Web Cache Poisoning: An In-Depth Investigation for Bug Hunters and Security Researchers

I. Executive Summary

Web Cache Poisoning (WCP) represents a critical cyberattack vector that leverages misconfigurations or vulnerabilities in web caching mechanisms to serve malicious content to unsuspecting users. This attack amplifies the impact of otherwise localized flaws, transforming them into widespread, persistent threats across web applications. The core of WCP lies in exploiting discrepancies between how caching layers (e.g., CDNs, reverse proxies) and origin servers interpret HTTP requests, particularly concerning "unkeyed" inputs—request components that influence the server's response but are not factored into the cache's key generation.

A significant characteristic of WCP is its ability to elevate the severity of vulnerabilities. For instance, a reflected Cross-Site Scripting (XSS) vulnerability, which typically requires a victim to click a specially crafted link for each instance of exploitation, can be transformed into a "stored" XSS via WCP. When a malicious response containing an XSS payload is cached, it becomes persistently available and is automatically served to all subsequent users whose requests match the poisoned cache key. This effectively automates the attack delivery, turning a client-side, user-specific vulnerability into a widespread, server-side-delivered threat. This dramatic increase in potential victims and attack duration significantly elevates the severity and potential damage, even without direct remote code execution (RCE). Consequently, even seemingly low-impact reflected vulnerabilities on pages likely to be cached warrant higher prioritization by bug hunters, as their potential for escalation through WCP is substantial. For defenders, this means that patching reflected XSS vulnerabilities, while necessary, is often insufficient; a comprehensive defense must also include rigorous auditing and secure configuration of all caching mechanisms to prevent such amplification.

Furthermore, a critical aspect of WCP is that poisoned content is served from a "trusted source"—the legitimate domain itself. Users are inherently less likely to question the authenticity or safety of content originating from a domain they trust, making phishing campaigns, malware distribution, or deceptive content highly effective. This exploitation of inherent trust undermines traditional security awareness training focused on identifying suspicious URLs or emails. This phenomenon suggests that traditional user education and endpoint security measures, while important, may be insufficient against WCP. Security solutions need to focus more on integrity verification at the caching layer itself, ensuring that the content being served from a trusted domain has not been tampered with. This necessitates a shift towards deeper inspection and validation within the web infrastructure.

Successful WCP attacks can lead to severe consequences, including widespread Cross-Site Scripting (XSS), Denial of Service (DoS), sensitive data theft, account takeovers, and the

distribution of malware. High-profile real-world incidents and substantial bug bounty payouts underscore the financial incentives and significant potential for damage. Effective defense against WCP necessitates a multi-faceted and proactive approach. This includes meticulous cache configuration, robust input validation and sanitization, strategic use of HTTP headers (Cache-Control, Vary), and the deployment of advanced security controls such as Web Application Firewalls (WAFs). Continuous monitoring and a deep understanding of the web stack's behavior are paramount to mitigating these evolving risks.

II. Fundamentals of Web Cache Poisoning

Definition and Core Concepts

Cache poisoning, in a broad sense, refers to a computer security vulnerability where invalid entries are maliciously introduced into a cache, which are then erroneously assumed to be valid when subsequently retrieved and used. This general concept branches into several specific attack types based on the targeted caching layer.

DNS Cache Poisoning, also known as DNS Spoofing, targets the Domain Name System. Attackers manipulate the cache memory of DNS resolvers, causing them to provide incorrect IP addresses for legitimate websites. This redirection sends users attempting to visit a genuine site to a different, often malicious, site without their knowledge or consent, exploiting the inherent trust between clients and DNS servers. The DNS system translates human-readable website addresses (e.g., example.com) into machine-readable IP addresses, and recursive DNS servers store information from previous requests in a local cache for faster responses. Attackers exploit this by replacing legitimate addresses with fake ones, hijacking browser sessions to redirect users to fraudulent or malicious sites for credential theft or data exfiltration.

Web Cache Poisoning (WCP), the primary focus of this report, specifically involves manipulating the caching mechanisms of web applications. Attackers craft specially designed HTTP requests that, when processed incorrectly by the application but still cached, inject harmful content into the cache. Subsequent legitimate users requesting the same resource then receive this poisoned content.

Web Caches are essential components of modern web infrastructure. They act as intermediate connections, such as reverse proxies or Content Delivery Networks (CDNs), positioned between clients and origin web servers. Their primary function is to store temporary copies of frequently accessed web content (like images, scripts, and pages) to significantly improve performance, reduce server load on origin servers, and accelerate content delivery to users.

Cache Keys are fundamental to how caches operate. When a cache receives an HTTP request, it calculates a unique "cache key" based on a predefined subset of the request's components. Typically, this key includes the HTTP method, URL path, query string, and Host

header. This key is used to determine if a matching response is already stored in the cache. If the key of an incoming request matches the key of a previous request, the cache considers them equivalent and serves the stored response directly to the user, without involving the backend server.

Unkeyed Parameters/Inputs are a critical vulnerability point in many caching systems. These are components of an HTTP request (e.g., certain headers, query parameters, or cookies) that can influence the backend server's response but are *not* included in the cache key. When such an unkeyed input causes a malicious change in the server's response, and that response is subsequently cached, it creates a WCP vulnerability. All subsequent requests matching the *same* (incomplete) cache key will then receive the poisoned content.

Relevance in Bug Hunting and Security Research

From a bug hunting and security research perspective, WCP is highly relevant because it transforms isolated vulnerabilities into widespread threats. Attackers can leverage a successful cache poison to alter the data users receive, redirect them to fraudulent websites, inject malware, or steal personal information on a mass scale. This fundamentally undermines the integrity and trustworthiness of the internet by manipulating the expected behavior of caching mechanisms to serve malicious content or redirect traffic without the end-user's knowledge or consent.

A key aspect of its relevance is its ability to escalate the impact of otherwise less severe flaws. For instance, a reflected Cross-Site Scripting (XSS) vulnerability, which typically requires a victim to click a crafted link, can be turned into a "stored" XSS via WCP. Once cached, this malicious payload affects all users accessing that resource, significantly amplifying its severity. This dramatic increase in potential victims and attack duration effectively escalates the vulnerability's severity from a typical medium to a high or even critical rating, even in the absence of direct remote code execution. This means that bug bounty programs and internal vulnerability assessments should assign a significantly higher severity to reflected vulnerabilities found on cacheable pages, acknowledging their potential for widespread, persistent impact through WCP. For developers, this underscores that any reflected user input on a page that might be cached should be treated with extreme caution and subjected to rigorous sanitization and validation, as it effectively becomes a potential stored attack vector, regardless of whether it is initially classified as a "reflected" flaw.

