Poisoned by the Host: Large-Scale Measurement of Host Name Poisoning in Web Applications

Rui Yang, Haoyu Wang, Zhicheng Sun, Zhengyu Liu, Yinzhi Cao
Johns Hopkins University
ryang54@jh.edu, hwang335@jh.edu, zsun54@jh.edu, zliu192@jhu.edu, yinzhi.cao@jhu.edu

***Abstract*—Host Name Poisoning (HNP) allows an adversary to craft malicious host names at the client side to hijack server-side web application's functionality. Prior works have studied potential consequences of HNP, such as password resetting, cache poisoning, and origin confusion, but they largely ignored other consequences, such as open redirects, OAuth link hijacking, server-side request forgery (SSRF), and authentication bypasses. A study of HNP and its consequences is challenging due to the multi-layer architecture of server-side web applications.**

In this paper, we design a novel measurement framework, called HALO, to understand why HNP exists and detect HNP vulnerabilities in real-world, open-source web applications. HALO breaks down the multi-layer structure into individual components and analyzes them using a combination of dynamic testing and static analysis to detect vulnerabilities. Our evaluation of 9,860 open-source applications uncovers 82 zero-day HNP vulnerabilities. We have responsibly disclosed all of them to their developers: So far, we have received 52 Common Vulnerabilities and Exposures (CVEs) and 20 confirmed fixes.

1. Introduction

A host name is a fundamental concept in web applications that not only defines a server's identity but also specifies the destination of a client request. For example, on the client side, the HTTP protocol allows host name to appear in many request headers, e.g., `Host` and `:authority` pseudo-header [1], [2], [3]. Thus, one challenging task for web servers is to differentiate the host name provided by the client in the HTTP request, which could be potentially malicious, and that exists on the server, which should be trusted. A misuse of the untrusted, client-provided host name will lead to so-called *Host Name Poisoning (HNP)* vulnerability defined in this paper.

Prior works have touched the surface of detecting and exploiting HNP vulnerabilities, given the severity of this issue. For example, Innocenti et al. [4] measured password-reset link hijacking in applications that trust unvalidated `Host` headers. For another example, Chen et al. [5] analyzed how web servers parse multiple `Host` headers and revealed inconsistencies that lead to cache poisoning and origin confusion. However, prior works only studied a limited set of potential consequences, such as password resetting, cache poisoning, and origin confusion. They do not provide a principled cross layer view of how host authority is reconstructed and trusted across servers, frameworks, and applications. As a result, it remains unclear whether HNP can lead to a broader consequence space, including open redirects, OAuth link hijacking, server side request forgery (SSRF), and authentication bypasses, and how such consequences arise from specific cross layer trust inconsistencies in real world applications.

The study of HNP consequences and reasons is a challenging problem, because of the multi-layer structure of web stacks involving multiple parties, e.g., from top to bottom, the web application (i.e., those programmed by web developers), the web framework (e.g., Django and Flask), and the web server[1] (e.g., Nginx and Apache). Different layers may choose and provide host names for the upper layer to use. For example, web servers may rewrite or inject host-related headers to support routing or multitenancy, which could be picked up by web frameworks. Then, web frameworks may reconstruct what they regard as the canonical host name for web applications to use. Such a multi-layer structure complicates not only the problem of HNP vulnerability, as each layer may think it is others' responsibility to provide trusted values, but also the detection of such vulnerabilities, as there are different combinations of these three layers with different host name trust relations.

In this paper, we design a novel measurement framework, called HALO, to detect and measure Host Name Poisoning (HNP) vulnerabilities in open-source web applications. Our key insight is to break down the multi-layer structure of web applications, test each individual layer separately, and provide an abstraction for upper layer measurement. Specifically, HALO first analyzes and tests different combinations of server-framework pairs with different host names in HTTP requests to understand which combination is vulnerable, called Vulnerable Server-Framework Pair (VSFP) , and how potential untrusted host names flow to web frameworks from web servers for this combination. Then, HALO performs static dataflow analysis of web frameworks to track and trace host name propagation to host sensitive APIs and infer their default guard states, producing per framework API-guard pairs, which we call Host-Sensitive API-Guard Pair (HAGP)

1. To simplify terminologies, we use a broader definition of web servers in the paper, which may include a reverse proxy.

in this paper.

Lastly, HALO performs another static analysis based on the aforementioned Vulnerable Server-Framework Pair (VSFP) and Host-Sensitive API-Guard Pair (HAGP) against open-source web applications to detect HNP vulnerabilities. Specifically, once a VSFP is matched, HALO starts from the API in a HAGP, matches a guard if it exists, and determines where the host name flows to in the web application. This design allows HALO to move beyond previously studied case-specific consequences and systematically reason about a broader consequence space. If the final sink is an API that may lead to an exploitable consequence, as determined by a Large Language Model (LLM), HALO considers this dataflow as potentially vulnerable.

We evaluate HALO against 9,860 open-source repositories, uncovering 361 potential HNP vulnerabilities. Our manual review of 100 top-ranked (by number of stars) cases confirms 82 exploitable zero-day HNP vulnerabilities covering many different consequences, such as Server-side Request Forgery (SSRF), Open Redirect, and Authentication Bypass. We responsibly disclosed all vulnerabilities to their developers: So far, we have received 52 CVE assignments and have 20 vulnerabilities being fixed.

In summary, this paper makes following contributions:

- We design a novel measurement framework, called HALO, that combines dynamic testing and static analysis to provide a principled cross-layer analysis of host authority reconstruction across web servers, frameworks, and applications, enabling the detection of zero-day HNP vulnerabilities together with their root causes, e.g., a misconfiguration of the web server and the framework.
- We perform the first large-scale measurement of Host Name Poisoning (HNP) in open-source applications, revealing that HNP leads to a broader consequence space than previously studied, including SSRF, open redirect, and authentication bypass, while also exposing clear cross-layer classification insights into when and why these vulnerabilities arise. We uncover zero-day vulnerabilities including 52 assigned CVEs, and we have responsibly disclosed all of our findings.
- We propose practical, secure-by-default guidance for configuring web servers, frameworks, and applications.

2. Overview

This section presents a real-world motivating example that illustrates HNP in a widely deployed system, and defines the threat model and scope that guide the rest of this paper.

2.1. Background

Modern web applications pass through several layers before reaching the application itself. A typical setup has a web server at the front, a framework behind it, and the application on top. As requests move through these layers, servers often add or rewrite forwarding headers such as `X-Forwarded-Host` or the standardized `Forwarded: host=` [6] to record the host name that the client originally saw. These headers help preserve routing and deployment context, but different servers and frameworks interpret them in different ways. As a result, the same request can produce different views of which host name should be trusted.

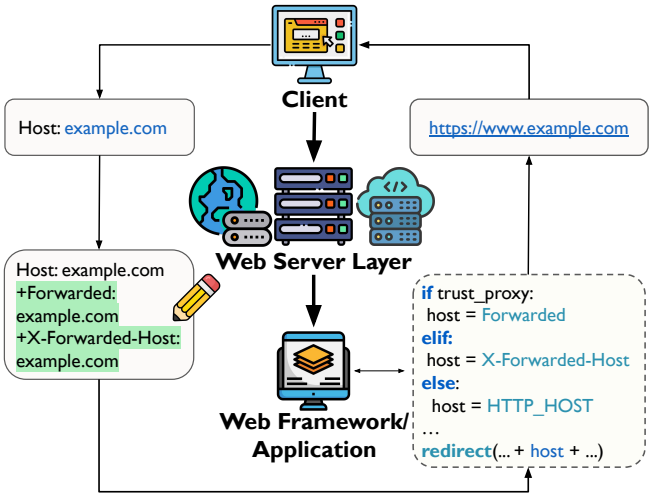

Figure 1: Host Name in Modern Web Stack

To reason about these differences, we use two common deployment models. In the direct-to-application model, the framework receives requests straight from the client and simply treats the incoming `Host` header as the trusted value. In the behind-proxy model, a web server sits between the client and the framework, forwarding the request and possibly changing or adding forwarding headers that shape how the framework interprets the host name. These two models capture how authority information is produced and passed along the stack.

HALO focuses on both reconstruction process across layers. It analyzes how host names are resolved, checked, and reused by servers, frameworks, and applications, and how these decisions shape the trust boundary for the rest of the system.

2.2. A Motivating Example

Listing 1 presents a real-world zero-day host-name poisoning vulnerability discovered by HALO in *easy-mock*, a widely used open-source platform with over 9k GitHub stars, and it is built on Koa2, a popular Node.js web framework with more than 35.7k stars. *easy-mock* has been deployed by many well-known organizations, including Meituan, 360 Enterprise Security, CGB Bank, ZTE Corporation, and East China Normal University. The vulnerability lies in the project's file-upload handler, which lets an attacker poison the host name and causes the application to generate persistent avatar and file URLs that point to attacker-controlled domains. We have responsibly disclosed the issue to the *easy-mock* maintainers.

Vulnerability Details. Koa2 combines the request protocol and host to construct `ctx.request.origin`; when no host allowlist or proxy constraints are configured, the host value is derived from the client-supplied `Host` header without any validation. *easy-mock* passes this origin into `new URL(filePath, origin).href` to generate an absolute link and wraps the result in `ctx.util.resuccess({path: ...})`, which is finally assigned to `ctx.body`. This establishes a direct dataflow from the inbound host value to a user-visible sink. When a request includes a poisoned host name (e.g., `attacker.com`), the service generates an upload link such as `http://attacker.com/upload/x.png`. As this link appears in project dashboards, shared pages, and user avatars, any user who follows it is sent to an attacker-controlled URL. This lets the adversary deliver their own content or collect application tokens embedded in the link, depending on how the application uses the generated URL.

```
/* Koa2: origin reflects protocol and host; host usually mirrors inbound Host header */
get origin() {
  return `${this.protocol}://${this.host}`;
}

/* Easy Mock: file-upload handler that builds absolute URL from ctx.request.origin */
static async upload (ctx) {
  const origin = ctx.request.origin;

/* derived from inbound Host header; validation omitted */

  const filePath = path.join('upload', date, fileName);
  ctx.body = ctx.util.resuccess({
    path: new URL(filePath, origin).href,
    expire: /* ... */
  });
}
```

Listing 1: A motivating example of a zero-day Host Name Poisoning vulnerability found in easy-mock. The code is simplified for explanation

Overall Solution. HALO organizes its analysis into three stages that cover servers, frameworks, and applications. The first two stages produce VSFP indicators and per-framework HAGPs. We illustrate how these results are used by examining the HNP issue in *easy-mock*. From the framework analysis, HALO knows that Koa 2 exposes `ctx.request.origin` as a host-bearing API. When scanning projects at scale, HALO encounters *easy-mock* and sees that the application uses this API, so it performs static analysis to inspect how the value moves through the code. The analysis reveals a complete dataflow from `ctx.request.origin` to `ctx.body`, meaning the generated URL is written into the HTTP response with no validation and no guard. Next, HALO checks the project's deployment settings to see whether any whitelist or trusted-host configuration is enabled. In *easy-mock*, no such guard is present, so the application falls into the VSFP where client-supplied host values are accepted. Combined with the host-controlled flow into `ctx.body`, HALO reports the case as an HNP vulnerability. After manual deployment in a controlled local environment, we sent a test request with `Host: attacker.com`. The generated URL in the response pointed to `attacker.com`. When another user of the same service accessed the file, the request was redirected to the attacker-controlled domain within the test setup, confirming a concrete instance of Host Name Poisoning without causing any real-world impact.

