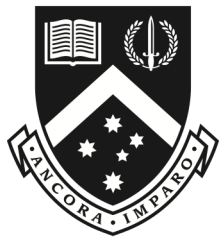


MONASH University

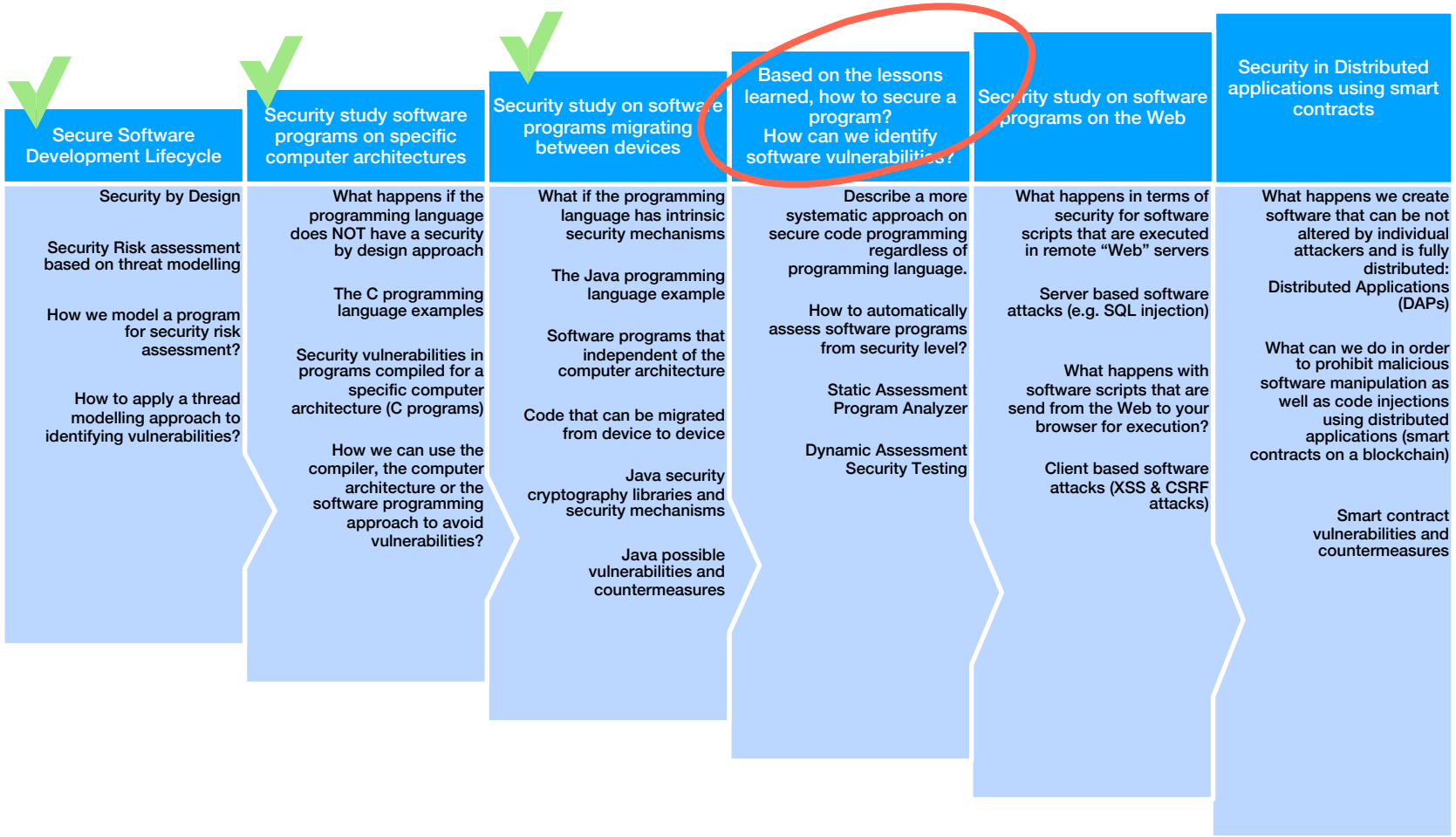
FIT5003: Secure Software Design and Code Practices

Dr Xiaoning Du
Department of Software Systems and Cybersecurity
Faculty of Information Technology

