Principles of Computer Security

Alvin Lin

January 2018 - May 2018

Secure Coding

Code that is weak in quality is likely to introduce security flaws and vulnerabilities that would otherwise not be an issue. Secure code is very much related to writing high quality code. Secure code is usually a three-way partnership between the code itself, the compiler, and the operating system it will run on. The programmer can utilize all three to ensure the behavior of a certain program.

Terminology

- Security Policy: the set of rules/practices that specify what security an organization provides.
- Security Mechanism: code that conforms to security policy.
- Assurance: proof that the security mechanism conforms to the security policy.
- Security Flaw: software defect posing a potential security risk.
- Vulnerability: conditions that allow attackers to violate security policy.
- Exploit: technique that takes advantage of security vulnerabilities to violate a security policy.
- Mitigation: methods, techniques, processes, tools, or runtime libraries that cna prevent or limit exploits against vulnerabilities.

Then vs Now

In the past, systems were smaller and operated on less code. Computers operated behind closed doors with limited network access. Computer users were mostly experts, backed by defense mechanisms such as "guns, gates, and guards". Today, everyone has a computer and they are becoming increasingly networked. Often systems are very large, and many have legacy code. For convenience, code is reused and sacrifices performance.

Trusted Computer Base

A trusted computing base is the part of a system that must work correctly to ensure proper function, for example, the OS kernel and hardware. We need to rely on languages, compilers, and the OS to help enforce protection and security mechanisms. Dijkstra, in addition to being known for his famous search algorithm, is known for his Technische Hogeschool Eindhoven (THE) operating system which conformed to this standard by creating a layered system in which higher levels ONLY depended on the lower layers.

Process Memory Layout

- Text: Read-only segment with instructions for CPU execution. No uncontrolled instruction modification, changing this segment results in a segmentation fault.
- Data: Global initialized static variables.
- Block Started by Symbol (BSS): Global uninitialized static variables. This section is zeroed out by the operating system before program execution.
- Heap: Dynamic memory allocated at runtime.
- Stack: Memory for function variables allocated automatically on call and deallocated on return. Process execution control information.

Strings

Strings are a major mechanism for information exchange between users and software and between software as well. They are used in text fields, command line arguments, environment variables, and other methods of information storage and propagation.

Even though they are so basic, they are not a built-in type in C or C++. The standard C library supports strings of type `char` and wide strings of type `wchar_t`. Software vulnerabilities arise due to weaknesses in string management, representation, and manipulation. They are represented with 8 bit ASCII or 16 bit Unicode and must be null terminated. Multiple string types (like `std::basic_string`, `std::string`, `std:wstring`) cause issues with secure coding in C++. Common errors arise from:

- Improperly bounded string copies when copying an unbounded string to a fixed length `char` array.
- Copying and concatenating strings, since many library calls such as `strcopy()`, `strcat()`, and `sprintf()` perform unbounded copy operations.
- Off-by-one errors
- Null termination errors
- Incorrect string truncation

Exploits

- Unsanitized data entered from a user outside of the program control can cause security issues.
- Buffer overflows from data written outside memory boundaries allow for users to leak or modify program data.
- Stack smashing allows users to overwrite data in the execution stack and execute arbitrary code.
- Code injection allows an overwritten stack return address to point to a malicious function.
- Arc injection allows for the transfer of control from the existing code to some in-process memory.

Concurrency

Concurrency is essential to modern computing. Multiple separate execution flows running simultaneously allow for speed and optimization. Uncontrolled concurrency can lead to non-determinism however.

Race Conditions

An unanticipated execution ordering in concurrent code flow can lead to a defect known as a race condition. Three conditions are necessary for a race condition to occur:

1. Concurrency: there must be at least two concurrent control flows
2. Shared Object: a shared race object must be accessed by both concurrent flows
3. State Change: at least one control flow must alter the state of the race object

Attackers can exploit a race condition to manipulate data in unpredictable ways. To prevent it, access to critical sections must be atomic and code accessing it must execute alone. Synchronization primitives such as locks, counting semaphores, binary semaphores, pipes, monitors, and condition variables help implement mutual exclusion.

Deadlock

Deadlock occurs whenever a set of tasks are blocked waiting for one another to finish. This often leads to a denial of service vulnerability. Four necessary conditions are necessary for a deadlock to occur:

1. Circular wait cycle
2. Mutual exclusion
3. No preemption
4. Hold and wait

Deadlocks can be exploited by altering processor speeds, changes in task scheduling algorithms, different memory constraints on execution, or any asynchronous event capable of interrupting program execution. Attacks often automate the creation of race conditions to cause load in executing a race window.

Time of Check/Time of Use (TOCTOU)

TOCTOU race conditions are caused by check and access conditions happening on a shared object at separate times. An untrusted script running in parallel can modify an object state after the check but before the access, influencing the execution of the following code. Consider the following C code:

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
  FILE* fd;
  if (access("/oldfile", W_OK) == 0) {
    printf("access granted.\n");
    fd = fopen("/oldfile", "wb+");
    /* write data to the file */
    fclose(fd);
  }
  return 0;
}
```

If a malicious entity removed the file `oldfile` and symlinked it to another file, they could gain access to the data written. These attacks are usually mitigated by combining the file check and open in an atomic operation.

You can find all my notes at `http://omgimanerd.tech/notes`. If you have any questions, comments, or concerns, please contact me at alvin@omgimanerd.tech