WCP facilitates the rapid spread of malicious content by exploiting the inherent trust users place in legitimate websites. It can also enable attackers to bypass traditional security controls (e.g., XSS and Cross-Site Request Forgery - CSRF protections) by manipulating cached content. This scale makes WCP a high-value target for bug hunters and a critical concern for security researchers. The continued prevalence of known WCP attack types, accounting for more than half of all identified vulnerabilities, further indicates that various vendors are still not adequately protecting against these established techniques. This

reinforces that WCP is far from a "solved" problem; it is a dynamic and persistent threat that requires continuous adaptation in security methodologies and proactive, adaptive defenses that go beyond static, one-time fixes.

III. Common Vulnerabilities and Misconfigurations

Web Cache Poisoning primarily arises from discrepancies in how different components of a web application's stack—the caching layer (e.g., CDN, reverse proxy) and the origin server—interpret HTTP requests. These inconsistencies, often stemming from misconfigurations or default behaviors, create opportunities for attackers to inject malicious content into the cache.

Discrepancies in Request Interpretation

The fundamental problem leading to WCP is that a cache system constructs its cache key based on a subset of HTTP request components, while other "unkeyed" inputs within the request can still influence the backend server's response. This mismatch in interpretation is the root cause of many WCP vulnerabilities.

- **GET Parameter Cloaking:** This vulnerability occurs when an attacker can separate query parameters using a semicolon (;) instead of the standard ampersand (&). While the backend server might correctly interpret the semicolon as a parameter separator (e.g., Python's parse_qsl method), the caching proxy, especially with default configurations, may not recognize it as such. This failure to include the subsequent parameters in the cache key allows malicious parameters to be cached as if they were part of a "safe" request, as observed in frameworks like Tornado, Bottle, and Rack.
- **GET Body Parameters (Fat GET):** Despite HTTP RFC stating that "A payload within a GET request message has no defined semantics," some proxies (e.g., NGINX) and application frameworks process body parameters in GET requests. If the cache layer ignores these body parameters when generating the cache key, an attacker can inject malicious content into the GET request body. This content can then override legitimate GET query parameters or inject extra parameters, leading to the caching and serving of a malicious response to other users. This issue was found in Tornado when used with NGINX (parameter override) and Flask (extra parameter injection). Cloudflare also caches contents of GET request bodies but does not include them in the cache key, making this a potential vulnerability for sites behind it.
- **URL Normalization Discrepancies:** Different components of the web stack (cache, web server, application) may apply varying levels of URL normalization (e.g., URL-decoding, path simplification) when processing requests. If the cache normalizes a URL for its cache key (e.g., fully URL-decoding it) but the backend server does not, an attacker can craft a request that appears different to the backend (e.g., using an

encoded question mark %3f instead of ?) but generates the same cache key. This can lead to a broken or malicious response being cached for the "normalized" URL, affecting legitimate users. A critical real-world example involved disabling Firefox updates globally by exploiting nginx's URL normalization process.

- **Path Mapping Discrepancies:** Similar to URL normalization, inconsistencies in how the cache and origin server map URL paths to resources or process delimiter characters can be exploited. This is a common vector for Web Cache Deception, where an attacker crafts a URL for sensitive dynamic content but appends a static file extension (e.g., .js, .css) that the origin server ignores. The cache, however, might interpret the extension as a caching directive, leading to the sensitive dynamic content being cached as a static resource and served to other users.
- **Cache Key Injection:** In certain configurations, if a cache bundles components of the cache key into a string without proper escaping of delimiters, an attacker can craft two semantically different requests that, due to the lack of escaping, result in identical cache keys. This allows for injection of malicious content into what should be a "keyed" header, making it exploitable.

The repeated observation that WCP vulnerabilities arise from "misconfigurations or default options that administrators might be unaware of" and "implementation and configuration quirks" highlights a systemic problem of configuration drift or interpretation mismatch. This occurs as complex web architectures are built by layering different technologies (web servers, reverse proxies, CDNs, application frameworks), each with its own parsing rules, default behaviors, and assumptions about HTTP requests. The fact that "many of the popular frameworks today are vulnerable to web cache poisoning out of the box" underscores that security is often not the default setting, especially when performance is prioritized. This implies that organizations need to implement robust configuration management and continuous security auditing that spans all layers of their web infrastructure, not just the application code. Understanding the precise interaction and interpretation rules between each component is critical. Simply deploying a WAF or patching known vulnerabilities might not be enough if the fundamental discrepancies in HTTP request processing across the stack persist. A comprehensive defense strategy for WCP must involve harmonizing configurations and validating behavior across the entire request-response flow.

Exploitable Unkeyed Input Vectors

Beyond the discrepancies in interpretation, specific HTTP headers, query parameters, and cookies are frequently abused due to not being included in the cache key while still influencing the backend response.

- **HTTP Headers:** Common unkeyed headers include X-Forwarded-Host (used to generate resource URLs),

User-Agent (used for dynamic content),

Content-Type, X-Forwarded-Scheme (can cause redirect loops),

X-HTTP-Method-Override (can force empty responses),

Origin, X-Forwarded-Proto, X-Forwarded-SSL, Accept-Encoding, Transfer-Encoding, If-Modified-Since, ETag, If-Match, If-Range, Upgrade, Authorization, X-Auth-User, and Auth-Key.

- **Query Parameters:** Unkeyed query parameters that are reflected unsanitized in the response are a common source of XSS vulnerabilities that can be amplified by WCP.
- **Cookies:** If cookies are used to dynamically generate content (e.g., preferred language) but are not part of the cache key, an attacker can manipulate them to poison the cache.
- **HTTP Header Oversize Attack:** This exploits differing limits on HTTP request header length between the cache server and the web server. An attacker sends a request with a header length that the cache server accepts but the web server rejects, leading to an error response being cached.
- **HTTP Meta Character Attack:** This involves injecting harmful metacharacters (e.g., newline, carriage return) into request headers. The cache server might tolerate these, but the web server processes them incorrectly, triggering an error page that then gets cached.
- **Blacklist Attack:** This exploits inconsistencies in blacklist support between cache servers and Web Application Firewalls (WAFs). An attacker sends a request with a malicious string (e.g., a security scanner User-Agent, phishing site Referer, or common attack payloads), which the web server's WAF blocks with a 403 Forbidden Access. If the cache server incorrectly caches this response, it can block normal users.

