CSE 127 Lecture 5: Isolation and Secure Design

These scribe notes were written by students in CSE 127 Winter 2021. They have been lightly edited but may still contain some errors.

1 Intro

Today, we often need to run buggy or untrusted code. This code can be the foundation for a number of things you use throughout any given day. This includes applications and systems such as:

- Desktop Applications like Zoom
- Mobile apps like TikTok
- Untrusted User code
- Websites, JavaScript, browser extensions
- PDF viewers
- Email clients like the Gmail Mobile App, Thunderbird
- VMs on cloud computing infrastructure

Because these issues can occur almost anywhere, systems have to be resilient in the face of vulnerabilities and malicious users. If not, systems could shut down or leak sensitive information to attackers.

There are several general principles of secure system design that can help mitigate the effects of vulnerabilities from buggy and untrusted code. We will go more in depth in each of these soon, but below is a basic overview of each principle.

- **Least Privilege**

  Ensure that components in a system run with the minimal privileges required to complete a given task.

- **Privilege Separation**

  Separate the levels of privilege between system components in order to limit access.

- **Complete Mediation**

  All communication across a security boundary should be controlled.

- **Fail Safe/Closed**

  If part of the system fails, it should do so in such a way that the system remains secure. (Stopping execution if there is an error is an example).

- **Defense In Depth**

  Deploy multiple countermeasures to make it more difficult for an attacker to fully compromise a system.

- **Keep It Simple**

  Build systems and trusted components that are simple enough to understand and audit.

Each of these principles can be used together. Now, let's dive up deeper into each one.

2 Least Privilege

The core idea behind the Principle of Least Privilege is that users should only have access to the data and resources that they need to perform their intended tasks. Some examples of this are teaching faculty only being able to change grades for classes they teach, or only letting employees with background checks have access to classified documents.

3 Privilege Separation

To implement least privilege, we must be able to divide a system into separate components with different access levels. A real life example of this would be the security clearance system in the United States government, where documents are classified at different security levels, and only those with the proper clearance and need are able to access documents with a given clearance level. At the operating system level, two common ideas are memory and resource isolation, where processes should not be able to access each others memory, and should only be able to access certain resources instead of the entire system's available resources. In Unix, processes are given their own virtual address space that is managed by the operating system, and is unavailable to other processes without the operating system's assistance. This ensures memory isolation across processes.

4 Principle of Complete Mediation

Privileges are the key to a resource, but the principle of complete mediation requires the key to be checked before the keyholder can access the resource. In other words, complete mediation necessitates a security checkpoint for all accesses to scan through.

4.1 Virtual Memory

Virtual memory is an example of an operating system performing complete mediation on process memory accesses. Each process has its own virtual memory address space. When a program or process calls load, store, or instruction fetch, they are accessing virtual memory addresses. The OS has to then translate the addresses to physical memory and execute the call if the program is allowed to do so. The OS mediates all process memory accesses and enforces control policies. Virtual memory allows a program to operate as if the computer had a larger memory space than it actually does, but also has the security benefit that malicious programs cannot access or modify data from other programs that they should not have access to.

4.2 Resource Isolation in the Unix security model

In Unix, every resource is a file: files, sockets, pipes, hardware devices, etc... How does the OS know which user or process can access which resources or files? Every user is associated with a user ID (UID) and every process is associated with a process ID (PID). Every file has associated permissions that act as an access control list (ACL). These permissions grant read/write/execute abilities on a file to a user based on their UIDs and roles (owner, group, other). In Unix, the root user (UID 0) can access everything. To access files and other resources, programs must go through the operating system for access, and the operating system mediates access to resources based on these permissions. This is another example of complete mediation.