Prior black-box approaches are limited by what they can observe: they can only test the specific behaviors and APIs that they happen to trigger. HALO brings together server behavior, framework semantics, and application-level flows, giving it a broader and clearer view of where host values are used and the range of HNP consequences that follow.

2.3. Threat Model

We consider a remote attacker who can send arbitrary HTTP(S) requests to a target service and control client-supplied metadata such as `Host`, `:authority`, `X-Forwarded-Host`, and `Forwarded: host=` [6]. The attacker cannot modify server, framework, or proxy configurations, nor obtain privileged access; their capability is limited to crafting and delivering normal network requests.

We assume deployments where an application runs on top of a web framework, either directly exposed to clients or placed behind a separate web server. An HNP vulnerability arises when these layers, taken together, accept an attacker-supplied host as canonical. In this setting, the attacker can steer how the service constructs absolute URLs, redirect targets, authentication links, or server-to-server requests, and can influence framework helpers that reuse request-derived host values. These effects produce a wide range of consequences: open redirects, authentication bypasses and OAuth callback hijacking, cross-origin token leakage [7], [8], [9], server-side request forgery, cookie-domain manipulation, corrupted resource URLs, and persistent links that continue to expose attacker-controlled domains in shared views. When such links appear in shared pages, a victim who follows them is taken to an attacker-controlled domain and may unknowingly interact with content derived from the poisoned host.

Our analysis focuses on how host values are reconstructed and trusted across servers, frameworks, and applications, covering both direct and behind-proxy deployments. All experiments were performed in isolated environments, and confirmed vulnerabilities were reported to maintainers under responsible disclosure.

3. Design

To study how Host Name Poisoning arises, how it propagates across layers, and what consequences it can cause in real deployments, we design HALO, a three-stage measurement system that tests server–framework interactions, extracts framework-level host semantics, and measures how

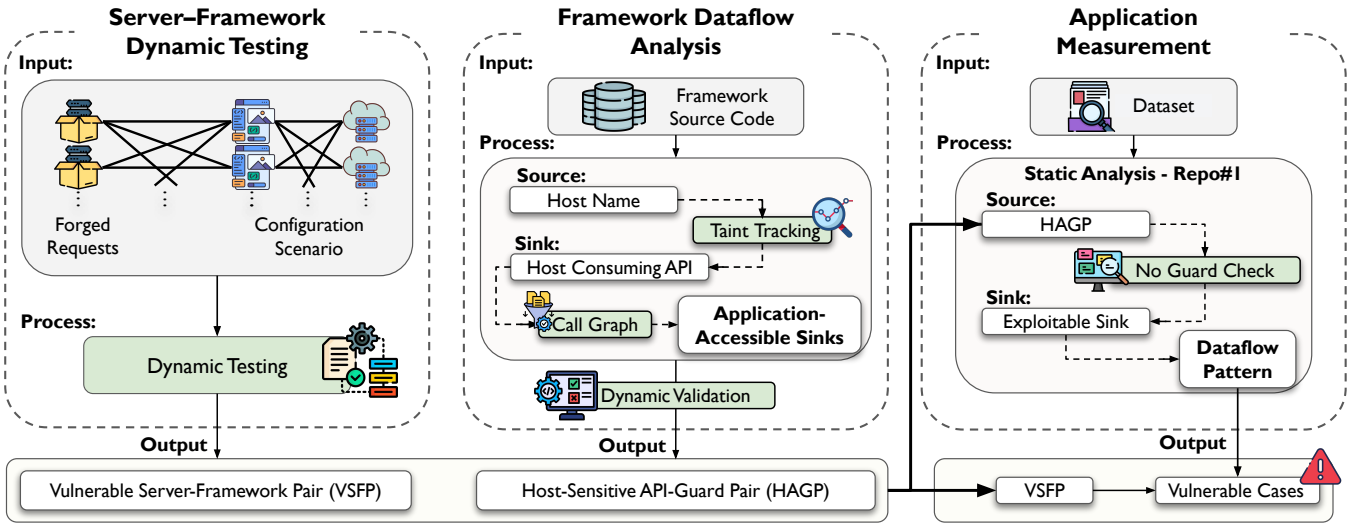

Figure 2: Overall Architecture of HALO

applications reuse these values. Figure 2 shows its overall architecture.

3.1. Server–Framework Dynamic Testing

The first stage of HALO performs dynamic testing of web server–framework combinations under different configurations to observe how servers rewrite or forward host-related request metadata and how the framework accepts, ignores, or overrides these fields (Figure 2, left).

HALO treats servers and frameworks as separate configuration axes. On the framework side, HALO enumerates all host-related options the platform exposes, such as whether the framework checks the `Host` header, accepts forwarded host headers, validates them, or trusts upstream proxies. We combine these settings into twelve A-profiles that cover the default behavior, each individual option, and the valid combinations. Each profile is first tested without any fronting server so we can see how the framework handles host metadata before adding server-side forwarding.

On the server side, HALO tests a representative set of host-handling profiles that reflect how web servers commonly process host information. These profiles include passing requests through unchanged, overwriting the host with a fixed value, accepting forwarded host headers only from trusted ranges, and combining an override with a trust policy. Each profile is instantiated on representative server deployments and paired with every framework configuration. We label each scenario as SC–XXX–D/A/N, where the suffix denotes a direct run (D) or a deployment that applies a specific server profile (A/N). This scheme captures both the framework's own behavior and the effects introduced by upstream rewriting and trust rules. Table 1 provides representative entries, and the appendix contains the full matrix.

Each scenario is exercised with HALO's canonical suite of crafted HTTP requests (TC) [10]. The goal of this suite is to trigger the different ways in which servers and frameworks may interpret or prioritize host-related metadata. To do this, we parameterize the four authority carriers: `Host`, HTTP/2 `:authority`, `X-Forwarded-Host`, and `Forwarded: host=`, and instantiate representative combinations drawn from several attack families. These families cover direct tampering of `Host` and forwarded-host fields, manipulation of individual forwarding headers, precedence stress tests using raw or encoded variants, and HTTP/2 or absolute-form request-line cases. Within each family, we include both benign and adversarial inputs. The benign cases serve as baselines and help isolate normal precedence behavior, since some scenarios only require observing which host field a server or framework prefers. The adversarial cases then vary these fields to test how the system reacts under manipulation. Table 2 provides representative examples, and the full suite appears in the appendix.

TABLE 1: Representative server-framework configuration scenarios (full matrix in the appendix).

Scenario	Server	Deployment	Policy tuple
SC-001-D	–	Direct	A0 (NoAppConfig)
SC-006-D	–	Direct	A4 (ProxyFix w/ validation)
SC-010-N	Nginx	Behind proxy	A4, B1 (XFH override)
SC-018-N	Nginx	Behind proxy	A2+A1, B2 (Host whitelist)
SC-025-A	Apache	Behind proxy	A1+A3, B0 (No override)
SC-033-A	Apache	Behind proxy	A2+A1+A4, B2

As Figure 3 shows, for each ⟨scenario, template⟩ pair, HALO runs a dynamic test with the crafted HTTP request and records the host value that the framework ultimately observes, together with the header or field it was derived from. HALO also records any safeguards that activate during the run, such as host allowlists, proxy trust checks, or header validation routines that cause the framework to override, ignore, or replace the incoming host value. The resulting traces form a matrix showing how each combination of server behavior, framework settings, and crafted inputs influences the host value delivered to the framework. HALO cross-checks these

TABLE 2: Representative cases of 55 crafted HTTP request test cases (full matrix in the appendix).

TC ID	Name	Attack Type
TC-001	Direct Host header tampering	Host manipulation
TC-003	Missing Host + malicious XFH	XFH injection
TC-005	Host header port injection	Port injection
TC-007	Wildcard/child-domain bypass	Subdomain injection
TC-011	benign Host, malicious XFH	Priority confusion
TC-025	Full X-Forwarded series spoofing	Full headers
TC-037	XFH priority vs Host/Forwarded	Priority test
TC-041	Forwarded quoted port	Forwarded port
TC-047	XFS + Forwarded host fallback	Server header

observations against raw network captures and framework logs to ensure that the results are deterministic and reflect the actual processing of the stack. This stage allows HALO to classify each server–framework configuration by whether client-controlled host fields are accepted, overwritten, or ignored, and which request patterns and settings lead to these outcomes.

From these observations, HALO groups the server–framework outcomes that accept client-supplied host values into a set of VSFPs. Each VSFP represents a distinct pattern in how a poisoned host becomes accepted by the stack. For each class, HALO identifies minimal request templates that trigger the behavior, giving a compact view of the conditions under which host drift occurs. HALO also extracts rules that describe the framework's priority order among host-related fields and how this order changes under different server behaviors. Taken together, the classes, rules, and templates serve as input to the next stages, which examine how host values propagate within frameworks and surface in application logic.

3.2. Framework Dataflow Analysis

The second stage of HALO analyzes how web frameworks read, pass, and apply host values once they reach the framework layer (Figure 2). This stage uses the trust and priority rules derived from Stage I together with each framewor's source code and documentation, analyzed in its native environment. The goal is to identify how host values move through framework abstractions, which APIs depend on them, and where checks are applied.

HALO then performs static dataflow analysis of web frameworks to track how host names move through framework code and reach host-bearing APIs. It uses lightweight taint tracking over abstract syntax tree (AST) and control-flow graph (CFG) representations [11], [12], and builds an interprocedural call graph that links user-facing APIs, middleware, and routing modules. This allows HALO to trace how host values pass through parameters, variable definitions, and return values, and how they are reconstructed or reused by components such as URL builders, redirect handlers, and authentication helpers. During this process, HALO records every guard it encounters, such as allowlists, normalization routines, or trusted-proxy checks, but does not stop the analysis at these points. Many guards are optional

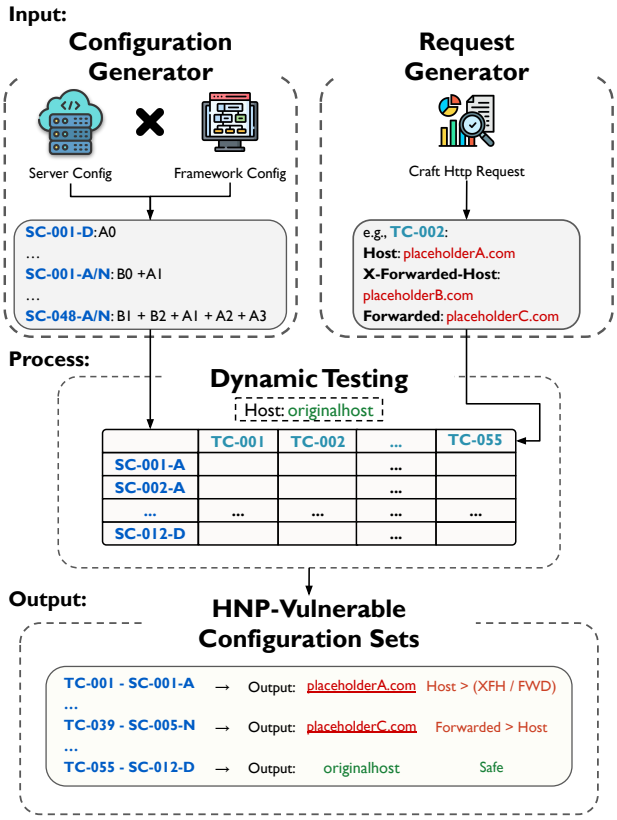

Figure 3: Server–Framework Composition Engine of HALO

or depend on deployment configuration, and a dataflow that passes through a guard may still reach an application sink if the guard is disabled or misconfigured. By modeling flows in this way, HALO captures both guarded and unguarded uses of host metadata and produces per-framework API–guard indicators (HAGPs) that characterize where validation is enforced by design and where it can be bypassed or deferred to the application.

