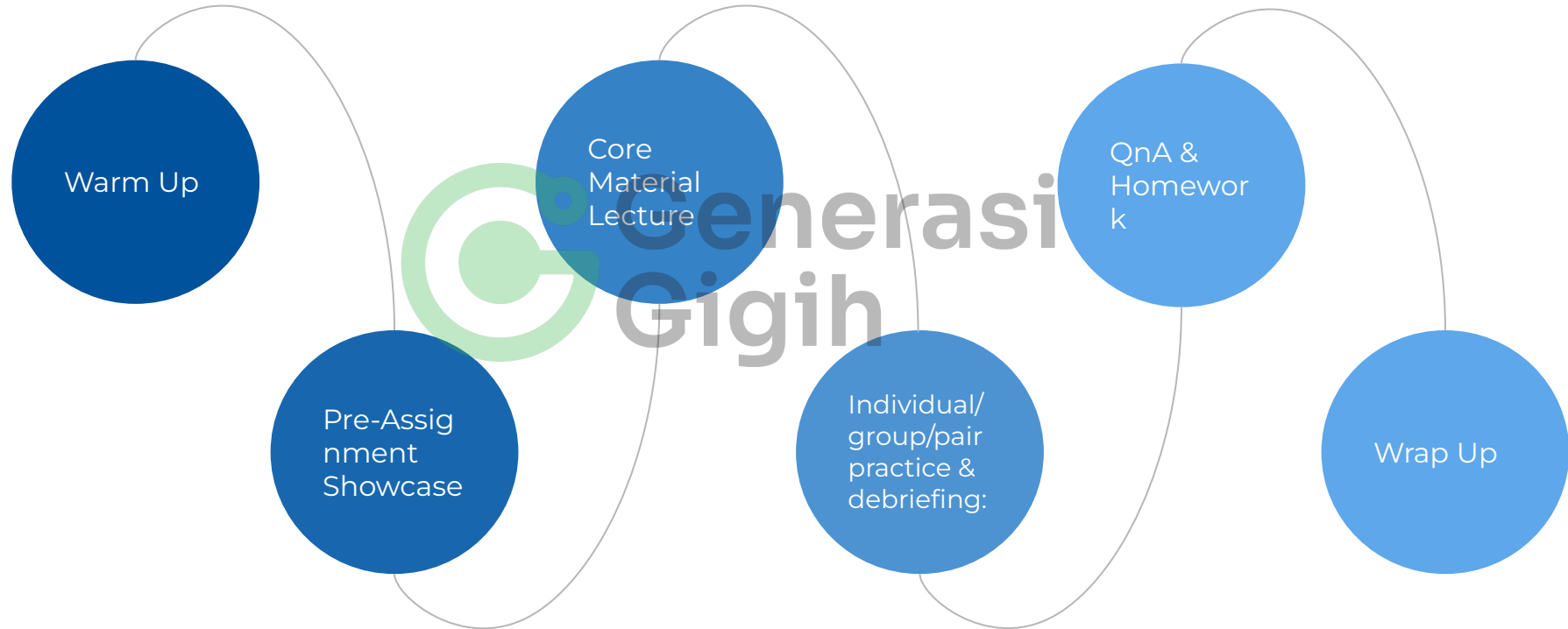


## Full Stack Engineer

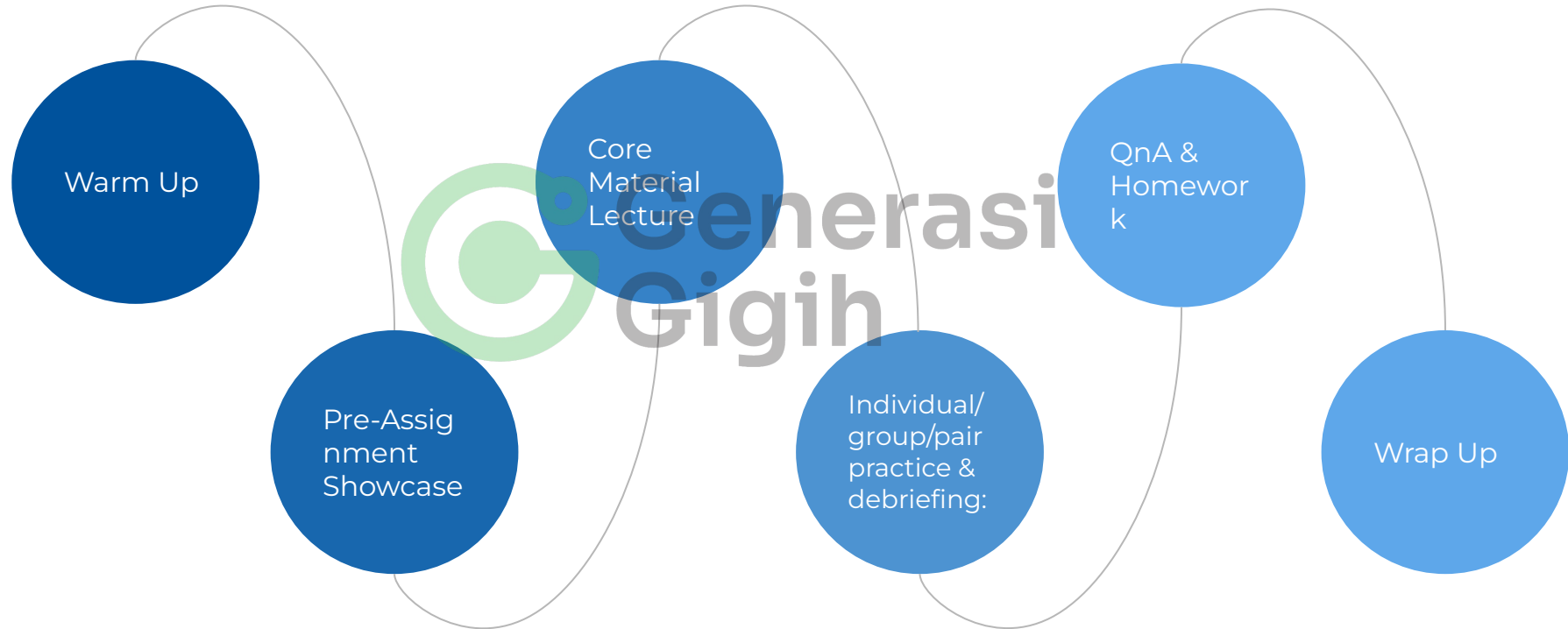
### Module 6.3: Introduction to Software Security



