# 14 Security

Hackers Are Like Artists, Who Wake Up In A Good Mood & Start Painting

Vladimir Putin

Computer security is the protection of hardware and software from unauthorized access or modification. Even if you don't work directly in the computer security field, the concepts are important to learn because all systems will have attackers given enough time. Even though this is introduced as a different chapter, it is important to note that most of these concepts and code examples have already been introduced at different points in the course. We won't go in depth about all of the common ways of attack and defense nor will we go into how to perform all of these attacks in an arbitrary system. Our goal is to introduce you to the field of making programs do what you want to do.

## 14.1 Security Terminology and Ethics

There is some terminology that needs to be explained to get someone who has little to no experience in computer security up to speed

1. An **Attacker** is typically the user who is trying to break into the system. Breaking into the system means performing an action that the developer of the system didn't intend. It could also mean accessing a system you shouldn't have access to.
2. A **Defender** is typically the user who is preventing the attacker from breaking into the system. This may be the developer of the system.
3. There are different types of attackers. There are **white hat** hackers who attempt to hack a defender with their consent. This is commonly a form of pre-emptive testing – in case a not-so-friendly attack comes along. The **black hat** hackers are hackers who hack without permission and the intent to use the information obtained for any purpose. Gray hat hacking differs because the hacker's intent is to inform the defender of the vulnerability – though this can be hard to judge at times.

**Danger Will Robinson** Before we let you go much further, it is important that we talk about ethics. Before you skip over this section, know that your career quite literally can be terminated over an unethical decision that you might make. The computer fraud and security act is a broad, and arguably terrible law, that casts any

non-authorized use of a 'protected computer' of a computer as a felony. Since most computers are involved in some interstate/international commerce (the internet) most computers fall under this category. It is important to think about your actions and have some ladder of accountability before executing any attack or defense. To be more concrete, make sure supervisors in your organization have given you their blessing before trying to execute an attack.

First if at all possible, get written permission from one of your superiors. We do realize that this is a cop-out and this puts the blame up a level, but at the risk of sounding cynical organizations will often put blame on an individual employee to avoid damages TODO: Citation Needed. If not possible, try to go through the engineering steps

1. Figure out what the problem is that you are trying to solve. You can't solve a problem that you don't fully understand.
2. Determine whetheryou need to "hack" the system. A hack is defined generally as trying to use a system unintendedly. First, you should determine if your use is intended or unintended or somewhere in the middle – get a decision for them. If you can't get that, make a reasonable judgement as to what the intended use.
3. Figure out a reasonable estimate of what the cost is to "hacking" the system. Get that reasonable estimate checked out with a few engineers so they can highlight things that you may have missed. Try to get someone to sign off on the plan.
4. Execute the plan with caution. If at any point something seems wrong, weigh the risks and execute the plan.

If there isn't a certain ethical guideline for the current application, then create some. This is often called a `policy vacuum`. This may seem like busy work and more on the "business side" than computer scientists are used to, but your career is at stake here. It is up to you as a computing professional to assess the risk and to decide whether to execute. Courts generally like sitting on precedent, but you can easily say that you aren't a legal scholar. In lieu, you must be able to say that you reacted as a "reasonable" engineer would react.

TODO: Link to some case studies of real engineers having to decide

### 14.1.1 CIA Triad

There are three commonly accepted goals to help understand if a system is secure.

1. Information Confidentiality means that only authorized parties are allowed to see a piece of information
2. Information Integrity means that only authorized parties are allowed the modify a piece of information, regardless of whether they are allowed to see it. It ensures that information remains in complete during transit.
3. Information Availability means information, or a service, is available when it is needed.
4. The triad above forms the Confidentiality, Integrity, and Availability (CIA) triad, often authenticity is added as well.

If any of these are broken, the security of a system (either a service or piece of information) has been compromised.

## 14.2 Security in C Programs

### 14.2.1 Stack Smashing

Consider the following code snippet

```
void greeting(const char *name) {
  char buf[32];
  strcpy(buf, name);
  printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
  if (argc < 2){
    return 1;
  }
  greeting(argv[1]);
  return 0;
}
```

There is no checking on the bounds of strcpy! This means that we could potentially pass in a large string and get the program to do something unintended, usually via replacing the return address of the function with the address of malicious code. Most strings will cause the program to exit with a segmentation fault

```
$ ./a.out john
Hello, john!
$ ./a.out JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Program received signal SIGSEGV, Segmentation fault.
...
```

If we manipulate the bytes in certain ways and the program was compiled with the correct flags, we can actually get access to a shell! Consider if that file is owned by root, we put in some valid bytecode (binary instructions) as the string. What will happen is we'll try to execute `execve('/bin/sh', '/bin/sh', NULL , NULL)` that is compiled to the bytecode of the operating system and pass it as part of our string. With some luck, we will get access to a root shell.

```
$ ./a.out <payload>
root#
```

The question arises, which parts of the triad does this break? Try to answer that question yourself. So how would we go about fixing this? We could ingrain into most programmers at the C level to use `strncpy` or `strlcpy` on OpenBSD systems. Turning on stack canaries as explained later will fix this issue as well.

