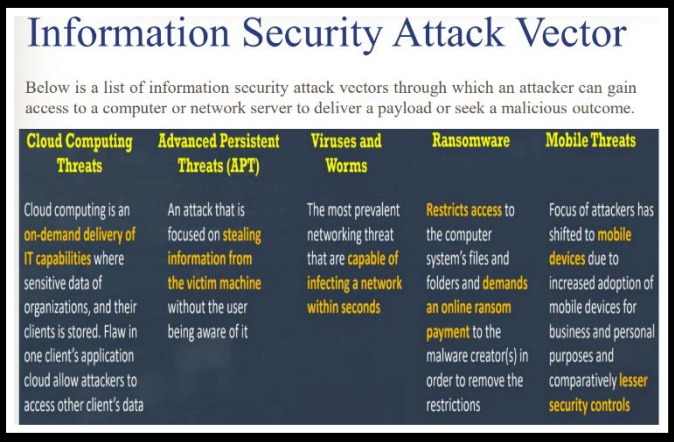
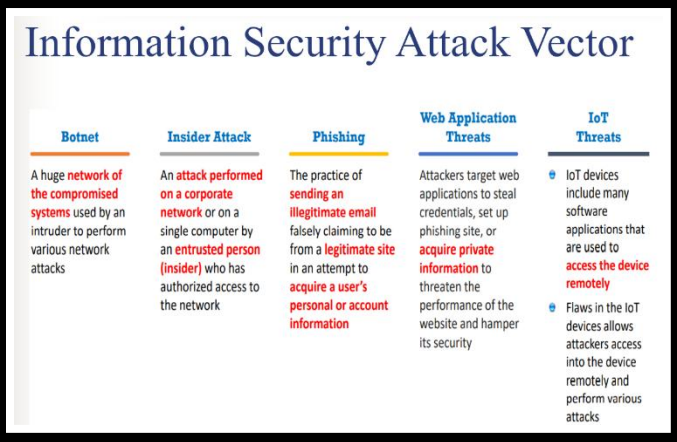
AST 1 Introduction.C





🏗️ Application Architecture – Summary Guide

Application Architecture is the structured design and organization of software components and their interactions. It defines how applications are built, deployed, scaled, and maintained.

🔑 Key Components of Application Architecture

Layer	Description
Presentation Layer	Manages the user interface (UI) and interactions. This is what users see — web pages, app screens, etc.
Business Logic Layer	Processes core application logic, rules, and workflows (e.g., checkout in an e-commerce app).
Data Layer	Handles data storage, retrieval, and database interaction (e.g., querying user profiles).

🧩 Types of Application Architectures

1. Model-View-Controller (MVC)

A widely-used pattern that separates application concerns into 3 components:

- **Model** – Manages data and logic (e.g., database interaction).
- **View** – Handles the display/UI (e.g., HTML page or UI component).
- **Controller** – Processes user input and updates the model or view.

🔄 **How MVC works together**:

User → View → Controller → Model → View

This separation allows **easier debugging, scaling, and testing**.

---

2. Microservices Architecture

Large applications are broken down into **independent services**, each responsible for a single function.

- Each microservice has **its own database and can be deployed separately**.
- Services **communicate via APIs** (e.g., REST/HTTP).

🧠 **Examples**:

- Netflix: Uses microservices to stream content and handle user accounts independently.
- Amazon: Manages product catalog, payment, and user login as separate services.

✅ Pros: Scalable, fault-isolated, easy to update.

⚠️ Cons: Complex management and deployment strategy.

---

3. Serverless Architecture

Serverless means developers write code without managing infrastructure.

- Code runs as **functions (FaaS)** or **backend services (BaaS)** triggered by events.
- Cloud providers (e.g., AWS Lambda, Azure Functions) handle execution.

🧠 **Examples**:

- Netflix: Uses serverless for video processing.
- Slack: Uses it for notifications and file uploads.

