CS 334 Computer Security **Fall 2014**

Common Implementation Flaws — Memory Safety

Prof. Szajda

Thanks to Vern Paxson and Dave Wagner of UC-Berkeley for providing this handout.

In the next few lectures we will be looking at software security problems associated with software implementation. Even with a perfect design, perfect specification, and perfect algorithms, you may still have implementation vulnerabilities. In fact, after configuration errors, implementation errors are probably the largest single class of security errors exploited in practice. The purpose of the next few lectures is to teach you about software security, and in particular about the kinds of implementation errors that can lead to serious vulnerabilities.

We will start by showing you some of the most common implementation flaws. Because many security-critical applications have been written in C, and because C has peculiar pitfalls of its own, many of these examples will be C-specific. Don't let this fool you into believing that other languages are safe. Some, like Java, take pains to eliminate certain types of implementation flaws. No language, however, is completely safe from these errors. In addition, implementation flaws can occur at all levels: in improper use of the programming language, the libraries, the operating system, or in the application logic. Since by far the most common class of implementation flaw is the buffer overrun, we will start there.

1 Buffer Overruns

A *buffer overflow* (also called *buffer overrun*) vulnerability is present when code has been written such that it is possible for the user to place more data into memory (typically a variable) than the amount of space that has been allocated for that data by the programmer. In effect, the programmer has neglected to check that what is being received fits into the space in which it is to be placed. The result is that memory adjacent to the desired storage space is overwritten. Buffer overflow vulnerabilities are a particular risk in C, since C does not, by default, provide bounds checks on many operations that write to memory. Since C is an especially widely used systems programming language, you might not be surprised to hear that buffer overflows are one of the most pervasive kind of implementation flaws around.

As a low-level language, we can think of C as a portable assembly language. The programmer is exposed to the bare machine (which is one reason that C is such a popular systems language). As mentioned above, a particular weakness is the absence of automatic bounds-checking, in particular for array or pointer access. If the programmer fails to perform adequate bounds checks, the user can trigger an out-of-bounds memory access that writes beyond the bounds of some memory region. Attackers can use these out-of-bounds memory accesses to corrupt the programs intended behavior.

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. 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 the password. Unfortunately, the `authenticated` flag is stored in memory right after `buf`. If the attacker can write 81 bytes of data to `buf` (with the 81st byte set to a non-zero value), then this will set the `authenticated` flag to true, and the attacker will gain access. The program above allows that to happen, because `gets` does no bounds-checking: it will write as much data to `buf` as is supplied to it. In other words, the code above is vulnerable: an attacker who can control the input to the program, can bypass the password checks.

Now consider another variation:

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

The function pointer `fnptr` is invoked elsewhere in the program (not shown). This enables a more serious attack: the attacker can overwrite `fnptr` with any address of her choosing, and redirecting program execution to some other memory location. A crafty attacker could supply an input that consists of malicious machine instructions, followed by a few bytes that overwrite `fnptr` with some address A. When `fnptr` is next invoked, the flow of control is redirected to address A. Notice that in this attack, the attacker can choose the address A however she likes—so, for instance, she can choose to overwrite `fnptr` with an address where the malicious machine instructions will be stored (e.g., the address `&buf[0]`). This is a *malicious code injection* attack. Of course, many variations on this attack are possible: for instance, the attacker could arrange to store the malicious code anywhere else (e.g., in some other input buffer), rather than in `buf`, and redirect execution to that other location.

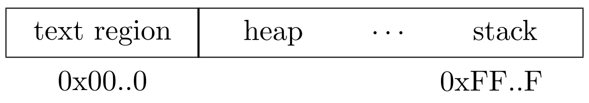
Malicious code injection attacks allow an attacker to sieze control of the program. At the conclusion of the attack, the program is still running, but now it is executing code chosen by the attacker, rather than the original software. For instance, consider a web server that receives requests from clients 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, once the attacker seizes control, the attacker can do anything that root can do; for instance, the attacker can leave a backdoor that allows her to log in as root later. At that point, the system has been "owned."

