# EECE7376: Operating Systems Interface and Implementation

## Security and Protection


# Introduction

- Operating systems needs to be secure by preventing **unauthorized access**, **malicious destruction** or **alteration of data**, and **accidental introduction of inconsistency**.
- We say that a system is **secure** if its resources are used and accessed as intended under all circumstances.
- Operating systems need to provide **protection mechanisms** that allow the **implementation of the security features**.
- Security violations of the system can be categorized as **intentional** (malicious) or **accidental**.


# Security Violations Forms

- **Breach of confidentiality**: It involves unauthorized theft of information (e.g., identity theft).
- **Breach of integrity**: It involves unauthorized modification of data.
- **Breach of availability**: It involves unauthorized destruction of data.
- **Theft of service**: It involves unauthorized use of resources (e.g., installing a daemon on a system that acts as a file server).
- **Denial of service (DoS)**: It involves preventing legitimate use of the system.


# Security Measures Levels

- **Physical**: Both the **machine rooms** and **the computers** that have access to the target resources must be secured.
- **Network**: Computer data frequently travel over shared lines of wired and wireless networks. Securing such networks is necessary as **intercepting** these data can be just as harmful as breaking into a computer.
- **Operating system**: The operating system may harbor **many vulnerabilities**. Operating systems must be "hardened to decrease the **attack surface**, which is the set of points at which an attacker can try to break into the system.
- **Application**: Third-party applications may also pose risks, especially if they **possess significant privileges**.


# Security Human Factor

- **Social engineering**: using deception to persuade authorized users to give up confidential information.

- **Phishing**: It is one type of social-engineering attack in which a legitimate-looking e-mail or web page misleads a user into entering confidential information.


# Program Threats

- **Malware** is software designed to exploit, disable or damage computer systems.
- **Trojan horse** is a program that acts in a malicious manner, rather than simply performing its stated function.
  - Example: program that emulates a login program to steal passwords.
- **Spyware** sometimes accompanies a program that the user has chosen to install (e.g., freeware).


# Principle of Least Privilege

- The *principle of least privilege* commonly occurs when the operating system allows by default more privileges than a normal user needs or when the user **runs by default as an administrator**.
- An operating system should allow fine-grained control of access and security, so that **only the privileges needed** to perform a task are available during the task's execution.


# Trap Door

- *Trap door* (or *back door*) is the type of security breach where the designer of a program or system **leaves a hole** in the software that only the designer is capable of using.
- A trap door may be set to operate only under a specific set of logic conditions, in which case it is referred to as a **logic bomb**.
- A trap door could be included in a compiler. The **compiler** could generate the intended object code as well as a trap door.
  - In 2015, malware that targets Apple's Xcode compiler suite (dubbed "XCodeGhost") affected many software developers who used compromised versions of XCode not downloaded directly from Apple.


# Code-Injection Attack

- Code-injection attack happens when a software harbors vulnerabilities that allow an attacker to take over the program code and run an alternative code.
- Such vulnerabilities include **buffer overflows**, allowing access to unauthorized memory space.