### 14.2.2 Buffer Overflow

Most of you are already familiar with Buffer Overflows! A lot of time they are fairly tame, leading to simple program crashes or funny mistakes. Here is a complete example

```
> cat main.c
#include <stdio.h>

int main() {
    char out[10];
    char in[10];
    fscanf(stdin, "%s", in);
    out[0] = 'a';
    out[9] = '\0';
    printf("%s\n", out);

    return 0;
}
> gcc main.c -fno-stack-protector # need the special flag otherwise
    won't work
# Stack protectors are explained later.
> ./a.out
hello
a
> ./a.out
hellloooooooo
aoo
>
```

What happens here should be clear if you recall the c memory model. Out and in are both next to each other in memory. If you read in a string from standard input that overflows in, then you end up printing aoo. It gets a little more serious if the snippet starts out as

```
int main() {
  char pass_hash[10];
  char in[10];
  read_user_password(pass_hash, 10);
  // ...
}
```

### 14.2.3 Out of order instructions & Spectre

Out of order execution is an amazing development that has been recently adopted by many hardware vendors (think 1990s) TODO: citation needed. Processors now instead of executing a sequence of instructions (let's say assigning a variable and then another variable) execute instructions before the current one is done [1, P. 45]. This is because modern processors spend a lot of time waiting for memory accesses and other I/O driven applications. This means that a processor, while it is waiting for an operation to complete, will execute the next few operations. If any of the operations would possibly alter the final result, there is a barrier, or if the re-ordering violates the data dependencies of the instructions, the processor keep the instructions in the stated order [1, P. 296].

Naturally, this allowed CPUs to become more energy-efficient while executing more instructions in real-time and increased security risks from complex architectures. What system programmers are worried about is that operation with mutex locks among threads are out of order – meaning that a pure software implementation of a mutex will fail without copious memory barriers. Therefore, the programmer has to acknowledge that updates may be missed among a series of threads, given that there is no barrier, on modern processors.

One of the most prominent bugs concerning this is Spectre [2]. Spectre is a bug where instructions that otherwise wouldn't be executed are speculatively executed due to out-of-order instruction execution. The following snippet is a high-level proof of concept.

```
char *a[10];
for (int i = 10; i != 1; --i) {
  a[i] = calloc(1, 1);
}
a[0] = 0xCAFE;
int val;
int j = 10; // This will be in a register
int i = 10; // This will be in main memory
for (int i = 10; i != 0; --i, --j) {
  if (i) {
    val = *a[j];
  }
}
```

Let's analyze this code. The first loop allocates 9 elements through a valid malloc. The last element is `0xCAFE`, meaning a dereference should result in a SEGFAULT. For the first 9 iterations, the branch is taken and `val` is assigned to a valid value. The interesting part happens in the last iteration. The resulting behavior of the program is to skip the last iteration. Therefore, `val` never gets assigned to the last value.

But under the right compilation conditions and compiler flags, the instructions will be speculatively executed. The processor thinks that the branch will be taken, since it has been taken in the last 9 iterations. As such, the processor will fetch those instructions. Due to out-of-order instruction execution, while the value of $i$ is being fetched from memory, we have to force it not to be in a register. Then, the processor will try to dereference that address. This should result in a SEGFAULT. Since the address was never logically reached by the program, the result is discarded.

Now here is the trick. Even though the value of the calculation would have resulted in a SEGFAULT, the bug doesn't clear the cache that refers to the physical memory where 0xCAFE is located. This is an inexact explanation,

but essentially how it works. Since it is still in the cache, if you again trick the processor to read form the cache using `val` then you will read a memory value that you wouldn't be able to read normally. This could include important information such as passwords, payment information, etc.

### 14.2.4 Operating Systems Security

1. Permissions. In POSIX systems, we have permissions everywhere. There are directories that you can and can't access, files that you can and can't access. Each user account is given access to each file and directory through the read-write-execute (RWX) bits. The user gets matched with either the owner, the group, or 'everyone else', and their access to the file is limited using these bits. Note that permissions work slightly differently on directories compared to files.

