Hackers' Playbook:

Top 10 Web Exploits Exposed in 2024

Daily Red Team

Table Of Contents

In the dynamic field of web security, staying informed about the latest vulnerabilities and exploitation techniques is crucial. PortSwigger's "Top 10 Web Hacking Techniques of 2024" offers a comprehensive overview of the most innovative and impactful web security research from the past year. This annual compilation serves as a valuable resource for security professionals seeking to understand emerging threats and the methodologies behind them."PortSwigger's Top 10 Web Hacking Techniques of 2024" is a yearly report that offers an overview of the most impactful web security research from the previous year. This summary of innovative exploits and methodologies is a valuable tool for security professionals to learn about current threats and how they are executed, which is essential in the ever-changing landscape of web security.

01

Understanding Cookie Tossing

In the evolving landscape of cybersecurity, red teamers continually explore innovative attack vectors to identify and mitigate potential vulnerabilities. One such emerging technique is Cookie Tossing, which can be exploited to hijack OAuth flows, potentially leading to account takeovers at Identity Providers (IdPs). This article delves into the intricacies of Cookie Tossing, its application in compromising OAuth flows, and strategies to defend against such attacks.

Cookie Tossing is an attack method where a malicious subdomain sets cookies that apply to its parent domain. This technique leverages the way browsers handle cookies, particularly the Domain and Path attributes, to manipulate the order and scope of cookies sent in HTTP requests.

Key Concepts

- Domain Attribute: Specifies the domain for which the cookie is valid. If set to a parent domain (e.g., example.com), the cookie is sent to all subdomains.
- Path Attribute: Defines the URL path that must exist in the requested resource before sending the Cookie header.
- Cookie Precedence: When multiple cookies with the same name are present, browsers determine which cookie to send based on the specificity of the Path attribute and the order of settings..

By carefully crafting these attributes, an attacker can ensure that their malicious cookie is sent instead of, or prior to, a legitimate one, thereby influencing the application's behavior.

Exploiting OAuth Flows via Cookie Tossing

OAuth is a widely adopted authorization framework that allows third-party applications to access user resources without exposing credentials. However, improper implementation can expose vulnerabilities.

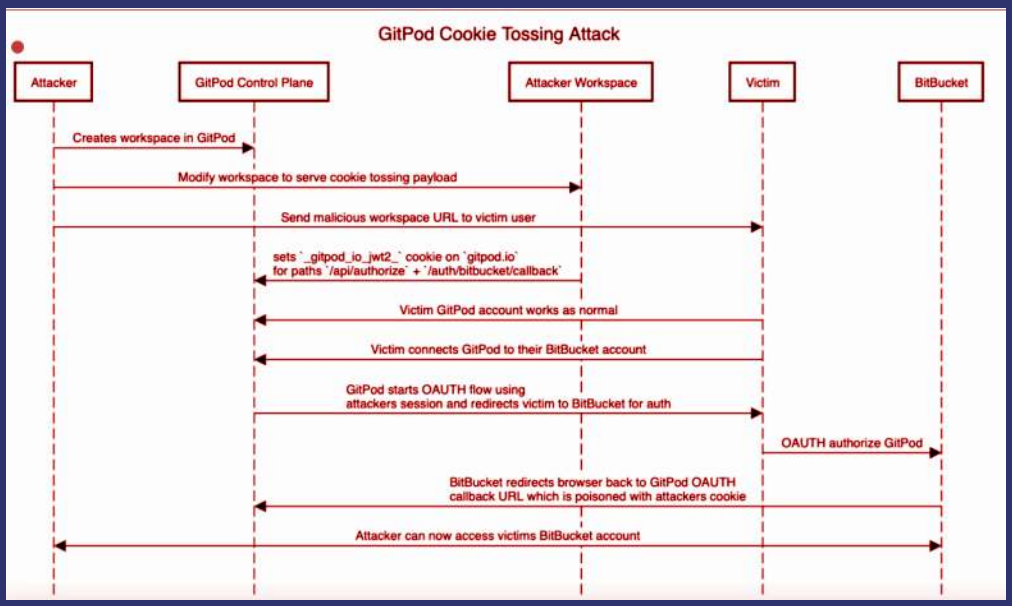


Image Credit: snyk.io

Attack Scenario

1. Setup: An attacker controls a subdomain of the target application (e.g., attacker.example.com).
2. Cookie Injection: The attacker sets a cookie with a name matching a legitimate session or state management cookie used in the OAuth flow, configuring the Domain attribute to the parent domain (example.com) and the Path attribute to target specific endpoints involved in the OAuth process.
3. User Interaction: The victim visits the malicious subdomain, causing their browser to store the attacker's cookie.
4. OAuth Flow Initiation: When the victim initiates an OAuth flow with a service provider, their browser includes the malicious cookie in requests to the parent domain.
5. Session Hijacking: The application processes the malicious cookie, leading to unauthorized actions such as session fixation or state manipulation, ultimately allowing the attacker to hijack the OAuth flow and gain access to the victim's account.

Snyk's research highlighted this method, demonstrating how Cookie Tossing can hijack OAuth flows and lead to account takeovers at the Identity Provider (IdP).