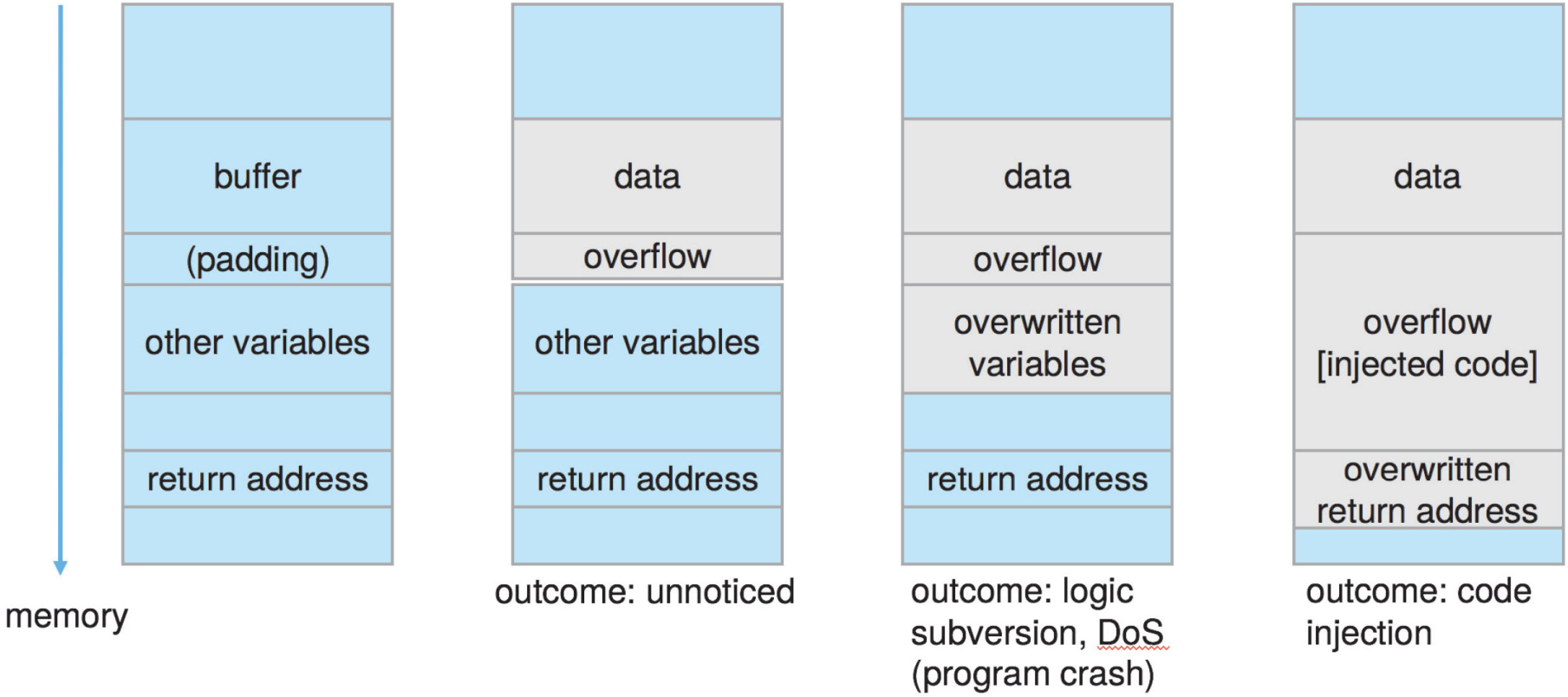

# Buffer Overflow

- The following is an example of a vulnerable code that causes a buffer overflow, which occurs due to an **unbounded copy operation**, the call to `strcpy()`.

```
#include <stdio.h>
#define BUFFER_SIZE 20
int main(int argc, char *argv[]) {
 char buffer[BUFFER_SIZE];
 //...
 strcpy(buffer,argv[1]);
 //...
 return 0;
}
```

- A careful programmer could have performed bounds checking on the size of `argv[1]` by using the `strncpy()`, replacing the line "`strcpy(buffer, argv[1]);`" with "`strncpy(buffer,argv[1], sizeof(buffer)-1);`".

# Buffer Overflow Possible Effects




- If the overflow greatly exceeds the padding, all of the current function's **stack frame** is overwritten. At the very top of the frame is the function's return address (where usually the **ra** register is pushed). The flow of the program can be redirected by the attacker to another region of memory where the injected code is then executed and can do anything allowed by the **privileges** of the attacked process.