The analysis identifies three key elements: sources, guards, and sinks. Sources extract host names from HTTP headers or configuration files. Guards enforce constraints such as host allowlists or trusted-proxy scopes. Sinks reuse host information in visible contexts such as URL generation, redirects, cookies, or origin checks. Instead of relying on a fixed set of predefined sinks (e.g., redirects, URL builders, or HTTP clients), HALO treats any API that consumes host metadata as a potential sink. HALO then uses the framework's call graph to collapse wrapper functions and duplicate call paths so that each API is counted once. This broader view allows it to reveal propagation paths that are not covered by traditional sink lists and would otherwise remain undocumented. Guards are detected from validation logic and configuration lookups, and their dominance is computed on the control-flow graph: a guard dominates a source–sink pair if all feasible propagation paths pass through it. When documentation and implementation differ, HALO prioritizes the actual code behavior to reflect real semantics. Reflection and middleware sequencing are

modeled conservatively to preserve execution order. Then HALO empirically validates these sinks using the Stage I harness. Each candidate is exercised in a minimal framework setup with controlled host values, allowing HALO to determine whether its output reflects the tainted host. This step confirms the sink's actual behavior.

Finally, HALO summarizes each framework as a reusable set of API–guard indicators (HAGPs), capturing its default trust behavior in a uniform form for later analysis. From these models, HALO derives ten evaluation dimensions that characterize the robustness of each framework's host-handling design. As shown in Table 3, eight categorical dimensions (FW-D1–FW-D8) describe whether the framework enforces host validation, applies trusted-host checks, accepts or verifies forwarded headers, normalizes host format, and documents these behaviors accurately. Two quantitative metrics (FW-AE1 and FW-AE2) measure the number of APIs that reconstruct or reuse host authority and the subset that lack validation or guard logic.

FW-D1 and FW-D2 capture the basic boundary checks applied to the `Host` header, indicating whether the framework validates it and whether an allowlist is enforced before the value is reused. FW-D3 and FW-D4 reflect how the framework processes different forms of host input, including whether it normalizes host format and whether forwarded headers are accepted by default. FW-D5 and FW-D6 show the degree of control and checking provided for these headers, which determine how upstream proxies influence reconstructed host authority. FW-D7 summarizes whether these protections sit on all host-flow paths, while FW-D8 records whether the official documentation alerts developers to these behaviors and their deployment impact. To compute these dimensions, HALO traverses each framework's interprocedural call graph to locate host-bearing APIs, analyzes the dominance of guards along these paths, and determines whether validation is mandatory, optional, or missing. Proxy- and normalization-related properties are inferred from framework code and confirmed through empirical tests using the Stage I harness. For FW-D8, HALO supplements static analysis with an LLM-assisted review of official documentation, extracting relevant descriptions of host and proxy settings and cross-checking them against implementation behavior.

Overall, these dimensions give a clear picture of how each framework interprets, checks, and exposes host values. They form the basis for Stage III, where HALO applies the resulting HAGPs to large-scale repository analysis and measures Host Name Poisoning exposure in real applications, showing how often these flows arise and the types of consequences they lead to.

TABLE 3: Framework-level evaluation dimensions for assessing host-handling robustness. Each dimension is rated as **Yes/Partial/No**; **AE1/AE2** are numerical counts.

ID	Dimension	Meaning
FW-D1	Host validation support	Built-in logic for checking host names.
FW-D2	Trusted-host enforcement	Mandatory allowlist for accepted hosts.
FW-D3	Host normalization	Canonicalization of host format before reuse.
FW-D4	Default header acceptance	How the framework treats forwarded headers by default.
FW-D5	Forwarded-header control	User-facing options to enable forwarded-header handling.
FW-D6	Forwarded-header checks	Built-in verification or filtering of forwarded headers.
FW-D7	Complete guard path	Whether all host flows pass through a protective guard.
FW-D8	Documentation warnings	Whether documentation alerts developers to host-related risks.
FW-AE1	Host-name API exposure	Count of APIs that reconstruct or reuse host authority.
FW-AE2	Unvalidated exposure	Count of such APIs without validation or guard logic.

3.3. Application Measurement

The final stage of HALO performs ecosystem-scale application measurement by applying the configuration classes (VSFPs) and framework semantics (HAGPs) derived from earlier stages to real-world repositories (Figure 2, right). Its goal is to quantify how often host-name trust inconsistencies arise in practice and, more importantly, to examine what consequences they cause in real-world applications. Specifically, HALO analyzes how untrusted host names flow into security-sensitive logic and manifest as concrete vulnerabilities. These results provide empirical evidence of the practical impact of Host Name Poisoning.

HALO takes three inputs. First, it uses the VSFP derived from Stage I, which summarizes how web servers and frameworks reconstruct or forward host names under different configurations. Second, it incorporates the per-framework HAGPs from Stage II, which capture how frameworks interpret, validate, and propagate host names in their default settings. Third, it analyzes a language-partitioned corpus of open-source repositories cloned from GitHub and organized by primary language and declared framework dependencies [13]. This setup allows each repository to be matched with the corresponding VSFP and HAGP, enabling consistent measurement across languages and frameworks. While framework-level configurations can be fully captured from the source code, such as whether a project enables proxy-related middleware or sets host validation options, the server layer is typically not visible in the repository. HALO bridges this gap by pairing these observable framework-level indicators (e.g., `ProxyFix`, `trust_proxy`) with the measured server behaviors captured in Stage I. In other words, although the repository may not explicitly specify whether it runs behind a proxy, HALO infers likely deployment models based on the framework's configuration, middleware usage, and supplementary evidence from project documentation or deployment guides. It then composes these inferred models with the corresponding VSFP. This design enables HALO to reason about realistic cross-layer deployments even when server configurations are not explicitly visible in the repository.

HALO performs scalable static analysis to identify end-to-end host-name flows in application code. In this stage, the API–guard indicators (HAGPs) derived from Stage II serve as the starting points for taint tracking. Each indicator defines a host-bearing API that introduces host metadata into the application, and HALO traces how these values propagate through function calls and data structures toward any reachable sink. For every repository, HALO builds lightweight program representations, including abstract syntax trees (ASTs), call graphs, and module dependencies, and searches for complete dataflow paths that match this propagation pattern. A flow is considered relevant when a host-bearing source reaches a sink without an intervening guard and the repository's configuration or documentation indicates that the corresponding VSFP applies. To determine whether the sink may lead to an exploitable consequence, HALO employs a large language model to analyze the semantics of the target API. If the final sink corresponds to an API that can produce observable effects, like redirects, URL generation, authentication, or external network requests, the LLM marks it as security-sensitive. These annotations are then manually reviewed for correctness.

To handle large-scale analysis, HALO applies several practical optimizations. Before dataflow exploration, HALO performs a prefiltering step that checks whether a repository invokes any APIs defined in the API–guard indicators (HAGPs). Repositories that contain no such references, and therefore do not process host names, are skipped to reduce unnecessary analysis. For the remaining projects, HALO runs static analysis in parallel and limits call depth and loop exploration to keep the analysis scalable. Each finding is stored with provenance information such as file path, relevant APIs, surrounding guard logic, and the assigned VSFP. This metadata makes the results reproducible and helps with efficient manual triage.

Stage III produces a searchable database of pattern-matched and validated cases, linking real repositories to concrete HNP exploit chains. For each framework, HALO reports metrics such as how many projects contain unguarded host flows, how VSFPs are distributed, and how often validation succeeds. By combining VSFPs, HAGPs, and corpus-scale validation, this stage summarizes the practical impact of Host Name Poisoning and provides an ecosystem-level view of how often these issues occur and what consequences they can produce.

4. Implementation

We implemented HALO as a measurement pipeline that follows the design in Section §3. In this section, we describe how we built the pipeline in practice.

4.1. Dataset Selection

HALO's dataset consists of web servers paired with major frameworks, together with a large GitHub corpus for ecosystem-scale analysis. The selection follows simple, publicly verifiable criteria to ensure coverage of common deployment setups and language diversity.

Web Servers. We use Nginx and Apache to represent origin-side request termination. As of November 2025, W3Techs reports Nginx and Apache serving 33.3% and 25.3% of websites with known servers, respectively; Netcraft independently observes Nginx at 25% [14], [15], [16]. These two components capture the dominant origin stacks on the web.

Programming Languages. We target nine major server-side languages: PHP, Ruby, Java, JavaScript (Node.js), Scala, ASP.NET (C#), Python, Go, and Rust. According to W3Techs, PHP remains by far the most widely used server-side language, with Ruby, Java, JavaScript, ASP.NET, Scala, and Python each contributing additional shares among websites with identifiable back-end stacks [17]. While Go and Rust appear only in a small fraction of surveyed sites, they are increasingly prominent in cloud-native and security-critical services. Together, these nine languages capture both traditional ecosystems and emerging high-assurance stacks.

Frameworks. Within each language, frameworks are chosen through a multi-metric procedure combining six rankings: package downloads, dependent count, GitHub stars, forks, repository dependents, and package dependents. We take the union of the top five in each ranking and apply maturity filters (downloads $\geq$10,000, stars $\geq$10,000, forks $\geq$1,000); frameworks without telemetry are retained if they meet GitHub criteria. This yields 32 frameworks across nine languages (see Table 6 for the final list). Package data comes from native registries (PyPI Stats, npm, Packagist, RubyGems, crates.io, NuGet) and deps.dev. Combining heterogeneous metrics balances popularity, reuse, and transitive adoption [18], [19].

Applications. For applications measurement, we sample public GitHub repositories created between 2015 and 2025, restricted to original (non-fork), active (archiving disabled) projects with at least one commit in the past 12 months, a primary language among our nine targets, and $\geq$100 stars. These filters emphasize maintained, non-trivial projects and align with prior large-scale mining practices using popularity and activity as validity signals [19].

4.2. HALO Implementation

HALO is implemented as a unified framework that combines dynamic composition testing, static taint analysis, and large-scale repository scanning. The prototype is written in Python and QL, using CodeQL v2.23.3 with Python 3.12 [20]. Because CodeQL does not support PHP, we use Semgrep in taint mode with custom rules that encode our API–guard indicator (HAGP) model [21]. Semgrep outputs are normalized to the same schema as CodeQL results so that VSFPs and HAGPs remain consistent across languages. We further extend CodeQL's dataflow libraries with rules tailored to host-handling behavior.

Host-related operations are modeled as taint steps that track host accessors, server-side guards, and authority-

TABLE 4: [RQ1] Representative zero-day HNP cases across languages.