Defense Mechanisms

To mitigate the risks associated with Cookie Tossing in OAuth flows, consider implementing the following strategies:

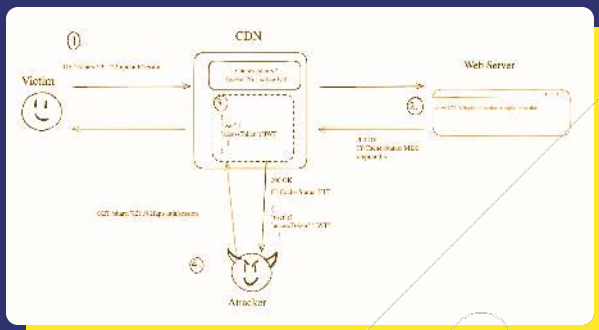
1. Use the __Host- Prefix: Cookies with the __Host- prefix are required to have the Path attribute set to /, must not have a Domain attribute, and must be secured with the Secure and HttpOnly flags. This restricts the cookie to the host that set it, preventing subdomains from setting cookies that affect the parent domain.
2. Implement SameSite Attributes: Setting the SameSite attribute to Strict or Lax can prevent browsers from sending cookies along with cross-site requests, mitigating CSRF attacks. However, be cautious, as this may interfere with legitimate cross-site OAuth flows.
3. Validate OAuth State Parameters: Ensure that the state parameter in OAuth flows is unique and validated upon return to prevent CSRF attacks.
4. Regularly Audit Subdomains: Monitor and control subdomain creations to prevent unauthorized or malicious subdomains from being used as attack vectors.
5. Educate Development Teams: Raise awareness about Cookie Tossing and related vulnerabilities among developers to promote secure coding practices.

By understanding and addressing the nuances of Cookie Tossing, security professionals can better protect OAuth implementations from sophisticated hijacking attempts.

02

The Vulnerability in ChatGPT

In the realm of cybersecurity, understanding the nuances of web caching mechanisms is crucial for both attackers and defenders. A recent vulnerability discovered in ChatGPT highlights the potential risks associated with Web Cache Deception (WCD), particularly when combined with path traversal and URL parser inconsistencies. This article delves into the specifics of this vulnerability, its exploitation, and the broader implications for web security.



In February 2024, a security researcher identified a critical vulnerability in ChatGPT's implementation of its "share" feature. This feature allows users to share their chats publicly via unique URLs structured as https://chat.openai.com/share/CHAT-UUID. Upon investigation, it was observed that these shared chat links were being cached, as indicated by the Cf-Cache-Status: HIT response header.

Image Credit: nokline.github.io

Notably, the caching mechanism did not rely on file extensions but rather on the URL path, leading to a caching rule resembling /share/*. This permissive caching rule raised concerns about potential exploitation through URL parser confusion and path traversal.

Exploitation via Path Traversal and URL Parser Confusion

The core of the exploitation lies in the discrepancy between how the Content Delivery Network (CDN) and the web server handle URL encoding and path normalization:

1. CDN Behavior: The CDN, in this case, Cloudflare, does not decode or normalize URL-encoded path traversal sequences (e.g., %2F..%2F). Consequently, it caches the response based on the literal URL provided.
2. Web Server Behavior: The web server decodes and normalizes these sequences, interpreting %2F..%2F as /../, effectively traversing directories in the file system.

By crafting a URL such as https://chat.openai.com/share/%2F..%2Fapi/auth/session?cachebuster=123, an attacker can exploit this discrepancy:

- Caching Mechanism: The CDN caches the response because the URL matches the /share/* pattern, without decoding the %2F..%2F sequence.
- Server Response: The web server decodes %2F..%2F to /../, accessing the /api/auth/session endpoint, which contains sensitive authentication tokens.

When a victim accesses this malicious URL, their authentication token is cached by the CDN. Subsequently, the attacker can retrieve the cached token by accessing the same URL, leading to a full account takeover.

Implications and Mitigation Strategies

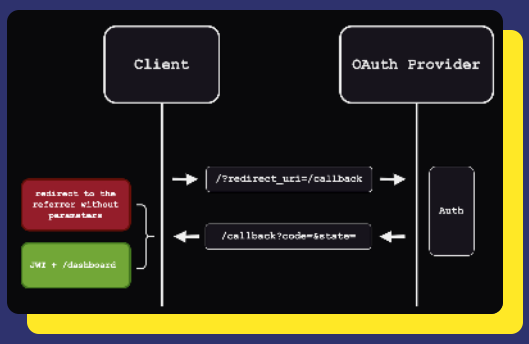
This vulnerability underscores the dangers of relaxed caching rules and inconsistencies in URL parsing between CDNs and web servers. To mitigate such risks:

1. Strict Caching Policies: Define precise caching rules that limit caching to intended content types and paths. Avoid wildcard patterns that encompass sensitive endpoints.
2. Consistent URL Parsing: Ensure uniform URL decoding and normalization processes between the CDN and the web server to prevent discrepancies.
3. Input Validation: Implement robust input validation to detect and reject malicious URL patterns, including encoded path traversal sequences.
4. Regular Security Audits: Conduct periodic security assessments focusing on caching mechanisms and URL handling to identify and remediate potential vulnerabilities.

