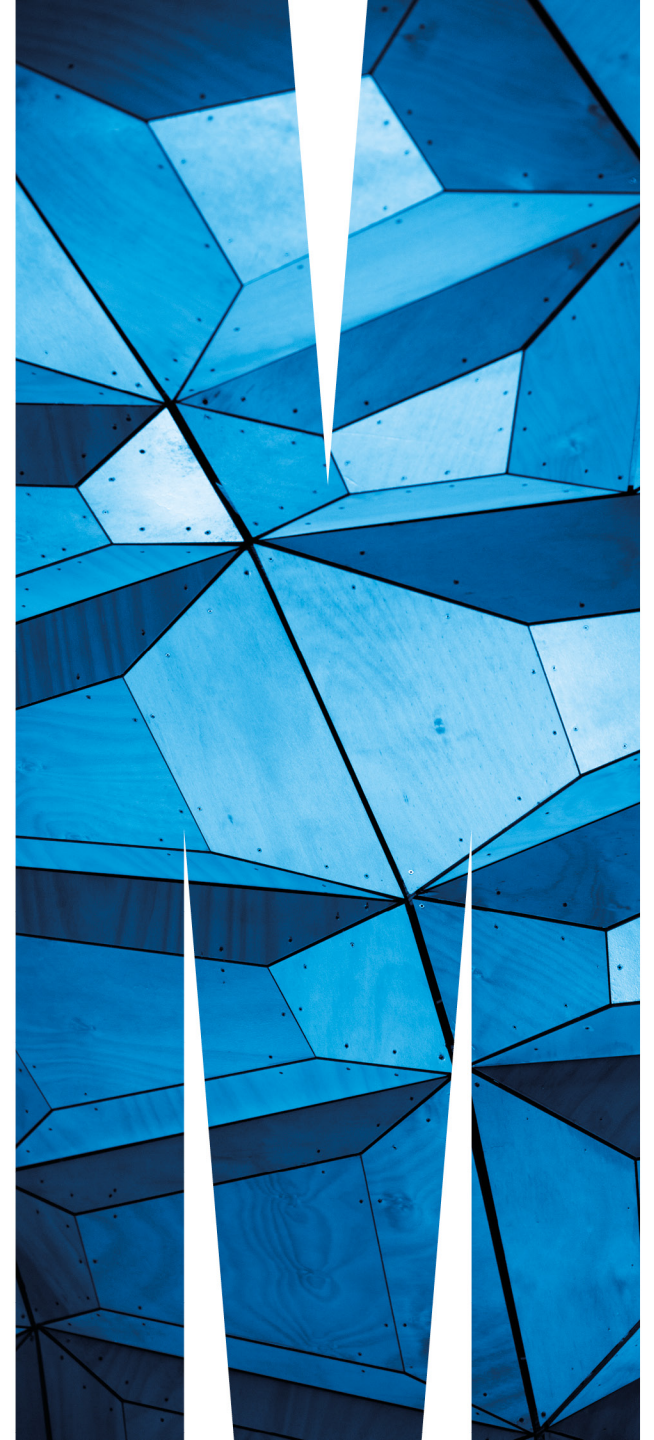


# FIT2100 Semester 2 2017

## Lecture 12: Operating System Security (Reading: Stallings, Chapter 15)

Jojo Wong

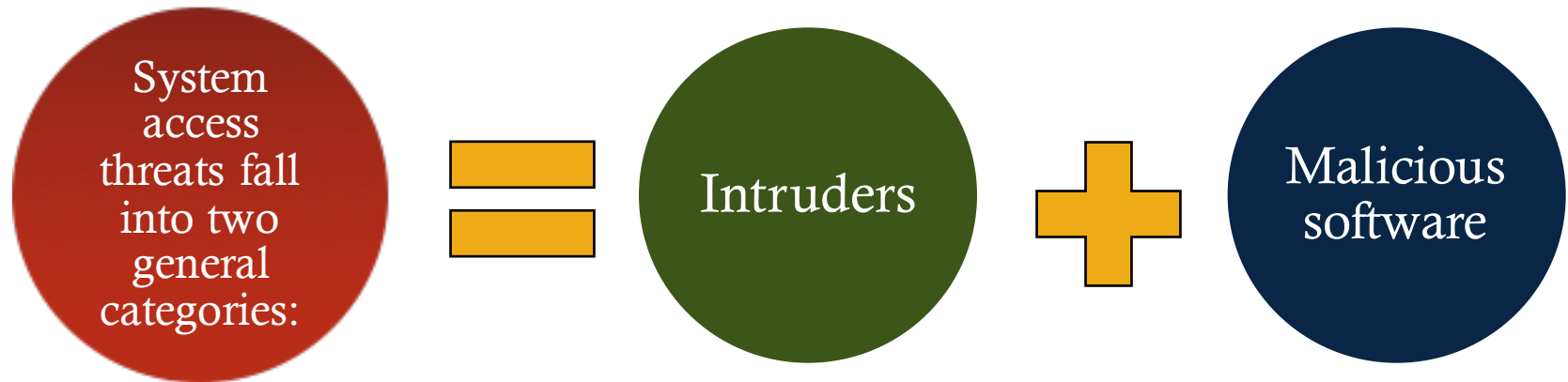


# Lecture 12: Learning Outcomes

- Upon the completion of this lecture, you should be able to:
  - Assess the key security issues related to operating systems (**intruders** and **malicious software**)
  - Discuss some common countermeasures for security threats
  - Compare and contrast two methods of **access control**
  - Understand how to defend against **buffer overflow** attacks

What are the common threats to system security?

# System Access Threats



# Intruders

## Masquerader

an individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account

## Misfeasor


a legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges

## Clandestine user

an individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

# Malicious Software

- Programs that exploit *vulnerabilities* in computing system
- Also referred to as **malware**
- Can be divided into two categories:
  - **Parasitic**
    - fragments of programs that cannot exist independently of some actual application program, utility, or system program
  - **Independent**
    - self-contained programs that can be scheduled and run by the OS
  - **Replication**
    - Threat fragments/programs replicate themselves