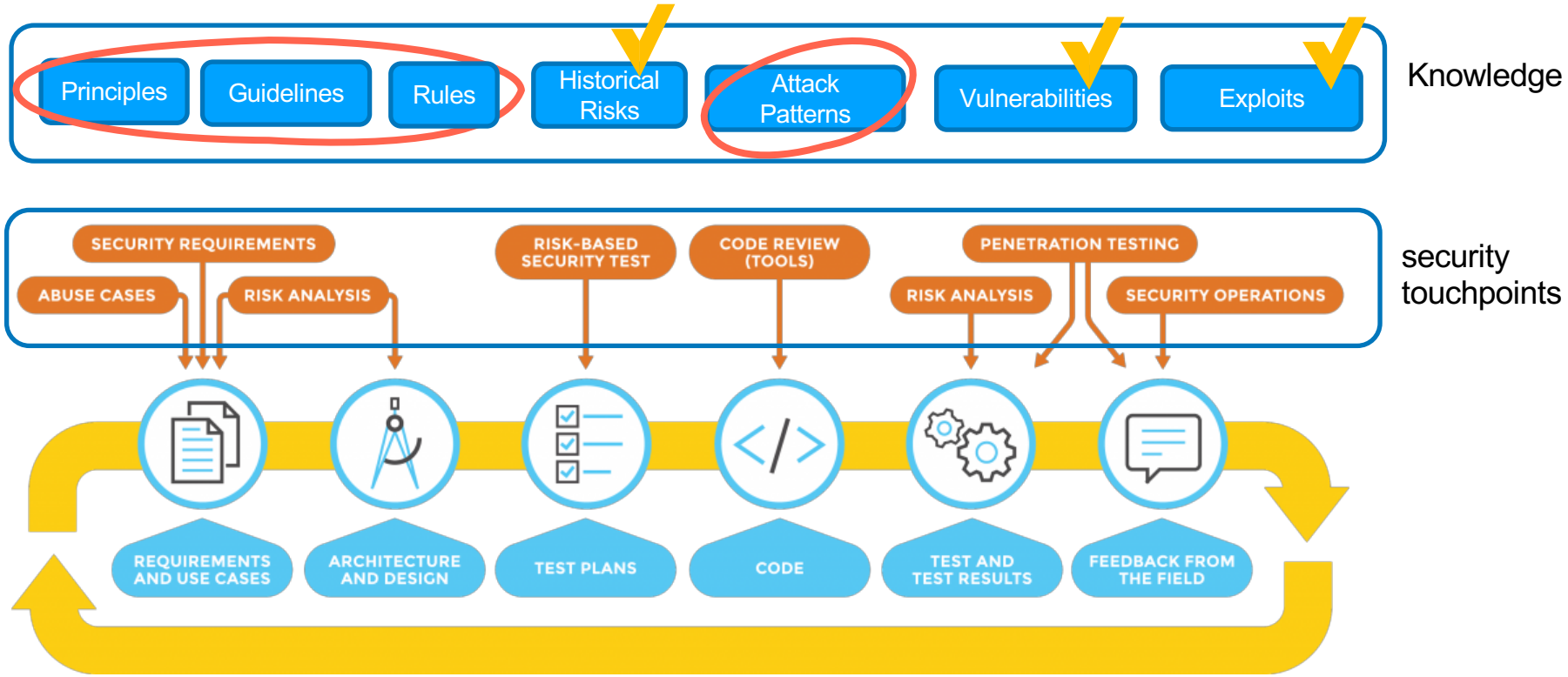
Learning Outcomes of This Lecture

- Generalize on the secure software design approaches
- Describe a generic attack methodology and pattern
- Introduce a secure software **design** principles
- Provide secure **coding** principles

Revisiting the Unit structure



Secure Software Development Lifecycle



MONASH University

Attack Methodology and Patterns

What does a typical program look like and what valuables does it contain?

What is the adversary's motivation for attacking your program?

