# SUBGRAPH

# FINDINGS REPORT

**Hushline**

December 30, 2025

Prepared for: Hushline

Subgraph Technologies, Inc.
345 Av. Victoria, Suite 400
Montreal, Quebec
https://subgraph.com

# Contents

## Overview

We conducted a penetration test of the Hushline application in December of 2025. This report documents our observations and security findings.

Most testing and review was performed on the `v0.4.76` version source repository in our own testing environment using Docker. We also made an account on the production application (https://tip.hushline.app) but no live testing was performed on production. The production application was used as a reference to compare against our testing environment.

## Summary

| No. | Title | Severity | Remediation |
|---|---|---|---|
| V-001 | Blind SSRF Private Network Enumeration in Profile Links | Medium | Unresolved |
| V-002 | Client-Side Encryption Timeout Encryption Failure | Low | Unresolved |
| V-003 | Sequential Message IDs | Low | Unresolved |

## Observations

As part of the test, we looked for common vulnerabilities using manual and automated techniques as well as source code review. This section, separate from the security findings, lists our observations.

### SQL injection

We performed automated testing and source code review of the front-end and back-end code to look for potential SQL injection vulnerabilities.

The database operations are implemented with `SQLAlchemy`. This library, when best practices are employed, will prevent SQL injection vulnerabilities. During our test, we looked for issues like raw SQL queries with user input, unsafe string concatenation of user input, and other known bad practices. We did not observe any cases where the application deviated from best practices in unsafe ways.

### Cross-site scripting

We looked for cross-site scripting issues using manual and automated methods in addition to reviewing source code.

The application renders user-supplied input using a combination of `Jinja` templating with the `markupsafe` and `Bleach` libraries. We didn't observe any cases where user-supplied input was rendered back unsafely.

### Cross-site request forgery

We performed manual tests and reviewed source code to determine if cross-site request forgery (CSRF) issues were present. This included tests like submitting forms with bad or missing CSRF tokens.

The application incorporates the `WTForms` library for CSRF protection and this proved mostly effective during our tests.

In one instance, we observed that the form for deleting messages did not include a CSRF token. However, we could not exploit the issue because the application sets `SameSite` headers when setting cookies. Therefore, submitting a malicious off-site form caused a redirect to the authentication page due to lack of cookie-based authentication credentials.

The developers should review the message deletion workflow to determine if the missing CSRF token is an oversight.

# Details

## V-001: Blind SSRF Private Network Enumeration in Profile Links

| Severity | Remediation |
|---|---|
| Medium | Unresolved |

### Discussion

Hushline includes a feature to perform basic checks for valid social media profiles included in user profiles.

There is a blind server-side request forgery (SSRF) vulnerability that is present when the application attempts to verify URLs submitted in user Profile Details. The issue is that the code that makes requests to external sites does not prevent private IP addresses from being requested.

The vulnerable code exists in `hushline/settings/common.py:verify_url()`.

We performed tests on our network to see what was possible, examining application logs, and network traffic from the application to our network. We found response timing patterns that enable enumeration of the private network where the application is hosted.

### Impact Analysis

The attacker must be authenticated to the application to exploit the issue. To exploit it, they will automate profile link submissions containing private IPs and observe the response metadata from the application.

There are a few limitations to exploitation:

- Only HTTPS URLs will be requested by the application
- The attacker must infer details about the network from the timing of application responses

We tested a few scenarios: - Looking for ports on the local machine 127.0.0.1 - Looking for live IP addresses in private IP ranges, using common ports for TLS-based and non-TLS based network services

Our tests revealed some response timing patterns that could be used to infer whether IP addresses were live and identify ports that were open and running services.

We explored a scenario where third-party services echo back full URLs, allowing an attacker to use the verification status to enumerate services. To take advantage of this scenario, the service must have a valid TLS certificate that is not self-signed and return the full URL in a response with a 200 status code.