# Viruses Categories

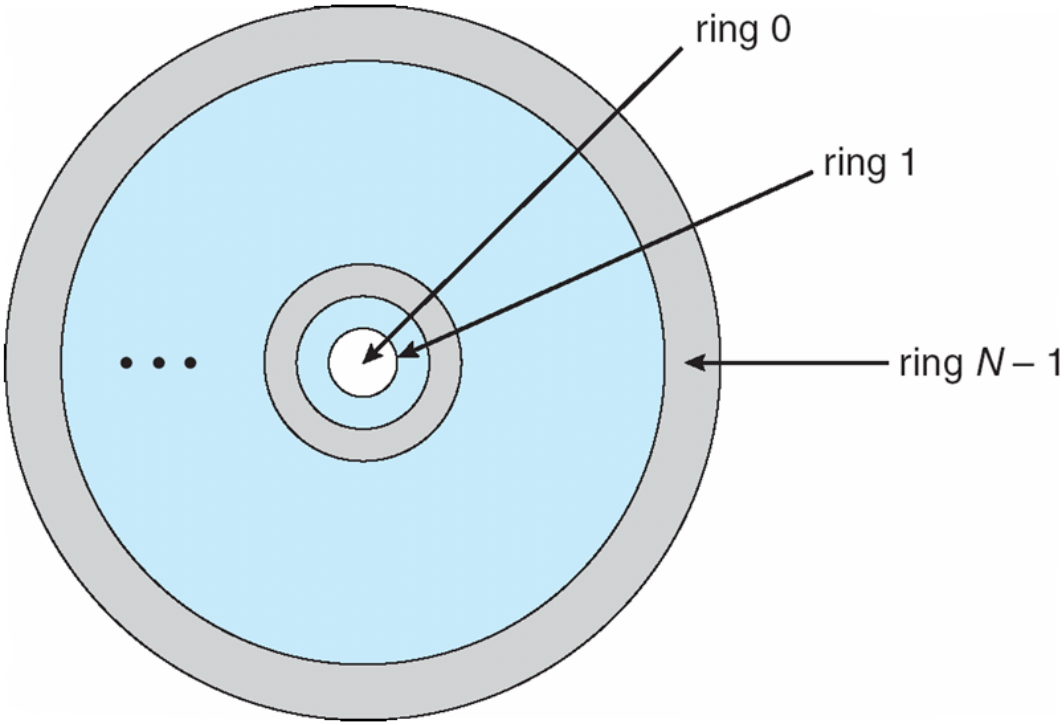
- **File**. A file virus infects a system by **appending itself to a program file**. It changes the start of the program to jump to its code. After it executes, it returns control to the program so that its execution is not noticed.
- **Boot**. A boot virus **infects the boot sector of the system**, executing every time the system is booted and **before the operating system is loaded**. At system boot, virus decreases physical memory limit and hides in memory above that limit.
- **Macro**. Most viruses are written in a low-level language, such as assembly or C. Macro viruses **are written in a high-level language**, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file


# Goal of Protection

- The role of protection in a computer system is to provide a **mechanism** for the enforcement of the **policies** governing resource use.
- **Policies** decide **what** will be done.
- **Mechanisms** determine **how** something will be done.


# Protection Rings

- The OS kernel is a trusted and privileged component and therefore must run with a higher level of privileges than user processes. To carry out this **privilege separation**, **hardware support** is required.
- The **protection rings** is a popular model of privilege separation where the **innermost** ring, ring 0 provides the **full set of privileges**.
- The system boots to the highest privilege level. Code at that level performs necessary initialization before dropping to a less privileged level.




# Portal Between Rings

- Code usually calls **a special instruction**, sometimes referred to as a gate, which provides a portal between rings.
- A **processor trap** (an interrupt) and **system calls** are examples of such gates to move to an inner ring (higher privilege level).
- Such gates allows the caller to specify only arguments (including the system call number), and **not arbitrary kernel addresses**. In this way, the integrity of the more privileged ring can generally be assured.


# Rings in Intel Architecture

- Intel architectures places user mode code in ring 3 and kernel mode code in ring 0.
- The distinction is made by two bits in the special **EFLAGS** register. Access to this register is not allowed in ring 3—thus preventing a malicious process from escalating privileges.
- With the advent of virtualization, Intel defined an additional ring (-1) to allow for **hypervisors**, or virtual machine managers, which create and run virtual machines.
- Hypervisors have more capabilities than the kernels of the guest operating systems.