The suggestion to restrict caching to "truly static" resources often overlooks a critical nuance. The detailed examples of "Fat GET" vulnerabilities and URL normalization discrepancies illustrate that what appears static to a developer based on file extensions or typical HTTP methods might be dynamically influenced by malicious input due to subtle protocol quirks or framework behaviors. For instance, a GET request with a body, while non-standard, can be processed by some backends, and if the cache does not key on the body, it becomes an attack vector. This implies that the definition of "static" for caching purposes is often oversimplified and does not account for the full spectrum of how various web components interpret HTTP requests. Therefore, security professionals and developers must rigorously re-evaluate what constitutes "static" content beyond superficial characteristics like file extensions. This requires a deep understanding of how all possible request components (headers, query parameters, body, path segments) are processed by every layer of the web stack—from the CDN to the application framework—and how these components might influence the response, even if they are not part of the standard cache key. This

necessitates a shift from a "blacklist" approach (do not cache dynamic content) to a "whitelist" approach (only cache content explicitly verified as truly static and immune to input manipulation).

Chaining with Application Flaws

WCP is rarely a standalone attack; it is typically used as a delivery mechanism to amplify the impact of other underlying web application vulnerabilities.

- The most common chain involves turning a reflected XSS into a stored XSS, affecting a broad user base.
- It can also be chained with Host Header attacks , open redirects , or unsafe handling of resource imports (e.g., dynamically generated URLs for JavaScript files).
- WCP can facilitate large-scale phishing campaigns by serving fake websites that appear to originate from legitimate, trusted domains.
- Another significant impact is Denial of Service (DoS) attacks. By caching error responses (such as HTTP 500, 400 Bad Request) or creating infinite redirect loops, attackers can render legitimate pages or entire sections of a website inaccessible to users.
- Furthermore, WCP can enable attackers to bypass certain security controls (e.g., XSS and CSRF protections) by manipulating the content served from the cache.

IV. Methodologies and Testing Techniques for Bug Hunters

Effective web cache poisoning (WCP) detection and exploitation require a systematic methodology that combines careful observation, manual crafting of requests, and automated assistance. The process moves from understanding the caching mechanism to identifying exploitable inputs and finally, successfully poisoning the cache.

Identifying Cache Behavior

A foundational step in WCP testing is to thoroughly understand how the target's caching mechanism operates. This involves discerning whether a response is being served from a cache or directly from the origin server.

- **Explicit Cache Headers:** The most straightforward method is to look for specific HTTP response headers that explicitly indicate whether a response was served from the cache (a "cache hit") or fetched from the origin server (a "cache miss"). Common headers include X-Cache (e.g., HIT, MISS), CF-Cache-Status (e.g., HIT), Age (time in seconds the resource has been stored in a proxy cache), and Cache-Control directives (max-age, public, private, no-store, no-cache, must-revalidate). For instance, an

X-Cache: HIT header confirms the response came from the cache.

- **Inferred Cache Behavior:** In the absence of explicit headers, cache behavior can sometimes be inferred through observation. This includes analyzing response timing (cached responses are typically faster) or looking for dynamic content that might change on a cache miss.
- **Testing for Cacheable Responses:** Actively testing if various types of responses are cached is crucial. This includes attempting to cache error responses (e.g., HTTP 400, 404, 500) by injecting malformed headers or invalid inputs, then checking if subsequent legitimate requests receive the cached error. Some services, like Akamai, are known to cache certain error responses by default for a short duration.

Discovering Unkeyed Inputs

The success of any WCP attack hinges on the manipulation of "unkeyed inputs"—request components that influence the server's response but are ignored by the cache key.[4]

- **Manual Identification:** This involves systematically adding random or speculative inputs (such as custom HTTP headers, new query parameters, or cookie values) to requests and carefully observing their effect on the backend server's response. The impact might be obvious (e.g., direct reflection of the input in the response) or more subtle (e.g., triggering a different response, altering a URL). Tools like Burp Comparer can assist in side-by-side comparison of responses to spot these subtle changes, though this method can be labor-intensive.
- **Automated Identification (Param Miner):** For greater efficiency, the Param Miner extension for Burp Suite is highly recommended. This tool automates the process of identifying unkeyed inputs by sending requests with an extensive, built-in list of speculative headers and parameters. It then logs any inputs that cause a change in the response, making the discovery process significantly faster. Param Miner can also perform specialized "Fat GET" scans to identify vulnerabilities related to GET requests with bodies.
- **Caution with Live Websites (Cache Busters):** When testing on live production websites, there is a significant risk of inadvertently poisoning the cache and serving malicious content to real users. To mitigate this, it is crucial to ensure that every test request has a unique cache key. This is achieved by adding a "cache buster"—a unique, random parameter (e.g., ?cb=1234) to the request line. Param Miner offers an option to automatically add a cache buster to every request, which is highly recommended for safe testing.

The recurring theme across multiple sources is that components are often excluded from the cache key for "performance reasons". This highlights a fundamental tension in web application design: optimizing for speed and efficiency often leads to simplified cache keys, which inadvertently create critical security vulnerabilities. The recommendation to "rewrite

the request instead" when considering excluding something from the cache key for performance highlights that secure performance optimization is possible, but it requires more deliberate architectural design than simply dropping parameters from the cache key. This indicates that security is often sacrificed for perceived performance gains, leading to a "fast but insecure" default. Security researchers and bug hunters should therefore specifically target web applications where performance optimization is a clear priority, as these environments are more likely to have simplified or misconfigured cache keys. They should look for instances where performance considerations have led to a compromise in the completeness of the cache key. For developers, this underscores the importance of integrating security considerations from the outset of caching design, understanding that performance gains should never come at the cost of ignoring inputs that can influence the response, even if they seem minor or non-standard. This necessitates a shift towards "secure by design" principles for caching.

Eliciting and Caching Harmful Responses

Once an unkeyed input is identified, the next step is to understand precisely how the website processes it. This understanding is essential for crafting a payload that will elicit a harmful response.

If the input is reflected in the server's response without proper sanitization, or if it is used to dynamically generate other data (e.g., URLs for resource imports), it represents a potential entry point for WCP. Attackers must then craft malicious requests containing the desired payload (e.g., an XSS payload, a redirect URL to a phishing site, or an input that triggers an error). They also need to manipulate cache control headers or other request elements to ensure that this malicious response is indeed cached by the intermediary proxy or CDN. This often involves repeatedly sending the crafted request and monitoring the response headers until the payload is reflected and an

X-Cache: HIT header is observed, confirming that the poisoned response has been successfully stored in the cache.

The Cache Oracle Approach (Advanced Technique)

For more sophisticated WCP testing, especially when dealing with complex caching behaviors or "Web Cache Entanglement" scenarios, the "cache oracle" approach is invaluable.