5 Role Based Access Control

The Unix permission model is an example of Role Based Access Control (RBAC). RBAC works by allocating a specific access based on the role of the user. The general structure of a RBAC is usually in the form of a matrix that specified specific permissions to each user. Below is an example. Here, certain permissions for read (r), write (w), and execute (x) are granted to each of the user's roles for each file.

	hw/	exams/	grades/	lectures/
cse127-instr	r/w	r/w	r/w	r/w
cse127-tas	r/w	read	-	r/w
cse127-students	read	-	-	read
cse-students	-	-	-	read

Lecture Example: CSE 127 Access Control Matrix [**?**]

6 Capabilities vs Access Control Lists (ACL)

One can view the permission matrix above column-wise, in which each resource is associated with an access control list, or row-wise, in which each user has capabilities granting them permissions on resources.

For capability systems, each user is associated with certain permissions for each resource. These can be specified in the form of an unforgeable "ticket" specifying "capabilities" for each file that can be presented when attempting to access the resource. These are represented by each row of the access control matrix. Each row denotes a user, and all their permissions for all the files which are listed in the columns. An analogy to this would be like a theater or concert ticket system. As long as a user has a ticket for access, they will be granted access.

An Access Control List controls entry by storing a list of users and their associated permissions with each resource. These are represented by each column in the access control matrix. An analogy to this would be something akin to guest lists for an event, the files themselves are the event, and users are guests on the lists. Based on this "guest list", the OS will then check permissions and allow users on the guess list entry/access to the resource.

Many modern operating systems now use a combination of both ACLs and capabilities for permissions. The Unix system of file permissions is an example of a simplified Access Control List. (Users are classified into either the user owner, group owner, or other for all other users). Each file then lists our file operations, read, write, or execute, for each categories. The OS will then know what user group can perform what action on a specific resource. Mobile phone app permissions where a user can grant, say, camera or microphone access to a given app might be an example of a capability.

7 Process UID and setuid

In Unix, each process has three associated UIDs that determine the priviliges that the program runs with. Fork and exec functions will inherit the 3 UIDs (RUID, EUID, SUID) of the parent process.

- **1. Real User ID (RUID)**

The RUID is the user who started that process. A program retains the User ID of the parent process. This can otherwise be known as the UID of the process owner.

- **2. Effective User ID (EUID)**

  EUID specifies the permissions that the process runs under. The default is that it is the same as the Real User ID, but it can be changed if the setuid bit is set.

- **3. Saved User ID (SUID)**

  SUID is the EUID prior to any changes.

- **setuidbit**

  If this bit is set, the EUID is the UID of the file owner rather than the UID of the user who launched the program.

  An example: `-rwsr-xr-x 1 root root 54256 Mar 26 2019 /usr/bin/passwd`

  Users can run the `passwd` program to update their system passwords. User passwords are stored together in the file `/etc/passwd` which lists user information together with users' hashed passwords. This file needs to be protected from malicious users by only root having read or write access to the file, so the owner of this file is root. In the `ls` example above, you can see the "s" modifier which sets the setuid bit on the `/usr/bin/passwd` executable. This means that when a non-root user runs the `passwd` program to update their password, the program runs with the permissions of the file owner (root), which gives the program the necessary permissions to edit the `/etc/passwd` file even though the non-root user does not.

  In other scenarios, this bit might be set is when you want to lower privileges. Applications such as web servers might need root privileges for accessing port 80, but might want to run other programs with lower privileges. Setting the bit and the program file owner means that these programs can be run by the root server with restricted privileges.

8 UID Bit Settings

There are three file permission bits that can be set in order to modify the User IDs (UID) of the executed resources. One type was already explored in the form on the setuid bit. It is reiterated below for a reminder.

- **1. setuid bit**

  Sets the EUID of process to the owner UID of executed file.

- **2. setgid bit**

  Sets the group EUID of process to the group UID of executed file.

- **3. Sticky Bit**

  A. IF ON, only a file owner or root can rename or remove files in a directory.

  B. IF OFF, as long as user has write permission in the directory, they can rename/remove files in directory even if not root/owner.

  An example: `drwxrwxrwt 10 root root 12288 Jan 18 20:55 tmp`

  Notice the "t" permission set on the last bit. This allows only the owner of a file in tmp to rename/remove the file. If Alice creates a file in tmp called test, then Bob will not be able to create another file with the same name. The test file has user owner and group owner for Alice, therefore Bob does not have permissions to rename or remove that file within `/tmp`.

