So far, we have discussed many conceptual techniques to secure systems. Of course, to use these techniques, we must use them. 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, and missing checks.

1 Memory Corruption

Consider the following C code:

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

This type of code would be relatively common several years ago, and still likely shows up today. This code snippet writes input bytes to the buffer until the input ends. When the input ends, gets will write a zero byte to the next location in buf. In C, arrays are passed as pointers to the start of the array, so all that gets receives is the address to start writing.

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

### 1.1 Bounds Checking

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:

```c
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 `sz`. We then may consider adding a check like the following:

```c
#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, integers in C are stored in 32 bits, and $2^{34}$ will not fit! 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.
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

'\texttt{SELECT name FROM users WHERE phone = "617236005"}'.

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

\begin{verbatim}
/* WRONG!!! */
'SELECT name FROM users WHERE phone ='' + phone + ''''.
\end{verbatim}

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

\begin{verbatim}
phone = '123"; DROP TABLE users; ''.
\end{verbatim}

The SQL engine will then receive the query:

\begin{verbatim}
'SELECT name FROM users WHERE phone = "123"; DROP TABLE users;',
\end{verbatim}

which will have the effect of deleting the users table.

To build SQL queries safely from user input, it is important to escape special characters like quotes. Modern libraries for interacting with databases perform this type of 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 the sign up, and may use some code like the following to render another user's list of friends:

\begin{verbatim}
def render_friends(friends: List[str]):
    print("<h3>Friends</h3>")
    print("<ul>")
    for name in friends:
        print("<li>" + name + "</li>")
    print("</ul>")
\end{verbatim}
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:

```html
<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 `&lt;` and `&gt;`. Now that `&` has a special meaning, we also must escape it as `&amp;`. Modern web frameworks have “templating” systems that automate this escaping process.

### 2.3 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:

```python
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.

4 Resource Usage

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

4.1 Hash Tables

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.

5 Dealing with Software Bugs

Software bugs are difficult to tackle, but there are several things to keep in mind:

Clear Specification. One way to avoid design-level bugs is to have a clear and complete specification.

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

Find and Prevent Bugs at Development Time.

Catch and Mitigate Bugs at Runtime

Deploy Bug Fixes Quickly