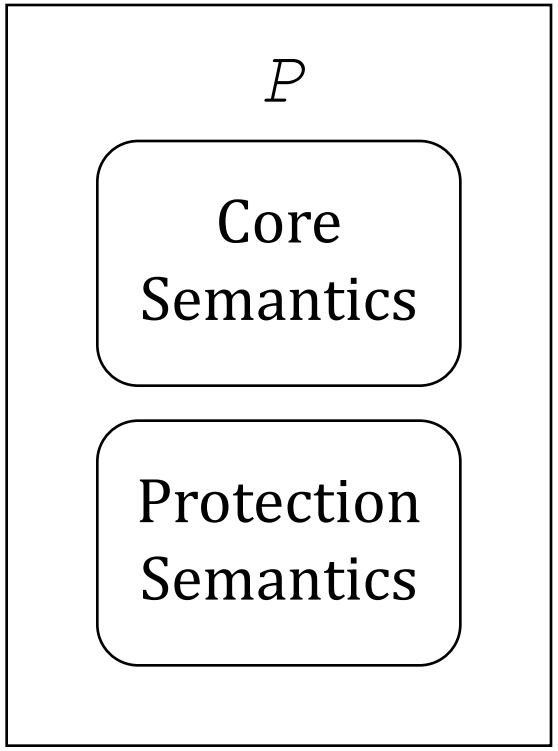
What information does he start out with as he attacks your program?

What is his overall strategy for reaching his goals?

What tools does he have at his disposal?

What specific techniques does he use to attack the program?

Attack Methodology and Patterns



Stripped executable do not include debugging information and other data included which is not necessary for execution in order to reduce the size of the executable

Stripped

`scanf, printf`

Unstripped

When the executable contains debugging information.

Eg.: If you compile an executable with gcc's -g flag, it contains debugging information.

`scanf, printf`

`test,…`

Architecture Neutral Format

Many languages, for example, Java and C#, are compiled not to native binary code (as C programs), but to an architecture-neutral format.

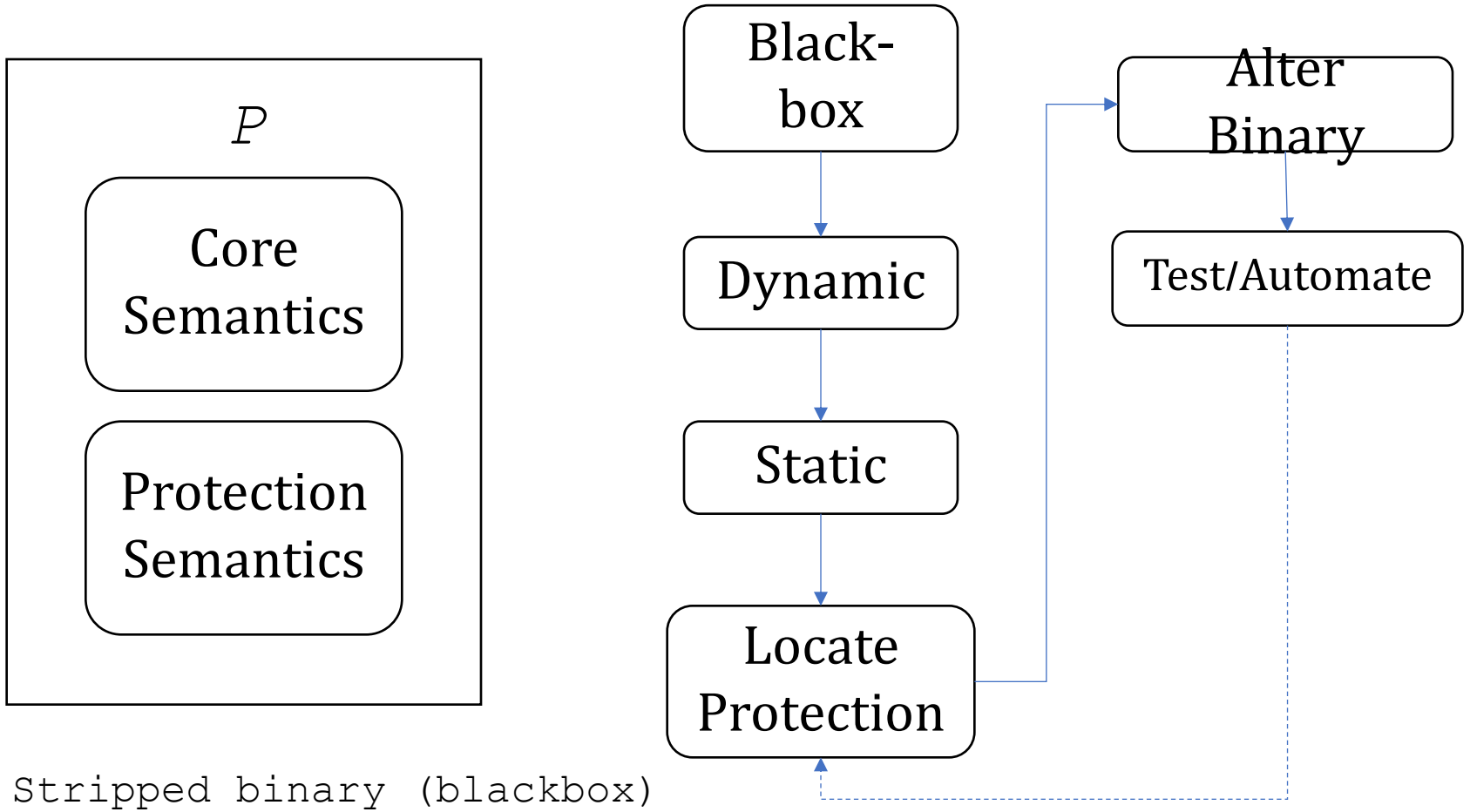
Eg. every class X.java is compiled into a classfile X.class that contains a symbol table and, for every method, a list of its bytecode instructions.