03

Understanding the "Non-Happy Path" in OAuth

In red teaming, exploring "non-happy paths"—scenarios where applications deviate from their expected workflows—can unveil critical vulnerabilities. A recent analysis by 0xrz from the Voorivex team highlights how such deviations in OAuth implementations can lead to account takeovers (ATO). This article delves into the intricacies of this vulnerability, its exploitation, and preventive measures.



The happy path refers to the standard, error-free execution flow of a software application. Conversely, a non-happy path involves alternative flows, often triggered by unexpected inputs or errors, leading to unforeseen application behaviors. In OAuth implementations, these non-happy paths can be exploited if not properly handled.

Case Study: Exploiting OAuth Non-Happy Paths

In a public bug bounty program, 0xrz examined an OAuth implementation used for user authentication. By manipulating the response_type parameter from code to id_token, the application deviated from its typical flow, resulting in the access token being included in the URL fragment. This deviation led to an open redirect based on the Referer header, which could be exploited for account takeover. Image credit: voorivex.team

Key Conditions for Exploitation

1. Redirect Behavior: The application redirects users based on the Referer header without additional parameters.
2. Attacker-Controlled Referer: The attacker can manipulate the Referer header to point to a malicious site.
3. Access Token Exposure: The access token is included in the URL fragment during the non-happy path flow.
4. Token Theft: The attacker can capture the access token by exploiting the redirect behavior.

Exploitation Process

1. Initiate OAuth Flow: The attacker initiates the OAuth flow with a manipulated response_type parameter (id_token instead of code).
2. Trigger Non-Happy Path: This manipulation causes the application to enter a non-happy path, resulting in the access token being placed in the URL fragment.
3. Manipulate Referer Header: The application redirects the user based on the Referer header, which the attacker has set to a malicious site.
4. Capture Access Token: The attacker captures the access token from the URL fragment during the redirect, leading to account takeover.

Mitigation Strategies

To prevent such vulnerabilities:

1. Strict Parameter Validation: Ensure that parameters like response_type are strictly validated and only accept expected values.
2. Secure Redirect Handling: Avoid using the Referer header for redirects. Instead, implement a whitelist of allowed URLs and validate redirect destinations.
3. Fragment Handling: Be cautious with URL fragments, as they are not sent to the server. Ensure sensitive information is not included in fragments.
4. Comprehensive Testing: Conduct thorough testing of both happy and non-happy paths in OAuth flows to identify and address potential vulnerabilities.

04

JavaScript Execution in PDF.js

Image Credit: codeanlabs.com

In May 2024, a critical vulnerability identified as CVE-2024-4367 was discovered in PDF.js, a widely used JavaScript-based PDF viewer maintained by Mozilla. This vulnerability allows attackers to execute arbitrary JavaScript code when a malicious PDF file is opened, posing significant risks to users and applications that utilize PDF.js for rendering PDFs.

PDF.js serves two primary purposes:

1. Browser Integration: It functions as the built-in PDF viewer in Mozilla Firefox.
2. Web and Application Embedding: Through the pdfjs-dist Node module, it enables websites and Electron-based applications to embed PDF viewing capabilities.

Given its extensive adoption, a vulnerability in PDF.js can have widespread implications.

Root Cause of the Vulnerability

The vulnerability stems from the font rendering process within PDF.js. To optimize performance, PDF.js pre-compiles path generator functions for glyphs (characters) by dynamically creating JavaScript functions. This involves constructing a function body (jsBuf) containing rendering commands, which is then converted into a new Function object.

An attacker can exploit this process by manipulating the FontMatrix attribute within a PDF's font object. By injecting a string into the FontMatrix array, the malicious input is incorporated directly into the dynamically generated function without proper sanitization, leading to arbitrary code execution.

Exploitation Details

To exploit this vulnerability, an attacker crafts a PDF with a malicious FontMatrix entry:

/FontMatrix [1 2 3 4 5 (0\); alert\('Exploit')]\n```

When this PDF is processed by PDF.js, the injected string is executed as JavaScript, resulting in the execution of arbitrary code.

Impact

The impact of this vulnerability is significant:

- Firefox Users: All versions prior to 126 are affected, as they include the vulnerable PDF.js component.
- Websites and Applications: Any web or Electron-based application that incorporates PDF.js for PDF rendering is at risk.

Depending on the application's context, successful exploitation can lead to unauthorized actions, data leakage, or full account compromise.

Mitigation Strategies

To protect against this vulnerability:

1. Update PDF.js: Upgrade to PDF.js version 4.2.67 or later, where the issue has been addressed.
2. Disable Dynamic Code Execution: Set the isEvalSupported configuration option to false to prevent the use of eval and the Function constructor.
3. Enforce Content Security Policies (CSP): Implement strict CSPs that disallow unsafe script execution methods.
4. Audit Dependencies: Review your project's dependencies to ensure that no vulnerable versions of PDF.js are in use, especially in nested packages.

