Cybersecurity Strategies For Web Applications

Defend Your Code in the Age of Digital Warfare

# whoami

I'm **Monish Kumar**, also known as `m0n1x90`. I'm a seasoned Security Engineer @ ZOHO with a focus on offensive cybersecurity.

My expertise lies in Red Teaming operations and advanced Malware Development. I've honed these skills through extensive experience in protecting digital assets and full stack development.

The Web Application Battlefield

Web applications are prime targets. Attackers relentlessly probe for vulnerabilities, exploiting every entry point from simple input fields to complex APIs and sophisticated template engines.

High Vulnerability Rate

90% of web applications have at least one critical vulnerability (Verizon DBIR 2024).

Diverse Attack Vectors

Exploiting input fields, APIs, and templating systems.

Steep Financial Stakes

Data breaches cost an average of **$4.45M** per incident (IBM 2025).

Why Web Apps Are Prime Targets ?

Web applications are constantly under siege due to their inherent accessibility and the vast amount of data they process. Attackers capitalise on common weaknesses, making web apps a goldmine for data breaches and system compromise.

Inherent Vulnerabilities

Web applications are complex by nature, often integrating various components and exposing numerous entry points. This complexity, coupled with common coding errors, leads to a high prevalence of known vulnerabilities like Cross-Site Scripting (XSS) and SQL Injection (SQLi).

Fact: OWASP's Top 10 consistently highlights critical web application security risks.

Misconfigurations & Defaults

Many breaches stem from simple misconfigurations: default credentials left unchanged, unnecessary services exposed, or inadequate access controls. Developers often prioritise functionality over security, leaving critical gaps that attackers quickly discover and exploit.

Stat: Over 70% of cloud security incidents are due to misconfiguration (IBM X-Force Threat Intelligence Index 2023).

Poor Security Practices

Rushed development cycles, lack of security training, and insufficient testing lead to insecure coding practices. Without a security-first mindset, applications become riddled with weaknesses, from weak authentication mechanisms to insecure API designs.

Impact: Data breaches caused by web application attacks cost organisations millions annually.

Core Cybersecurity Strategies for Web Apps

Building a robust defence requires a multi-layered approach. Each strategy acts as a critical barrier, reinforcing the overall security posture of your web applications.

1 Input Validation & Sanitisation

Strictly filter and clean all incoming data to neutralise malicious payloads at the earliest possible stage.

2 Authentication & Session Management

Implement strong authentication protocols and secure session handling to prevent unauthorised access and session hijacking.

3 Encryption & Secure Communication

Encrypt all sensitive data, both in transit (e.g., TLS) and at rest (e.g., database encryption), to protect against eavesdropping and data exposure.

4 Regular Patching & Dependency Management

Continuously update all software components and libraries to patch known vulnerabilities and mitigate supply chain risks.

Introducing Template Engines: Power & Peril

Template engines are indispensable tools for modern web development, facilitating the dynamic generation of HTML pages by separating logic from presentation. However, their power can be double-edged, introducing subtle yet critical vulnerabilities if mishandled.

What They Do

- Separate design (HTML) from data (dynamic content).
- Streamline development workflows.
- Enable rapid UI changes without altering backend logic.

Popular Examples

- Jinja2 (Python)
- Handlebars (JavaScript)
- EJS (JavaScript)
- Twig (PHP)

What Are Templating Attacks?

Templating attacks involve injecting malicious code into template systems to manipulate application output or execute arbitrary commands. These attacks leverage the dynamic nature of template engines to bypass traditional security controls.

1

Injection-Based

Attackers inject template syntax or expressions that the engine then processes, rather than rendering as static text.

2

Client-Side (CSTI)

Exploits vulnerabilities in client-side rendering engines, leading to execution in the victim's browser.

3

Server-Side (SSTI)

More severe, exploiting server-side engines that can lead to remote code execution (RCE) on the server itself.