- **Identifying a Cache Oracle:** This involves selecting a specific, cacheable endpoint on the target site that ideally reflects the entire URL and at least one query parameter. Crucially, this endpoint must provide a clear indication of whether a request resulted in a cache hit or miss (e.g., via CF-Cache-Status: HIT or by observing dynamic content changes or response timing).

- **Probing Key Handling:** Once a cache oracle is identified, the next step is to systematically "probe key handling." This involves sending a series of slightly different requests to the cache oracle and observing if subsequent requests result in a cache hit despite the subtle differences. This process helps to identify if and how the request is transformed or normalized when it is saved in the cache key. Common exploitable transformations detectable through this probing include the removal of specific query parameters, the exclusion of the entire query string, the removal of the port from the Host header, or URL-decoding behaviors. This deep understanding of cache key generation is critical for crafting advanced WCP exploits.

The methodologies emphasize the importance of actively identifying cache hits/misses and the critical use of cache busters. The statement that "the only way to actually know for sure is to go ahead and try to poison it, then follow up with a second request to verify" reveals that WCP testing is inherently active, iterative, and requires direct, empirical observation of cache behavior, rather than relying solely on theoretical analysis or passive scanning. This highlights a practical, almost scientific, approach to bug hunting, where hypotheses about cache behavior are constantly tested and refined. This implies that fully automated vulnerability scanners, while useful for initial broad sweeps, may not be sufficient for identifying and confirming subtle WCP vulnerabilities. Manual testing, augmented by intelligent tooling like Param Miner, remains critical for deep-dive WCP analysis. Bug hunters must develop a systematic, iterative testing process, being prepared for multiple steps of probing, payload crafting, and verification to confirm cacheability and successful exploitation. This also underscores the need for safe testing practices (cache busters) to prevent unintended impact on live systems.

V. Tools and Scripts for Web Cache Poisoning

The effective identification and exploitation of web cache poisoning vulnerabilities are significantly aided by specialized tools and scripts. These range from general-purpose web penetration testing suites to dedicated scanners and extensions.

Manual and Semi-Automated Tools

- **Burp Suite:** This is the industry-standard toolkit for web penetration testing and bug bounty hunting, offering a comprehensive suite of tools invaluable for WCP research.
    - **Repeater:** Essential for crafting and re-sending individual HTTP requests with modified headers, parameters, or bodies, and meticulously observing the backend server's responses and cache behavior.
    - **Comparer:** Used for side-by-side comparison of HTTP responses to identify subtle differences or reflections caused by injected inputs, aiding in the discovery of unkeyed parameters.

    - **Intruder:** Can be used for automated brute-forcing of parameters and headers, as well as testing for delimiter discrepancies by systematically injecting various characters and observing response changes.
    - **Scanner (DAST):** Burp Scanner, particularly the Professional and Enterprise editions, includes dynamic application security testing (DAST) capabilities that can help identify generic web cache poisoning vulnerabilities. It aims to improve security posture and prioritize manual testing.
    - **Extender:** Allows for extending Burp Suite's functionality through custom extensions. This framework enables the integration of specialized tools like Param Miner.
    - **Param Miner:** This is a highly recommended Burp Suite extension available from the BApp store. It automates the process of identifying hidden, unlinked, and unkeyed parameters by sending requests with a vast, built-in list of speculative headers and parameters. It uses advanced diffing logic and a binary search technique to guess a large number of parameter names per request. Param Miner logs any inputs that affect the response, making the discovery process significantly faster and more comprehensive than manual methods. It also offers options for automatically adding a cache buster to every request, crucial for safe testing on live environments.

Automated Scanners

- **Web Cache Vulnerability Scanner (WCVS):** Developed by Hackmanit and Maximilian Hildebrand, WCVS is a fast and versatile Go-based command-line interface (CLI) tool specifically designed for testing web cache poisoning and web cache deception.
    - **Features:** WCVS supports a wide array of WCP techniques, including unkeyed header poisoning, unkeyed parameter poisoning, parameter cloaking, Fat GET, HTTP response splitting, HTTP request smuggling, HTTP header oversize (HHO), HTTP meta character (HMC), and HTTP method override (HMO). It also supports multiple web cache deception techniques. The scanner can analyze a web cache before testing to adapt for more efficient operations, generate JSON reports, crawl websites for additional URLs, route traffic through a proxy (e.g., Burp Suite), and limit requests per second to bypass rate limiting.
    - **Integration:** Its customizable nature allows for easy integration into existing CI/CD pipelines, enabling automated security assessments.
- **ZAP (OWASP Zed Attack Proxy) - Param Digger:** Param Digger is a new add-on for ZAP that focuses on discovering unlinked, unreferenced, or hidden parameters that can influence application behavior. Similar to Param Miner, it aims to increase the

attack surface by identifying these parameters, which can then be used in WCP-based attacks, as well as SQL injection, XSS, and SSRF. The add-on plans to incorporate "Parameter Mining" using "FAT GET" requests and other techniques suggested in "Practical Web Cache Poisoning" and "Web Cache Entanglement".

- **Tenable Web App Scanning:** Tenable's web application scanning solution can identify web cache poisoning and web cache deception vulnerabilities through its classic and API scan features. It includes dedicated plugins for generic web cache deception generic web cache poisoning and DoS web cache poisoning .
- **Snyk:** Snyk offers vulnerability scanners that can automatically identify common cache poisoning vulnerabilities, aiding developers in decreasing their threat exposure.

VI. Detection and Mitigation Strategies

Effective defense against web cache poisoning requires a multi-layered approach, encompassing robust detection mechanisms and comprehensive mitigation strategies. Attackers, however, are constantly seeking ways to bypass these defenses.

Detection Strategies

Detecting web cache poisoning can be challenging because the malicious data is often served from a legitimate cache, making it appear as trusted content. However, several methods can help identify potential attacks:

- **Monitoring Cache Headers:** Regularly checking HTTP response headers like X-Cache (e.g., HIT, MISS), Age, Cache-Control, and Pragma can provide crucial information about caching behavior and indicate if a response is being served from the cache. Unexpected HIT responses for dynamic content or rapid changes in Age headers might signal an issue.
- **Traffic Analysis:** Implementing comprehensive traffic analysis tools can help detect abnormal traffic patterns indicative of a cache poisoning attempt. Unusual spikes in requests or anomalous response patterns can serve as red flags. Automated DNS security tools are particularly effective for detecting DNS cache poisoning by analyzing queries and responses in real-time for anomalies.
- **Log Monitoring:** Regularly monitoring web server and proxy logs for suspicious activities is vital. This includes looking for unexpected changes in cache content, altered headers, or content mismatches, which can indicate a cache poisoning attack in progress.
- **Content Validation:** Periodically validating cached content against the original source helps ensure integrity. This can involve comparing hashes or content to detect unauthorized modifications.