✅ Pros: Scalable, low maintenance.

⚠️ Cons: Debugging and cold start latency can be issues.

---

4. Single Page Application (SPA)

SPAs load a single HTML page and dynamically update content via JavaScript — no full-page reloads.

- Initial page load fetches HTML/CSS/JS.
- Further interactions load JSON data via **AJAX or Fetch API**.
- Routing is handled on the client side (e.g., React Router).

🧠 **Examples**:

- Gmail: Loads once; switching between emails is seamless.
- Google Maps: Dynamic zoom/search with no page reload.
- Facebook, Trello: Live updates without refreshing the page.

✅ Pros: Fast, responsive, fluid UI.

⚠️ Cons: Harder to secure and optimize for SEO.

🆕 **BONUS: Trends in Modern Web Architecture**

- **Legacy Monoliths** combined server-side scripts, HTML, and CSS into one file.
- **MVC patterns** helped separate concerns and improve maintainability.
- **Microservices** decoupled apps for independent deployment.
- **SPAs** offered better UX but required stronger client-side security.
- **Serverless** simplified backend concerns and improved scalability for event-driven systems.

🛡️ **Application Security Concepts**

- Definition of Application Security
- Need for Secure Development Life Cycle (SDLC)
- Integrating security in each phase of SDLC
- Secure coding practices
-

🔍 **Types of Security Testing**

- **Black Box Testing**: No internal knowledge
- **White Box Testing**: Full access to source code
- **Gray Box Testing**: Partial knowledge of internals

🔐 **Web Application Firewall (WAF)**
Monitors HTTP traffic and blocks common attacks like SQLi or XSS.

🧠 **Runtime Application Self-Protection (RASP)**
Acts inside the app to monitor and block malicious behavior in real-time.

📦 **Software Composition Analysis (SCA)**
Scans open-source libraries and components for known vulnerabilities.

📄 **Static Application Security Testing (SAST)**
White-box testing; scans code for flaws before deployment.

⚙️ **Dynamic Application Security Testing (DAST)**
Black-box testing; simulates external attacks on running applications.

🔬 **Interactive Application Security Testing (IAST)**
Combines SAST + DAST from within a running app using sensors and agents.

📱 **Mobile Application Security Testing (MAST)**
Focuses on mobile-specific threats like insecure storage, rooting, or rogue apps.

☁️ **CNAPP (Cloud-Native Application Protection Platforms)**
Unifies cloud workload protection and posture management for DevOps pipelines.

AST 9 OAuth Authentication.C

OAuth

🧩 **PART 1: BROKEN AUTHENTICATION (OWASP A07:2021)**

🔐 What is Broken Authentication?

Broken Authentication is one of the most dangerous and common web vulnerabilities outlined in the **OWASP Top 10 2021**, categorized under **A07: Identification and Authentication Failures**. This issue arises when the authentication mechanism of a web application is poorly implemented or misconfigured, allowing attackers to bypass login mechanisms, hijack sessions, or impersonate users. Authentication is the process by which a system verifies the identity of a user, typically through credentials like usernames and passwords. However, if any component of this process—such as credential handling, session management, or password recovery—is weak or broken, it becomes an attack vector.

A typical web application creates a **session ID** once a user logs in. This ID is used to maintain the user's logged-in state across different pages. Broken Authentication occurs when that session ID is exposed, guessable, or not invalidated properly, enabling attackers to hijack sessions or access accounts without proper credentials. The core of the problem lies in how identity is established and managed over time, especially after authentication.

🚪 Two Main Categories of Vulnerability

1. Poor Session Management

Session management refers to how a web application controls and protects a user's session after they log in. In a vulnerable application, session IDs are predictable, not rotated after login, or not tied to the user's context (like IP address or browser fingerprint). This can allow attackers to hijack the session and act as the legitimate user.