Client-Side Template Injection (CSTI)

CSTI occurs when an attacker can inject malicious code into client-side templates, which are then rendered by the user's browser. This often happens when web applications use frameworks like Angular or Vue.js to dynamically generate UI based on untrusted input without proper sanitisation.

How It Works

- Untrusted user input is embedded directly into client-side template expressions.
- The browser's JavaScript engine then parses and executes this malicious code.

Example (AngularJS)

Injecting `{{constructor.constructor('alert(1)')()}}` in a vulnerable field can pop an alert box.

Risks & Impact

- Data theft (e.g., stealing cookies, local storage).
- Session hijacking.
- Website defacement.
- Phishing attacks.

Client-Side Template Injection (CSTI)

Client-Side Template Injection (CSTI) occurs when a web application incorporates user-supplied input directly into a client-side template, which is then rendered by the user's browser. This allows attackers to execute malicious code within the victim's browser context.

1. Malicious Input

An attacker injects client-side template syntax (e.g., `{{alert(1)}}` for AngularJS) into an input field or URL parameter.

2. Server Echoes Input

The server incorporates this unprocessed, malicious input directly into the HTML response sent back to the client.

3. Client-Side Rendering

The victim's browser receives the response, and its client-side template engine attempts to render the dynamic content, including the malicious payload.

4. Code Execution

The template engine executes the attacker's payload within the user's browser, leading to XSS, data exfiltration, or UI defacement.

CSTI vs. XSS: Dissecting the Nuances

While often conflated due to similar outcomes, Client-Side Template Injection (CSTI) is distinct from traditional Cross-Site Scripting (XSS), particularly when considering how vulnerabilities are exploited and their specific attack vectors, including the 'Self-XSS' variant.

Cross-Site Scripting

Injects malicious scripts directly into a webpage's HTML, leading to execution when the victim's browser renders the page. Exploits browser's trust in rendered HTML.

Client-Side Template Injection

Specifically targets client-side template engines (e.g., AngularJS, Vue.js). Attackers inject template syntax (like `{{alert(1)}}`) which the template engine then parses and executes, leading to JavaScript execution.

Self-XSS

A social engineering attack where the victim is tricked into executing malicious code in their own browser, typically by pasting it into the developer console. Does not automatically affect other users.

Server-Side Template Injection (SSTI)

SSTI is a far more critical vulnerability, as it allows attackers to execute code directly on the server. This can lead to complete server compromise, data exfiltration, or even pivoting to other systems within the network.

Mechanism

- Unsafe embedding of user-controlled input into server-side templates.
- The server-side template engine evaluates and executes the injected payload.

Example (Jinja2)

Exploiting Jinja2 to execute `{{ config.items() }}` can reveal sensitive server configuration secrets.

Severe Consequences

- Remote Code Execution (RCE).
- Full server takeover.
- Database dumps.
- Lateral movement within the network.

Server-Side Template Injection (SSTI)

SSTI is a critical server-side vulnerability that occurs when an attacker can inject malicious code directly into a server-side template, tricking the template engine into executing arbitrary commands on the server. This often leads to remote code execution (RCE).

1. Malicious Input

Attacker submits crafted input containing template syntax.

2. Direct Inclusion

Application directly embeds input into a server-side template.

3. Server-Side Execution

Template engine parses and executes the injected malicious code.

4. System Compromise

Attacker gains control, often leading to Remote Code Execution (RCE).

Real-World CSTI: The AngularJS Case Study

One prominent example of Client-Side Template Injection involved applications utilising AngularJS. Due to its powerful two-way data binding and expression evaluation, if user-controlled input was improperly sanitised and directly rendered within AngularJS templates, attackers could inject malicious expressions.

> This led to severe Cross-Site Scripting (XSS) vulnerabilities, allowing session hijacking, sensitive data exfiltration, and UI manipulation within the victim's browser.

Lesson Learned

The AngularJS CSTI cases underscore the critical importance of treating all user input as untrusted, even when rendered client-side. Strict input sanitisation and context-aware escaping are paramount to prevent the injection of malicious template syntax.

