Software Security

6.1600 Course Staff

Fall 2026

Software vulnerabilities are at the root of some of the most serious security failures in real-world software. A huge number of security issues come from software-implementation bugs. A rule of thumb to keep in mind is that, for reasonably carefully-written code, there will be around one bug for every 1000 lines of code. Many of these bugs may seem minor, but surprisingly, almost any kind of bug can be used in a security exploit and lead to compromise—even bugs in code that may not seem security-critical. One principle to remember is then:

> "Any bug can be a security bug."

Protecting the security of computer systems thus requires eliminating software bugs.

Many bugs can lead to security exploits, but some of the most common types of exploited bugs include memory corruption bugs (buffer overrun, use after free, etc.), encoding and decoding errors, cryptographic implementation bugs, race conditions and resource-consumption bugs.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

In terms of practical consequences on deployed computer systems, software issues are perhaps second only to phishing attacks.

A common strategy to exploit seemingly benign software bugs is to string together a number of benign bugs into an exploit that does something very bad from a security perspective.

1 *Memory Corruption*

Memory-corruption bugs show up in languages, such as C and C++, that do not guarantee any sort of memory- or type-safety properties. The two most common types of memory-corruption bugs are: (1) buffer overflows and (2) use-after-free bugs.

1.1 *Buffer overflow*

As an example of a memory-corruption bug, consider the following C code:

```
void f() {
        char buf[128];
        gets(buf); // write bytes from stdin starting
                   // at &buf[0], followed by a '\0'
}
```

This code allocates an array of 128 bytes and then uses `gets` to read a string from standard input into the buffer. The `gets` routine will read the input until it reaches the end of the string (indicated in C with a zero byte), then it will write the corresponding string

into the buffer. Arrays in C have no length information attached to them, so the `gets` code will happily accept an input string of any length—possibly much larger than 127 bytes.

So, what happens if the input is longer than 128 bytes? The `gets` function simply writes until it finds a NULL character (`'\0'`) in the input. So if the input string is too long, `gets` will simply write past the end of the memory allocated for `buf`. If an attacker gives an input that is longer than 128 bytes, the attacker will be able to write bytes of its choice into the memory after `buf`.

In C, the `buf` array will be allocated on the call stack, which will be laid out in memory like this:

```
... rest of stack ...
---------------
return address of f's caller
---------------
buf[127]
...
buf[2]
buf[1]
buf[0]
---------------
```

Once the `f` function ends execution, the program will jump to the return address sitting at the end of the buffer. So if an attacker can write data past the end of the `buf` array, the attacker can overwrite the return address of `f` on the stack and the attacker can cause the program to jump to and begin executing code at an arbitrary location in memory.

You might think that this problem is caused in part by having the stack in C grow down, so that running off the end of the buffer can cause the attacker to overwrite the return address. It turns out that similar attacks are possible even on architectures in which the stack grows up. The fundamental problem is that the attacker can scribble over data in the stack.

One simple mitigation is to modify the compiler to refuse to compile programs that use routines such as `gets`. We discuss other mitigations below.

Controlling the return address. What can an attacker actually do once it is able to overwrite the return address? A common strategy is to place its own machine code—often called *shellcode*, because it frequently just spawns a command shell—inside the very bytes that it writes into `buf`, and then to overwrite the return address of `f` with the address of `buf` itself. When `f` returns, the program counter jumps into the buffer and the attacker's code begins executing with all of the privileges of the vulnerable program.

It is worth looking more carefully at *which* return address the attacker overwrites. When `f` calls `gets(buf)`, there are two stack frames in play at once: the frame for `f`, which contains `buf` and the return address back into `f`'s caller, and the frame for `gets`, which

contains the return address that `gets` uses to return back into `f`. On a typical architecture, where the stack grows downward, `gets` fills `buf` in the direction of increasing addresses, so an overflow runs past the end of `buf` and over `f`'s saved frame pointer and the return address of `f`'s caller:

```
        ... rest of stack ...
        ---------------------------
        return address (f's caller)    ^
        ---------------------------    |
        saved frame pointer            |  overflow
        ---------------------------    |  grows
 buf -> buf[127]                       |  this way
        ...                            |
        buf[0]                         |
        ---------------------------    <- gets's frame
        return address (back into f)
        ---------------------------
        ... gets's locals ...
        ---------------------------
        (stack grows down)
```

Having the stack grow downward is not what makes the attack possible. Even on an architecture whose stack grows *upward*, an overflow of `buf` would instead run into `gets`'s frame and overwrite the return address that `gets` uses to return into `f`. Either way, the attacker ends up controlling some return address, and therefore the program counter. The fundamental problem is simply that the attacker can scribble over control data sitting on the stack right next to the buffer.