9 UNIX File Security Mechanism

The UNIX File Security Mechanism has several pros and cons. The UNIX FS Mechanism is simple and flexible in that it acts as an Access Control List (ACL) for files and can create many users on the system with different roles, each with different access privileges. On the other hand, some disadvantages of this mechanism are that it is coarse-grained, requires root for nearly all system operations, and runs many services as root, meaning there are many potential vulnerabilities to exploit and run as root.

10 Kernel Isolation

One example of privilege separation and complete mediation is the CPU/hardware enforced isolation of the kernel. The kernel is isolated from user processes and has separate page tables. This isolation is to ensure that the user space cannot use privileged instructions. In addition, user programs who require kernel operations are checked when they enter the kernel protection domain (i.e. system calls). In some operating systems, even the user processes are isolated from each other, only allowing verified message channels to communicate (e.g. Software Isolated Processes in the Singularity operating system).

11 Process Confinement: System Call Interposition

The idea behind system call interposition is the observation that in order to damage the host system, an app must make system calls. For example, to

delete or overwrite files, one can use the commands: *unlink*, *open*, or *write*. In order to communicate over the network, the application must use system calls like *socket*, *bind*, *connect*, or *send*. System call interposition is an extra layer of protection around a program that monitors system calls and prevents a program from making unauthorized system calls. Even if a program behaves maliciously, this would make it more difficult for the program to act maliciously on the system.

12 Key Component: Reference Monitor

A reference monitor is an example of complete mediation across security boundaries in an operating system. The properties of a reference monitor are: it must mediate requests, it must be invoked for every access, it should not be able to be killed, or if it is killed, then the monitored process is killed too. It should also be small enough to be analyzed and validated.

13 System Call Interposition (SCI) in Linux

seccomp-bpf is a system call filtering facility that allows configurable policies written in Berkeley Packet Filter rules. It is used in Chromium and Docker containers. A container is used for process-level isolation and is prevented from making syscalls filtered by *seccomp-bpf*.

14 Smartphone OS Design

How does the threat model for a smartphone differ from a desktop? Here are a few possible threats that a smartphone designer might take into account:

- Malicious applications and/or application writers
- Users rooting their smartphones
- Applications misrepresenting functionalities
- Remote adversaries exploiting hardware (i.e. Radio, bluetooth, WiFi, cell towers)

In the smartphone context, assets to be protected could include user data, the camera(s) and microphone(s) built into the device, and network communications. For the security properties that need to be preserved, these might include preserving isolation across apps, protecting user privacy, giving users control over permissions, and others.

Here are some additional orthogonal security facts about phones that we discussed in class:

- *Cell phones continuously broadcast unique identifiers associated with a phone that are recorded by nearby cell towers or other entities with appropriate antennas. This cell tower data can be used to track a phone's location to varying levels of granularity, and can be accessed by law enforcement with a search warrant.*
- *Cell phone applications can track user location at a very fine granularity using a combination of cell tower location, GPS, wifi networks, and other information. By default, Google keeps an extremely detailed record of location data for all users.*

14.1 Android Process Isolation

Android uses the Linux security model and sandboxing in order to isolate apps from each other and the system. Each application runs under its own UID. Applications are able to request permissions, which are essentially capabilities, by using an API to request any information outside their sandbox. Then, the reference monitor in the operating system checks the permissions and determine whether the permission is allowed.

15 Software Fault Isolation (SFI)

There is overhead in having dedicated memory address space and isolation for every component. The idea of SFI is to be able partition apps running in the same address space so that:

- Kernel modules should not corrupt the Kernel.
- Native libraries should not corrupt the JVM.

The SFI approach is to partition memory into segments. By doing so, we can implement **memory isolation** by mediating all load and store instructions on memory. This memory isolation gives us **complete mediation** by disallowing privileged instructions for processes without kernel privileges. By segmenting the memory, we can ensure **control flow integrity** by restricting all control flow to the control flow graph (CFG) that checks these loads and stores. As such, we get this **syscall-like interface** between isolated code.