For this to work, the attacker would submit a URL like the following:

```
https://private-ip/<a href="http://127.0.0.1:8080/to/subgraph-user" rel="me">verify</a>
```

In a case where all of these conditions are met, the application will set the verified symbol in the profile. This would allow enumeration of the service without measuring response times. While possible, it doesn't seem like this scenario is at all that common.

## Remediation Recommendations

The verification function should not be able to request private IP addresses. It could check against a block list such as the following:

- localhost
- 127.0.0.0/8
- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16
- ::1/128
- fc00::/7
- fe80::/10

## Additional Information

N/A

## V-002: Client-Side Encryption Timeout Encryption Failure

| Severity | Remediation |
|---|---|
| Low | Unresolved |

### Discussion

There is an issue in the client-side JavaScript message encryption operations that could allow unencrypted text to be submitted to the server. If the encryption operations take too long when upon submission then unencrypted text will be submitted to the server.

The issue is present in `assets/js/client-side-encryption.js:document.addEventListener()`.

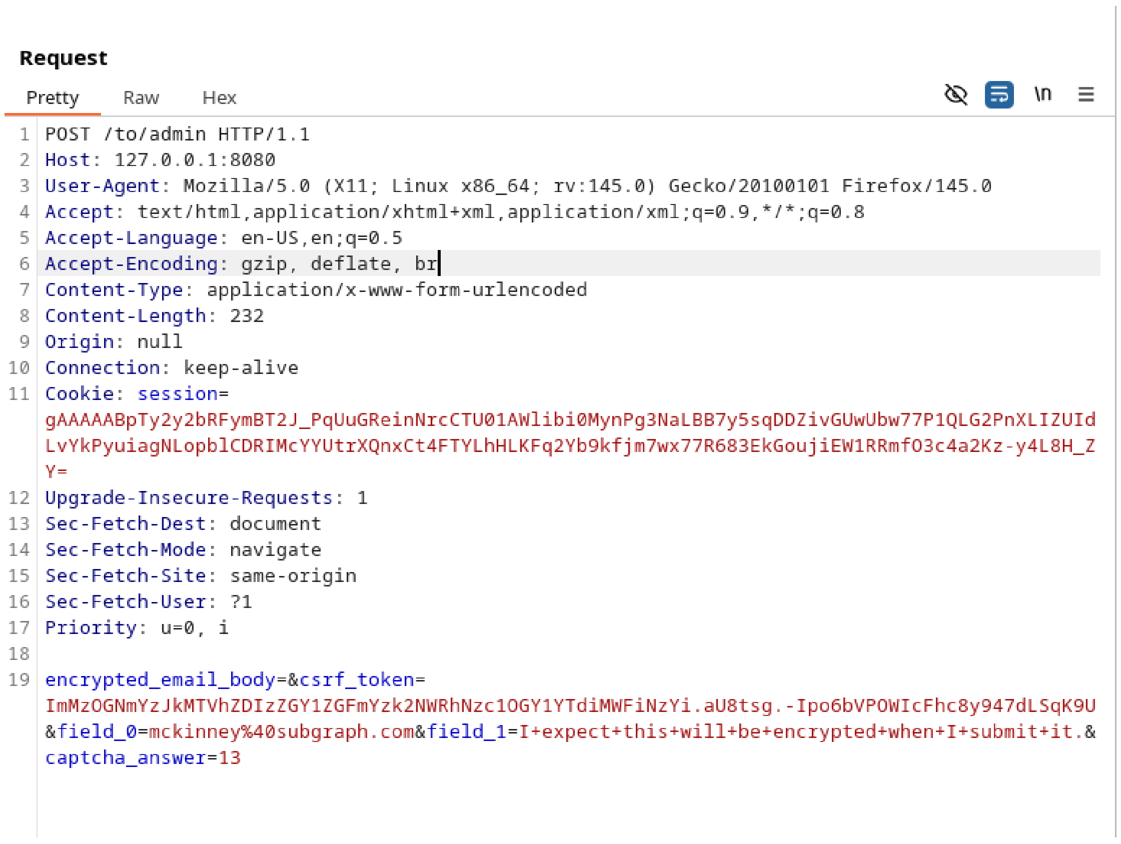
The affected function adds an event listener on form submission that perform the necessary steps to encrypt the form submission fields but sets a timeout (100ms) at the end of the listener to wait for the page's DOM to update with the encrypted fields before submission. If the operations take longer, the form will still be submitted even if the DOM has not been updated with encrypted text.