Buffer overflow vulnerabilities and malicious code injection are a favorite method used by worm writers and attackers. A pure example such as we showed above is relatively rare. However, it does occur: for instance, it formed the vectors used by the first major Internet worm (the Morris worm). Morris took advantage of a buffer overflow in `in.fingerd` (the network finger daemon) to overwrite the filename of a command executed by `in.fingerd`, similar to the example above involving an overwrite of an "authenticated" flag. But pure attacks, as illustrated above, are only possible when the code satisfies certain special conditions: the buffer that can be overflowed must be followed in memory by some security-critical data (e.g., a function pointer, a flag that has a critical influence on the subsequent flow of execution of the program). Because these conditions occur only rarely in practice, attackers have developed more effective methods of malicious code injection.

2 Stack smashing

One powerful method for exploiting buffer overrun vulnerabilities takes advantage of the way local variables are layed out on the stack.

We need to review some background material first. Let's recall C's memory layout:



The text region contains the executable code of the program. The heap stores dynamically allocated data (and grows and shrinks as objects are allocated and freed). Local variables are stored and other information associated with each function call is stored on the stack (which grows and shrinks with function calls and returns). In the picture above, the text region starts at smaller-numbered memory addresses (e.g., 0x00..0), and the stack region ends at larger-numbered memory addresses (0xFF..F).

Function calls push new stack frames onto the stack. A stack frame includes space for all the local variables used by that function, and other book-keeping information used by the compiler for this function invocation. On Intel (x86) machines, the stack grows down. This means that the stack grows towards smaller memory addresses. 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 the memory allocated for the process. Pushing a new frame on the stack involves subtracting the length of that frame from SP.

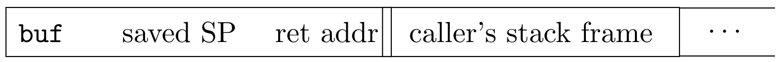
Intel (x86) machines have another special register, called the instruction pointer (IP), that points to the next machine instruction to execute. For most machine instructions, the machine reads the instruction pointed to by IP, executes that instruction, and then increments the IP. Function calls cause the IP to be updated differently: the current value of IP is pushed onto the stack (this will be called the return address), and the program then jumps to the beginning of the function to be called. The compiler inserts a function prologue—some automatically generated code that performs the above operations—into each function, so it is the first thing to be executed when the function is called. The prologue pushes the current value of SP onto the stack and allocates stack space for local variables by decrementing the SP by some appropriate amount.

When the function returns, the old SP and return address are retrieved from the stack, and the stack frame is popped from the stack (by restoring the old SP value). Execution continues 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. The stack will look something like this:



If the input is too long, the code will write past the end of `buf` and the saved SP and return address will be overwritten. This is a *stack smashing* attack.

Stack smashing can be used for malicious code injection. First, the attacker arranges to infiltrate a malicious code sequence somewhere in the program's address space, at a known address (perhaps using techniques previously mentioned). Next, the attacker provides a carefully-chosen 88-byte input, where the last four bytes hold the address of the malicious code[1]. The `gets()` call will overwrite the return address on the stack with the last 4 bytes of the input—in other words, with the address of the malicious code. When `vulnerable()` returns, the CPU will retrieve the return address stored on the stack and transfer control to that address, handing control over to the attacker's malicious code.

The discussion above has barely scratched the surface of techniques for exploiting buffer overrun bugs. Stack smashing was first well-documented in 1996 (see "Smashing the Stack for Fun and Profit" Aleph One), though the general idea of overflowing a buffer to control execution has been known since at least 1972. Modern methods are considerably more sophisticated and powerful. These attacks may seem esoteric, but attackers have become highly skilled at exploiting them. Indeed, you can find tutorials on the web explaining how to deal with complications such as:

- The malicious code is stored at an unknown location.
- There is no way to introduce malicious code into the program's address space.
- The buffer is stored on the heap instead of on the stack.

[1]In this example, I am assuming a 32-bit architecture, so that SP and the return address are 4 bytes long. On a 64-bit architecture, SP and the return address would be 8 bytes long, and the attacker would need to provide a 96-byte input whose last 8 bytes were chosen carefully to contain the address of the malicious code.