05

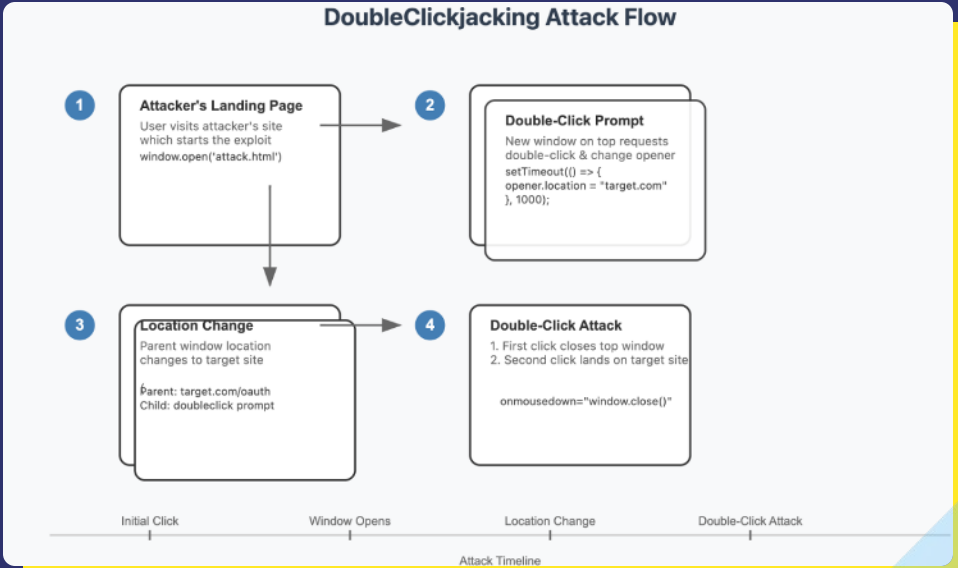
DoubleClickjacking

In the evolving landscape of web security, traditional clickjacking defenses are becoming less effective due to advancements in browser security features. A recent technique, termed DoubleClickjacking, has emerged as a novel method of user interface (UI) redressing that exploits double-click sequences to bypass existing protections. This article delves into the mechanics of DoubleClickjacking, its potential impacts, and strategies for mitigation.

DoubleClickjacking is an advanced form of clickjacking that leverages the timing and event-order differences between mousedown and click events. Unlike traditional clickjacking, which relies on a single click, this technique manipulates a double-click sequence to execute unauthorized actions. Notably, DoubleClickjacking can circumvent established defenses such as the X-Frame-Options header, Content Security Policy (CSP) frame-ancestors, and SameSite cookie attributes.

Exploitation Process

Image Credit: paulosyibelo.com



The DoubleClickjacking attack unfolds through the following steps:

1. Initiation: An attacker creates a webpage containing a button that, when clicked, opens a new window or tab.
2. Window Manipulation:
    - Upon the user's click, a new window appears, prompting the user to perform a double-click action.
    - Simultaneously, this new window utilizes window.opener.location to change the parent window's location to a target page, such as an OAuth authorization prompt.
3. Double-Click Execution:
    - The user's first click (mousedown event) triggers the top window to close.
    - The second click lands on the now-exposed sensitive UI element in the parent window, such as an "Authorize" button.

This sequence results in the user inadvertently performing actions like authorizing a malicious application with extensive privileges. The attack exploits the brief interval between the mousedown and click events, allowing the attacker to manipulate the UI without the user's awareness.

Potential Impacts

DoubleClickjacking poses significant security risks, including:

- Unauthorized Application Authorization: Attackers can trick users into granting extensive permissions to malicious applications, leading to account takeovers.
- Account Setting Modifications: Similar to classic clickjacking, this technique can be used to alter account settings, disable security features, or initiate unauthorized transactions.

The effectiveness of DoubleClickjacking is heightened by its ability to bypass traditional clickjacking defenses, making it a pervasive threat across various platforms.

Mitigation Strategies

To defend against DoubleClickjacking attacks, consider implementing the following measures:

1. Client-Side Protections:
    - Disable Critical Buttons by Default: Implement JavaScript to disable form buttons until a legitimate user interaction, such as mouse movement or keyboard input, is detected.
    - Event Handling: Favor mousedown event handlers over click events to reduce the exploitable time window.
2. Randomize UI Element Identifiers: Assign unpredictable identifiers to critical UI elements to prevent attackers from accurately targeting them.
3. User Interaction Verification: Require additional user verification steps for sensitive actions, such as confirming a transaction or changing account settings.
4. Browser-Level Defenses: Advocate for browser vendors to develop new standards and headers that mitigate double-click exploitation, similar to existing protections against classic clickjacking.

06

Exploring the DOMPurify Library

In the realm of web security, DOMPurify stands as a widely adopted client-side library designed to sanitize HTML and prevent Cross-Site Scripting (XSS) attacks. However, recent research has uncovered several bypasses in versions 3.1.0 through 3.1.2, highlighting the evolving challenges in HTML sanitization. This article delves into these bypasses, their underlying mechanisms, and the subsequent fixes implemented to enhance DOMPurify's resilience.