This is a snippet of the code at the end of form's submit event listener:

```
setTimeout(() => {
      form.submit();
    }, 100);
```

We were able to reproduce the issue by changing the timeout value in our browser console to 1ms and then submitting the message form.

The following screenshot shows that the unencrypted text was submitted to the server:



```
POST /to/admin HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:145.0) Gecko/20100101 Firefox/145.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 232
Origin: null
Connection: keep-alive
Cookie: session=
gAAAAABpTy2y2bRFymBT2J_PqUuGReinNrcCTU01AWlibi0MynPg3NaLBB7y5sqDDZivGUwUbw77P1QLG2PnXLIZUId
LvYkPyuiagNLopblCDRIMcYYUtrXQnxCt4FTYLhHLKFq2Yb9kfjm7wx77R683EkGoujiEW1RRmfO3c4a2Kz-y4L8H_Z
Y=
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i

encrypted_email_body=&csrf_token=
ImMzOGNmYzJkMTVhZDIzZGY1ZGFmYzk2NWRhNzc1OGY1YTdiMWFiNzYi.aU8tsg.-Ipo6bVPOWIcFhc8y947dLSqK9U
&field_0=mckinney%40subgraph.com&field_1=I+expect+this+will+be+encrypted+when+I+submit+it.&
captcha_answer=13
```

### Impact Analysis

This issue isn't likely to occur very often but it could happen when a very large message is submitted and/or a message is submitted from a low-powered device.

The consequences are the following:

1. A message that is supposed to be unencrypted is sent to the server in plaintext. This could appear in server logs. The application does not successfully save the message for the Hushline user recipient.
2. As the message is not saved but the application, the person who sent the message may expect that it was received.

In the case of #2, the normal UX of showing the encrypted content prior to submission does not occur. A user who is familiar with the application may suspect that something didn't work. But it all happens so quickly that they could miss the visual cues. A first-time user would have no idea what is supposed to happen when they submit a message.

### Remediation Recommendations

The submit event listener could be written to avoid using timeouts. For example:

```
form.addEventListener("submit", async function (event) {
  event.preventDefault();

  try {
    // Build an email body with all fields, encrypt it, and add it to
     ↪  the DOM as a hidden field
    let emailBody = "";
    document.querySelectorAll(".form-field").forEach((field) => {
      const value = getFieldValue(field);
      const label = getFieldLabel(field);
      emailBody += `#
↪  ${label}\n\n${value}\n\n====================\n\n`;
    });

    const encryptedEmailBody = await encryptMessage(
      publicKeyArmored,
      emailBody,
    );

    if (encryptedEmailBody) {
      const encryptedEmailBodyEl = document.getElementById(
        "encrypted_email_body",
      );
      encryptedEmailBodyEl.value = encryptedEmailBody;
    } else {
      throw new Error("Client-side encryption failed for email body");
    }

    // Loop through all encrypted fields and encrypt them
    // Use Promise.all to wait for all encryptions to complete
    const encryptionPromises = Array.from(
      document.querySelectorAll(".encrypted-field")
    ).map(async (field) => {
     // Do the encryption steps

  } else {
        throw new Error(
          `Client-side encryption failed for field: ${field.name ||
           ↪  "unknown"}`
        );
      }
    });
```

```
      // Wait for all encryption operations to complete before
      ↪  submitting
      await Promise.all(encryptionPromises);

      console.log("All encryptions completed successfully - submitting
      ↪  form");

      // Now it's safe to submit the form
      form.submit();

    } catch (error) {
      console.error("Encryption error:", error);
      alert(
        "Message encryption failed. Your message was NOT submitted for
        ↪  security reasons. Please try again."
      );
    }
  });
```

UX enhancements could also address the issue. For example, some indication that the message was encrypted successfully and a prompt to confirm submission.