For more details, refer to: **PortSwigger: Client-Side Template Injection Overview**

Another Common CSTI Scenario: User Profile Customisation

Beyond specific framework vulnerabilities, Client-Side Template Injection frequently emerges in web applications that allow users to customise their profiles or content, such as social media platforms, e-commerce sites, or blogging platforms. If user-supplied data (like a custom bio or a display name) is directly incorporated into a client-side template without proper sanitisation, it becomes a vector for attack.

The Vulnerability

Applications fetch user-defined content (e.g., a "About Me" section) and directly bind it into a template that is rendered in the user's browser. If this content contains malicious template expressions, they are executed.

Exploitation

An attacker inputs template syntax (e.g., `<img src="" onerror="alert(document.cookie)">` within a rich text editor that gets evaluated by a client-side templating engine), which is then rendered by the victim's browser, leading to XSS.

Impact

This can result in session hijacking, data theft, or defacement of the user interface for the victim, all within their own browser context, often without needing a server-side exploit.

This highlights that even seemingly innocuous features, when combined with client-side templating, require stringent input validation and output encoding to mitigate CSTI risks.

Real-World SSTI: Apache Struts 2 RCE

A notable series of Server-Side Template Injection vulnerabilities plagued Apache Struts 2, a widely used open-source web application framework. These flaws often stemmed from improper handling of user-supplied data in various HTTP components, allowing direct injection into its OGNL (Object-Graph Navigation Language) expression evaluation.

The Vulnerability

Attackers exploited how Apache Struts 2 processed user input (e.g., via HTTP headers or parameters), leading to the execution of malicious OGNL expressions directly on the server.

Catastrophic Impact

These vulnerabilities, such as CVE-2017-5638, permitted Remote Code Execution (RCE), enabling attackers to take full control of affected servers and access sensitive data.

Critical Lesson

The Struts 2 incidents underscore the extreme danger of SSTI, emphasising that any user-controlled input processed by server-side expression or template engines must be rigorously validated and sanitised.

For further details on this class of vulnerability, refer to: **PortSwigger: Server-Side Template Injection Explained**

Defending Against Templating Attacks

Preventing template injection requires a rigorous and proactive approach to development and security. Implementing these measures can significantly harden your web applications against both CSTI and SSTI.

1

Strict Input Validation

Never trust user input. Implement comprehensive sanitisation and validation for all data fed into templates.

2

Safe Template Rendering

Utilise built-in security features like auto escaping, sandboxing, and strict variable filters provided by template engines.

3

Content Security Policy (CSP)

Deploy robust CSP headers to restrict the sources from which client-side scripts can be executed, mitigating CSTI impacts.

4

Regular Audits & Monitoring

Conduct frequent security audits of template usage, dependencies, and implement runtime monitoring for suspicious template behaviour.

Live Demonstration

Let's put theory into practice. We'll explore Client-Side and Server-Side Template Injection firsthand on intentionally vulnerable applications.

Client-Side Template Injection (CSTI)

Witness how malicious code can be executed within a user's browser via client-side template vulnerabilities.

Server-Side Template Injection (SSTI)

Observe the severe impact of SSTI, demonstrating remote code execution capabilities on the server.

Prepare to see these attacks in action and understand their real-world implications.

The Future of Web App Security: Vigilance & Innovation

As template engines continue to evolve, so too will the ingenuity of attackers. Our defence must match this pace, combining robust technical measures with a security-first culture.

Combine secure coding practices with automated scanning.

Empower developers with a security-first mindset and cutting-edge tools.

Together, we can build resilient web applications that withstand the hacker's arsenal.

Thank you. Questions?

Follow & Connect with Me

Thank you for your attention and engagement.

The journey to robust web application security is continuous. Let's keep the conversation going!

X (formerly Twitter)	GitHub	LinkedIn
@m0n1x90	@m0n1x90	Monish Kumar