# SUBGRAPH

# FINAL FINDINGS REPORT

**Hypothesis**

Feb 16, 2024

Prepared for: Hypothesis

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

# Contents

# Overview

## Summary of Activities

Hypothesis contracted an independent security assessment of several components that comprise the Hypothesis service. These included:

- The *h* application backend
- The LMS integration
- The via Proxy
- Checkmate
- The Hypothesis client

The following classes of vulnerabilities were in scope for testing:

- Injection ("cross-site scripting", "SQL injection", and all other related attacks)
- Business logic
- Trust relationships between application components
- Authentication and authorization enforcement
- Configuration
- Use of cryptography by the application

Functional security testing of the *h* application was performed in the following manner:

Subgraph interacted with a non-production instance deployed within their multi-tenant environment. Testing of the *h* application was from the perspective of a user of the public Hypothesis service. This functional testing of the live application simulated an adversarial user account without administrative privileges. The public source code of the *h* application was also reviewed to cover key areas of the total attack surface.

Functional security testing of the *lms* integration was performed in the following manner:

The LMS integration was tested through a configuration that was established between the Hypothesis Beta environment and a LMS platform. Subgraph performed testing simulating a user interacting with the Hypothesis backend as an authenticated user originating from within the LMS. The testing simulated an adversarial attacker originating in the LMS integrated with the Hypothesis infrastructure. Elements of the public source code of the LMS integration was also reviewed.

The via proxy was tested through the application, simulating a malicious user attempting to circumvent the security controls implemented in the proxy.

Functional testing of the *viahtml* proxy, the *client* components, and *checkmate* was performed through the *h* application and the *lms* integration, from within the LMS.

## Hosts Tested

- https://qa-lms-via.hypothes.is/
- https://qa-checkmate.hypothes.is
- https://qa-via.hypothes.is
- https://qa-viahtml.hypothes.is
- https://qa-web.hypothes.is
- https://qa.hypothes.is

## Tools Used

- Burpsuite Pro
- Firefox

## Code Review

Tactical auditing of source code was performed. This means that, rather than a full code review, key areas: security boundaries, authentication, input validation, credential handling, etc, were examined in source code. The public commit logs were also surveyed.

Third-party dependencies were scanned for vulnerabilities or issues with dependency management using automated tools.

## Artifacts Examined

The *h* application source tree at commit 323c11eb08d6777669651fcf0fce04a1341f4043.

The *checkmate* source tree at commit d8826ce2cb0e82a98e1725a1e65bea2bc74c7b58.

The *client* source tree at commit f23ae27dc527489da5d480de9446c57196879756.

The *viahtml* source tree at commit f9d1565c357ce08e63756ec4e885db9edae733d9.

The *lms* source tree at commit fb4db8b687b448f03a152476ed3384f69b3093eb

## Infrastructure

A control plane audit of the the AWS environment in which the Hypothesis application was deployed. For this a role was created for Subgraph with the built-in SecurityAudit managed policy permissions.

## Duration

The duration of the engagement was approximately 10 days.

## Observations on Authentication

The Hypothesis public application accepts a username and password as authentication credentials. Upon successful authentication, cookies are set for both authentication and the session. Cookies are set with the *HttpOnly* flag, but not the *secure* flag. Not setting the *secure* flag on authentication/session cookies is highlighted in this report as a finding (V-002), however modern browser defenses heavily mitigate the risk.

Username enumeration is possible, i.e., the application does inform clients attempting to login that a provided username exists or not. This would typically be reported as an issue however usernames are revealed quite openly within the application.

Request intent is authenticated using a CSRF token. Subgraph observed that the CSRF token is validated.

## Observations on Input Validation

Subgraph tested for both SQL injection and content injection, such as cross-site scripting. The *h* application uses an ORM abstraction library (Sqlalchemy) for database queries and does so very consistently. Numerous tests were also performed to test for HTML and script injection. No instances were identified.

## Observations on Authentication

The *h* backend sets a session cookie on successful authentication. Attempts to connect without this cookie cause a redirect to the login page. This appears to function as designed.

Hypothesis when integrated with an LMS (learning management system) has a more complicated authentication system. Hypothesis supports LTI 1.1 and 1.3. One of the key differences in these versions is that LMS 1.3 uses OAUTH2/OIDC with JWTs for message signing. LTI 1.1 uses OAUTH 1.0a. Testing of a live deployment used an LMS integrated with the LTI 1.1 adapter. Tactical code review was performed on some of the implementation code as it was not possible to fully simulate the role of an LMS (or that of Hypothesis) due to limitations in the configuration flexibility in the testing environment. Some of the potential complexity was simplified in the functional testing scenario, and tactical code review was applied to compensate, though even this had some limitations in total code coverage.

Password hashing uses bcrypt with 12 rounds. This is the current default value for the implementation that is used by *h* and is considered a reasonable standard. The number of "rounds" is the computational cost to produce a hash from a source string. The default value raises the effort cost of a password cracking attack, however the value of this is ultimately undermined by the application permitting very weak passwords: see V-001.