### Additional Information

N/A

## V-003: Sequential Message IDs

| Severity | Remediation |
|---|---|
| Low | Unresolved |

### Discussion

The IDs for submitted messages are sequential and instance-wide.

While the messages are encrypted and we tested that one user cannot access another user's messages, there are a few issues to consider.

1. Latent vulnerabilities in the future where being able to guess message IDs is useful.
2. It also may not be desirable for users to know how many messages have been processed by the service.

### Impact Analysis

Guessable message IDs may be useful in potential attack scenarios.

The volume of submitted messages is not private to any application user.

### Remediation Recommendations

UUIDs could be used instead of sequential message IDs. Alternatively, sequential message IDs on a per-user basis is an improvement though it may not be effective if a future vulnerability relies on guessable IDs.

### Additional Information

N/A

# Appendix

## Methodology

Our approach to testing is designed to understand the design, behavior, and security considerations of the assets being tested. This helps us to achieve the best coverage over the duration of the test.

To accomplish this, Subgraph employs automated, manual and custom testing methods. We conduct our automated tests using the industry standard security tools. This may include using multiple tools to test for the same types of issues. We perform manual tests in cases where the automated tools are not adequate or reveal behavior that must be tested manually. Where required, we also develop custom tools to perform tests or reproduce test findings.

The goals of our testing methodology are to:

- Understand the expected behavior and business logic of the assets being tested
- Map out the attack surface
- Understand how authentication, authorization, and other security controls are implemented
- Test for flaws in the security controls based on our understanding
- Test every point of input against a large number of variables and observe the resulting behavior
- Reproduce and re-test findings
- Gather enough supporting information about findings to enable us to classify, report, and suggest remediations

## Description of testing activities

Depending on the type and scope of the engagement, our methodology may include any of the following testing activities:

1. **Information Gathering:** Information will be gathered from publicly availble sources to help increase the success of attacks or discover new vulnerabilities
2. **Network discovery:** The networks in scope will be scanned for active, reachable hosts that could be vulnerable to compromise
3. **Host Vulnerability Assessment:** Hosts applications and services will be assessed for known or possible vulnerabilities
4. **Application Exploration:** The application will be explored using manual and automated methods to better understand the attack surface and expected behavior
5. **Session Management:** Session management in web applications will be tested for security flaws that may allow unauthorized access
6. **Authentication System Review:** The authentication system will be reviewed to determine if it can be bypassed
7. **Privilege Escalation:** Privilege escalation checks will be performed to determine if it is possible for an authenticated user to gain access to the privileges assigned to another role or administrator

8. **Input Validation:** Input validation tests will be performed on all endpoints and fields within scope, including tests for injection vulnerabilities (SQL injection, cross-site scripting, command injection, etc.)
9. **Business Logic Review:** Business logic will be reviewed, including attempts to subvert the intended design to cause unexpected behavior or bypass security controls

## Reporting

Findings reports are peer-reviewed within Subgraph to produce the highest quality findings. The report includes an itemized list of findings, classified by their severity and remediation status.

### Severity ratings

Severity ratings are a metric to help organizations prioritize security findings. The severity ratings we provide are simple by design so that at a high-level they can be understood by different audiences. In lieu of a complex rating system, we quantify the various factors and considerations in the body of the security findings. For example, if there are mitigating factors that would reduce the severity of a vulnerability, the finding will include a description of those mitigations and our reasoning for adjusting the rating.