Project	Language	Stars	Source → Sink	Consequence	CVE ID
webssh	Python	5K	`get_redirect_url()` → `self.redirect()`	HTTPS redirect hijack	CVE-2025-65455
flask-base	Python	3.1K	`url_for(..., _external=True)` → `mail.send()`	Authentication Bypass	CVE-2025-63771
carbon	JS	35.8K	`req.headers.host` → `https://$host/api`	SSRF	CVE-2025-65461
easy-mock	JS	9.03K	`ctx.request.origin` → `ctx.body.path`	Resource URLs Corruption	–
apihub	JS	8.5K	`req.get("host")` → `link()`	Open Redirect	CVE-2025-69892
BrowserBox	JS	3.7K	`req.get('host')` → `makeLoginLink()`	Link hijacking	CVE-2025-63773
WDScanner	PHP	2.1K	`$_SERVER['HTTP_HOST']` → `url()`	SSRF	CVE-2025-65452
wordpress-12factor	PHP	274	`$_SERVER['REQUEST_URI']` → `header("Location: $url")`	Stored HNP	–
DataSphereStudio	Java	3.2K	`DomainUtils.getCookieDomain()` → `Set-Cookie: Domain`	Cookie Manipulation	CVE-2025-65447
url-shortener-app	C#	150	`Request.Host` → `ShortenUrls.Add()`	Persisted Link Hijack	CVE-2025-65460
traefik-forward-auth	Go	2.3K	`r.Header.Get` → `AuthCallbackHandler()`	OAuth Redirect	CVE-2025-63774
actix-web-lab	Rust	122	`req.connection_info().host()` → `https://{hostname}{path}`	Open Redirect	–
Hunt3r	Ruby	170	`request.host_with_port` → `meshs.url()`	Authentication Bypass	CVE-2025-63764

bearing sinks. Stage II performs static analysis on each framework using CodeQL and Semgrep for PHP, with inter-procedural rules and flow-state annotations to identify paths where host values reach security-sensitive APIs without validation. Stage II produces structured flow patterns describing sources, guards, sinks, and context.

Stage III applies static analysis to applications. Framework-level sinks found in Stage II become new taint sources, and we track how applications reuse them across thousands of repositories. This stage automates repository retrieval, database construction, taint-query execution, and result aggregation in a distributed workflow. Stage I provides configuration context for forwarded-header behavior, and Stage II patterns are serialized as JSON guides for application-level analysis. To preserve realistic call relationships, HALO extends CodeQL's resolution logic to account for framework abstractions such as middleware chains, dependency injection, and routing contexts. We use a large language model (GPT-5) in two narrowly scoped auxiliary tasks. First, given official framework documentation, the model acts as a reading assistant to locate and summarize host and proxy handling semantics (e.g., forwarded header processing). These outputs only support rule implementation, are cross-checked against code, and are not treated as vulnerability evidence. Second, in Stage III, given static flow context, the model labels whether a candidate sink corresponds to a security-relevant consequence under our threat model, with a short rationale. This signal is used only for triage. HALO does not rely on LLM output alone; all reported vulnerabilities are validated by the pipeline and manual review. All components interoperate through versioned JSON schemas that record flow patterns, VSFPs, HAGPs, and provenance information, ensuring consistent and reproducible integration across dynamic, static, and documentation-based analyses. The full implementation and workflow are available in our anonymous artifact repository (https://anonymous.4open.science/r/halo-demo-7D50).

5. Evaluation

We evaluate HALO to understand the prevalence, causes, and real-world impact of HNP across modern web ecosystems, and to examine where false positives arise in our analysis. This section reports our measurement results and addresses five research questions.

RQ1 [Zero-day]. How many previously unknown HNP vulnerabilities can HALO uncover, and what attack scenarios do they represent?

RQ2 [Root Cause]. What cross-layer inconsistencies between servers and frameworks give rise to HNP?

RQ3 [Framework Measurement]. How do mainstream frameworks process host metadata, and how many expose host-bearing APIs?

RQ4 [Ecosystem Distribution]. How is HNP distributed across languages, frameworks, and real-world project?

RQ5 [False Positives]. When HALO flags a host-related flow, how often does it not constitute a real vulnerability, and what causes these false positives?

5.1. RQ1: Zero-day Vulnerabilities

HALO analyzes nine languages, 32 frameworks, and 9,860 open-source repositories, uncovering 361 potential HNP vulnerabilities. From a manual review of 100 top-ranked cases, we confirmed 82 exploitable zero-days. To explain how these vulnerabilities arise in practice, we organize the confirmed cases into three categories based on how applications consume the host value: user-driven flows, which directly affect links or redirects shown to users; server-driven flows, where the poisoned host influences backend decisions; and persistent or cross-request flows, where the forged value persists beyond the triggering request.

5.1.1. User-driven Flows. These cases change links or redirects that users see and follow. A forged host directly poison the URLs that applications present to users, so the impact emerges immediately when a victim clicks the generated link.

Authentication Bypass. Framework helpers that generate absolute URLs for password resets or invitation flows often rely on the request host. In several Flask and FastAPI projects, a forged `Host` header caused these helpers to construct reset or login links that pointed to attacker domains. The application believes it is producing a valid callback target, so no further checks are triggered. When users follow these links, their reset tokens or login callbacks are sent directly to the attacker, allowing account takeover even though the underlying reset logic remains unchanged.

OAuth and Callback Manipulation. OAuth handlers frequently reconstruct the origin to check redirect parameters or to issue authorization callbacks. In `traefik-forward-auth`, forwarded headers are used as provided, allowing an attacker to replace the origin used when completing the OAuth flow. This causes the framework to send the authorization callback to an attacker-controlled endpoint, exposing authorization codes or tokens without requiring any additional interaction from the victim.

Link Hijacking and Open Redirects. In several JavaScript and Rust middlewares that generate absolute redirect URLs the target is constructed from connection metadata such as `Forwarded`, `X-Forwarded-Host`, or `Host`. If proxy-trust settings are misconfigured, the resulting `Location` header can be influenced by an attacker-controlled origin by simply appending the request path. When applications surface such URLs to users like login links, confirmation pages, or notification emails, this behavior creates a practical open-redirect and phishing vector. Developers should prefer relative redirects when possible, and otherwise enforce an allowlist for the canonical origin while configuring trusted-proxy settings correctly.

5.1.2. Server-driven Flows. Here the poisoned host influences internal server behavior rather than user-facing links. The request appears normal to the user, but the server later relies on the forged value when generating resource URLs or issuing backend requests, causing internal actions to be steered by attacker-controlled input.

Resource-URL Corruption. Some services reconstruct storage paths or API endpoints from a derived origin. In `easy-mock`, the system combined a base origin with a file path to form an upload URL. A forged host replaced the origin entirely, so the application served attacker-controlled URLs as if they were its own resources, causing asset breakage and enabling malicious file delivery in downstream workflows.

Server-side Request Forgery (SSRF). In `WDScanner`, PHP superglobals containing the host value were fed directly into backend scanning logic. A forged host changed where the scanner sent its internal requests. Because these scanners often run with elevated privileges or have access to internal networks, the forged value exposed internal admin panels and diagnostic endpoints that were never meant to handle user-controlled input.

5.1.3. Persistent and Cross-request Flows. These cases persist across requests, so a single forged host can influence future interactions or affect users who never saw the original spoof.

Persistent Host Name Poisoning. Some applications store generated URLs in databases, logs, or content fields. In `wordpress-12factor`, poisoned URLs written into post content or metadata reappeared when other users viewed or edited the entry. This turned one spoofed request into a durable redirect or content-modification issue that survived restarts and caching layers.

Cookie-domain Manipulation. Some enterprise systems derive the `Domain=` attribute of cookies from the request's host value to support multi-tenant deployments. In `DataSphereStudio`, a forged `Host` header caused the server to emit a session cookie whose domain matched the attacker-controlled hostname. Because browsers store this cookie and send it on any later request to that domain, the attacker only needs the victim to visit the attacker domain at any point—no crafted link is required. The browser then attaches the mis-issued session cookie to the attacker's site, giving the attacker direct access to the victim's session [22].

Persistent Link Hijack. Link-management systems that store absolute URLs are vulnerable at creation time. In `url-shortener-app`, submitting a request with a forged host caused the system to record a poisoned origin inside the shortlink entry. All subsequent users who followed that short link were redirected to attacker pages until the entry was manually corrected, creating a long-lived phishing surface.

These cases illustrate how reusing the host without validation leads to different consequences depending on where the value is consumed. They also show that unsafe host handling arises across many parts of the stack, including utilities, dashboards, authentication modules, AI panels, and infrastructure components. All confirmed vulnerabilities were reported to maintainers under coordinated responsible disclosure.

TABLE 5: [RQ2] Vulnerable scenarios and triggering test cases. Each row lists a scenario (SC) that exhibited HNP, the number of distinct test cases (TC) that triggered it, and the corresponding TC identifiers.

Scenario (SC)	#Polluted Cases (TC Count)	Triggering Test Cases (TC)
SC-001-D	27	TC-001–002, 005–010, 012–013, 016–018, 026–036, 039–041, 042–045
SC-004-D	33	TC-001–005, 006–011, 012, 014–018, 026–027, 031–035, 037, 039–043, 045–046, 054
SC-005-D	30	TC-001–012, 013–018, 026–027, 031–035, 037, 039–043, 045–046, 054
SC-001-A/N	32	TC-001–002, 005–010, 012–013, 016–018, 026–036, 039–041, 045, 048, 051–053, 055
SC-005-A/N	29	TC-001–003, 005–012, 014–018, 026–027, 031–035, 037, 039–041, 045, 048–049, 051
SC-006-A/N	29	TC-001–003, 005–012, 014–018, 026–027, 031–035, 037, 039–041, 045, 048–049, 051
SC-009-A/N	7	TC-003, 011, 015, 037, 046, 049, 054
SC-012-A/N	30	TC-001–002, 005–010, 012–013, 016–018, 026–036, 039–041, 045, 048, 051–052
SC-015-N	7	TC-003, 011, 015, 037, 046, 049, 054
SC-007-A	31	TC-001–002, 005–010, 012, 016–018, 026–030, 031–036, 039–041, 043–045, 048, 051–053, 055
SC-010-A	31	TC-001–002, 005–010, 012, 016–018, 026–030, 031–036, 039–041, 043–045, 048, 051–053, 055
SC-013-A	31	TC-001–002, 005–010, 012, 016–018, 026–030, 031–036, 039–041, 043–045, 048, 051–053, 055
SC-014-A	31	TC-001–002, 005–010, 012, 016–018, 026–030, 031–036, 039–041, 043–045, 048, 051–053, 055
SC-015-A	31	TC-001–002, 005–010, 012, 016–018, 026–030, 031–036, 039–041, 043–045, 048, 051–053, 055

5.2. RQ2: Cross-Layer Inconsistency

We investigate how interactions between web servers and frameworks introduce inconsistent host authority semantics that can enable HNP. Stage I systematically pairs each framework configuration with 108 deployment variants, replays 55 canonical requests, and observes whether the framework accepts a forged host as authoritative. Across all combinations, HALO identifies 19 recurring scenarios with authority drift, which are summarized as distinct VSFPs in Table 5. These 19 scenarios represent the intersection of all tested frameworks and serve as representative results: each authority-drift pattern appears in at least one framework, but some frameworks expose fewer variants due to missing or disabled proxy-aware modes.

We find three structural causes of cross-layer inconsistency. First, trust assumptions differ across layers. In our evaluated deployments, servers and frameworks do not always treat forwarded host bearing inputs the same way: some server configurations pass such values downstream with limited interpretation, while some framework modes may use them when deriving the effective request host. This mismatch can allow a client supplied value to become authoritative at the framework layer. Second, layers differ in what host bearing information they preserve and how they resolve precedence among it. When multiple values such as `Host`, `X-Forwarded-Host`, or `:authority` remain visible downstream, host validation and host resolution may consult different fields, creating ambiguity and enabling authority drift. In other deployments, host information is collapsed earlier, which reduces the opportunity for drift to accumulate across layers. Third, proxy related behavior varies across stacks. In some deployments, protocol and proxy metadata can influence the host value ultimately exposed to the application; in others, the effective host remains more stable unless the framework explicitly trusts forwarded host bearing headers.