# Rings in ARMv8 Architecture

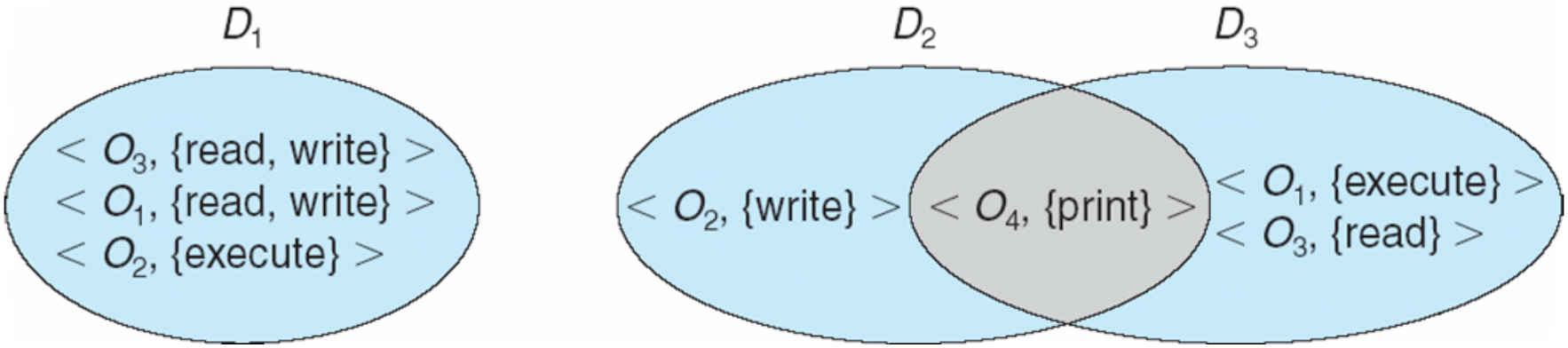
- ARMv8 supports four rings, called "exception levels," numbered EL0 through EL3. **User mode** runs in EL0, and **kernel mode** in EL1. EL2 is reserved for **hypervisors**, and EL3 (the most privileged) is reserved for the secure monitor (the **TrustZone** layer).
- The **secure monitor** runs at a higher execution level than general purpose kernels, which makes it the perfect place to deploy **code that will check the kernels' integrity**.
- **TrustZone** also has exclusive access to hardware-backed cryptographic features. Even the kernel itself has no access to the on-chip key, and it can only request encryption and decryption services from the TrustZone environment (by means of a specialized instruction **Secure Monitor Call** (SMC).


# Domain of Protection

- A computer system can be treated as a collection of processes accessing different objects.
  - Hardware objects examples: CPU, memory segments, printers, and disks.
  - Software objects examples: files and programs.
- A process should be allowed to access only those objects for which it has authorization.
- The ability to execute an operation on an object is an **access right**.
- A **domain of protection** is a collection of access rights, each of which is an ordered pair <object-name, rights-set>.
  - *Example:* if domain *D* has the access right <file *F*, {read,write}>, then **any process executing in domain *D*** can read and write file *F*. It cannot, however, perform any other operation on that object.

# Protection Domains with Shared Rights

- The figure shows a system with three protection domains: $D_1$, $D_2$, and $D_3$.
- The access right <$O_4$, {print}> is shared by $D_2$ and $D_3$, implying that a process executing in either of these two domains can print object $O_4$.




# User Domain in Unix

- Each ***user*** may be a separate domain. In this case, the set of objects that can be accessed depends on the identity of the user.
- Unix Domain switch accomplished via passwords
  - **su** (substitute user) command temporarily switches to another user's domain when other domain's password is provided.
  - It originally stood for "superuser" as that was the default user to switch to.
- Unix Domain switching via commands
  - **sudo** command prefix executes specified command in another domain (if original domain has privilege or password given).


# Access Matrix

- Access matrix is an abstract view of the general model of protection. As shown in the figure:
  - Rows represent domains
  - Columns represent objects
  - **`access(i, j)`** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

- We must ensure that a process executing in domain $D_i$ can access only those objects specified in row *i*, and then only as allowed by the access-matrix entries.

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read<br>write | | read<br>write | |


# Domains Switching

- When we switch a process from one domain to another, we are executing an operation (**`switch`**) on an object (the target domain in this case).
- We can control domain switching by including the domains themselves among the objects of the access matrix as shown below.
  - Example: a process executing in domain $D_1$ can switch only to domain $D_2$

| domain \ object | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | read write | | read write | | switch | | | |


# Code Signing

- Code signing is used for operating-system distributions, updates, and third-party tools alike.
- How can we be sure these tools were not modified on their way from where they were created to our systems?
- Currently, code signing is the best tool in the protection arsenal for solving these problems.
- **Code signing** is the digital signing of programs and executables to confirm that they have not been changed since the author created them and it uses a **cryptographic hash** to test for integrity and authenticity.
- It can also enhance system functionality in other ways.
  - For example, Apple can disable all programs written for a now-obsolete version of iOS by stopping its signing of those programs when they are downloaded from the App Store.


# Readings and Resources List

- From Silberschatz, Galvin, and Gagne "Operating System Concepts", 10/e textbook:
  - Chapter 16: 16.4 Cryptography as a Security Tool
  - Chapter 17: 17.8 Role-Based Access Control
  - Chapter 17: 17.9 Mandatory Access Control (MAC)
  - Chapter 17: 17.11 Other Protection Improvement Methods