DOMPurify operates by leveraging the browser's native HTML parser to cleanse potentially malicious code from user inputs. By parsing and sanitizing HTML content in the client-side environment, it aims to mitigate the risk of XSS attacks that exploit vulnerabilities in web applications.

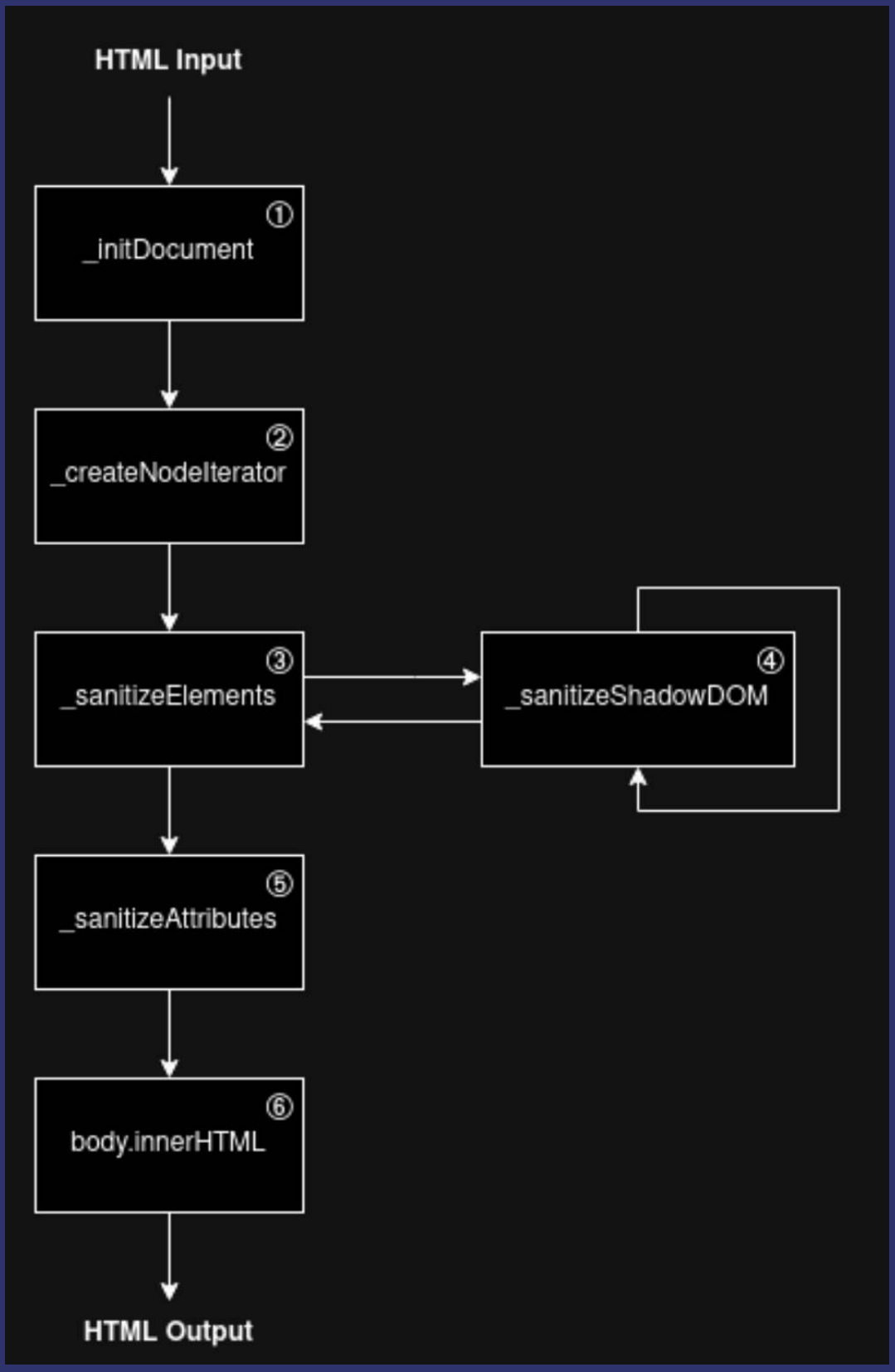

HTML Input
①
_initDocument
②
_createNodeIterator
③
_sanitizeElements
④
_sanitizeShadowDOM
⑤
_sanitizeAttributes
⑥
body.innerHTML
HTML Output

Image Credit: mizu.re

I. DOMPurify 3.1.0 Bypass: Deep Nesting

Discovery: In April 2024, researcher @IcesFont identified a bypass in DOMPurify version 3.1.0. The exploit involved creating deeply nested HTML structures that exceeded the library's parsing depth, allowing malicious scripts to evade sanitization.

Mechanism: By nesting elements beyond the library's processing capacity, the sanitizer failed to traverse and cleanse the innermost nodes, resulting in the execution of harmful scripts.

Fix: To address this, DOMPurify introduced a depth counter limiting the maximum nested depth to 255 levels. This measure ensures that excessively deep structures are sanitized appropriately.