Viruses, logic bombs, and backdoors



Worms and bot programs



MONASH  
University

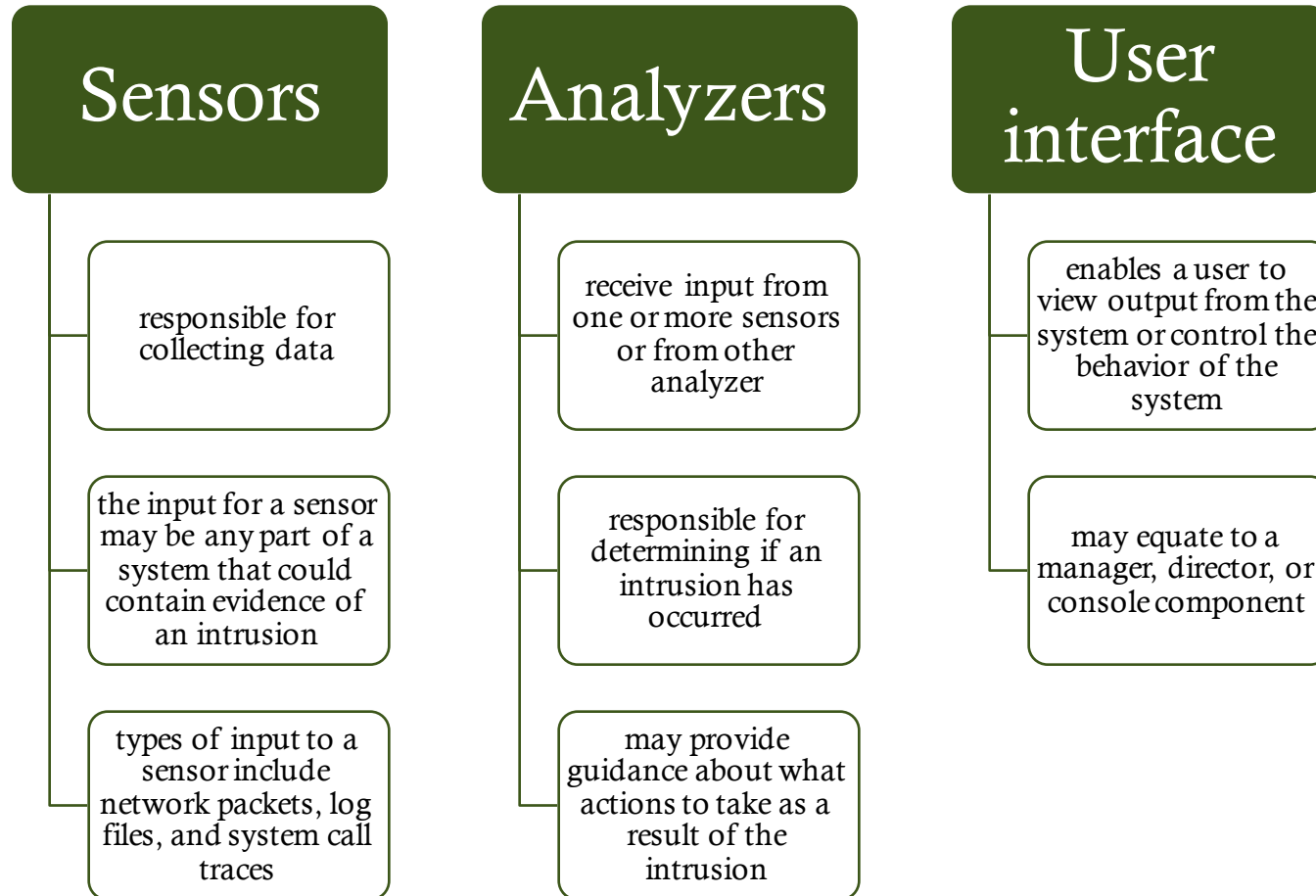
What are the security  
countermeasures?

# Intrusion Detection

- **RFC 4949** (*Internet Security Glossary*) defines **intrusion detection** as:
  - A security service that monitors and analyses system events for the purpose of finding, and providing real-time or near real-time warning of, attempts to access system resources in an unauthorised manner
- Intrusion detection systems (IDSs) can be classified as:
  - **Host-based IDS**: monitors the characteristics of a single host and the events occurring within that host for suspicious activity
  - **Network-based IDS**: monitors network traffic for particular network segments or devices and analyses network, transport, and application protocols to identify suspicious activity



# Intrusion Detection System: Components



# Firewalls

- **Protecting** a local system or network of systems from network-based security threats — while affording access to the outside world via wide area networks and the Internet
- **Design goals:**
  1. The firewall acts as a **choke point** — all incoming traffic and all outgoing traffic must pass through the firewall
  2. The firewall enforces the local security policy — defines the traffic that is authorised to pass
  3. The firewall is secure against attacks

If the firewall has been compromised, the system will be susceptible to attacks

- **User authentication** is the fundamental building block and the primary line of *defense*
- **RFC 4949** defines user authentication as:
  - The process of verifying an identity claimed by or for a system entity
- An authentication process consists of two steps:
  1. **Identification step** — presenting an *identifier* to the security system
  2. **Verification step** — presenting or generating authentication information that corroborates the binding between the entity and the identifier

# Authentication: Approaches

- Something the individual **knows**
  - E.g. a password, a personal identification number (PIN), or answers to a pre-arranged set of questions
- Something the individual is (**static biometrics**)
  - E.g. recognition by fingerprint, retina, and face