16 Browser Design Example

What is the threat model?

- Malicious web pages and software written in javascript.

What are the assets?

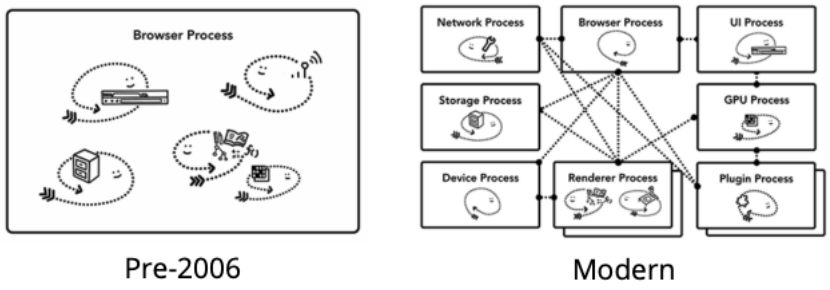
- Different web pages visited, currently open tabs, and the information within each of them.

What security properties do we want to preserve?

- That one corrupt component does not affect others.
- That one site cannot read the data from others.
- Web pages and tabs need to be isolated from each other.

17 Chrome Security Architecture

Pre-2006, chrome was a monolithic architecture with different threads handling things like file system access or rendering. More modern chrome browsers are completely sandboxed, such that every single process runs in isolation with a complicated sequence of IPC calls to communicate between them.



Source: Google Web Updates September, 2018. [**?**]

18 Modern Browser Security Model

There are different processes running, and they need to be isolated from each other. For example, by separating browser processes and rendering processes, web browsers try to protect bugs in one component to affect other parts of the browser, which is a constant arms race.

Browser process

- The browser process handles the privileged parts of browser (e.g. network requests, address bar, bookmarks)

Renderer process

- The renderer process handles untrusted attacker content: JS engine, DOM, etc.
- Communication with the renderer process is restricted to remote procedure calls, in order to prevent attackers to access privileged code, like in the browser process.

There are many other processes like GPU, plugin, UI, and Storage. These processes are also isolated in their sandboxes to prevent attacks on them or other processes.

19 Virtual Machines (VM)

Virtual machines allow a single piece of hardware to emulate multiple machines by creating seperate independent virtual environments. Such technology is useful in cloud computing and for isolating processes. For example, isolation is useful when:

- Cloud compute customers (AWS, Azure, Google Cloud) are on the same server and they need to seperate customers and their data
- Processes prevent malicous interactions in interprocess communication by running every process in a virtual machine (Qubes OS: A desktop OS where everything is a VM)

The hypervisor is the software interface that VMs use to communicate with the host OS. Since it controls access to host OS resources, the hypervisor enforces isolation by mediating these requests. Additionally, there is hardware support for virtualization. Intel has hardware support for x86 virtualization by implementing Virtual Machine Manager (VMM) support in hardware so that the operating system can be run in ring 0 of the kernel without requiring VMM intervention for syscalls.

20 Hardware Isolation

Modern computer systems also contain hardware security mechanisms to enforce different isolation properties. For example, Intel Software Guard eXtensons (SGX) can run code in an **enclave**, an encrypted region of memory that is only accessible to the CPU. This can protect program execution and memory even from a malicous OS. Examples of when a program needs protection from a malicous OS include:

- DRM (Digital Rights Management)
- Secure remote computation
- Protecting crypto keys or sensitive information from root-level attackers

21 iOS Secure Boot

Some mobile devices use secure enclaves to prevent malicous code execution during sensitive events. For example, Apple devices use a secure enclave coprocessor as part of its boot chain. The purpose is to create a hardware-based root