To avoid this kind of *buffer overflow* bug, we need to somehow ensure that `gets` only writes within the bounds of the buffer.

Mitigation: Runtime checks Modern compilers can try to check in real time whether a memory access goes past the end of the buffer and will crash the program if so. These defenses are imperfect but prevent the most naive type of bug.

Mitigation: Bounds Checking Another mitigation is to ensure that the input is not too large before reading it in. To do this, we can insert a check before writing.

Consider the following slightly more complex code, which receives n records that are each 16 bytes long and writes them into the buffer:

```
void f() {
        char buf[256];
```

```
        uint32_t n = get_input();
        for (uint32_t i=0; i < n; i++) {
                // read record i into
                // buf[i*16] .. buf[i*16+15]
        }
}
```

This code will write beyond the end of the buffer if `n*16` is greater than `256`. We then may consider adding a check like the following:

```
#define sz 256

void f() {
        char buf[sz];
        uint32_t n = get_input();
        if (n * 16 > sz) {
                // input too long!
                return;
        }
        for (uint32_t i=0; i < n; i++) {
                // read record i into buf[i*16] .. buf[i
                    *16+15]
        }
}
```

However, consider an adversary that inputs data such that `n = 2^30`. If we were computing on paper, our check would work just fine: $2^{30} \cdot 16 = 2^{34}$, which is certainly greater than `sz`. However, `uint32_t` is a 32-bit value and 2^{34} will not fit into the 32 bits allocated for that integer—the computation will *overflow*. It turns out that if you try to compute `n*16` in C when `n` is 2^{30}, the answer is zero! Thus, our check will pass but our code will still write beyond the end of the buffer.

To prevent this type of overflow, the program can explicitly check for overflow—it's tricky to do, but important when accepting user-provided input.

Mitigation: Stack canaries Rather than trying to prevent an overflow from happening in the first place, we can instead try to *detect* it before the corrupted return address is ever used. This is the idea behind defenses such as StackGuard and gcc's stack-smashing protector. When a function is entered, the compiler-generated code pushes a known value—called a *canary*—onto the stack, positioned between the local buffers and the saved return address. Just before the function returns, the code pops the canary and checks that it still holds its original value; if it has changed, the program concludes that the stack

has been smashed and aborts instead of returning.

```
---------------------------
return address
---------------------------
saved frame pointer
---------------------------
CANARY                      ^ overflow must
---------------------------   | overwrite the
buf[127]                    | canary before it
...                         | can reach the
buf[0]                      | return address
---------------------------
```

Because an overflow that reaches the return address must first write over the canary, the check will almost always fail and stop the attack. For this to work, the attacker must not be able to guess the canary value, so it is typically a random number chosen when the program starts and kept secret. Canaries are not a complete defense—an attacker who can read or guess the canary value, or who can corrupt the return address without writing through the canary, may still succeed—but they raise the bar considerably.

1.2 Use after free

Another common type of bug is a use-after-free bug, in which a programmer frees a chunk of heap memory and then reads or writes it after freeing it.

In C this happens when a programmer uses `malloc` to allocate some memory, then calls `free` to free it, and then accesses it. An example piece of code with this bug is here:

```
void f() {
  char *req = malloc(1024);
  int err = read(0, req, 1024);
  if (err) free(req);
                          // ***
  if (err) printf("Error␣%d:␣%s\n", err, req);
}
```

If some other thread in the program calls `malloc` when the code is at point `***`, the other thread of execution may use the array `req` for something else. Then the `printf` line could print out some other contents of memory—possibly exposing cryptographic keys or other sensitive data.

These bugs are very difficult to track down since it is difficult for a compiler to figure out which memory a piece of code should or

should not have access to. One way to defend against these bugs is to use a programming language, such as Rust, that explicitly associates a "lifetime" with each piece of memory and can prevent code from accessing free'd memory.

Preventing use-after-free bugs. Use-after-free bugs are, today, among the most prevalent memory-safety errors in deployed software. Depending on what the freed memory is reused for, they can lead either to the leakage of sensitive data, as in the example above, or to the corruption of data the program relies on—for example, overwriting a function pointer that is later called. Unlike buffer overflows, these bugs are hard to stop with simple range or bounds checks: the access itself is in bounds, and the only thing wrong with it is its *timing* relative to the `free`. What we really need are *lifetime* checks that track when a piece of memory is still valid to use.