- **Security Information and Event Management (SIEM) Systems:** SIEM solutions can be configured to detect cache poisoning events by correlating data from various sources across the infrastructure and triggering alerts when suspicious activities occur.
- **Cache-Busting Techniques:** While primarily a testing technique, cache-busting (adding unique query strings or headers to requests) can also help detect potential cache attacks by preventing the cache from serving stale or potentially poisoned data, forcing a fresh response from the origin. If a page behaves differently with a cache buster, it suggests caching is active and potentially exploitable.
- **Vulnerability Scanners:** Dynamic Application Security Testing (DAST) tools, such as Tenable Web App Scanning, can identify generic web cache poisoning and web cache deception vulnerabilities using dedicated plugins. Snyk also offers vulnerability scanners that can automatically identify common cache poisoning vulnerabilities.[8]

Mitigation Strategies (Defenders' Perspective)

Preventing web cache poisoning requires a proactive and multi-faceted approach, addressing both configuration and application-level security.

- **Cache Key Normalization:** This method standardizes the processing of HTTP requests by removing variability in how they are handled. This limits the exploitation of the caching mechanism, making it more difficult for attackers to inject or alter cached content. Normalization focuses on treating similar requests as identical, reducing opportunities for cache poisoning and enhancing the resilience of web caching systems.
- **Cache-Control Headers:** These HTTP headers play a crucial role in defining caching policies for web applications. Web developers can use directives to control how and whether caching mechanisms should store responses. For example, Cache-Control: no-store instructs caches not to store a copy of the response, while Cache-Control: private ensures the response is stored only in private caches (like a user's browser) and not in shared caches (like an ISP or CDN cache), which are more susceptible to poisoning attacks. Properly configuring these headers helps reduce the risk of poisoning, ensuring that only the intended, unaltered content reaches the end-user.
- **Validate User Input:** This is a critical defense mechanism against web cache poisoning. It involves checking and sanitizing any data received from users before the web application processes it or stores it in the cache. The goal is to ensure that only valid and expected data is accepted, thereby preventing attackers from injecting malicious content through crafted requests. Input validation can block attempts to exploit vulnerabilities in the application's logic that might lead to cache poisoning, such as unexpected combinations of parameters or malformed URLs. Effective input validation typically uses an allowlist approach, accepting only known, safe input

patterns, and may include strict type checking, length verification, and pattern matching against regular expressions.

- **Use Web Application Firewalls (WAFs):** WAFs are an essential security layer for defending against web cache poisoning and other web-based threats. A WAF inspects incoming web traffic and filters out malicious requests based on predefined or dynamic rulesets. By placing a WAF between the internet and the web application, it acts as a gatekeeper, preventing harmful requests from reaching the server or corrupting the cache. To counter WCP, WAFs can be configured to detect and block requests that attempt to exploit known vulnerabilities or exhibit patterns indicative of an attack, such as unusual query parameters, unexpected HTTP headers, or anomalous request paths. WAFs can also be regularly updated with new rules to adapt to emerging threats, providing an evolving defense mechanism.
- **Vary Header:** The Vary header helps mitigate WCP by allowing the server to specify which parts of the HTTP request should be taken into account by the cache. This means the cache will store different versions of a resource based on the values of the specified headers. For example, if a server includes

Vary: User-Agent in its response, the cache will store separate copies of the resource for different User-Agent strings. If an attacker tries to poison the cache by sending a malicious User-Agent header, that malicious content will only be served to users who send the *exact same* User-Agent header, preventing widespread impact.

- **DNSSEC Adoption:** For DNS cache poisoning, adopting Domain Name System Security Extensions (DNSSEC) is a key prevention method. DNSSEC uses public key cryptography to verify the integrity and authenticity of DNS data, making it difficult for attackers to insert falsified information into the DNS cache.
- **HTTPS for DNS Traffic (DoH):** Encrypting DNS traffic using DNS over HTTPS (DoH) improves privacy and hides DNS queries from view, making it harder for attackers to intercept or alter DNS information.
- **Zero Trust Approach for DNS Servers:** Applying Zero Trust principles to DNS server configuration ensures that all users, devices, applications, and requests are considered compromised until authenticated and continuously validated. DNS becomes a valuable control point for scanning internet addresses for malicious behavior.
- **Secure Caching Mechanisms:** Using cryptographic signatures or hashes to verify the integrity of cached data can help ensure that data stored in caches is not tampered with or modified.

- **Regular Software Updates and Patching:** Establishing an optimal cadence for updating and patching DNS and web application software reduces the chance that attackers can exploit known or zero-day vulnerabilities.
- **Runtime Application Self-Protection (RASP):** RASP provides real-time attack detection and prevention from within the application's runtime environment, stopping external attacks and injections and reducing vulnerability backlogs.
- **Restrict Fat GET Requests:** Organizations should configure their systems to not accept GET requests with bodies, as some third-party technologies might permit this by default, creating a vulnerability.
- **Patch Client-Side Vulnerabilities:** Even seemingly unexploitable client-side vulnerabilities should be patched, as unpredictable quirks in cache behavior or other future discoveries could make them exploitable.

Attacker Bypass Techniques

Attackers constantly seek ways to circumvent defenses.

- **Automated Re-poisoning:** If a cache has a short Time-to-Live (TTL) for error responses (e.g., Akamai caching 400 errors for 10 seconds), an attacker can use a script to automate the re-injection of the malicious payload every few seconds, effectively maintaining a Denial of Service.
- **Exploiting Configuration Gaps:** Attackers target discrepancies between how the cache and the origin server interpret requests, or how different layers of caching interact. Even if a WAF is in place, if the WAF's rules or the cache's key generation do not account for subtle HTTP protocol variations or unkeyed inputs, bypasses are possible.
- **Chaining Vulnerabilities:** Attackers often chain WCP with other vulnerabilities (e.g., WAF bypasses for XSS payloads, as seen in the Expedia case) to achieve their objectives, making defense more complex.

VII. Learning Resources and Relevant Classifications

For those seeking to deepen their understanding and practical skills in web cache poisoning, a variety of learning resources and standardized classifications are available.