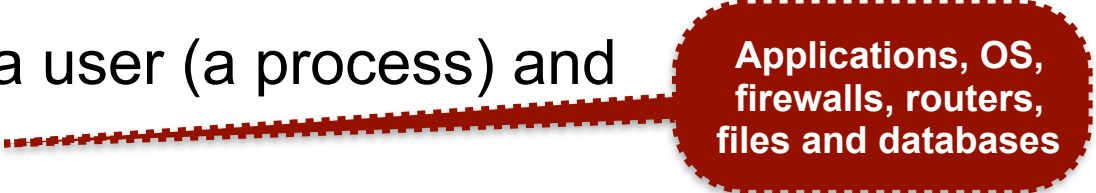
- Something the individual **possesses**
  - E.g. electronic keycards, smart cards, and physical keys
- Something the individual does (**dynamic biometrics**)
- E.g. recognition by voice pattern, handwriting characteristics, and typing rhythm



Token

What are the approaches to access control?

# Access Control

- Specifies **who or what** may have access to each specific **system resource** and the type of access that is permitted in each instance
- *Mediates* between a user (a process) and system resources 

Applications, OS, firewalls, routers, files and databases
- Security administrator maintains an **authorisation database** — specifies what type of access to which resources is allowed for a user
  - **Access control function** consults the database to determine whether to grant access
  - **Auditing function** monitors and keeps a record of user accesses to system resources

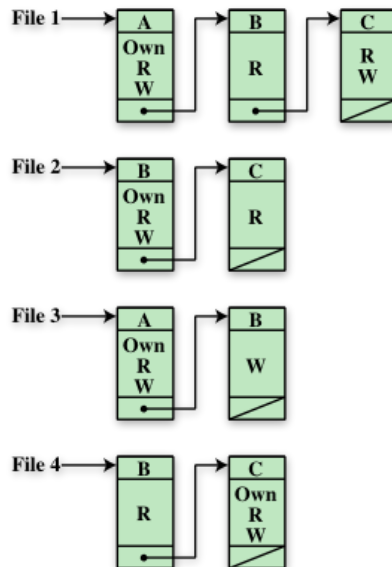
# Access Control: File System

- Identifies a user to the system
- A profile can be associated with each user — specifies permissible operations and file accesses
  - OS can then enforce rules based on the user profiles
- Database management system (DMS) must control access to specific records or even portions of records
- DMS decision for access depends on:
  1. the user's identity
  2. the specific parts of the data being accessed
  3. the information already divulged to the user

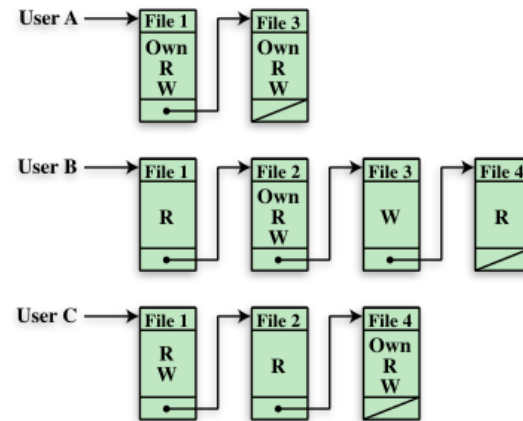
# Access Control: Data Structures

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry Credit	
User B	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
User C	R W	R		Own R W		Inquiry Debit

(a) Access matrix



(b) Access control lists for files of part (a)



(c) Capability lists for files of part (a)



# What is the buffer overflow attack?

- One of the most prevalent and dangerous types of security attacks
- Defined in the NIST Glossary of Key Information Security Teams as:

“A condition at an interface under which more input can be placed into a buffer or data-holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system”