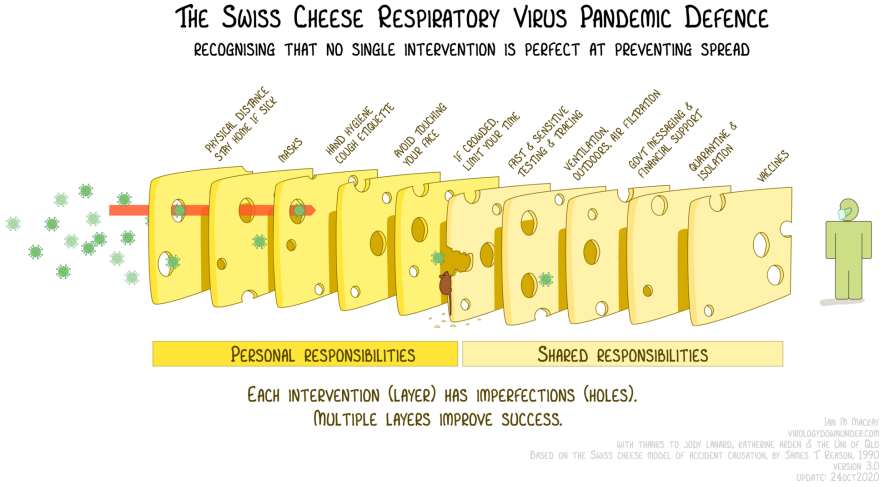
of trust wherein code and code-verifying keys are baked into the boot ROM, a read-only section of memory. Each step of the boot process verifies that the bootloader and kernel are signed by Apple. This prevents a malicous user from executing malicous code during the boot sequence of an Apple device, that is, it prevents rooting/jailbreaking.

22 Physical Isolation: Air Gap

The most extreme form of isolation is complete physical isolation between systems, called an air gap. Physically isolated systems should only be connected to completely physically separated networks. This is one way to attempt to ensure that a misbehaving app on one system cannot affect the other system. We would want to completely separate extremely sensitive systems whose data should not be shared to other machines, such as voting machines or simply systems that should not connect to the internet (e.g. power plants, industrial control systems, or access to nuclear weaponary).

The downsides for a physically isolated setup is that they are difficult to actually implement in practice, and thus only effectively used for high-resource, high-security scenarios. Even so, even physically air-gapped systems have been compromised by well-resourced attackers: a prominent example is the Stuxnet malware.

23 Principle of Defense in Depth



Wikipedia: Swiss cheese model [**?**]

It is unreasonable to expect that any one of our defenses is flawless. The swiss cheese model of pandemic measures shown in lecture gives us a good analogy. You can think of each cheese as a layer of securing the system, the person as the

system, and the many viruses as the attacker's toolbox. Any single defense may be imperfect, but using multiple defenses together is likely to be more effective against attacks.

24 Principle of Fail Safe/Closed

If a reference monitor fails, a fail open system would let communications flow through. Fail closed, however, would block the communication if the monitor fails.

25 Principle of Keeping It Simple

In the end, we have to trust some components of the system, because any system must have a Trusted Computing Base (TCB). In general, we would want to keep the TCB small and simple enough that it can be tested and verified. If the TCB enforces sufficient isolation across the rest of the system, a fully verified TCB can allow bootstrapped trust for a much more complex system. For example, a small kernel has fewer possible entry points than a large, monolithic kernel with many features. The idea is similar to unit testing a few functions rather than hundreds of functions, where there is a higher probability of missing a vulnerability.

26 Conclusion

In conclusion, there are a range of software and hardware isolation techniques to secure systems. We covered sandboxing, access control, containers, virtualization, secure enclaves, and physical isolation. There are a variety of approaches and techniques, which are chosen based on the threat model, costs, and system usage. Overall, the lesson to take away from all of this is that complete isolation is often inappropriate and in real systems it is often sufficient for applications to communicate through regulated interfaces.

References

[1] Nadia Heninger. Isolation and secure design. `http://cseweb.ucsd.edu/classes/wi21/cse127-a/slides/5-isolation.pdf`. Accessed: 2021-02-13.

[2] Mariko Kosaka. Inside look at modern web browser (part 1). `https://developers.google.com/web/updates/2018/09/inside-browser-part1`. Accessed: 2021-02-13.

[3] Wikipedia. Swiss cheese model. `https://en.wikipedia.org/wiki/Swiss_cheese_model`. Accessed: 2021-02-13.