One form of session attack is **Session Hijacking**, where an attacker uses a stolen or leaked session ID to gain access to the application as another user. For instance, if a user logs into an application but does not log out and walks away from their computer, an attacker could take over the session. Another common vulnerability is **Session ID URL Rewriting**, where session tokens are embedded in URLs instead of cookies. If someone shares or logs such a URL, it could expose the session token to others. This is how early vulnerabilities in Zoom led to "Zoombombing."

2. Poor Credential Management

Credential management flaws involve weak or insecure handling of user login details. This includes issues like **credential stuffing**, where attackers use leaked username-password pairs (from data breaches) and try them on different websites, taking advantage of password reuse. Another threat is **password spraying**, which uses common passwords like "123456" or "password" across many user accounts rather than brute-forcing a single account. This works effectively when password complexity is not enforced. Lastly, **phishing attacks** are used to deceive users into providing their login credentials to attackers via fake websites that resemble the original service.

💉 Practical Exploitation: Cookie Manipulation Attack

The document includes an example of exploiting a broken authentication vulnerability using **cookie-based session manipulation**. The attack begins by creating an account on the vulnerable web application. Using a proxy tool such as **Burp Suite**, the attacker intercepts the login request to analyze the response headers or cookies issued by the server. These cookies often include a session or user ID token.

In this case, the server issued a UserID cookie that uniquely identified the user. By modifying this UserID value (for example, incrementing it numerically), the attacker could brute-force other valid session IDs. Each attempt would involve sending a modified cookie with a different UserID value and observing the server's response. If a request results in a 200 OK response, it indicates a valid session—successfully impersonating another user. The document cites an example where the UserID value 10411 mapped to a user named usAndyPaul, whose default password was simply "PASSWORD", demonstrating the compounding danger of weak credential practices.

This kind of session manipulation allows attackers to enumerate users, access their profiles, and potentially extract sensitive information — all without needing to guess or break passwords.

⚠️ Impacts of Broken Authentication

The consequences of broken authentication can be severe:

- **Account Takeover**: Attackers gain unauthorized access to user accounts.
- **Data Breaches**: Sensitive user or business data can be stolen.
- **Privilege Escalation**: If administrative accounts are compromised, attackers gain system-wide control.
- **Financial and Legal Ramifications**: Companies face heavy fines and damage to their reputation.
- **Identity Theft**: Stolen personal information can be used in further attacks.

🛡️ Mitigations for Broken Authentication

Session Management Controls

To defend against session-related attacks, applications must **regulate session duration** and **regenerate session IDs** after authentication. The session should expire after a period of inactivity (especially in financial or sensitive applications), and logout actions should invalidate the session. **Session IDs should never be exposed in URLs**, and cookies should be marked as Secure, HttpOnly, and with appropriate SameSite settings to prevent XSS or CSRF exploitation.

Authentication Hardening

Implementing **Multi-Factor Authentication (MFA)** is a highly recommended control. MFA introduces an extra layer (like OTP, biometrics, or device confirmation) that makes it harder for attackers to log in even with a valid password. Enforcing strong password policies—minimum length, character variety, and no use of common passwords—is essential. Passwords must be stored securely using salted hashes with secure algorithms such as bcrypt, PBKDF2, or Argon2.

Attack Surface Reduction

Applications must detect brute-force attempts and implement **lockout mechanisms** after multiple failed logins. They can also monitor IP addresses for abnormal activity and apply **rate limiting** to login endpoints. Furthermore, the **credential recovery process** must be secure, with identity verification and temporary tokens, to prevent attackers from resetting other users' passwords.

🔐 Single-Factor vs Multi-Factor Authentication

Single-factor authentication, the traditional model, relies solely on a username and password combination. This method is vulnerable to theft, brute-force, and phishing attacks. On the other hand, **multi-factor authentication (MFA)** includes something the user knows (password), something the user has (mobile device or token), and/or something the user is (biometric data). MFA significantly reduces the risk of account compromise even if one factor is exposed.