2. Capabilities. In addition to permissions on files, each user has a certain set of permissions that they can do. For a full list, you can check capabilities(7). In short, allowing a capability allows a user to perform a set of actions. Some examples include controlling networking devices, creating special files, and peering into IPC or interprocess communication.

3. Address Space Layout Randomization (ASLR). ASLR causes the address spaces of important sections of a process, including the base address of the executable and the positions of the stack, heap and libraries, to start at randomized values, on every run. This is so that an attacker with a running executable has to randomly guess where sensitive information could be hidden. For example, an attacker may use this to easily perform a `return-to-libc` attack.

4. Stack Protectors. Let's say you've programmed a buffer overflow as above. In most cases, what happens? Unless specifically turned off, the compiler will put in stack protectors or stack canaries. This is a value that resides in the stack and must remain constant for the duration of the function call. If that protector is overwritten at the end of the function call, the run time will abort and report to the user that stack smashing was detected.

5. Write xor Execute, also known as `Data Execution Prevention` (DEP). This is a protection that was covered in the IPC section that distinguishes code from data. A page can either be written to or executed but not both. This is to prevent buffer overflows where attackers write arbitrary code, often stored on the stack or heap, and execute with the user's permissions.

6. Firewall. The Linux kernel provides the netfilter module as a way of deciding whetheran incoming connection should be allowed and various other restrictions on connections. This can help with a DDOS attack (explained later).

7. AppArmor. AppArmor is a suite of operating system tools at the userspace level to restrict applications to certain operations.

OpenBSD is an arguably better system for security. It has many security oriented features. Some of these features have been touched upon earlier. An exhaustive list of features is at `https://www.openbsd.org/innovations.html`

1. pledge. Pledge is a powerful command that restricts system calls. This means if you have a simple program like `cat` which only reads to and from files, one can reasonably restrict all network access, pipe access, and write access to files. This is known as the process of "hardening" an executable or system, giving the smallest amount of permissions to the least number of executables needed to run a system. Pledge is also useful in case one tries to perform an injection attack.

2. unveil. Unveil is a system call that restricts the access of a current program to a few directories. Those permissions apply to all forked programs as well. This means if you have a suspicious executable that you want to run whose description is "creates a new file and outputs random words" one could use this call to restrict access to a safe subdirectory and watch it receive the SIGKILL signal if it tries to access system files in the root directory, for example. This could be useful for your program as well. If you want to ensure that no user data is lost during an update (which is what happened with a Steam system update), then the system could only reveal the program's installation directory. If an attacker manages to find an exploit in the executable, it can only compromise the installation directory.

3. sudo. Sudo is an openBSD project that runs everywhere! Before to run commands as root, one would have to drop to a root shell. Some times that would also mean giving users scary system capabilities. Sudo gives you access to perform commands as root for one-offs without giving a long list of capabilities to all of your users.

### 14.2.5 Virtualization Security

Virtualization is the act of creating a virtual version of an environment for a program to run on. Though that definition might be bent a little with the advent of new-age bare metal Virtual Machines, the abstraction is still there. One can imagine a single Operating System per motherboard. Virtualization in the software sense is providing "virtual" motherboard features like USB ports or monitors that another program (the bridge) communicates with the actual hardware to perform a task. A simple example is running a virtual machine on your host desktop! One can spin up an entirely different operating system whose instructions are fed through another program and executed on the host system. There are many forms of virtualization that we use today. We will discuss two popular forms below. One form is virtual machines. These programs emulate *all* forms of motherboard peripherals to create a full machine. Another form is containers. Virtual machines are good but are often bulky and programs only need a certain level of protection. Containers are virtual machines that don't emulate all motherboard peripherals and instead share with the host operating system, adding in additional layers of security.

Now, you can't have proper virtualization without security. One of the reasons to have virtualization is to ensure that the virtualized environment doesn't maliciously leak back into the host environment. We say maliciously because there are intended ways of communication that we want to keep in check. Here are some simple examples of security provided through virtualization

1. chroot is a contrived way of creating a virtualization environment. chroot is short for change root. This changes where a program believes that (/) is mounted on the system. For example with chroot, one can make a hello world program believe `/home/user/` is actually the root directory. This is useful because no other files are exposed. This is contrived because Linux still needs additional tools (think the c standard library) to come from different directories such as `/usr/lib` which means those could still be vulnerable.