Overall, deployments that enforce a single precedence and a validated proxy chain can remove authority drift. Section 5.3 then analyzes how these inconsistencies propagate into framework APIs and application logic.

5.3. RQ3: Framework Measurement

We examine how mainstream web frameworks handle host metadata using the eight defense dimensions and the two API-level metrics introduced earlier. Table 6 summarizes the results across 32 frameworks.

FW-D1 and FW-D2 capture the basic boundary checks applied to the `Host` header. Many frameworks offer a validation function or an allowlist, but these mechanisms are often optional and must be enabled by developers. Only a few frameworks, such as Django and Play, enforce trusted-host policies by default, and these systems show noticeably lower exposure [23], [24].

FW-D3 and FW-D4 describe how frameworks process host input before reuse. Most frameworks preserve the host value as received and perform only minimal normalization; lowercasing or port stripping typically occurs only when explicitly implemented by the framework or enabled through configuration [25], [26]. Default handling of forwarded headers also varies widely across ecosystems. Some frameworks ignore forwarded headers unless proxy-related settings are enabled, while others expose middleware or configuration options that enable proxy-aware behavior. [27], [28], [29]. These differences determine whether upstream proxies participate in reconstructing the effective host name or whether the framework trusts incoming headers directly.

FW-D5 and FW-D6 measure how frameworks control and check forwarded headers. Although several frameworks allow trusted-proxy configuration, deeper verification of forwarded fields remains uncommon. In our evaluated settings, most accept `X-Forwarded-Host`, `X-Forwarded-Proto`, and related headers without verifying their provenance, leaving safety dependent on deployment choices [30], [31], [32], [33]. Go and Rust frameworks generally expose fewer controls and rely more heavily on

TABLE 6: Framework-level evaluation across 32 frameworks, scored as **Yes** (●), **Partial** (◐), or **No** (○). *Host-name API Exposure* columns are **AE1** (API count) and **AE2** (Unvalidated), both as counts.

Framework Info					Defense Dimensions (D1–D8)								API Exposure	
#	Name	Lang.	Stars (k)	Forks (k)	D1	D2	D3	D4	D5	D6	D7	D8	AE1	AE2
1	FastAPI	Python	91.4	8.1	●	○	○	●	○	○	●	●	16	4
2	Django	Python	85.6	33.2	●	●	●	●	◐	○	●	●	19	3
3	Flask	Python	70.7	16.6	●	○	●	●	○	○	●	●	15	4
4	Tornado	Python	22.3	5.5	○	○	●	●	●	○	○	○	18	18
5	Sanic	Python	18.5	1.6	○	○	○	●	○	●	○	◐	11	11
6	aiohttp	Python	16.1	2.1	○	○	○	●	○	○	○	◐	7	7
7	Starlette	Python	11.6	1.1	●	○	○	●	○	○	●	●	16	4
8	Laravel	PHP	34.2	11.6	●	○	●	●	○	●	●	●	50	2
9	Symfony	PHP	30.7	9.7	●	○	●	●	○	●	●	●	20	4
10	Yii	PHP	14.3	6.9	●	○	○	●	○	●	●	●	21	4
11	Slim	PHP	12.2	2.0	○	○	●	●	○	○	○	○	17	17
12	NestJS	JS	73.3	8.1	○	○	○	●	●	●	○	○	13	13
13	Express	JS	68.1	21.4	○	○	○	●	●	●	○	○	6	6
14	Koa	JS	35.7	3.2	○	○	○	●	●	◐	○	◐	17	17
15	Fastify	JS	34.9	2.5	○	○	○	●	●	◐	○	◐	7	7
16	Sails	JS	22.9	1.9	○	○	○	●	●	●	○	○	8	8
17	Hapi	JS	14.7	1.4	○	○	○	●	○	○	○	○	9	9
18	Rails	Ruby	57.8	22.0	●	◐	○	○	○	◐	●	●	18	2
19	Sinatra	Ruby	12.4	2.1	●	◐	○	○	○	◐	●	●	17	2
20	Rocket	Rust	25.5	1.6	●	○	○	●	○	◐	○	●	15	13
21	Actix Web	Rust	23.8	1.8	○	○	○	○	○	○	○	◐	18	18
22	Axum	Rust	23.6	1.3	○	○	○	○	○	○	○	◐	44	44
23	ASP.NET Core	C#	37.3	10.5	●	○	●	●	●	●	●	●	34	4
24	Spring Boot	Java	78.8	41.6	○	○	○	◐	●	●	○	◐	3	3
25	Spring MVC	Java	59.1	38.8	○	○	○	●	●	○	○	●	21	21
26	Play Framework	Java	12.6	4.1	●	●	●	●	○	●	●	●	14	2
27	Play Framework	Scala	12.6	4.1	●	●	●	●	○	●	●	●	14	2
28	Gin	Go	86.8	8.5	○	○	○	○	○	●	○	◐	14	14
29	Fiber	Go	38.3	1.9	○	○	○	○	●	●	○	◐	17	11
30	Beego	Go	32.3	5.6	○	○	○	○	○	○	○	○	19	19
31	Echo	Go	31.7	2.3	○	○	○	○	○	◐	○	◐	16	16
32	Iris	Go	25.6	2.5	○	○	○	●	○	◐	○	◐	14	14

upstream servers, which leads to predictable patterns in their exposure.

FW-D7 reflects whether host flows pass through a required guard. Mandatory coverage is unusual: middleware chains, route-specific logic, and optional components often create bypasses that let untrusted host values reach URL builders, redirection helpers, or authentication code even when a guard exists elsewhere.

FW-D8 captures whether documentation warns developers about host-handling risks. Documentation quality varies markedly across ecosystems. Some frameworks provide clear guidance on host validation and proxy configuration, while others describe defaults that appear safer than what the implementation actually enforces [34], [35], [36], [37]. Notably, Ruby on Rails offers detailed and prominently surfaced guidance on both `Host` header validation and trusted-proxy configuration. This clarity helps reduce misconfiguration risk and is consistent with fewer observed host related issues in Rails based projects.

The API-level metrics reinforce these trends. Across all frameworks, HALO identifies 548 APIs that reuse host metadata, and more than half propagate host values without a guard. Python, JavaScript, PHP, and Go account for most of this unvalidated surface, while frameworks such as ASP.NET Core exposes a broader but more consistently protected host-flow surface, while Spring Boot exposes a narrower API surface. Across ecosystems, host-handling remains inconsistent and mostly opt-in. JavaScript, Go, and Rust frameworks rely on incomplete mechanisms and permissive defaults, while PHP, C#, and Scala exhibit clearer defenses and more explicit security policies [38], [39], [40]. Python, Java, and Ruby fall between these extremes: each avoids some parsing issues through different protective mechanisms but seldom enforce canonicalization or proxy checks. As a result, similar patterns appear within ecosystems but differ sharply across them.

Insights. Viewed as a whole, these results reveal deeper structural reasons for why HNP persists across frameworks. Modern ecosystems were built on different assumptions about where trust should begin, and those assumptions continue to shape host-handling behavior. Some frameworks treat the host name as part of their security boundary and route all flows through a central authority, while others follow lighter middleware-first designs that expose raw request fields and leave trust to the surrounding environment [41], [42], [43]. Once these patterns took hold, they propagated through the helpers that applications rely on: functions

like `request.host` or `req.headers.host` became the default way to construct absolute URLs, redirects, or callback links, and applications inherited their assumptions without revisiting them. Forwarded headers introduce further variation, as different frameworks interpret them differently and documentation often suggests safer defaults than the implementation provides. Taken together, these structural choices explain the uneven defenses observed in D1–D8 and the concentration of unguarded flows in AE1 and AE2. HNP persists because the abstractions that carry host metadata were never designed around a common trust model, and applications continue to rely on them without a clear sense of where authority should be established.

5.4. RQ4: Ecosystem-Scale Distribution

Most HNP cases come from JavaScript and Python, which together account for over 81% of the affected repositories. This skew reflects both the scale of these ecosystems and the way their frameworks handle host metadata. Express, Hapi, Flask, and FastAPI often read host fields directly and reuse them when constructing redirects, absolute URLs, or authentication links, making unvalidated hosts surface easily in common application flows [44], [45], [46]. PHP shows a similar pattern in account-recovery and login logic that assembles links from host-related server variables.

Smaller ecosystems appear less often, but for different reasons. In Java, C#, and Rust, frameworks such as Play and ASP.NET Core provide clearer guidance and stricter defaults, which reduces the number of unvalidated host flows that reach application code [47], [48]. In Go and Rust, host handling is often concentrated in gateway or infrastructure components rather than in general-purpose applications, and many frameworks provide only minimal validation by default. Scala stands out because its primary web framework, Play, enforces strict host filtering by default, and projects built on top of it rarely expose host-dependent logic; as a result, no Scala applications in our dataset exhibited HNP issues. As a result, these ecosystems show fewer cases not because they are inherently safer, but because host reconstruction is concentrated in a narrower class of projects. The ten most frequent match rules account for more than half of all cases, and most of them come from familiar helpers, such as `request.host` and `url_for` in Flask or `req.headers.host` in Express [49], [50]. These helpers appear in templates, starter kits, and routine development patterns, so the same assumptions about host trust are reproduced across many projects.

Once these patterns are present, the type of application determines how the issue shows up in practice. A large share of recent cases comes from AI agent interfaces and automation consoles, where frameworks such as FastAPI and Hapi often echo `request.headers.host` in API responses or status pages [44], [46]. Because these tools present absolute URLs directly to users or scripts, any poisoned host is immediately visible and easy to follow.

High-star consumer-facing tools like code-sharing services, visualization dashboards, and modern AI UIs also feature prominently, as they routinely return absolute URLs to browsers or automation scripts, making host misuse more visible and more likely to propagate. These application-level patterns align with broader ecosystem signals. Frameworks with permissive proxy defaults or limited documentation on host handling tend to expose more projects, while frameworks with large dependency networks, for example Express, FastAPI, Rails, and Gin, spread these assumptions widely across downstream repositories.

Overall, the data show that HNP arises from shared assumptions about how host metadata is handled across many ecosystems, and HALO reveals how these assumptions are carried through languages, frameworks, and application types, shaping the distribution of HNP in real-world code.

5.5. RQ5: False Positives

To understand how often HALO produces false positives, we validated 100 representative repositories. Each project was deployed in a controlled local environment using its documented configuration. We exercised the running instance with the Stage I request templates and checked whether forged host values appeared in boundary-visible outputs. All tests were performed in isolated sandboxes, and only public code was used.

Among the 100 cases, 82 reproduced the expected HNP behavior. The remaining 18 propagated the host value correctly but reached sinks that fall outside our attacker model. 16 cases required capabilities we do not assume, such as observing victim traffic or accessing internal responses, and 2 cases could not be fully exercised due to setup constraints. We therefore do not count these flows as vulnerabilities even though HALO extracted their host paths correctly.

These results indicate that the false positives come mainly from how consequences are classified. HALO consistently identifies the code paths through which host data moves, and the filtering step keeps its outputs aligned with the threat model. The validation confirms that HALO gives a reliable view of HNP without overstating risk.