# Basic Buffer Overflow: Example

Stack  
overflow

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

# Basic Buffer Overflow: Stack Values

Memory Address	Before gets(str2)	After gets(str2)	Contains Value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000 . . . .	01000000 . . . .	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf . . . .	08fcffbf . . . .	old base ptr
bffffbe4	00000000 . . . .	01000000 . . . .	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408 . . . .	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

# Buffer Overflow: Exploitation

- To exploit any type of buffer overflow, the attacker needs:
  1. To **identify a buffer overflow vulnerability** in some program that can be triggered using externally sourced data under the attackers control
  2. To **understand how that buffer will be stored in the processes memory** — hence the potential for corrupting adjacent memory locations and potentially altering the flow of execution of the program

# Buffer Overflow: Countermeasures

- Countermeasures to buffer overflow attacks can be broadly classified into two categories:
  1. **Compile-time defenses** — which aim to harden programs to resist attacks
  2. **Runtime defenses** — which aim to detect and abort attacks in executing programs

# Buffer Overflow: Defenses

## Compile-time

- Prevent or detect buffer overflows by instrumenting programs when they are compiled
- Possibilities:
  - choose a high-level language that does not permit buffer overflows
  - encourage safe coding standards
  - use safe standard libraries
  - include additional code to detect corruption of the stack frame

## Runtime

- Can be deployed in as OS updates and can provide some protection for existing vulnerable programs
- Involve changes to the memory management of the virtual address space of processes
- Changes act either to:
  - alter the properties of regions of memory; or
  - to make predicting the location of targeted buffers sufficiently difficult to thwart many types of attacks

# Buffer Overflow: Compile-Time Defenses

- Choice of programming language

- Write the program using a high-level programming that has a strong notion of variable type and what constitutes permissible operations on them

- Use of safe libraries

- Augment compilers to automatically insert range checks on pointer references

**Libsafe**

- Safe coding techniques

- Programmers need to inspect the code and rewrite any unsafe coding constructs — undertake an extensive audit of the existing code base

- Stack protection mechanisms

- Instrument the function entry and exit code to set up and then check its stack frame for any evidence of corruption

**Stackguard**



# Buffer Overflow: Runtime Defenses

- Executable address space protection
    - Block the execution of code on the stack — assuming that executable code should only be found elsewhere in the processes address space
  - Guard pages
    - Guard pages are placed between critical regions of memory in processes address space — flagged in MMU as illegal addresses
- Address space randomisation
    - Manipulate the location of key data structures in the address space of a process
    - Move the stack memory region around by a megabyte (or so) — make predicting the target buffer's address most impossible
    - Use a security extension that randomises the order of loading standard libraries by a program and their virtual memory address locations

How could we secure the operating system?

# Operating Systems Hardening

- Basic steps to use to secure the operating system (OS):
  - Install and patch the OS
  - Harden and configure the OS to adequately address the identified security needs of the system by:
    1. removing unnecessary services, applications, and protocols
    2. configuring users, groups and permissions (authentication)
    3. configuring resource controls
  - Install and configure additional security controls — such as antivirus, host-based firewalls, and intrusion detection systems (IDS)
  - Test the security of the basic OS to ensure that the steps taken adequately address its security needs

# OS Installation: Initial Setup and Patching

System security begins with the installation of the operating system



Ideally new systems should be constructed on a protected network



The initial installation should comprise the minimum necessary for the desired system, with additional software packages included only if they are required for the function of the system



The overall boot process must also be secured



Care is also required with the selection and installation of any additional device driver code, since this executes with full kernel level privileges, but is often supplied by a third party

# Summary of Lecture 12

- The most prominent issue for OS security is countering threats from **intruders** and **malicious software**.
- Intruders attempt to gain unauthorised access to system resources.
- Malicious software is designed to penetrate system defences and become executable on target systems.
- Countermeasures include **intrusion detection systems**, **authentication protocols**, **access control mechanisms**, and **firewalls**.
- **Buffer overflow** is one of the most common techniques for compromising OS security and a variety of compile-time and runtime defenses are used as countermeasures.