- The attack can only overflow the buffer by a single byte[2].
- The characters that can be written to the buffer are limited (e.g., to only lowercase letters)[3].

Buffer overrun attacks may appear mysterious or complex or hard to exploit, but in reality, they are none of the above. Attackers exploit these bugs all the time. For example, the Code Red II worm compromised 250K machines by exploiting a buffer overflow bug in the IIS web server. In the past, many security researcher have underestimated the opportunities for obscure and sophisticated attacks, only to later discover that the ability of attacker to find clever ways to exploit these bugs exceeded their imaginations. Attacks once thought to be esoteric to worry about are now considered easy and routinely mounted by attackers. The bottom line is this: If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.

How do you avoid buffer overflows in your code? One way is to check that there is sufficient space for what you will write before performing the write.

3 Format String Vulnerabilities

Let's look at another type of 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 be `printf("%s",buf)`. Instead, `buf` was wrongly passed 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

[2]In one of the most impressive instances of this, the Apache webserver at one point had an off-by-one bug that allowed an attacker to zero out the next byte immediately after the end of the buffer. The Apache developers downplayed the impact of this bug and assumed it was harmless—until someone came up with a clever and sneaky way to mount a malicious code injection attack, using only the ability to write a single zero byte one byte past the end of the buffer.

[3]Imagine writing a malicious sequence of instructions where every byte in the machine code has to be in the range 0x61 to 0x7A ('a' to 'z'). Yes, its been done.

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 `"x04x03x02x01:%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.

The bottom line: if your program has a format string bug, assume that the attacker can learn all secrets stored in memory, and assume that the attacker can take control of your program.

4 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:  bad dog, no cookie for you!");
          return;
     }
     memcpy(buf, p, len);
}
```

Here's a hint. The prototype for `memcpy()` is:

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

And the definition of `size_t` is:

```
typedef unsigned int size_t;
```

Do you see the bug now? If the attacker provides a negative value for `len`, the `if` statement won't notice anything wrong, and `memcpy()` will be executed with a negative third argument. C will cast this negative value to an `unsigned int` and it will become a very large positive integer. Thus `memcpy()` will copy a huge amount of memory into `buf`, overflowing the buffer.

Note that the C compiler won't warn about the type mismatch between `signed int` and `unsigned int`; it silently inserts an implicit cast. This kind of bug can be hard to spot. The above example is particularly nasty, because on the surface it appears that the programmer has applied the correct bounds checks, but they are flawed.

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

```
void vulnerable() {
     size_t len;
     char *buf;

     len = read_int_from_network();
     buf = malloc(len+5);
     read(fd, buf, len);
     ...
}
```

This code seems to avoid buffer overflow problems, (indeed, it allocates 5 more bytes than necessary). And there are no signed/unsigned problems here, since every integer in sight is unsigned. But, there is a subtle problem: len+5 can wrap around if len is too large. For instance, if len = 0xFFFFFFFF, then the value of len+5 is 4 (on 32-bit platforms). In this case, the code allocates a 4-byte buffer and then writes a lot more than 4 bytes into it: a classic buffer overflow. You have to know the semantics of your programming language very well to avoid all the pitfalls.

5 Memory Safety

Buffer overflow, format string, and the other examples above are examples of *memory safety* bugs: cases where an attacker can read or write beyond the valid range of memory regions. Other examples of memory safety violations include using a dangling pointer (a pointer into a memory region that has been freed and is no longer valid) and double-free bugs (where a dynamically allocated object is explicitly freed multiple times). C and C++ rely upon the programmer to preserve memory safety, but bugs in the code can lead to violations of memory safety. History has taught us that memory safety violations often enable malicious code injection and other kinds of attacks.

Some modern languages are designed to be intrinsically memory-safe, no matter what the programmer does. Java is one example. Thus, memory-safe languages eliminate the opportunity for one kind of programming mistake that has been known to cause serious security problems.

We've only scratched the surface of implementation vulnerabilities. If this makes you a bit more cautious when you write code, then good! In future lectures we'll discuss how to prevent (or reduce the likelihood) of these types of flaws and to improve the odds of surviving any flaws that do creep in.