6. Discussion

Mitigation Principles. HNP reflects a fundamental inconsistency in how different parts of the web stack reconstruct and trust host names. Effective mitigation requires unified and consistent semantics at every layer, including servers that forward requests, frameworks that rebuild origins, and applications that use the host name. Frameworks should provide built-in host validation, disable forwarded-header trust by default, and define clear precedence among `Host`, `:authority`, `X-Forwarded-Host` and `Forwarded` fields. Validation should occur early in the request path, with proxy scopes explicitly bounded by allowlists or trusted network ranges. Developers need to keep configuration consistent across servers, frameworks, and applications. All external URLs, such as redirects, OAuth callbacks, password resets, and resource links, should come from verified configuration values rather than from request headers. Enabling

and testing built-in validation or trusted-proxy mechanisms as part of deployment helps ensure deterministic authority reconstruction and prevents the propagation of unverified host data across layers.

Ethics and Disclosure. We ensure that our study complies with established ethical standards for security research. All analyzed code was obtained from publicly available sources, such as open-source GitHub repositories, in accordance with the platforms' terms of service. At no point did our analysis access private data, credentials, or user information. For vulnerability disclosure, we followed responsible reporting practices and notified affected maintainers of all identified HNP vulnerabilities prior to any public disclosure.

Prompt Sensitivity Experiments. Since HALO uses GPT-5 for final sink classification, its outputs may in principle be sensitive to prompt wording. To mitigate this risk, we treat LLM predictions as auxiliary signals and manually review all labels before reporting vulnerabilities. We conduct a small sanity check using three representative sink examples (redirect, URL generation, and non-sensitive local operation) and three semantically equivalent prompt variants, resulting in nine test instances. Across all cases, the model produces consistent sensitivity labels, indicating limited sensitivity to modest prompt variation in this setting. For reproducibility, we include the prompt variants and example inputs in Appendix C.

Limitations. Our study provides a large-scale view of Host Name Poisoning across modern web stacks; yet several limitations remain. First, our dataset covers applications built with 32 mainstream web frameworks across nine major programming languages. While this scope captures much of today's open-source web ecosystem, smaller languages, niche frameworks, and non-framework-based applications fall outside our measurement scope and may exhibit different host-handling behaviors. Second, the dynamic validation stage reproduces realistic proxy and framework compositions but does not include proprietary deployments, commercial appliances, or multi CDN infrastructures that are inaccessible for testing. Moreover, deployment models are inferred from repository-visible artifacts (e.g., code, configuration, and documentation), which may not fully reflect external infrastructure settings.

7. Related Work

Host-Header Security. Prior work has examined individual symptoms of improper host handling. Chen et al. systematically mapped ambiguities in how servers and frameworks process `Host` and related authority fields, showing how multiple or malformed values surface as routing inconsistencies, cache poisoning, and origin confusion [5]. At the application layer, Innocenti et al. measured password-reset workflows at scale and found that reset links constructed from unvalidated `Host` values can be redirected to attacker domains, enabling account takeover [4]. Additional studies show that host metadata can affect redirect handlers, HTTPS front ends, and CDN deployments, leading to origin confusion and redirection abuse [51], [52], [53]. These efforts demonstrate real security failures but focus on a limited set of consequences. Broader effects, including open redirect, OAuth link hijacking, SSRF, and authentication bypass, remain less explored, as do the cross-layer conditions that enable them. Our work addresses this gap by systematically measuring host-handling semantics across server, framework, and application layers, and by characterizing when and why Host Name Poisoning becomes exploitable in real deployments. Compared with prior studies that focus on individual manifestations, HALO provides a broader consequence space and a principled cross-layer explanation of the underlying trust inconsistencies.

Security Studies on Web Frameworks. Prior work on web framework security spans several directions. Oliveira et al. benchmark frameworks under DoS attacks, producing a resilience ranking [54]. Salas-Zárate et al. compare frameworks and their design choices [55]. Duisebekova et al. analyze Django's built-in protections, including CSRF defenses and `ALLOWED_HOSTS` validation [56]. Other studies examine isolated mechanisms such as proxy handling or URL construction, but remain framework-specific and lack cross-stack comparison. Our work complements these directions by analyzing how frameworks interpret and reuse host information across stacks. We study their handling of `Host` and `X-Forwarded-*`, identify dependent APIs, and measure how server signals influence URL construction and callbacks. This cross-framework view connects host-handling behavior to downstream consequences and clarifies the server–framework conditions under which they emerge.

8. Conclusion

We introduced HALO, a system that measures how inconsistencies in host handling across servers, frameworks, and applications lead to Host Name Poisoning. By combining dynamic tests with static dataflow analysis, HALO exposes where host values are reconstructed, trusted, and reused. Applied to 9,860 open-source projects, it surfaces 82 previously unknown vulnerabilities, 52 of which have been assigned CVEs, with consequences such as open redirects, OAuth hijacking, SSRF, and authentication bypasses. Our findings clarify the root causes of host-name trust drift and offer guidance for building safer defaults in servers, frameworks, and applications.

Ethics considerations

We follow standard ethical guidelines for security research. All analyzed code came from publicly available open-source repositories in compliance with platform terms of service. No private data, credentials, or production systems were accessed. Dynamic testing was conducted in isolated environments with rate-limited traffic. Identified vulnerabilities were responsibly disclosed to affected maintainers before any public release, and all artifacts were anonymized to preserve review anonymity.

LLM Usage Considerations

LLMs were used in this study to support methodological and editorial tasks, and all outputs were reviewed and verified by the authors. Methodologically, an LLM assisted in summarizing framework documentation and helping prioritize potential sink APIs for further analysis. LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. All technical decisions, classifications, and measurements were independently validated by the authors. No private data was provided to any model, and no model training was involved.

Acknowledgment

We would like to thank anonymous reviewers and the shepherd for their helpful comments and feedback. This work was supported in part by the National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] R. T. Fielding, M. Nottingham, and J. Reschke, "Http/1.1," IETF, Tech. Rep. RFC 9112, 2022.

[2] M. Thomson and C. Benfield, "Http/2," IETF, Tech. Rep. RFC 9113, 2022.

[3] M. Bishop, "Http/3," IETF, Tech. Rep. RFC 9114, 2022.

[4] T. Innocenti, S. A. Mirheidari, A. Kharraz, B. Crispo, and E. Kirda, "You've got (a reset) mail: A security analysis of email-based password reset procedures," in *Proc. Int'l Conf. on Detection of Intrusions, Malware, and Vulnerability Assessment (DIMVA)*, 2021.

[5] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in http implementations," in *Proc. ACM Conf. on Computer and Communications Security (CCS)*, 2016.

[6] A. Petersson and M. Nilsson, "Forwarded http extension," IETF, Tech. Rep. RFC 7239, 2014.

[7] K. Saric, F. Savins, G. S. Ramachandran, R. Jurdak, and S. Nepal, "Hyperlink hijacking: Exploiting erroneous URL links to phantom domains," in *Proc. The Web Conference (WWW)*, 2024.

[8] S. Khodayari, K. Glauber, and G. Pellegrino, "Do (not) follow the white rabbit: Challenging the myth of harmless open redirection," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2025.

[9] T. Innocenti, M. Golinelli, K. Onarlioglu, B. Crispo, and E. Kirda, "Oauth 2.0 redirect uri validation falls short, literally," in *Proc. Annual Computer Security Applications Conf. (ACSAC)*, 2023.

[10] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations," in *Proc. IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*, 2022.

[11] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Tystahl, B. Reaves, A. Bianchi, W. Enck, A. Kapravelos, and A. Machiry, "Argus: A framework for staged static taint analysis of github workflows and actions," in *Proc. USENIX Security Symposium*, 2023.

[12] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2014.

[13] G. Gousios and D. Spinellis, "Ghtorrent: GitHub's data from a firehose," in *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, 2012.

[14] W3Techs, "Usage statistics and market share of nginx," https://w3techs.com/technologies/details/ws-nginx, 2025, statistics page for Nginx usage, accessed Nov. 10, 2025.

[15] ——, "Usage statistics and market share of apache," https://w3techs.com/technologies/details/ws-apache, 2025, statistics page for Apache usage, accessed Nov. 10, 2025.

[16] Netcraft, "October 2025 web server survey," https://news.netcraft.com/archives/category/web-server-survey/, 2025, monthly web server market share report, accessed Nov. 10, 2025.

[17] W3Techs, "Usage statistics of server-side programming languages for websites," https://w3techs.com/technologies/overview/programming_language, 2025, server-side language distribution, accessed Nov. 10, 2025.

[18] Google Open Source Insights, "deps.dev: Api and public dataset," https://deps.dev/, 2025, database and official documentation, accessed Nov. 10, 2025.

[19] S. Koch, D. Klein, and M. Johns, "The fault in our stars: An analysis of github stars as an importance metric for web source code," in *Proc. MADWeb Workshop (co-located with NDSS)*, 2024.

[20] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, 2016.

[21] G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proc. 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*, 2024.

[22] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2019.

[23] Django Software Foundation, "Allowed_hosts — django settings reference," https://docs.djangoproject.com/en/5.2/ref/settings/#allowed-hosts, 2025, official documentation, accessed Nov. 10, 2025.

[24] Lightbend, "Allowedhostsfilter — play framework 3.0 documentation," https://www.playframework.com/documentation/3.0.x/AllowedHostsFilter, 2025, official documentation, accessed Nov. 10, 2025.

[25] Sanic Framework, "Sanic documentation: Core api," https://sanic.readthedocs.io/en/stable/sanic/api/core.html, 2025, official documentation, accessed Nov. 10, 2025.

[26] The Tornado Authors, "Tornado documentation: httputil module," https://www.tornadoweb.org/en/stable/_modules/tornado/httputil.html, 2025, official documentation, accessed Nov. 10, 2025.

[27] Gin maintainers, "Gin documentation: Deployment guide," https://gin-gonic.com/en/docs/deployment/, 2025, official documentation, accessed Nov. 10, 2025.

[28] Microsoft, "Asp.net core documentation: Proxy and load balancer scenarios," https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/proxy-load-balancer?view=aspnetcore-9.0#other-proxy-server-and-load-balancer-scenarios, 2025, official documentation, accessed Nov. 10, 2025.

[29] Fastify Project, "Fastify documentation: Server reference — trustproxy option," https://fastify.dev/docs/latest/Reference/Server/#trustproxy, 2025, official documentation, accessed Nov. 10, 2025.

[30] Django Software Foundation, "Django documentation: Use_x_forwarded_host setting," https://docs.djangoproject.com/en/5.2/ref/settings/#use-x-forwarded-host, 2025, official documentation, accessed Nov. 10, 2025.

[31] The Tornado Authors, "Tornado documentation: httpserver module," https://www.tornadoweb.org/en/latest/_modules/tornado/httpserver.html, 2025, official documentation, accessed Nov. 10, 2025.

[32] Koa.js Project, "Koa documentation: Application settings," https://koajs.com/#settings, 2025, official documentation, accessed Nov. 10, 2025.

[33] Spring Framework Team, "Spring documentation: Forwardedheaderfilter," https://docs.enterprise.spring.io/spring-framework/docs/6.0.24/javadoc-api/org/springframework/web/filter/ForwardedHeaderFilter.html, 2025, official documentation, accessed Nov. 10, 2025.

[34] Pallets Team, "Flask documentation: Host header validation," https://flask.palletsprojects.com/en/stable/web-security/#host-header-validation, 2025, official documentation, accessed Nov. 10, 2025.