At an organization's request, we will also provide third-party ratings and classifications. For example, we can analyze the findings to produce *Common Vulnerability Scoring System* (CVSS)[1] scores or *OWASP Top 10*[2] classifications.

The following is a list of the severity ratings we use with some example impacts:

**Critical**

Exploitation could compromise hosts or highly sensitive information

Critical Exploitation could compromise hosts or highly sensitive information

**High**

Exploitation could compromise the application or moderately sensitive information

High Exploitation could compromise the application or moderately sensitive information

**Medium**

Exploitation compromises multiple security properties (confidentiality, integrity, or availability)

Medium Exploitation compromises multiple security properties (confidentiality, integrity, or availability)

[1] https://www.first.org/cvss/
[2] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

**Low**

Exploitation compromises a single security property (confidentiality, integrity, or availability)

Low Exploitation compromises a single security property (confidentiality, integrity, or availability)

**Info**

Finding does not directly pose a security risk but merits further investigation

Info Finding does not directly pose a security risk but merits further investigation

The severity of a finding is often a product of the impact to general security properties of an application, host, network, or other information system.

The properties that can be impacted are:

**Confidentiality** Exploitation results in authorized access to data

**Integrity** Exploitation results in the unauthorized modification of data or state

**Availability** Exploitation results in a degradation of performance or an inability to access resources

The actual severity of a finding may be higher or lower depending on a number of other factors that may mitigate or exacerbate it. These include the context of the finding in relation to the organization as well as the likelihood of exploitation. These are described in further detail below.

### Contextual factors

Confidentiality, integrity, and availability are one dimension of the potential risk of a security finding. In some cases, we must also consider contextual factors that are unique to the organization and the assets tested.

The following is a list of those factors:

**Financial** Exploitation may result in financial losses

**Reputation** Exploitation may result in damage to the reputation of the organization

**Regulatory** Exploitation may expose the organization to regulatory liability (e.g. make them non-compliant)

**Organizational** Exploitation may disrupt the operations of the organization

### Likelihood

Likelihood measures how probable it is that an attacker exploit a finding.

This is determined by numerous factors, the most influential of which are listed below:

**Authentication** Whether or not the attack must be authenticated

**Privileges** Whether or not an authenticated attacker requires special privileges

**Public exploit** Whether or not exploit code is publicly available

**Public knowledge** Whether or not the finding is publicly known

**Exploit complexity** How complex it is for a skilled attacker to exploit the finding

**Local vs. remote** Whether or not the finding is exposed to the network

**Accessibility** Whether or not the affected asset is exposed on the public Internet

**Discoverability** How easy it is for the finding to be discovered by an attacker

**Dependencies** Whether or not exploitation is dependant on other findings such as information leaks

## Remediation status

As part of our reporting, remediation recommendations are provided to the client. To help track the issues, we also provide a remediation status rating in the findings report.

In some cases, the organization may be confident to remediate the issue and test it internally. In other cases, Subgraph works with the organization to re-test the findings, resulting in a subsequent report reflecting remediation status updates.

If requested to re-test findings, we determine the remediation status based on our ability to reproduce the finding. This is based on our understanding of the finding and our awareness of potential variants at that time. To reproduce the results, the re-test environment should be as close to the original test environment as possible.

Security findings are often due to unexpected or unanticipated behavior that is not always understood by the testers or the developers. Therefore, it is possible that a finding or variations of the finding may still be present even if it is not reproducible during a re-test. While we will do our best to work with the organization to avoid this, it is still possible.

The findings report includes the following remediation status information:

**Resolved**

Finding is believed to be remediated, we can no longer reproduce it

Resolved Finding is believed to be remediation, we can no longer reproduce it

**In progress**

Finding is in the process of being remediated

In progress Finding is in the process of being remediated

**Unresolved**

Finding is unresolved – used in initial report or when the organization chooses not to resolve

Unresolved Finding is unresolved – used in initial report or when the organization chooses not to resolve

**Not applicable**

There is nothing to resolve, this may be the case with informational findings