usually pitfalls for the unwary.

4 TOCTTOU vulnerabilities

Here's one for you that isn't language-specific.

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only allowed to regular files; nice try!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

This code is trying to open a file, but only if it is a regular file (e.g., not a symlink, not a directory, not a special device). On Unix, the `stat()` call is used to extract meta-data about the file, including whether it is a regular file or not. Then, the `open()` call is used to open the file.

The flaw in the above code is that it assumes the state of the filesystem will remain unchanged between the `stat()` and the `open()`. However, this assumption is not justified. In fact, an attacker might arrange to change the file referred to by `path` in between the `stat()` and `open()`. If `path` refers to a regular file when the `stat()` is executed, but refers to some other kind of file when the `open()` is executed, this bypasses the check in the code. If that check was there for a security reason, the attacker may be able to subvert system security.

This is known as a time-of-check to time-of-use (TOCTTOU) vulnerability, because the meaning of `path` changed from the time when it is checked (the `stat()`) and the time when it is used (the `open()`). In Unix, this often comes up with filesystem calls, because system calls are not atomic and the filesystem is where most long-lived state is stored. However, this is not specific to files. In general, TOCTTOU vulnerabilities can arise anywhere that there is mutable state that is shared between two or more entities. For instance, multi-threaded Java servlets and applications are at risk for this kind of flaw.

5 Many more

This is not, by any means, intended to be a complete list of implementation flaws. We've only scratched the surface. However, hopefully this illustrates some of the most prevalent examples and shows what can go wrong if implementations contain defects. If it makes you just a bit more cautious about how you write code, good! In future lectures, we will discuss what you can do to prevent (or reduce the likelihood) of these kinds of flaws, and to improve the odds of surviving any flaws that do creep in.