Learning Resources, Labs, and Tutorials

- **PortSwigger Web Security Academy:** This is a highly recommended resource for learning about web cache poisoning. It offers comprehensive tutorials that break down the process of constructing a WCP attack, including identifying unkeyed inputs, eliciting harmful responses, and getting responses cached. The academy also provides interactive labs to practice exploiting cache design flaws and

implementation quirks on deliberately vulnerable targets. Key research papers like "Practical Web Cache Poisoning" and "Web Cache Entanglement" are also referenced, providing theoretical depth.

- **PentesterLab:** Offers exercises like "Cache Poisoning 01," which details how to exploit applications vulnerable to cache poisoning. This includes practical steps such as identifying sensitive information, locating self-XSS, preparing payloads, and manipulating Varnish to cache and trigger a full XSS attack.
- **Hack The Box Academy:** Provides modules on "Abusing HTTP Misconfigurations," which cover web cache poisoning in detail. This includes explaining what web caches are, how they work, configuration, identifying keyed and unkeyed parameters, and exploiting basic and advanced WCP vulnerabilities.
- **Cobalt.io Blog:** Features deep dives into web cache poisoning attacks, providing step-by-step guides for executing various WCP techniques, such as those via unkeyed query strings or fat GET requests, often using Burp Suite for demonstration.
- **Snyk Blog:** Offers articles on cache poisoning, including explanations of common vulnerabilities and how to avoid them.
- **OWASP Community:** The OWASP website provides a dedicated page for "Cache Poisoning" under its "Attacks" category, offering a description, examples, and general remediation advice.

Relevant OWASP Categories and CVEs

Standardized classifications help categorize and track web cache poisoning vulnerabilities.

- **OWASP Categories:**
    - **Attacks:** Web cache poisoning is categorized under "Attacks" within the OWASP framework, reflecting its nature as a technique used by malicious actors to exploit weaknesses.
    - **Vulnerabilities:** The underlying weaknesses that enable cache poisoning, such as HTTP Response Splitting or flaws in web application logic, would fall under "Vulnerabilities".
    - **Controls:** Mitigation and prevention strategies are covered under "Controls".
- **CVEs (Common Vulnerabilities and Exposures):** Specific WCP vulnerabilities are assigned CVE IDs to facilitate tracking and communication.
    - **CVE-2021-23336:** This CVE relates to a GET parameter cloaking vulnerability found in Python's parse_qsl method, which parses URL query parameters using both semicolons and ampersands. This vulnerability affected frameworks like Tornado, Bottle, and Rack, leading to web cache poisoning.

    - **CVE-2024-12314:** This CVE affects the Rapid Cache plugin for WordPress (versions up to 1.2.3). It is vulnerable to cache poisoning due to the plugin storing HTTP headers in cached data, which can lead to unsanitized custom HTTP headers causing Cross-Site Scripting. It has a CVSS 3.1 score of 7.2 (High severity).
    - **CVE-2025-49826:** This critical cache poisoning bug was found in Next.js (versions 15.1.0 to before 15.1.8, and backported to 15.0.4-canary.51). It could lead to a Denial of Service (DoS) condition where HTTP 204 "No Content" responses are erroneously cached for static pages, rendering critical content inaccessible. This vulnerability has a CVSS score of 7.5 (High severity).
    - **CVE-2025-36852 (CREEP):** This critical vulnerability, "Cache Race-condition Exploit Enables Poisoning," affects remote cache plugins across numerous build systems, including Nx. It allows contributors with pull request privileges to inject compromised artifacts into production environments without detection. This is fundamentally different from traditional WCP as it exploits the artifact construction phase before transit or storage security measures. It is caused by a race condition where "first-to-cache-wins," meaning a malicious branch can upload a build artifact that will be used everywhere that source state appears, including production deployments. It requires low privileges and can lead to code execution, data exfiltration, and lateral movement, bypassing traditional security protections like checksums, encryption, and access control.

- **CWE (Common Weakness Enumeration):** CVE-2025-49826 is associated with CWE-444: Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling'), highlighting the underlying weakness.
- **CAPEC (Common Attack Pattern Enumeration and Classification):** Cache poisoning is classified under CAPEC-141.

VIII. Platform Handling and Ethical Considerations

The handling of web cache poisoning vulnerabilities varies across bug bounty platforms like HackerOne and Bugcrowd, and ethical considerations are paramount for security researchers engaged in testing.

How Platforms Handle WCP (HackerOne, Bugcrowd)

Bug bounty platforms serve as intermediaries between security researchers and organizations, defining the scope and rules for vulnerability disclosure.

- **Scope Variations:** The most significant factor in how WCP is handled by platforms is the program's scope. Some programs explicitly exclude "Cache poisoning" from their

in-scope vulnerabilities. This means that even if a WCP vulnerability is found, it may not be eligible for a bounty or points-based compensation.

- **Denial of Service (DoS) Classification:** WCP often leads to Denial of Service (DoS) attacks, where poisoned cache entries render legitimate pages or entire sections of a website inaccessible. However, DoS attacks are frequently listed as out of scope for many bug bounty programs, particularly traditional DoS attacks that rely on resource exhaustion.[26] This can create ambiguity for WCP-related DoS.
    - **Distinction from Traditional DoS:** Researchers often argue that WCP-based DoS (sometimes referred to as Cache Poisoning Denial of Service or CPDoS) is fundamentally different from traditional DoS. CPDoS exploits processing differences between HTTP parsers or misconfigurations, rather than simply exhausting resources. Companies like HackerOne, Glassdoor, and GitLab have reportedly rewarded CPDoS reports generously, even when traditional DoS attacks were out of scope, recognizing this distinction.
    - **Impact Assessment:** The severity and eligibility for WCP-related DoS often depend on the *criticality* of the impact. If the issue of making files inaccessible is not deemed a "critical security issue" by the program, it may not be eligible for a bounty, even if it is a bug in cache configuration.
- **Reward Structures:** When WCP vulnerabilities are in scope and deemed impactful, they can lead to significant payouts. Examples include PayPal's $9,700 bounty for a DoS via WCP , GitHub's $10,000 for a Fat GET poisoning vulnerability , and Expedia Group's reward for a WCP leading to stored XSS and account takeover. The payout is heavily influenced by the actual impact (e.g., widespread XSS, critical data theft) and the popularity of the affected page.
- **Safe Harbor:** Platforms like HackerOne offer "Safe Harbor" policies, which commit to protecting researchers who operate within the defined rules. The more closely a researcher's behavior matches the program guidelines, the more protection they can expect if a difficult disclosure situation escalates.

Legal and Ethical Considerations in Testing This Area

Testing for web cache poisoning, especially on live systems, carries significant legal and ethical responsibilities.