[35] FastAPI Project, "Fastapi documentation: Trustedhostmiddleware," https://fastapi.tiangolo.com/advanced/middleware/?h=trustedhostmiddleware#trustedhostmiddleware, 2025, official documentation, accessed Nov. 10, 2025.

[36] Gin Web Framework Team, "Gin documentation: Deployment configuration options," https://gin-gonic.com/en/docs/deployment/#configuration-options, 2025, official documentation, accessed Nov. 10, 2025.

[37] Actix Web Project, "Actix web documentation: Connectioninfo," https://mozilla-services.github.io/merino/rustdoc/actix_web/dev/struct.ConnectionInfo.html, 2025, official documentation, accessed Nov. 10, 2025.

[38] Starlette Project, "Starlette documentation: Other middleware," https://starlette.dev/middleware/#other-middleware, 2025, official documentation, accessed Nov. 12, 2025.

[39] aiohttp Developers, "aiohttp documentation: Forwarded header support," https://docs.aiohttp.org/en/stable/web_advanced.html#aiohttp-web-forwarded-support, 2025, official documentation, accessed Nov. 12, 2025.

[40] Microsoft, "Asp.net core documentation: Host filtering and allowedhosts," https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/host-filtering?view=aspnetcore-9.0, 2025, official documentation, accessed Nov. 12, 2025.

[41] Ruby on Rails Team, "Rails security guide: Dns rebinding and host header attacks," https://guides.rubyonrails.org/security.html#dns-rebinding-and-host-header-attacks, 2025, official documentation, accessed Nov. 10, 2025.

[42] Pallets Team, "Flask documentation: Web security guide," https://flask.palletsprojects.com/en/stable/web-security/, 2025, official documentation, accessed Nov. 10, 2025.

[43] Django Software Foundation, "Django security topics: Host header validation," https://docs.djangoproject.com/en/5.2/topics/security/#host-header-validation, 2025, official documentation, accessed Nov. 10, 2025.

[44] HapiJS Project, "Hapi documentation: request.url," https://hapi.dev/api/?v=21.4.3#request.url, 2025, official documentation, accessed Nov. 10, 2025.

[45] Pallets Team, "Flask api reference: Request.url," https://flask.palletsprojects.com/en/stable/api/#flask.Request.url, 2025, official documentation, accessed Nov. 10, 2025.

[46] FastAPI Project, "Fastapi documentation: Using request directly," https://fastapi.tiangolo.com/advanced/using-request-directly/, 2025, official documentation, accessed Nov. 10, 2025.

[47] Lightbend, "Play framework documentation: Allowedhosts-filter," https://www.playframework.com/documentation/3.0.x/AllowedHostsFilter?#Enabling-the-allowed-hosts-filter, 2025, official documentation, accessed Nov. 10, 2025.

[48] Microsoft, "Asp.net core documentation: Host filtering and allowedhosts," https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/host-filtering?view=aspnetcore-10.0, 2025, official documentation, accessed Nov. 10, 2025.

[49] Pallets Projects, "Flask api reference: url_for," https://flask.palletsprojects.com/en/stable/api/#flask.Flask.url_for, 2025, official documentation, accessed Nov. 10, 2025.

[50] ——, "Flask api reference: Request.host," https://flask.palletsprojects.com/en/stable/api/#flask.Request.host, 2025, official documentation, accessed Nov. 10, 2025.

[51] A. Delignat-Lavaud and K. Bhargavan, "Network-based origin confusion attacks against HTTPS virtual hosting," in *WWW*, 2015.

[52] S. Hao, Y. Zhang, H. Wang, and A. Stavrou, "End-users get maneuvered: Empirical analysis of redirection hijacking in cdns," in *USENIX Security*, 2018.

[53] S. Pletinckx, C. Kruegel, and G. Vigna, "A large-scale measurement study of the proxy protocol and its security implications," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2025.

[54] R. A. Oliveira, M. M. Raga, N. Laranjeiro, and M. Vieira, "An approach for benchmarking the security of web service frameworks," *Future Generation Computer Systems*, 2020.

[55] M. del Pilar Salas-Zárate, G. Alor-Hernández, R. Valencia-García, L. Rodríguez-Mazahua, A. Rodríguez-González, and J. L. L. Cuadrado, "Analyzing best practices on web development frameworks: The lift approach," *Science of Computer Programming*, 2015.

[56] K. Duisebekova, R. Khabirov, and A. Zholzhan, "Django as secure web-framework in practice," *The Bulletin of Kazakh Academy of Transport and Communications named after M. Tynyshpayev*, 2021.

Appendix A. Scenario Matrix

This appendix lists all Stage I deployment scenarios. They cover all combinations of framework trust settings (A0–A4) and proxy modes (B0–B2). We separate *direct baselines* (suffix D) with no proxy from *behind-proxy configurations* (suffix N/A) tested with Nginx (N) and Apache (A). Table 7 gives the direct baselines, and Table 8 gives the Nginx and Apache variants, together forming the full set of 60 scenarios.

TABLE 7: Direct scenarios and their corresponding SC-IDs.

D-ID → SC-ID	D-ID → SC-ID
SC-001-D → SC-001	SC-007-D → SC-019
SC-002-D → SC-002	SC-008-D → SC-022
SC-003-D → SC-003	SC-009-D → SC-025
SC-004-D → SC-005	SC-010-D → SC-028
SC-005-D → SC-006	SC-011-D → SC-031
SC-006-D → SC-016	SC-012-D → SC-045

Appendix B. Request Templates

Table 9 and Table 10 summarizes all 55 crafted HTTP request templates used in Stage I. Each row lists the test case ID, short category, illustrative header fields (or absolute-form request line), and a concise explanation. The full expanded catalog is included in our artifact.

TABLE 8: 48 behind-proxy scenarios shared by Nginx (N) and Apache (A).

ID	Name	Brief explanation
SC-001	NoAppConfig+NoOverride	Transparent edge; no validation.
SC-002	TRUSTED_HOSTS+NoOverride	Allowlist filters hosts.
SC-003	SERVER_NAME+NoOverride	Canonical host enforced.
SC-004	NoAppConfig+HostWhitelist	Proxy whitelist only; misalignment possible.
SC-005	ProxyFixNoValidation+NoOverride	Trusts any `X-Forwarded-*`.
SC-006	ProxyFixWithValidation+NoOverride	Checks exist; proxy scope absent.
SC-007	ProxyFixNoValidation+XFHOverride	Proxy injects host; unscoped trust.
SC-008	SERVER_NAME+XFHOverride	Canonical host wins if proxy not trusted.
SC-009	ProxyFixNoValidation+HostWhitelist	Proxy filters; unscoped trust may override.
SC-010	ProxyFixWithValidation+XFHOverride	Proxy injects host; scope decides.
SC-011	TRUSTED_HOSTS+XFHOverride	Allowlist blocks injected host.
SC-012	NoAppConfig+XFHOverride	No guards; forwarded host accepted.
SC-013	TRUSTED_HOSTS+HostWhitelist	Dual allowlists; mismatch risk.
SC-014	SERVER_NAME+HostWhitelist	Canonical + proxy lists must align.
SC-015	ProxyFixWithValidation+HostWhitelist	Filters both ends; scope decides.
SC-016	SERVER_NAME+TRUSTED_HOSTS+NoOverride	Two guards, no proxy.
SC-017	SERVER_NAME+TRUSTED_HOSTS+XFHOverride	Guards active; forwarded host ignored.
SC-018	SERVER_NAME+TRUSTED_HOSTS+HostWhitelist	Three filters; alignment needed.
SC-019	SERVER_NAME+ProxyFixNoValidation+NoOverride	Forwarded host overrides canonical.
SC-020	SERVER_NAME+ProxyFixNoValidation+XFHOverride	Proxy injects host; unscoped trust.
SC-021	SERVER_NAME+ProxyFixNoValidation+HostWhitelist	Filtering partial; unscoped wins.
SC-022	SERVER_NAME+ProxyFixWithValidation+NoOverride	Nominal checks only.
SC-023	SERVER_NAME+ProxyFixWithValidation+XFHOverride	Proxy injects host; scope matters.
SC-024	SERVER_NAME+ProxyFixWithValidation+HostWhitelist	Filters both; consistent config needed.
SC-025	TRUSTED_HOSTS+ProxyFixNoValidation+NoOverride	Unscoped proxy bypasses allowlist.
SC-026	TRUSTED_HOSTS+ProxyFixNoValidation+XFHOverride	Proxy injects host; unscoped trust.
SC-027	TRUSTED_HOSTS+ProxyFixNoValidation+HostWhitelist	Dual filters; unscoped override.
SC-028	TRUSTED_HOSTS+ProxyFixWithValidation+NoOverride	Nominal checks; no scoping.
SC-029	TRUSTED_HOSTS+ProxyFixWithValidation+XFHOverride	Scoped trust keeps allowlist intact.
SC-030	TRUSTED_HOSTS+ProxyFixWithValidation+HostWhitelist	Two filters; scope alignment required.
SC-031	SERVER_NAME+TRUSTED_HOSTS+ProxyFix+NoOverride	All guards; proxy order matters.
SC-032	SERVER_NAME+TRUSTED_HOSTS+ProxyFix+XFHOverride	Scoped forwarding preserves authority.
SC-033	SERVER_NAME+TRUSTED_HOSTS+ProxyFix+HostWhitelist	Fully filtered; consistency needed.
SC-034	NoAppConfig+XFHOverride+HostWhitelist	Proxy injects + filters; no guards.
SC-035	TRUSTED_HOSTS+XFHOverride+HostWhitelist	Proxy injects host; allowlist blocks.
SC-036	SERVER_NAME+XFHOverride+HostWhitelist	Proxy injects host; canonical wins.
SC-037	ProxyFixNoValidation+XFHOverride+HostWhitelist	Proxy injects host; unscoped wins.
SC-038	ProxyFixWithValidation+XFHOverride+HostWhitelist	Proxy injects host; scope decides.
SC-039	SERVER_NAME+TRUSTED_HOSTS+XFHOverride+HostWhitelist	Guards prevent drift.
SC-040	SERVER_NAME+ProxyFix+XFHOverride+HostWhitelist	Scoped proxy needed.
SC-041	TRUSTED_HOSTS+ProxyFix+XFHOverride+HostWhitelist	Scoped proxy + allowlist safe.
SC-042	SERVER_NAME+TRUSTED_HOSTS+ProxyFix+XFHOverride+HostWhitelist	Fully guarded; misalignment risky.
SC-043	SERVER_NAME+ProxyFixNoValidation+XFHOverride+HostWhitelist	Proxy injects host; unscoped overrides.
SC-044	TRUSTED_HOSTS+ProxyFixNoValidation+XFHOverride+HostWhitelist	Proxy injects host; unscoped defeats allowlist.
SC-045	SERVER_NAME+TRUSTED_HOSTS+ProxyFixNoValidation+NoOverride	Unscoped forwarding dominates.
SC-046	SERVER_NAME+TRUSTED_HOSTS+ProxyFixNoValidation+XFHOverride	Proxy injects host; unscoped wins.
SC-047	SERVER_NAME+TRUSTED_HOSTS+ProxyFixNoValidation+HostWhitelist	Unscoped forwarding overrides filter.
SC-048	SERVER_NAME+TRUSTED_HOSTS+ProxyFixNoValidation+XFHOverride+HostWhitelist	Max config; strict scoping required.