# Our Agenda



# Our Agenda



# Let's Warm Up!



# Let's Discuss



Generasi  
Gigih

[Session outline]

# Let's Talk About The Materials

# Software Security, Why?

1. Maintaining user trust
2. Protecting Organizational Reputation
3. Compliance and Legal Requirements
4. Business Continuity
5. Competitive Advantage
6. Ethical Responsibility

And many more.



# Fundamentals: CIA Triad

CIA Triad is the pillars of creating a secure software systems.

1. Confidentiality
2. Integrity
3. Availability





# Confidentiality

Confidentiality focuses on protecting sensitive data from unauthorized access.

Techniques such as **encryption**, **access controls**, and **secure authentication mechanisms** are used to achieve confidentiality.

Sample of Attack → Solution

- Network sniffing (e.g. Wireshark) → End-to-end Encryption (HTTPS, TLS)
- Unauthorized access → 2FA authentication

# Integrity

Integrity ensures the accuracy and trustworthiness of data and resources within the software system. It ensures that data remains unaltered and consistent throughout its lifecycle.

Sample of Attack → Solution

- Man-in-the-middle attack → End-to-end Encryption (HTTPS, TLS)

# Availability

Availability guarantees that the software and its resources are accessible and functional when needed

Sample of Attack → Solution

- Denial-of-Service (DoS/DDoS) → Scaling Infra, Blocking unusual traffic.

# Security in Software Development

Top security risks ([OWASP Top 10](#)):

1. [Broken Access Control](#)
2. [Cryptographic Failures](#)
3. [Injection](#)
4. [Insecure Design](#)
5. [Security Misconfiguration](#)
6. [Vulnerable and Outdated Components](#)
7. [Identification and Authentication Failures](#)
8. [Software and Data Integrity Failures](#)
9. [Security Logging and Monitoring Failures](#)
10. [Server-Side Request Forgery](#)

# Example: Injection

Injection, notably SQL, was on top of the list up until 2017 release of OWASP top 10. It can cause a catastrophic issue on the backend side of a web application with a simple attack.

Let's say you have this piece of code:

```
function login(username, password) {  
  let query = "SELECT * FROM users where username = '"+ username + "'  
  and password='"+password+"'";  
  console.log(query)  
  db.all(query, (err, row) => {  
    console.log(row)  
  })  
}
```

	username	password
	<input type="text" value="Search column..."/>	<input type="text" value="Search column..."/>
1	User0	Password0
2	User1	Password1
3	User2	Password2
4	User3	Password3
5	User4	Password4
6	User5	Password5
7	User6	Password6
8	User7	Password7
9	User8	Password8
10	User9	Password9

## Example: Injection (2)

```
//Standard Case
> login("User0", "Password0")
{ username: 'User0', password: 'Password0' }

//Standard Case - Wrong Password
> login("User0", "Password1")
{ username: 'User0', password: 'Password1' }
Undefined

//Injection, force login
> login("User0", "' OR 1=1;--")
{ username: 'User0', password: 'Password0' },
{ username: 'User1', password: 'Password1' },
{ username: 'User2', password: 'Password2' },
{ username: 'User3', password: 'Password3' },
{ username: 'User4', password: 'Password4' },
...
```

The query building code is injection prone.