II. DOMPurify 3.1.1 Bypass: DOM Clobbering

Discovery: Following the initial fix, a new bypass was discovered that exploited DOM clobbering techniques. Attackers manipulated the parentNode property to reset the depth counter, circumventing the newly implemented nesting limit.

Mechanism: By injecting elements that clobbered the parentNode property, attackers could deceive the sanitizer into miscalculating the depth of nested elements, allowing malicious code to slip through.

Fix: The DOMPurify team addressed this by enhancing the clobbering checks and ensuring that the parentNode property is accessed securely, preventing unauthorized modifications.

III. DOMPurify 3.1.2 Bypass: Second-Order DOM Clobbering and "Elevator" HTML Mutation

Discovery: In subsequent research, a more sophisticated bypass was identified, combining second-order DOM clobbering with "elevator" HTML mutations.

Mechanism: This exploit involved manipulating the order of sanitization processes and leveraging browser-specific HTML parsing behaviors. Attackers crafted payloads that mutated during parsing, effectively "elevating" malicious code past the sanitizer's defenses.

Fix: To mitigate this, DOMPurify implemented stricter sanitization routines, including blocking specific HTML integration points and enhancing attribute validation to prevent such mutations.

The identified bypasses underscore the complexity of HTML sanitization and the need for continuous vigilance in web security. The DOMPurify team's prompt responses to these vulnerabilities highlight the importance of collaborative efforts between researchers and developers to maintain robust defenses against evolving threats.

07

Hidden Transformers in Windows ANSI

In January 2025, security researcher Orange Tsai unveiled a novel attack surface in Windows systems, termed WorstFit, which exploits the internal character conversion feature known as "Best-Fit." This research, presented at Black Hat Europe 2024, demonstrates how the Best-Fit behavior can be transformed into practical attacks, including path traversal, argument injection, and remote code execution (RCE), affecting numerous well-known applications.

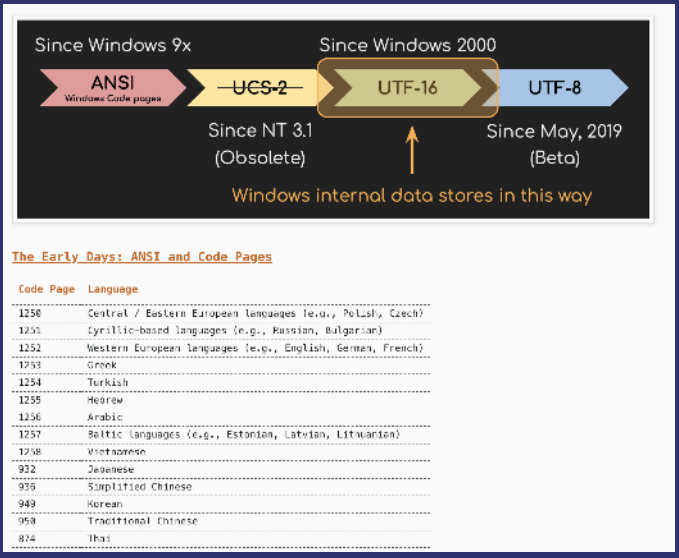


Image Credit: orange.tw

Windows systems support Unicode, allowing the use of diverse characters, including emojis and accented letters. Historically, Windows utilized ANSI encoding with various code pages to represent different languages. To maintain backward compatibility, Windows implemented two versions of APIs:

- ANSI APIs: Suffixed with 'A' (e.g., GetEnvironmentVariableA), these handle strings using the system's ANSI code page.
- Unicode APIs: Suffixed with 'W' (e.g., GetEnvironmentVariableW), these handle strings in Unicode.

When converting characters from UTF-16 (used in Unicode APIs) to ANSI (used in ANSI APIs), Windows employs a "Best-Fit" strategy. This means that if a character doesn't have a direct equivalent in the target code page, Windows maps it to the closest resembling character. For example, the infinity symbol (∞) might be mapped to the number '8' in certain code pages.

Exploiting Best-Fit Behavior: The WorstFit Attack Surface

The WorstFit research identifies how attackers can leverage the Best-Fit character mappings to perform various exploits:

1. Path Traversal: By crafting filenames with Unicode characters that, when converted via Best-Fit mapping, result in path traversal sequences (e.g., '../'), attackers can access unauthorized directories and files.
2. Argument Injection: Manipulating command-line arguments with specially chosen Unicode characters can lead to the injection of unintended commands or options, potentially resulting in arbitrary code execution.
3. Environment Variable Confusion: Altering environment variables using characters that transform into critical symbols (like '=' or '/') can disrupt program behavior or security mechanisms.

These vulnerabilities arise because many applications inadvertently use ANSI APIs or rely on libraries that do, leading to unintended character transformations and security gaps.

Mitigation Strategies

To defend against WorstFit attacks, consider the following measures:

1. Prefer Unicode APIs: Developers should use Unicode (W-suffixed) APIs to handle strings, minimizing the risk of unintended character conversions.
2. Input Validation: Implement strict validation for all user inputs, especially those involving file paths or command-line arguments, to detect and reject potentially malicious characters.
3. Awareness of Code Page Configurations: Understand the system's code page settings and how they might influence character conversions, particularly in internationalized applications.
4. Regular Security Audits: Conduct thorough code reviews and security assessments to identify and remediate instances where ANSI APIs are used or where Best-Fit mappings could introduce vulnerabilities.