- **Responsible Disclosure:** Researchers are expected to report vulnerabilities responsibly. This involves:
    - **Following Rules of Engagement:** Adhering strictly to the scope and rules set forth by the security team or bug bounty program. Testing outside the defined scope can lead to legal repercussions.

    - **Avoiding Exploitation for Personal Gain:** Using discovered vulnerabilities for personal gain, accessing unauthorized data, or causing harm is illegal and unethical. The goal is to improve security, not to gain unauthorized access.
    - **Minimizing Impact:** Researchers should use the least invasive test possible (e.g., calling a 1x1 image or a non-existent page on a web server) and avoid destructive testing.
    - **Data Protection:** If sensitive data is accidentally accessed, it must not be further exposed. Researchers should immediately report it and avoid accessing it further.
    - **Maintaining Confidentiality:** Vulnerabilities should be kept confidential until the organization has had reasonable time to fix them. Public disclosure without permission is generally discouraged.
    - **Clear Documentation:** Keeping detailed records of testing activities, including what was tested, how, and the results, is crucial for clarity and legal defense if questions arise.
- **Computer Fraud and Abuse Act (CFAA):** This federal anti-hacking law in the United States makes it illegal to intentionally access a computer "without authorization" or "in excess of authorization". The ambiguity of these terms has historically allowed prosecutors to bring criminal charges for actions like violating terms of service, which is a significant concern for security researchers.
- **GDPR Implications:** For organizations operating in or serving the EU, the General Data Protection Regulation (GDPR) mandates strict protection of personal data. If a WCP attack leads to a data breach involving users' identifiable data being cached and exposed, the organization could face significant fines (up to 4% of annual revenue) and must notify data protection authorities. This underscores the importance of encryption for cached data and robust security measures to prevent such leaks.
- **Safe Testing Practices:** When testing for WCP on live websites, there is an inherent risk of inadvertently causing the cache to serve generated responses to real users. To prevent this, it is critical to use a "cache buster"—a unique parameter added to the request line—to ensure that test requests have a unique cache key and will only be served to the researcher.This practice is a cornerstone of ethical WCP testing.

IX. Trends, Automation, and Future Projections

The landscape of web cache poisoning is continuously evolving, driven by new research, increasing automation, and the dynamic nature of web technologies.

Current Trends

- **Discovery of New Attack Vectors:** Recent systematic, large-scale evaluations, such as the HCache research, have identified new attack vectors. This study found 7 new attack vectors stemming from previously unexplored caching headers and HTTP request manipulations. These include:
    - **Internal Route Header Attack:** Abusing special CDN headers for internal transmission (e.g., Fastly-Client-Ip, X-Amz-Website-Redirect-Location) to trigger CDN exceptions.
    - **HTTP Authentication Header Attack:** Manipulating authentication headers (e.g., Authorization, Auth-Key) with illegal values to cause cached denial-of-access responses.
    - **HTTP Protocol Header Attack:** Exploiting headers like X-Forwarded-Scheme or X-Forwarded-Proto that identify client connection protocols, leading to cached redirects or DoS.
    - **HTTP Range Header Attack:** Causing cached error responses by sending malformed Range requests.
    - **HTTP If Header Attack:** Triggering cached 4xx/5xx errors by manipulating conditional request headers like If-Match or If-Modified-Since.
    - **HTTP Upgrade Header Attack:** Initiating unsupported or malformed Upgrade requests (e.g., Upgrade: HTTP/3.0) to cause cached incorrect status codes.
    - **HTTP Coding Header Attack:** Injecting malformed values into encoding headers (e.g., Accept-Encoding, Transfer-Encoding) to trigger cached server exceptions.

These new vectors highlight that protection against a single attack method is insufficient; any non-cache key field could potentially be at risk.

- **Continued Prevalence of Known Attacks:** Despite being disclosed in previous research, known WCP attack types still account for more than half of all identified vulnerabilities. This indicates that various vendors are still not adequately protecting against these established WCP attacks. Common persistent vectors include HTTP Header Oversize, HTTP Method Override, HTTP Meta Character, Fat GET, HTTP Parameters, HTTP Forwarded Header, and Blacklist attacks.
- **Prevalence in HTTP/2:** WCP issues are still prevalent in HTTP/2. All vulnerabilities that existed in HTTP/1.1 were also present in HTTP/2. Furthermore, about 90% of websites share caches between HTTP/1.1 and HTTP/2, meaning that poisoning a cache with an HTTP/1.1 request can affect subsequent HTTP/2 requests, and vice versa. This suggests that attacks targeting HTTP/1.1 can still impact services utilizing HTTP/2 due to potential HTTP/2 to HTTP/1.1 transitions.

- **Cache Race-condition Exploit Enables Poisoning (CREEP):** A critical new development is CVE-2025-36852, known as CREEP. This vulnerability affects remote cache plugins across numerous build systems (e.g., Nx, and other bucket-based caching solutions). CREEP allows any contributor with pull request privileges to inject compromised artifacts into production environments without detection. It exploits a race condition where the "first-to-cache-wins" principle applies, meaning a malicious branch can upload a build artifact for a particular source file state that will then be used everywhere that source state appears, including production deployments. This is distinct because poisoning happens during artifact construction, before traditional transit or storage security measures. It bypasses checksums, encryption, and access control, requiring only low privileges but enabling code execution, data exfiltration, and lateral movement.

Automation in Exploitation and Protection

- **Automated Exploitation:** Tools like Burp Suite (with extensions like Param Miner) and Web Cache Vulnerability Scanner (WCVS) automate the discovery of unkeyed inputs and the testing of various WCP techniques. This automation allows attackers to scale their efforts, rapidly identifying and probing potential targets.
- **Automated Protection:** On the defensive side, solutions like Web Application Firewalls (WAFs) can be configured with dynamic rulesets to detect and block requests attempting to exploit known WCP vulnerabilities or exhibiting suspicious patterns. Dynamic Application Security Testing (DAST) tools are increasingly used in CI/CD pipelines to automatically scan for WCP vulnerabilities early in the development lifecycle. Runtime Application Self-Protection (RASP) offers real-time detection and prevention from within the application runtime.

Future Projections

The evolution of web cache poisoning is likely to continue along several trajectories:

- **Increased Sophistication of Attacks:** As caching mechanisms become more complex and integrated into modern web architectures (e.g., serverless, microservices), attackers will continue to find subtle discrepancies in how different layers process HTTP requests. The emergence of "Web Cache Entanglement" and CREEP illustrates this trend towards exploiting deeper, more nuanced architectural flaws rather than simple misconfigurations.
- **AI and Machine Learning in Attack and Defense:** Artificial intelligence (AI) and machine learning (ML) will likely play an increasing role. Attackers may use AI to identify complex unkeyed input patterns or to craft highly evasive payloads that bypass traditional WAFs. Defenders, in turn, will leverage AI/ML for more sophisticated anomaly detection in cache behavior, real-time threat intelligence, and

adaptive WAF rulesets. The concept of "memory poisoning" in AI agents, where false information is stored and influences future operations, highlights a related future challenge in AI-driven systems.