🔑 PART 2: OAUTH — Open Authorization Framework

🧭 What is OAuth?

OAuth (Open Authorization) is an **authorization** framework that allows third-party applications to access user resources **without sharing credentials**. Unlike authentication (which confirms identity), OAuth is about granting **limited access to a resource** for a specific purpose.
A practical example is when you use a "Login with Google" button to log into a third-party app. The app doesn't see your Google credentials; instead, it receives a **token** that allows limited access to your profile data or email, as you authorized.

🧱 OAuth Core Components

1. **OAuth Provider**: The platform hosting your data (e.g., Google, Facebook)
2. **OAuth Client**: The app requesting access to your data (e.g., Spotify, ESPN)
3. **Resource Owner**: You — the person who owns the data and gives permission
4. **Authorization Server**: Validates the user and issues access tokens
5. **Resource Server**: Stores the data (e.g., Google Drive, Twitter)

🔄 OAuth Workflow (Step-by-Step)

Example 1: Login with Facebook

1. User clicks “Login with Facebook” on a website (OAuth Client).
2. The client redirects the user to Facebook’s **Authorization Server**.
3. The user logs in and approves access.
4. Facebook returns an **authorization code** to the client.
5. The client exchanges this code for an **access token**.
6. The client uses the access token to access user data (via the Resource Server).
7. The user is now authenticated with the third-party app, without revealing their Facebook password.

🧹 Real-World OAuth Scenarios

- **Facebook App**: You use a Facebook-connected quiz app, which can read your name and friends list — only what you allowed.
- **Cloud File Sharing**: You send Google Drive files via Gmail. OAuth lets Gmail access your Google Drive temporarily without asking you to log in again.
- **Smart Devices**: Smart thermostats or speakers use OAuth to access your cloud accounts securely without requiring re-authentication each time.

⚖️ SAML vs OAuth

Feature	SAML	OAuth
Purpose	Authentication	Authorization
Format	XML	JSON
Best For	Enterprise SSO	APIs, mobile apps, third-party integrations
Use Case	Accessing multiple corporate apps after one login	Granting app access to Google/Facebook info

OAuth is better suited for consumer apps and mobile environments, while **SAML** is used for enterprise-grade **Single Sign-On (SSO)** between internal systems.

🧪 OAuth Risks & Precautions

Although OAuth improves security by avoiding credential sharing, it has its own risks if not implemented properly:

- **Insecure Token Storage**: Tokens must be protected like passwords.
- **Improper Scopes**: Apps requesting more permissions than needed.
- **Phishing**: Fake OAuth approval pages trick users into granting access.
- **Token Leakage**: Through URL redirects, browser storage, or log files.

To mitigate these, always use **HTTPS**, implement **token expiration**, support **revocation**, and restrict **token scopes**.

✅ Final Thoughts

The document emphasizes that both **Broken Authentication** and **OAuth** are critical security areas in modern web applications.

- **Broken Authentication** occurs when systems fail to properly verify and manage user identity, leading to session hijacking, brute-force attacks, and account takeovers.
- **OAuth** is a powerful framework for safely delegating access across applications without sharing passwords, but it must be implemented with strict controls to avoid misuse.

AST 7 SSTI_HRS.C

Scenario	Description	Corrected Understanding
CL.TE	Frontend drops extra data, backend reads full	✅ Dropped data becomes next request
TE.CL	Frontend sends full, backend cuts early	✅ Backend leftover is appended to next request (Start)
Both drop	Frontend and backend read partial	✅ Their dropped parts together = smuggled request

✅ What is SSTI?

Server-Side Template Injection (SSTI) is a vulnerability that occurs when **user-controlled input** is **unsafely embedded into a server-side template**. Template engines (e.g., **Jinja2**, **Twig**, **Freemarker**) are used in web applications to render dynamic HTML pages. When developers improperly embed user input into templates, it can lead to code injection on the server.