2. namespaces are Linux's better way to create a virtualization environment. We won't go into this too much, just know that they exist.

3. Hardware virtualization technology. Hardware vendors have become increasingly aware that physical protections are needed when emulating instructions. As such, there can be switches enabled by the user that allows the operating system to flip into a virtualization mode where instructions are run as normal but are monitored for malicious activity. This helps the performance and increases the security of virtualized environments.

## 14.3 Cyber Security

Cyber Security is arguably the most popular area of security. More and more of our systems are hacked over the web, it is important to understand how we can protect against these attacks

### 14.3.1 Security at the TCP Level

1. Encryption. **TCP is unencrypted!** This means any data that is sent over a TCP connection is in plain text. If one needs to send encrypted data, one needs to use a higher level protocol such as HTTPs or develop their own.
2. Identity Verification. In TCP, there is no way to verify the identity of who the program is connecting to. There are no checks or federated databases in place. One just has to trust the DNS server gave a reasonable response which is almost always the incorrect answer. Apart from systems that have an approved white list or a "secret" connection protocol, there is little at the TCP level that one can do to stop.
3. Syn-Ack Sequence Number. This is a security improvement. TCP features what we call sequence numbers. That means that during the SYN-SYN/ACK-ACK dance, a connection starts at a random integer. This is important because if an attacker is trying to spoof packets (pretend those packets are coming from your program) that means that the attacker must either correctly guess – which is hard – or be in the route that your packet takes to the destination – much more likely. ISPs help out with the destination problem because it may send a connection through varying routers which makes it hard for an attacker to sit anywhere and be sure that they will receive your packets – this is why security experts usually advise against using coffee shop wifi for sensitive tasks.
4. Syn-Flood. Before the first synchronization packet is acknowledged, there is no connection. That means a malicious attacker can write a bad TCP implementation that sends out a flood of SYN packets to a hapless server. The SYN flood is easily mitigated by using IPTABLES or another netfilter module to drop all incoming connections from an IP address after a certain volume of traffic is reached for a certain period.
5. Denial of Service, Distributed Denial of Service is the hardest form of attack to stop. Companies today are still trying to find good ways to ease these attacks. This involves sending all sorts of network traffic forward to servers in the hopes that the traffic will clog them up and slow down the servers. In big systems, this can lead to cascading failures. If a system is engineered poorly, one server's failure causes all the other servers to pick up more work which increases the probability that they will fail and so on and so forth.

### 14.3.2 Security at the DNS Level

As of 2019, the United States Department of Homeland Security released a directive to switch all services from DNS to DNSSec `https://cyber.dhs.gov/assets/report/ed-19-01.pdf`. This directive is an inherent flaw of the DNS system. First, DNS doesn't offer any sort of verification on domain name requests. That is, it is easy to spoof DNS nameservers such that they point your browser to potentially malicious servers. Remember that DNS requests are sent as unsecured UDP packets, which are prone to tampering. This means that if an attacker snags a plain-text request for a DNS server, that attacker can now send the result back to the requester. More commonly instead of just attacking one person, they will connect to a public wifi station and poison the cache of the router – meaning that all who are connected will get a bad IP address when requesting a domain name. This can get into serious spoofing attacks if one tries to pretend they are a major bank.

## 14.4 Topics

1. Security Terminology
2. Security in local C programs
3. Security in CyberSpace

## 14.5 Review

1. What is a chmod statement to break only the confidentiality of your data?
2. What is a chmod statement to break only the confidentiality and availability of your data?
3. An attacker gains root access on a Linux system that you use to store private information. Does this affect confidentiality, integrity, or availability of your information, or all three?
4. Hackers brute force your git username and password. Who is affected?
5. Why is privilege separation useful in RPC applications?
6. Is it easier to forge a UDP or TCP packet and why?
7. Why are TCP sequence numbers initialized to a random number?
8. What is the impact if the RAM used to hold a shared library (e.g. the C standard library) was writable by any process?
9. Is creating and implementing client-server protocols that are secure and invulnerable to malicious attackers easy?
10. Which is harder to defend against: Syn-Flooding or Distributed Denial of Service?
11. Does deadlock affect the availability of a service?
12. Do buffer overflows / underflows affect the integrity of a data?
13. Why shouldn't stack memory be executable.
14. HeartBleed is an example of what kind of security issue? Which one(s) of the triad does it break?
15. Meltdown and Spectre is an example of what kind of security issue? Which one(s) of the triad does it break?

# Bibliography

[1] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[2] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.