These lifetime checks can be enforced in one of two broad ways. The first is to check lifetimes at compile time through the language's type system, as Rust does: the compiler refuses to compile a program that could use memory after it has been freed, so the bug can never arise at runtime. The second is to add "band-aid" checks at runtime that attempt to catch accesses to freed memory. These runtime checks are tricky to get right, however—once memory has been freed and then reallocated for some other purpose, a runtime check can no longer easily tell that accessing it is invalid.

Memory-safe languages. Taking a step back, an appealing way to avoid all of these memory-corruption bugs is to write code in a language that is memory-safe in the first place. This helps, but it is not a complete escape. Suppose you write all of your code in Python: the Python runtime itself is written in C, many of the library modules you import are written in C, and the underlying operating-system kernel is written in C as well. A memory-corruption bug anywhere in that stack can still compromise your program. The more hopeful development is the emergence of newer languages such as Rust, which aim to provide strong memory-safety guarantees while remaining suitable for low-level systems code, making it much harder to write the kinds of mistakes that lead to memory corruption in the first place.

2 *Encoding Bugs*

Another common source of security bugs comes from encoding and decoding data from and to language data structures.

2.1 SQL Injection

Most web sites that we interact with consist of some application code—for example, a Flask app—that communicates with a database via SQL queries. For example, a phone-number-to-name lookup site would likely use SQL queries that look like

```
'SELECT name FROM users WHERE phone = "6172536005"'.
```

When accepting a phone number, stored in variable `phone`, from a user, the same query might look like:

```
/* WRONG!!! */
'SELECT name FROM users WHERE phone = "' + phone + '"'.
```

The problem with using string interpolation is that an adversarial user can supply a phone number like

```
123"; DROP TABLE users; "
```

The SQL engine will then receive the query:

```
'SELECT name FROM users WHERE phone = "123"; DROP TABLE
users;',
```

which will have the effect of deleting the `users` table.

The principled way to solve this problem is to have a strategy for *unambiguously encoding* data. In SQL, if you have a quote character `"` in a data string, the programmer writes it as `\"`. This is called "escaping" a string. A SQL library can automatically escape characters such as quotes, but escaping is not as easy as replacing each quote with its escaped equivalent—you need to worry about escaping the escape character "`\\`" and all sorts of other subtleties.

Modern libraries for interacting with databases perform escaping automatically to avoid these "SQL injection" attacks.

2.2 Cross-Site Scripting

Another common behavior is to take input from the user via a form, save it to the database, and later render that input to another user as HTML. For example, a social media site will have each user enter their name when they sign up, and may use some code like the following to render another user's list of friends:

```
def render_friends(friends: List[str]):
        print("<h3>Friends</h3>")
        print("<ul>")
        for name in friends:
                print("<li>" + name + "</li>")
        print("</ul>")
```

If a friend's name is something expected, like "Alice" or "Bob", this works fine. However, what if a friend sets their name to something like `<script>send_to_adversary(document.cookie)</script>`? Now, the rendered HTML will look something like the following:

```
<h3>Friends</h3>
<ul>
        <li>Alice</li>
        <li>Bob</li>
        <li><script>
                send_to_adversary(document.cookie)
        </script></li>
</ul>
```

This `script` tag runs the contained Javascript code in the viewer's browser. Anyone who views a friends list with this adversary in it, then, will have their authentication cookie sent to the adversary, potentially allowing the adversary to log in as that user.

The core of this issue is similar to the SQL injection attack: an attacker is able to insert code that the victim's browser will run. To prevent this, the solution is again escaping: we typically replace the angle brackets (< and >) used to denote HTML tags with the sequences `<` and `>`. Now that `&` has a special meaning, we also must escape it as `&`. Modern web frameworks have "templating" systems that automate this escaping process.

2.3 Decoding: Python Pickling

Decoding untrusted data can be just as dangerous as encoding it. Python's `pickle` module, for example, lets a program serialize almost any object into a sequence of bytes and later reconstruct it. In order to support arbitrary objects, the module lets each object customize how it is pickled and, crucially, how it is *unpickled*. As a consequence, reconstructing an object from its pickled bytes can cause code to run.

This means that an adversary who can supply the bytes that a program unpickles can craft a byte sequence that causes the program to execute arbitrary code of the adversary's choosing. The lesson is that general-purpose serialization formats like `pickle` should never be used to decode data coming from an untrusted source; instead, one should use a format with a simple, well-specified decoder that has no ability to invoke arbitrary code.