🧠 How It Works

A web app might generate dynamic content using templates.

For example, a bulk email sender might personalize messages:

$output = $twig->render("Dear {first_name}", array("first_name" => $user.first_name));

This is **safe**—first_name is passed as data.

Now look at this vulnerable version:

$output = $twig->render("Dear " . $_GET['name']);

Here, the name parameter is directly injected into the template string.

If an attacker sends this:

http://vulnerable-website.com/?name={{7*7}}

The server might render:

Dear 49

If deeper access is possible, the attacker can run **commands or read files**, escalating it to **remote code execution (RCE)**.

⚙️ Real-world Example: Twig SSTI

A PHP app using the Twig template engine:

Payloads and Impact:

Input Payload	Server Response	Impact
{{1338-1}}	Thanks 1337.	Confirms code execution
{{_self.env.registerUndefinedFilterCallback("exec")}} {{_self.env.getFilter("id")}}	uid=33(www-data)...	Executes shell command
{{_self.env.getFilter("ls /")}}	Shows / directory listing	Filesystem exposure
{{_self.env.getFilter("cat /secret/flag.txt")}}	Reads flag	Full access to sensitive files

---

⚔️ SSTI vs XSS

Feature	SSTI	XSS
Injection Type	Server-side	Client-side
Target	Application/Server	User's browser
Impact	RCE, data exfiltration, full server control	Session hijacking, defacement, phishing
Cause	Unsafely injecting user input into templates	Unsanitized output reflected in HTML/JS
Attack Surface	Templating engines (e.g., Jinja, Twig)	JavaScript, HTML DOM

🛡️ Mitigation Strategies

1. **Never concatenate user input** into templates. Always pass data as variables.
2. Use **"logic-less" template engines** (e.g., Mustache), which separate logic and view.
3. Employ **sandboxing or whitelisting** within your template engine.
4. Apply **input validation and encoding**, and implement **least privilege** for server-side code execution.
5. **Perform regular security testing** using tools like Burp Suite or manual payload injection.

🔍 Threat Modeling

🎯 What is Threat Modeling?

Threat modeling is a structured method to **identify, assess, and prioritize risks** in an application **before and during development**.

Threat modeling is the process of identifying potential security threats early in the design phase.It helps teams proactively plan defenses before actual vulnerabilities are introduced.
It helps security teams and developers to:

- Visualize security flaws
- Understand attacker's perspective
- Prioritize fixes and mitigations

🧩 Steps in Threat Modeling (Based on NIST 800-30):

1. **Decompose the Application**
   Understand the system: data flow, architecture, authentication points, APIs, and more.

2. **Define and Classify Assets**
   Identify valuable components: databases, user data, credentials. Rank them by business impact.

3. **Identify Vulnerabilities**
   These could be **technical** (e.g., injection flaws), **operational** (misconfigurations), or **managerial** (lack of policies).

4. **Identify Threats**
   Model realistic attack vectors using **attack trees**, **STRIDE model**, or **abuse cases**.

5. **Create Mitigation Strategies**
   Design solutions for each threat—e.g., input sanitization, proper access controls, secure defaults.

📈 Outputs of Threat Modeling

- **Diagrams** showing data flows and trust boundaries
- **Lists** of identified threats and mitigations
- **Security test plans** focused on critical areas

✅ **Advantages of Threat Modeling**

- Gives a **practical attacker's view** of the system
- Helps catch issues **early in the SDLC**
- **Flexible**—can adapt to any architecture

❌ **Disadvantages**

- Still a **relatively new** approach for many teams
- A good model ≠ guaranteed security unless mitigations are properly implemented

---

🧠 **Summary**

Concept	Key Idea
SSTI	Exploiting template engines by injecting user-controlled input into templates
Threat Modeling	A proactive approach to identify and mitigate potential threats early in development