We can make the query into:

```
SELECT * FROM users where username = 'User0' and
password=' ' OR 1=1;--';
```

This query injected with OR 1=1 which always return true hence it will return all the data in the table.

- - Double hyphen means comment on SQL, so the rest of the query will be commented out.

# Example: Injection (3) - Solutions

## Prepared Statement

```
function login(username, password) {  
  stmt = db.prepare("SELECT * FROM users WHERE username=? AND password=?");  
  stmt.each(username, password, (err, row) => {  
    console.log(row)  
  })  
}
```

Prepared statement create a clear distinction between query and parameters, so user won't be able to tamper with the query.

It is the default behavior for many modern framework, hence on 2021 SQL Injection risk level drop to No. 3

# Website Security



# Most Common Website Security Threats

1. Cross-Site Scripting (XSS)
2. SQL injection
3. Cross-Site Request Forgery (CSRF)
4. Other threats



# Cross-Site Scripting / XSS (1)

- **What** | attacks that allow an attacker to inject client-side scripts through the website into the browsers of other users
- **Risk** | the injected code comes to the browser from the site → the code is trusted
  - can do things like send the user's site authorization cookie to the attacker.
  - Attacker then can log into a site as though they were the user and do anything the user can

# Cross-Site Scripting / XSS (2)

- **Types**
  - Reflected XSS
  - Persistent XSS
- **Defense | input sanitation**
  - remove or disable any markup that can potentially contain instructions to run the code. For HTML this includes elements, such as `<script>`, `<object>`, `<embed>`, and `<link>`


# SQL Injection (1)

- **What** | attackers execute arbitrary SQL code on a database, allowing data to be accessed, modified, or deleted irrespective of the user's permissions
- **Risk** | spoof identities, create new identities with administration rights, access all data on the server, or destroy/modify the data to make it unusable



# SQL Injection (2)

user input that is passed to an underlying SQL statement can change the meaning of the statement. See SQL statement below



```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

## SQL Injection (3)

the modified statement below creates a valid SQL statement that:

- deletes the users table
- selects all data from the userinfo table (which reveals the information of every user).

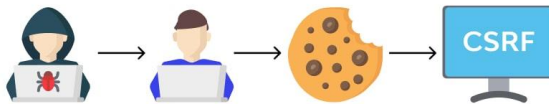
```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';
```

Defense | prepared statement / escape all the characters in the user input that have a special meaning in SQL.

# Cross-Site Request Forgery / CSRF (1)

- What | allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent
- Risk | spoof identities, create new identities with administration rights, access all data on the server, or destroy/modify the data to make it unusable

CSRF - Cross site request forgery  
attack



# Cross-Site Request Forgery /CSRF (2)

- How | best explained with example:
  - Attacker knows that a particular site allows **logged-in users to send money** to a specified account using an HTTP POST request that includes the account name and an amount of money.
  - Attacker constructs a form that includes his bank details and an amount of money as hidden fields, and emails it to other site users (with the Submit button disguised as a link to a "get rich quick" site).
  - Any user who clicks the Submit button while they are logged in to the trading site will make the transaction
  - Attacker gets rich
- Defense | the server to require that POST requests include a user-specific site-generated secret



# Others

- **Clickjacking** – hijacks clicks meant for a visible top-level site and routes them to a hidden page beneath
- **Denial of Service (DoS)** – flooding a target site with fake requests so that access to a site is disrupted for legitimate users
- **Directory Traversal (File and disclosure)** – attempts to access parts of the web server file system that they should not be able to access.
- **File Inclusion** – a user is able to specify an "unintended" file for display or execution in data passed to the server. When loaded, this file might be executed on the web server or the client-side (leading to an XSS attack)
- **Command Injection** – allow a malicious user to execute arbitrary system commands on the host operating system

# **Discussion/Hands On Assignment/Study Case about the material**

# Reminder

Give some examples of

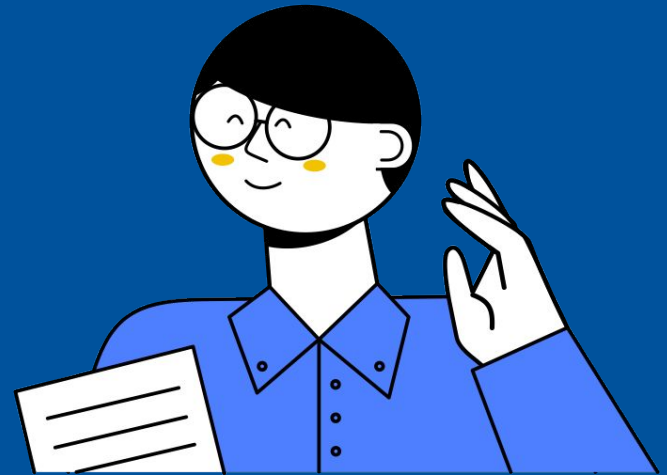
- CSRF
- XSS





# Showcase Time!

# Q&A!



# Finally, Let's Wrap Up!

**See you in the  
next session!**