2.4 Decoding: Android Apps

The application-installation process on Android also suffered from decoding errors. Apps on Android (`.apk` files) are just renamed ZIP

files.

Apps shipped with a signature. When installing an app, Android would first check that the contents of the ZIP file match the signature. If the signature checked out, Android would then install the app. However, the signature checking code and the installation code used different ZIP decoders. An attacker was able to take advantage of a historical quirk of the ZIP format that meant that ZIP holds two lists of files: the signature checker used one list, while the installer used the other list. By pointing to real files from one list but to malicious files in the other list, an attacker was able to bypass this signature check and cause a user to install malicious code.

3 Concurrency Bugs

When systems have code running in parallel, things become much more difficult to reason about, and as a result, many bugs can occur. Consider the following code running on a bank server:

```
def xfer(src, dst, amt):
        s = bal[src]
        d = bal[dst]
        if s < amt:
                raise InsufficientBalanceError
        balances[dst] = d + amt
        balances[src] = s - amt
```

By executing two of these `xfer` requests in parallel, an attacker can cause unexpected behavior. For example, if an attacker tries to send money from a single source to two destinations at once, the check `s < amt` may pass in both executions, and both destinations will then be updated to have money. However, the source will only be deducted once, since `s` is stored before any money is transferred.

The fix is to make sure that there is some kind of locking or concurrency-control strategy in place. The important high-level bit is to be thinking carefully about how concurrent execution can affect your software, in cases when multiple threads of execution can access the same data at the same time.

3.1 File system races / Time-of-check-time-of-use (TOCTOU) bugs

When dealing with files, symbolic links can cause all sorts of chaos. A piece of code might want to check that it is dealing with a regular file—rather than a symbolic link—before opening it. A bad way to implement would be like this:

```
// WRONG
```

```
if(lstat(path, &st) < 0) error();
if(!S_ISREG(st.st_mode)) error();
// ***
int f = open(path, O_RDWR);
...
```

The problem is that if an attacker can replace the file when the code is at line ***, the attacker could swap out the regular file with a symbolic link that points to somewhere else.

The way to defend against this bug is to change the interface. Newer versions of the POSIX file-system APIs enable checking for this type of property in a way that defeats race conditions.

The `openat()` API call gives a way to open a file while ensuring that the file is of a particular type.

4 *Resource Usage*

Other bugs allow attackers to consume many resources on a system, denying service to honest users.

For example, hash tables typically use a hash function designed to be very fast when deciding which bin to place an input in. These hash functions are typically not collision-resistant in the cryptographic sense. Hash tables work well when inputs get distributed evenly across all of the bins. However, if multiple inputs get mapped to the same hash value, the performance of a hash table deviates from the constant-time idea we have of a hash table—hash tables typically revert to using a linked list of all the values for a given hash value.

If an attacker is able to predict which bin their input will end up in, they can maliciously craft inputs that create a huge linked list, resulting in very poor performance for the hash table.

The way to defend against this type of attack is to use a *keyed* hash function—essentially a pseudorandom function—where the implementation does not reveal the secret key to the attacker (over the network, etc.).

5 *Dealing with Software Bugs*

As we have seen, there are many types of bugs so there are many ways to defend against them. Here are some general rules:

Clear Specification. One way to avoid design-level bugs is to have a clear and complete specification about what your program is supposed to do. In particular, for things such as encoders and decoders, a precise specification can make it easier to check that you have handled all of the important corner cases.

Design. A simpler design is easier to understand, and thus bugs are easier to find.

Limit Bug Impact. Some techniques that we will discuss, such as privilege separation, allow us to protect security even when software bugs arise.

Find and Prevent Bugs at Development Time. Techniques such as fuzzing can find bugs before they hit production systems.

Catch and Mitigate Bugs at Runtime Even when a bug makes it into a running program, a number of runtime defenses can catch it or blunt its impact. For memory-corruption bugs, these include the stack canaries discussed above, as well as address-space layout randomization (ASLR), which randomizes where code and data are placed in memory so that an attacker cannot easily predict useful addresses, and marking the stack and other data regions as non-executable (NX) so that injected data cannot be run as code. Running in a memory-safe language adds its own runtime bounds checking. Control-flow integrity (CFI) defends against code-injection attacks by ensuring that the program only follows legitimate control-flow paths, and taint tracking can catch injection attacks by following untrusted data as it flows through the program. We discuss several of these techniques in more detail in the next lecture.

Deploy Bug Fixes Quickly A big advance in browser security came from mechanisms for pushing out software updates quickly.