08

HTTP Request Smuggling

In July 2024, security researchers Paolo Arnolfo (@sw33tLie), Guillermo Gregorio (@bsysop), and @medusa_I unveiled a critical vulnerability affecting thousands of Google Cloud-hosted websites utilizing the Google Cloud Platform (GCP) Load Balancer. This vulnerability, termed TE.0 HTTP Request Smuggling, exploits discrepancies in HTTP request parsing between front-end and back-end servers, leading to potential unauthorized access and data exposure.

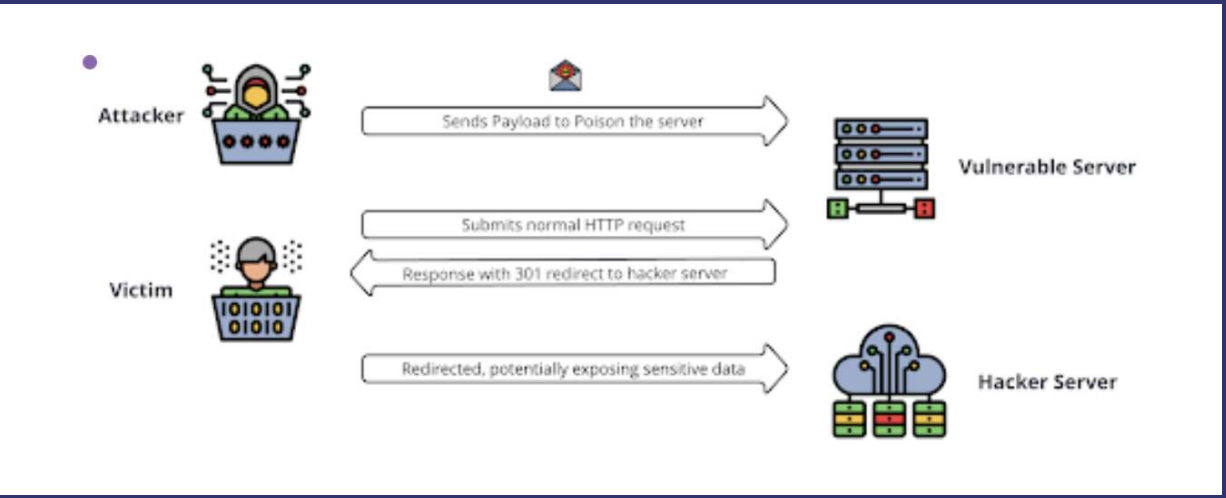


Image Credit: bugcrowd.com

HTTP Request Smuggling occurs when an attacker exploits inconsistencies in the processing of HTTP headers between two servers (typically a front-end proxy and a back-end server). By crafting malicious requests, attackers can manipulate how requests are interpreted, leading to various security issues, including unauthorized access, cache poisoning, and session hijacking.

The TE.0 Variant

The researchers identified a novel variant of HTTP Request Smuggling, dubbed TE.0, characterized by the following:

- Transfer-Encoding Header: The attack involves crafting requests with a Transfer-Encoding: chunked header.
- Chunk Size of Zero: A chunk size of zero (0) is used, which can cause the front-end server to misinterpret the end of the request body.
- Request Pipelining: By leveraging HTTP/1.1's pipelining feature, attackers can smuggle additional requests within the same connection.

This method exploits the differences in how the front-end and back-end servers handle the Transfer-Encoding header and the end of the request body, allowing attackers to inject and execute unauthorized requests.

Impact on Google Cloud Platform

The vulnerability was found to affect numerous websites hosted on GCP that utilized the GCP Load Balancer configured to default to HTTP/1.1 instead of HTTP/2. Notably, many of these sites were protected by Google's Identity-Aware Proxy (IAP), a security service that controls access to web applications. The exploitation of this vulnerability could lead to unauthorized access to protected resources, effectively bypassing the security measures provided by IAP.

Mitigation Strategies

To protect against TE.0 HTTP Request Smuggling attacks, consider the following measures:

1. Upgrade to HTTP/2: Configure GCP Load Balancers to use HTTP/2, which is less susceptible to this type of request smuggling.
2. Disable HTTP/1.1 Pipelining: If HTTP/1.1 must be used, disable request pipelining to prevent multiple requests from being sent in a single connection.
3. Strict Header Validation: Implement strict validation of HTTP headers to ensure that malformed or malicious headers are rejected.
4. Monitor for Anomalous Traffic: Deploy monitoring tools to detect unusual patterns in HTTP requests that may indicate an attempt at request smuggling.

09

Advanced SQL Injection Techniques

In August 2024, at DEF CON 32, security researcher Paul Gerste presented "SQL Injection Isn't Dead: Smuggling Queries at the Protocol Level," shedding light on advanced SQL injection techniques that exploit protocol-level vulnerabilities.