In essence, Java executables are unstripped

Which means they provide a lot of information to the attacker

Eg. Java Decompiler (java-decompiler.github.io)

Attack Methodology Patterns



Attack Phases

1. **black box** phase, the attacker treats the program as an "*oracle,*" feeding it inputs, recording its outputs, and drawing conclusions about its *external* behavior.
2. In the **dynamic analysis phase**, the attacker gets a first glimpse of the program's *internal* behavior. Again, he executes the program, but this time he records which parts of it get executed for different inputs.
3. **static analysis phase** where the attacker analyzes the program by examining the executable code directly
4. **editing phase** where the attacker uses his understanding of the internals of the program to modify the executable
5. **In the automate phase**, the attacker encapsulates his knowledge of the attack in an automated *script* that can be used, with little or no manual intervention, in future attacks

Attack Techniques for identifying code

- Learning about the executable and its function
- Static Pattern Matching
- Dynamic Pattern Matching
- Watching Memory
- When appropriate tools are available: Decompilation

What the Defender can do?

Secure programming Design Practices and Secure Coding practices

Secure Software Design Good Practices

1. **Model threats.** Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases

2. **Define security requirements.** Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.

Define security requirements.

- Identify and document security requirements early in the development life cycle
- Make sure that subsequent development artifacts are evaluated for compliance with those requirements.
- When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.
- OWASP: “A security requirement is a statement of needed security functionality that ensures one of many different security properties of software is being satisfied. Security requirements are derived from industry standards, applicable laws, and a history of past vulnerabilities”

Some standards with requirements:

- ISO/IEC 27034; https://www.iso.org/obp/ui/#iso:std:iso-iec:27034:-1:ed-1:v1:en
- OWASP Application Security Verification Standard; https://owasp.org/www-project-application-security-verification-standard/
- SAFECode “Fundamental Practices”; https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf
- SAFECode “Tactical Threat Modeling”;
- SAMM; BSIMM; CWSS; CAPEC;
- OWASP Threat Modeling Cheat Sheet

Security Requirement Example

Category	Subcategory	Diagnostic Statement	Comments on Implementation	Relevant Standards and Informative Resources
SECURE DEVELOPMENT				
Secure Coding (SC) *(continued)*	**SC.2.** Software is developed according to recognized, enforceable coding standards.	**SC.2-1.** Standards are formally identified and documented.		ISO/IEC TS 17961; SEI CERT C Coding Standard; SEI CERT C++ Coding Standard; SEI CERT Java Coding Standard; NCSC
		SC.2-2. Software uses canonical data formats.		SAFECode "Fundamental Practices"; CWE-21; CWE-22; CWE-35; CWE-36; CWE-37; CWE-38; CWE-39; CWE-40
	SC.3. The software is secure against known vulnerabilities, unsafe functions, and unsafe libraries.	**SC.3-1.** Software avoids, or includes documented mitigations for, known security vulnerabilities in included functions and libraries.	Software should avoid known vulnerabilities to the greatest extent possible. In some instances, there may be reasons for software to incorporate functions or libraries known to include vulnerabilities; such functions or libraries should only be incorporated when developers include documented mitigations that ensure the vulnerabilities are not exploitable.	NIST NVD; CWE/SANS Top 25 Most Dangerous Software Errors; OWASP Top 10; CWE-1006; CWE-242
		SC.3-2. Software validates input and output to mitigate common vulnerabilities in software.		SAFECode "Fundamental Practices"; OWASP Input Validation Cheat Sheet; CWE-20; CWE-89; CWE-119; CWE-120; CWE-183; CWE-184; CWE-242; CWE-625; CWE-675; CWE-805
		SC.3-3. Software encodes data and/ or uses anti-cross site scripting (XSS) libraries.		SAFECode "Fundamental Practices"; CWE-79