- **Focus on Supply Chain and Build System Caches:** The CREEP vulnerability points to a critical future trend: the targeting of remote caches within software supply chains and build systems. As organizations increasingly rely on shared build artifacts and remote caching for efficiency, these systems become high-value targets. Traditional security measures are insufficient against such attacks, necessitating new approaches to secure the artifact construction and caching phases.
- **Continued Challenge of "Static" Content:** The inherent tension between performance optimization and security will persist. Developers will continue to grapple with accurately defining and securing "static" content in dynamic environments, and attackers will continue to exploit the hidden complexities of how such content is cached and served.
- **Broader Impact on Internet Integrity:** Given the critical role of caching in internet infrastructure, successful WCP attacks will continue to undermine the integrity and trustworthiness of online interactions, leading to more sophisticated phishing, malware distribution, and DoS campaigns.

X. Potential Impact and Severity Ratings

The potential impact of web cache poisoning is significant and can range from minor content defacement to widespread denial of service, sensitive data theft, and full account compromise. The severity rating typically reflects the worst-case scenario of a successful exploitation.

Impact Categories

- **Cross-Site Scripting (XSS) Amplification:** One of the most common and impactful consequences. WCP can transform a reflected XSS, which normally requires user interaction for each instance, into a "stored" XSS. Once cached, the malicious script is served to all subsequent users accessing the poisoned resource, leading to widespread session hijacking, account theft, or sensitive data exfiltration.
- **Denial of Service (DoS):** Attackers can cause legitimate pages or entire sections of a website to become inaccessible. This can be achieved by caching error responses (e.g., HTTP 500, 400 Bad Request, 204 No Content) or by creating infinite redirect loops. This disrupts service availability, potentially leading to financial losses and reputational damage.
- **Data Theft and Phishing:** WCP can facilitate large-scale phishing campaigns by redirecting users to fraudulent websites that appear legitimate. Users, believing they

are on a trusted site, may enter credentials or sensitive information, leading to data theft, financial theft, or identity theft.

- **Malware Distribution:** By redirecting users to malicious sites or injecting harmful scripts, WCP can lead to widespread malware infection on end-user devices. This can result in devices becoming part of botnets or being subjected to ransomware attacks.
- **Account Takeover:** When combined with other vulnerabilities (e.g., XSS that exposes session cookies or sensitive variables), WCP can enable attackers to hijack user sessions and take over accounts.
- **Interference with Security Updates:** DNS cache poisoning can redirect users away from legitimate update sites to malicious ones offering fake updates, thereby compromising users' systems and preventing critical security patches.
- **Bypassing Security Controls:** By manipulating cached content, attackers can circumvent website security measures designed to protect against various web-based attacks, including XSS and CSRF.
- **Bug Bounty Payouts:** Bug bounty platforms reflect the perceived severity and business impact of WCP vulnerabilities through their payouts.
    - **High Payouts:** Examples like GitHub's $10,000 and PayPal's $9,700 demonstrate that WCP can command top-tier rewards, especially when it leads to widespread DoS, XSS, or account takeover.
    - **Variable Payouts:** Payouts can vary based on the specific program's scope and how they classify DoS. Even if DoS is generally out of scope, if the WCP leads to a critical security impact (e.g., total unavailability of a critical service, or chaining to other high-impact vulnerabilities), it may still be rewarded.

The impact of WCP is heavily dependent on two key factors: the harmfulness of the injected payload and the amount of traffic on the affected page. A poisoned response on the homepage of a major website, for example, could affect thousands of users without any subsequent interaction from the attacker, leading to massive impact.

Conclusions

Web cache poisoning stands as a pervasive and evolving threat in the cybersecurity landscape, consistently demonstrating its capacity to transform localized vulnerabilities into widespread, high-impact attacks. The fundamental mechanism, rooted in the discrepancy between how caching layers and origin servers interpret HTTP requests, particularly concerning "unkeyed" inputs, remains a persistent challenge across modern web architectures.

The analysis underscores that WCP is not merely a theoretical concern but a practical reality, evidenced by numerous real-world exploits and substantial bug bounty payouts. Its ability to

amplify the severity of other vulnerabilities, such as turning a reflected XSS into a persistent, stored attack affecting a broad user base, fundamentally alters the threat model for web applications. Furthermore, the exploitation of inherent trust, where malicious content is served from a legitimate domain, bypasses traditional user security awareness and necessitates a deeper focus on integrity verification within the web infrastructure.

The continuous discovery of new attack vectors, coupled with the enduring prevalence of known vulnerabilities even in newer protocols like HTTP/2, highlights that the web cache attack surface is dynamic and expanding. The emergence of sophisticated techniques like "Web Cache Entanglement" and "CREEP" further demonstrates that attackers are targeting more nuanced architectural flaws and race conditions within build systems and supply chains, moving beyond simple misconfigurations. This suggests that the problem of WCP is far from being "solved" and demands ongoing vigilance.

For bug hunters and security researchers, WCP presents a high-value target. The methodologies for identifying and exploiting these vulnerabilities require a systematic and iterative approach, combining manual observation with intelligent tooling like Burp Suite's Param Miner. The emphasis on "cache busters" for safe testing on live environments is a critical ethical consideration, reflecting the potential for unintended widespread impact.

From a defensive standpoint, a multi-layered strategy is indispensable. This includes meticulous cache key normalization, judicious use of Cache-Control and Vary headers, rigorous input validation, and the deployment of advanced security controls such as Web Application Firewalls (WAFs) and Runtime Application Self-Protection (RASP). The need to re-evaluate what constitutes "static" content, considering the complex interplay of various web components, is paramount. Moreover, organizations must implement robust configuration management and continuous security auditing across their entire web stack to mitigate configuration drift and interpretation mismatches.

In conclusion, web cache poisoning is a complex and persistent threat that exploits the very mechanisms designed to improve web performance. Its evolving nature, significant impact potential, and the continuous tension between performance and security necessitate a proactive, adaptive, and comprehensive approach from both offensive and defensive security professionals. Continuous research, education, and the adoption of secure-by-design principles for caching will be crucial in mitigating this critical vulnerability in the evolving digital landscape.

BY - bithowl