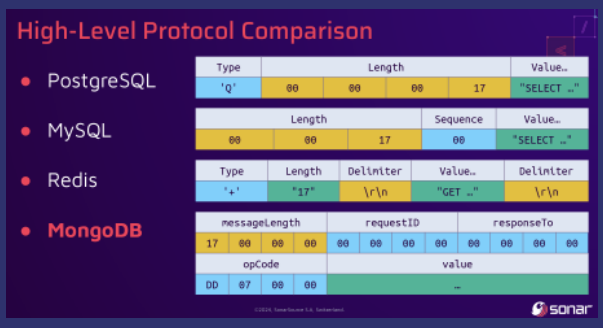


Image Credit: defcon.com

Gerste's research emphasizes that SQL injection vulnerabilities persist, particularly when attackers manipulate the database protocol directly. By crafting malicious payloads that exploit nuances in the communication between applications and databases, attackers can execute unauthorized queries, potentially leading to data breaches and system compromises.

Key Takeaways

- Protocol-Level Exploitation: Attackers can bypass traditional input validation and parameterization by targeting the underlying database protocol, allowing them to inject malicious queries.
- Complex Attack Vectors: These advanced techniques often involve an intricate understanding of database communication protocols, making detection and prevention more challenging.

Mitigation Strategies

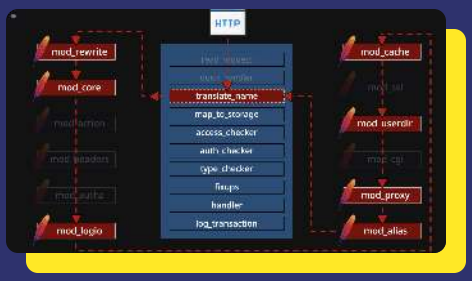
To defend against these sophisticated SQL injection attacks:

1. Deep Protocol Analysis: Implement monitoring tools that analyze database communication at the protocol level to detect anomalies indicative of injection attempts.
2. Strict Access Controls: Enforce least privilege principles for database accounts to minimize potential damage from successful injections.
3. Regular Security Audits: Conduct comprehensive security assessments that include protocol-level analysis to identify and remediate vulnerabilities.

10

Confusion Attacks

In August 2024, security researcher Orange Tsai unveiled a series of vulnerabilities in the Apache HTTP Server, collectively termed Confusion Attacks. These attacks exploit hidden semantic ambiguities within the server's architecture, leading to potential security breaches such as unauthorized access, information disclosure, and remote code execution (RCE).



The research identifies three primary types of Confusion Attacks:

1. Filename Confusion: This attack leverages inconsistencies in how the server interprets filenames, potentially allowing attackers to bypass access controls or execute unauthorized commands.
2. DocumentRoot Confusion: By exploiting ambiguities in the server's DocumentRoot configuration, attackers can access sensitive files outside the intended web directory, leading to information disclosure or code execution.
3. Handler Confusion: This involves manipulating the server's request handlers to process malicious requests, resulting in unintended behavior or security vulnerabilities.

Key Vulnerabilities and Exploitation Techniques

The research highlights several critical vulnerabilities, including:

- CVE-2024-38472: A Server-Side Request Forgery (SSRF) vulnerability on Windows platforms, allowing attackers to initiate unauthorized requests.
- CVE-2024-39573: A proxy encoding issue that can be exploited to bypass security restrictions.

Additionally, the study presents over 20 exploitation techniques and more than 30 case studies demonstrating the practical impact of these vulnerabilities.

Mitigation Strategies

To defend against Confusion Attacks, consider implementing the following measures:

1. Strict Input Validation: Ensure that all user inputs are thoroughly validated to prevent malicious data from being processed by the server.
2. Secure Configuration: Review and harden server configurations, particularly concerning filename handling, DocumentRoot settings, and request handlers.
3. Regular Updates: Keep the Apache HTTP Server and its modules up-to-date with the latest security patches to mitigate known vulnerabilities.
4. Comprehensive Logging and Monitoring: Implement robust logging and monitoring to detect and respond to suspicious activities promptly.

II

References and Credits

- Hijacking OAUTH flows via Cookie Tossing
- ChatGPT Account Takeover - Wildcard Web Cache Deception
- OAuth Non-Happy Path to ATO
- CVE-2024-4367 — Arbitrary JavaScript execution in PDF.js
- DoubleClickjacking: A New Era of UI Redressing
- Exploring the DOMPurify library: Bypasses and Fixes
- WorstFit: Unveiling Hidden Transformers in Windows ANSI!
- Unveiling TE.0 HTTP Request Smuggling: Discovering a Critical Vulnerability in Thousands of Google Cloud Websites
- SQL Injection Isn't Dead: Smuggling Queries at the Protocol Level
- Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server!
- Top 10 web hacking techniques of 2024

The "Top 10 Web Hacking Techniques of 2024" not only highlights significant research but also underscores the collaborative efforts within the security community to identify, analyze, and mitigate emerging threats. By studying these cutting-edge techniques, security professionals can enhance their understanding of current vulnerabilities and strengthen their defenses against future attacks.