Secure Design Principles

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Secure Design Principles:
economy of mechanism

- **Keep it simple.** Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Secure Design Principles:
Fail-safe defaults

- **Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Secure Design Principles:
Least Privilege policy

- Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Compromise recording: it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Compromise recording: it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

Defense in depth: design the system so that it can resist attack even if a single security vulnerability is discovered or a single security feature is bypassed. Defense in depth may involve including multiple levels of security mechanisms or designing a system so that it crashes rather than allowing an attacker to gain complete control.

Defense in depth

- **Practice defense in depth.** Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit.

- For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Compromise recording: it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

Defense in depth: design the system so that it can resist attack even if a single security vulnerability is discovered or a single security feature is bypassed. Defense in depth may involve including multiple levels of security mechanisms or designing a system so that it crashes rather than allowing an attacker to gain complete control.

Fail securely: a counterpoint to defense in depth is that a system should be designed to remain secure even if it encounters an error or crashes.

Secure Design Principles

Economy of mechanism: keep the design of the system as simple and small as possible.

Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).

Complete mediation: every access to every object must be checked for authorization.

Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.

Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Compromise recording: it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

Defense in depth: design the system so that it can resist attack even if a single security vulnerability is discovered or a single security feature is bypassed. Defense in depth may involve including multiple levels of security mechanisms or designing a system so that it crashes rather than allowing an attacker to gain complete control.

Fail securely: a counterpoint to defense in depth is that a system should be designed to remain secure even if it encounters an error or crashes.

Design for updating: no system is likely to remain free from security vulnerabilities forever, so developers should plan for the safe and reliable installation of security updates

Secure Design Principles:

Standardize Identity and Access Management

- This provides consistency between components as well as clear guidance on how to verify the presence of the controls.

- What to include/consider?
  - The mechanism by which users (both end-users and organization administrators) authenticate their identities.
  - The mechanism(s) by which one service or logical component authenticates to another, how the credentials are stored, and how they are rotated in a timely fashion
  - The mechanism(s) that authorizes the actions of each principal. → Complete mediation
- Authorization/authentication is linked to:
  - The lease privilege policy
  - Fail-safe defaults, and
  - complete mediation

Secure Design Principles:

Establish Log Requirements and Audit Practices

- Use the operating system logging mechanism to capture the status changes of the software and record software critical events
- What is a security critical event?

- The content of the log files should always be determined by the group or groups that will need to consume the log file contents.

- It is important not only to capture the critical information but to restrict information capture to only the needed data.
- It is equally important to carefully identify what security information is relevant and needs to be logged, where the logs will be stored, for how long the logs will be retained and how the logs will be protected

Why is logging and auditing needed?

- In the event of a security-related incident, it is important to be able to piece together relevant details to determine what happened.
- Well-designed application, system and security log files provide the ability to understand an application's behavior and how it has been used at any moment in time.
- They are the fundamental data sources that inform automated Security Information and Event Management (SIEM) systems alerting.
- Example: Linux logging
  - **/var/log/syslog** or **/var/log/messages**: Generic log mechanism
  - **/var/log/auth.log** or **/var/log/secure**: store authentication logs, including both successful and failed logins and authentication methods.
  - **/var/log/boot.log**: a repository of all information related to booting and any messages logged during startup.
  - **/var/log/kern**: stores Kernel logs and warning data. This log is valuable for troubleshooting custom kernels as well.
  - **/var/log/dmesg**: messages relating to device drivers. The command **dmesg** can be used to view messages in this file.
  - **/var/log/faillog:** contains information all failed login attempts, which is useful for gaining insights on attempted security breaches, such as those attempting to hack login credentials as well as brute-force attacks.
  - **/var/log/mysqld.log** or **/var/log/mysql.log**

What about the code?