Nginx scenario IDs (48).

SC-001-N, SC-002-N, SC-003-N, SC-004-N, SC-005-N,
SC-006-N, SC-007-N, SC-008-N, SC-009-N, SC-010-N,
SC-011-N, SC-012-N, SC-013-N, SC-014-N, SC-015-N,
SC-016-N, SC-017-N, SC-018-N, SC-019-N, SC-020-N,
SC-021-N, SC-022-N, SC-023-N, SC-024-N, SC-025-N,
SC-026-N, SC-027-N, SC-028-N, SC-029-N, SC-030-N,
SC-031-N, SC-032-N, SC-033-N, SC-034-N, SC-035-N,
SC-036-N, SC-037-N, SC-038-N, SC-039-N, SC-040-N,
SC-041-N, SC-042-N, SC-043-N, SC-044-N, SC-045-N,
SC-046-N, SC-047-N, SC-048-N.

Apache scenario IDs (48).

SC-001-A, SC-002-A, SC-003-A, SC-004-A, SC-005-A,
SC-006-A, SC-007-A, SC-008-A, SC-009-A, SC-010-A,
SC-011-A, SC-012-A, SC-013-A, SC-014-A, SC-015-A,
SC-016-A, SC-017-A, SC-018-A, SC-019-A, SC-020-A,
SC-021-A, SC-022-A, SC-023-A, SC-024-A, SC-025-A,
SC-026-A, SC-027-A, SC-028-A, SC-029-A, SC-030-A,
SC-031-A, SC-032-A, SC-033-A, SC-034-A, SC-035-A,
SC-036-A, SC-037-A, SC-038-A, SC-039-A, SC-040-A,
SC-041-A, SC-042-A, SC-043-A, SC-044-A, SC-045-A,
SC-046-A, SC-047-A, SC-048-A.

TABLE 9: Crafted Request Templates (TC-001–TC-028)

ID	Category	Headers
TC-001	Host manipulation	`Host: example.com`
TC-002	Host + XFH	`Host: example.com; XFH: attacker.com`
TC-003	XFH-only	`X-Forwarded-Host: attacker.com`
TC-004	Empty Host + XFH	`Host: ; XFH: attacker.com`
TC-005	Port injection	`Host: example.com:80`
TC-006	Routing bypass	`Host: example.com; X-Real-IP: 192.168.1.100`
TC-007	Subdomain bypass	`Host: evil.example.com`
TC-008	Default vhost	`Host: example.com; XFH: attacker.com; XFF: 10.0.0.1`
TC-009	Apache-specific	`Host: example.com; XFH: attacker.com; XFP: http`
TC-010	Nginx-specific	`Host: example.com; XFH: attacker.com; X-Original-Host: localhost`
TC-011	Priority confusion	`Host: localhost; XFH: attacker.com`
TC-012	Alt port	`Host: example.com:8080`
TC-013	User-info Host	`Host: localhost@example.com`
TC-014	IP Host + XFH	`Host: 127.0.0.1; XFH: attacker.com`
TC-015	Case confusion	`Host: LoCaLhOsT; XFH: attacker.com`
TC-016	XFS compare	`Host: example.com; XFS: attacker.com`
TC-017	X-Host variant	`Host: example.com; X-Host: attacker.com`
TC-018	Env pollution	`Host: example.com; XFH: attacker.com; XFF: 1.1.1.1`
TC-019	XFH spoofing	`Host: localhost; XFH: attacker.com; XFF: 127.0.0.1`
TC-020	XFH + proto conflict	`Host: localhost; XFH: attacker.com; XFP: https`
TC-021	XFH + XRI	`X-Forwarded-Host: attacker.com; X-Real-IP: 192.168.1.1`
TC-022	XFF spoofing	`Host: localhost; XFF: 192.168.1.100; XFH: attacker.com`
TC-023	XFP alteration	`Host: localhost; XFP: https; XFH: attacker.com`
TC-024	Multi-value XFH	`Host: localhost; XFH: example.com, attacker.com`
TC-025	Full X-Forwarded set	`Host: example.com; XFH/XFF/XFP/XRI all attacker`
TC-026	Forwarded host override	`Forwarded: host=evil.com`
TC-027	Forwarded proto+host	`Forwarded: proto=https; host=evil.com; for=1.2.3.4`
TC-028	Forwarded vs XFH	`Forwarded: host=evil.com; XFH: localhost`

TABLE 10: Crafted Request Templates (TC-029–TC-055)

ID	Category	Headers
TC-029	Repeated XFH	`X-Forwarded-Host: attacker.com, localhost`
TC-030	Case-mixed XFH	raw `x-forwarded-host` vs `X-FORWARDED-HOST`
TC-031	Whitespace XFH	raw `X-Forwarded-Host: \tattacker.com`
TC-032	Mixed multi-value	`XFH: example.com, attacker.com; Forwarded: host="evil.com:8080"`
TC-033	X-Original-Host	`Host: example.com; X-Original-Host: attacker.com`
TC-034	XFS + X-Host	`X-Forwarded-Server / X-Host`
TC-035	Forwarded IPv6	`Forwarded: for="[2001:db8::1]"; host="evil.com"`
TC-036	Host priority test	`Host: example.com; XFH: localhost; Forwarded: host=benign`
TC-037	XFH priority test	`Host: localhost; XFH: attacker.com; Forwarded: host=benign`
TC-038	Forwarded priority test	`Host: localhost; XFH: localhost; Forwarded: host=evil.com`
TC-039	Forwarded chain	`Forwarded: for=1.1.1.1; host=evil.com, for=2.2.2.2; host=benign`
TC-040	Forwarded casing	`Forwarded: HOST=evil; host=benign; Host=example`
TC-041	Forwarded quoted port	`Forwarded: host="evil.com:443"`
TC-042	Control characters	raw `XFH: \tattacker.com\r`
TC-043	User-info XFH	`X-Forwarded-Host: attacker.com@localhost`
TC-044	Percent-encoded XFH	`X-Forwarded-Host: attacker%2ecom`
TC-045	Dual override	`XFH: attacker; Forwarded: host=evil`
TC-046	Triple override	`X-Original-Host/XFH/Forwarded all attacker`
TC-047	Server-header fallback	`X-Forwarded-Server: attacker; Forwarded: host=evil`
TC-048	HTTP/2 authority override	`:authority: attacker.com`
TC-049	HTTP/2 vs XFH	`:authority: localhost; XFH: attacker.com`
TC-050	HTTP/2 vs Forwarded	`:authority: localhost; Forwarded: host=evil.com`
TC-051	HTTP/2 Forwarded chain	`:authority: attacker.com; Forwarded: for=1.1.1.1; ...`
TC-052	HTTP/2 duplicate authority	`:authority: attacker.com; XFH: localhost`
TC-053	Absolute URI override	`GET http://attacker.com/...; Host: localhost`
TC-054	Absolute URI + XFH	`GET http://localhost/...; XFH: attacker.com`
TC-055	Absolute URI + Forwarded	`Host: localhost; Forwarded: host=evil.com`

Appendix C.
Prompt Variants for Sensitivity Check

This appendix documents the three semantically equivalent prompt variants used in our sink classification sensitivity check for the final sink filtering step.

Input. Each prompt receives lightweight information already available from our static analysis: the programming framework, the candidate sink symbol, the callsite code with line numbers, and a small amount of local context from surrounding code. We do not provide full taint traces, broad interprocedural summaries, or external documentation at this stage, since the goal is only to help filter candidate sinks rather than to confirm end-to-end vulnerabilities.

Output. The model returns a JSON object with the following structure.

```
{ "sink_name": "<sink symbol>",
  "effect_category": "<category>",
  "is_security_sensitive": "<
    ↪ SECURITY_SENSITIVE |
    ↪ NOT_SECURITY_SENSITIVE |
       UNCERTAIN>",
  "effect_explanation": "<brief
    ↪ explanation>",
  "evidence_lines": ["Lx", "Ly-Lz"],
  "confidence": "<high|medium|low>"
```

Prompt Variants (A/B/C). The three variants differ in role description, decision phrasing, and terminology, while preserving the same task semantics.

```
[A] You are a neutral security reviewer.
Decide whether a cited API invocation is a
final sink with externally observable
impact.

[B] You are an impartial security
analyst. Determine whether the cited API
call is a terminal sink with externally
observable security impact.

[C] You are a security triage reviewer.
Judge whether the referenced API
invocation is the final sink and has
externally observable impact.

Externally observable = any action whose
result escapes the current process or
request.
Consider the following common sensitive
classes (examples, not exhaustive):
- REDIRECT: HTTP redirects, client
navigation helpers, and Location setters.
- URL_GENERATION: APIs used to create
externally reachable URLs (e.g., signed
or presigned links, object URLs, helpers
with `_external=True`).
- AUTHENTICATION: APIs that issue or
validate credentials, mutate sessions, or
change login state.
- EXTERNAL_REQUEST: APIs that initiate
outbound network I/O
  (e.g., HTTP(S), gRPC, WebSocket, DNS).
Non-sensitive examples include local
string processing, logging, error
construction, or utility helpers with no
redirect, authentication, or network side
effects.
Return exactly one of:
- SECURITY_SENSITIVE: the sink falls into
a sensitive class above.
- NOT_SECURITY_SENSITIVE: no externally
observable security effect is evident.
- UNCERTAIN: the effect is unclear (e.g.,
due to wrappers or aliases).
Reason silently. Output valid JSON only,
following the schema above.
```

Representative example. All prompt variants use the same few-shot examples. For brevity, we show one representative example here.

```
Language/Framework: Express
Sink symbol: express.Response.redirect
Callsite code (with line numbers):
67:   return res.redirect(avatarUrl);
Local context from nearby lines:
62: async function
finishAvatarUpload(req, res) {
63:   const user = await
Users.findById(req.session.userId);
64:   const fileName =
`${user.id}/${Date.now()}-
${req.file.filename}`;
65:   const avatarHost = user.profileHost
|| config.assetHost;
66:   const avatarUrl =
`https://${avatarHost}/avatars/${fileName}`;
67:   return res.redirect(avatarUrl);}
```

Representative exemplar output.

```
{ "sink_name": "express.Response.redirect"
    ↪ ,
  "effect_category": "REDIRECT",
  "is_security_sensitive": "
    ↪ SECURITY_SENSITIVE",
  "effect_explanation": "Redirect issues a
    ↪  response whose destination is
    ↪ derived from avatarUrl, which is
    ↪ constructed using avatarHost and
    ↪ fileName.",
  "evidence_lines": ["L65-L67"],
  "confidence": "high"
```

Across prompt variants, the model consistently assigns the same sensitivity label to this example, with only minor wording differences in the explanation.

Appendix D.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

The paper studies host name poisoning which exploits how different servers reconstruct and reuse the host name. The HALO framework presented in the paper focuses on how host values are reconstructed and trusted across servers, frameworks, and applications, covering both direct and behind-proxy deployments and how clients can manipulate host names to trigger vulnerabilities.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

D.3. Reasons for Acceptance

1) The HALO framework proposed in the paper combines static analysis with dynamic testing to detect Host Name Poisoning (HNP) attacks. The value of this framework is evidenced in the number of CVEs that this work has identified (19 so far) especially in popular open source software.
2) The HNP attack surface is under-researched and the this paper is a valuable step forward in a principled analysis of how different layers of the modern web can interact with each other inconsistently to provide an opening for attacks.

D.4. Noteworthy Concerns

None.