Secure Coding practices

MONASH
University

Secure Coding Practices

Establish Coding Standards and Conventions:

- use built-in security features in the frameworks and tools selected and ensure that these are on by default
- Create a secure code writing policy and enforce it to the software development team. In principle there will be mostly the same for various programming language. It might need some fine tuning for each programming language.

- **Architect and design for security policies.** Create a software architecture and design to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.

- There are secure coding standards to help
- Eg.: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

Secure Coding Practices

Handle Data securely→ Input Validation

- **All user-originated input should be treated as untrusted**

1. Validate input:

Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files

- Ensure that each area in the application stack defends itself against the malicious input it may be vulnerable to.
- Even with input validation, vulnerabilities remain as a result of processing data in unsafe ways, and therefore input validation should be considered with the defense in depth approach.
- Additional steps:
  - **Encoding,** which ensures the data is transformed so that it will be interpreted purely as data in the context where it is being used
  - **Data binding**, which prevents data from being interpreted as control logic by binding it to a specific data type

Secure Coding Practices

Input Validation

- Validation can be broken down into **checking origin, data size, lexical content, syntactic format, and semantics**.
- ***Origin checks*** can be done by checking the origin IP or requiring an access key to counteract DDoS attacks.
- ***Data size checks*** can be done both at the system border and at object creation.
- ***Lexical content checks*** can be done with a simple regular expression (regexp).
- ***Syntax format checks*** might require a parser, which is more expensive in terms of CPU and memory.
- ***Semantic checks*** often require looking into a white or black list. E.g. look at the data in the database, such as searching for an entity with a specific ID
- Earlier steps in the validation order are more economical to perform and protect the later, more expensive steps. If early checks fail, later steps can be skipped

Secure Coding Practices

Is Input validation enough?

After initial input validation and filtering ….still attacks are possible:

Attackers use alternative formulation of data input that can bypass the validation process.

Thus, we need:

2. Canonicalization:
is the process for converting data that establishes how these various equivalent forms of data are resolved into a "standard," "normal" or canonical form. Canonical representation ensures that the various forms of an expression do not bypass any security or filter mechanisms

- Input validation should be done AFTER canonicalization.

3. Sanitization:
can involve removing, replacing or encoding unwanted characters or escaping characters. Input from untrusted sources should always be sanitized

- **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components.
- Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks..

Secure Coding Practices

Use Safe Functions Only

- Many programming languages have functions and APIs whose security implications were not appreciated when initially introduced but are now widely regarded as dangerous.
  - Eg. Strcpy, strcat in C is vulnerable to buffer overflow
  - Javascript functions generate new code at runtime (eval, seTtimeout, etc.) and are a frequent source of code execution vulnerabilities.
- The developers should counsel a security standard so that they avoid vulnerable functions

Secure Coding Practices

Handle Errors

- Generated Errors can be exploited by attackers:
  - Information about the program
  - Data input manipulation
  - Even secret information disclosure

- It's better to fail fast in a controlled manner than to risk uncontrolled failures later. Fail fast by checking preconditions early in each method.

- When notifying the user of an error, the technical details of the problem should not be revealed.
  - Details such as a stack trace or exception message provide little utility to most users, and thus degrade their user experience, but they provide insight to the attacker about the inner workings of the application
  - Error messages to users should be generic, and ideally from a usability perspective should direct users to perform an action helpful to them

Secure Coding Practices

Latest Compiler and Secure Compilers

1. **Using the latest versions of compilers, linkers, interpreters and runtime environments:** Commonly, as languages evolve over time they incorporate security features, and developers using previous compiler and toolchain versions cannot make use of these security improvements in their software

2. **Secure compiler options:** Enable secure compiler options and do not disable secure defaults for the sake of performance or backwards compatibility.

3. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws.

Beyond Secure Designing and Coding: Testing and Quality assurance

- **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities.
- Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program.
- Independent security reviews can lead to more secure systems.
- External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions

Beyond Secure coding: Security Testing/Auditing

- Static Security Analysis
- Dynamic Security Testing
- Audits
- Penetration Testing (red team)
- Run Time Monitoring