The *h* backend also supports OAuth for authentication. This is important for supporting LTI, which partially uses OAuth 1.0a for LTI 1.1 and OAuth 2.0 and OIDC for LTI 1.3. Functional testing and tactical code review was employed to identify possible vulnerabilities. One issue that was deemed essentially theoretical (and was not verified with any functional testing) was identified in the code. This issue is described in V-004.

TLS provides endpoint authentication for network interfaces of the system. The service did not appear to be usable over HTTP. Requests sent to the application over HTTP result in a redirect to the HTTPS service. The application server endpoint also leverages HSTS by sending a *Strict-Transport-Security* response header that will informs clients to upgrade connections to TLS. The *max-age* value supplied is 6 months, which is a reasonable value.

## Observations on Cloud Infrastructure

Subgraph performed an audit of the AWS configuration state. Noteworthy observations include the following:

- The password policy does not expire passwords
- There were two access keys found that do not appear to have been used in the past 90 days.
- A console user account that does not appear to have been used in the past 90 days.

None of these issues are necessarily vulnerabilities. They all introduce some risk of unused credentials that could eventually be compromised or leaked. Subgraph recommends periodic reevaluation of privileged access.

## Observations on Authorization

The typical *h* user does not exist in a context where there is complex role-based access control. The testing performed by Subgraph explored enforcement within the multitenant application and did not identify and breaches of user authorization.

Numerous attempts were made to access URLs that should not be accessible through the via proxy. These included addresses such as the loopback interface, non-routeable internal addresses, the address for the EC2 instance metadata service, IPV6 loopback, and others. The URL validation in place did not permit access to any addresses of this type.

Wombat is a Javascript framework that is injected via the via proxy. Wombat attempts to safely enable scripting and AJAX in pages that are rendered through the proxy. Subgraph did not identify a way to break the wombat framework within the duration of the engagement, however it may be possible. Subgraph believes there is some residual risk in this component that could allow for malicious pages to intefere with the operation of the annotation client that overlays proxied pages. The URL authorization mitigates the risk of this, as does the use of the browser-plugin client.

## Observations on Intrusion Prevention

The public Hypothesis service uses Cloudflare as a reverse proxy that provides additional security services. On numerous occasions during testing the Cloudflare proxy detected and prevented what it determined to be malicious requests: these included known attack strings, as well as more general patterns such as a high volume of repeated requests from a single source address.

## Observations on Third-Party Dependency Management

Automated dependency audits were executed for all Python-based components. No critically vulnerable dependencies were identified. Additionally, commit history was reviewed for components in scope. Subgraph documented use of Dependabot to track updates of dependencies, as well as integration of updates on a regular basis.

Here is one example of a security vulnerability in a dependency within *client* being addressed through tracking and follow-up (merges):

Merged update that addresses CVE-2023-32681: Bump requests from 2.28.1 to 2.31.0 in /requirements #5583

## Summary

| No. | Title | Severity | CVSS | Remediation |
|---|---|---|---|---|
| V-001 | Minimum Password Length Too Low | Medium | 6.3 | Resolved |
| V-002 | Cookies Set Without Secure Flag | Medium | 5.0 | Resolved |
| V-003 | Stale AWS Access Keys | Medium | 5.0 | Resolved |
| V-004 | Potential OAuth Authentication Bypass | Low | 5.0 | Resolved |

## Details

### V-001: Minimum Password Length Too Low

| Severity | Remediation | CVSS Score | CVSS Vector |
|---|---|---|---|
| Medium | Resolved | 6.3 | CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:L/A:L |

#### Discussion

*Update: This was confirmed fixed in the source code.*

The minimum password length for users of *h* who create accounts is 2 characters. This issue was discovered in the source code and verified in the deployed application through the creation of an account with a 2 character password. The source code indicates that the developers were at some point aware of this issue but have not implemented a fix to date.

The issue can be seen in the public repository here:

```
PASSWORD_MIN_LENGTH = 2  # FIXME: this is ridiculous
```

Note that this issue affects the *h* service when deployed using its own internal identity store, as is the case with the public service. Deployments that use external identity providers would not be affected, as those organizations can and will have their own password and authentication policies.

#### Impact Analysis

Users of the *h* application are not barred from creating accounts with extremely weak passwords. Since usernames are trivial to gather due to visibility as public annotators, attacks against users who have opted for simple passwords seem plausible.

#### Remediation Recommendations

This issue will require a change to the code to implement greater minimum complexity.

Beyond this, there is the issue of users who currently have very low complexity passwords. Those users could be identified through a password strength exercise and then encouraged or forced to do a password change upon login. Something slightly similar was done in the past when the password hashing mechanism was improved, as evidenced by this check, though it would be a little more challenging as user interaction could be required, and the information that a user is affected would need to be stored somewhere.

## Additional Information

N/A

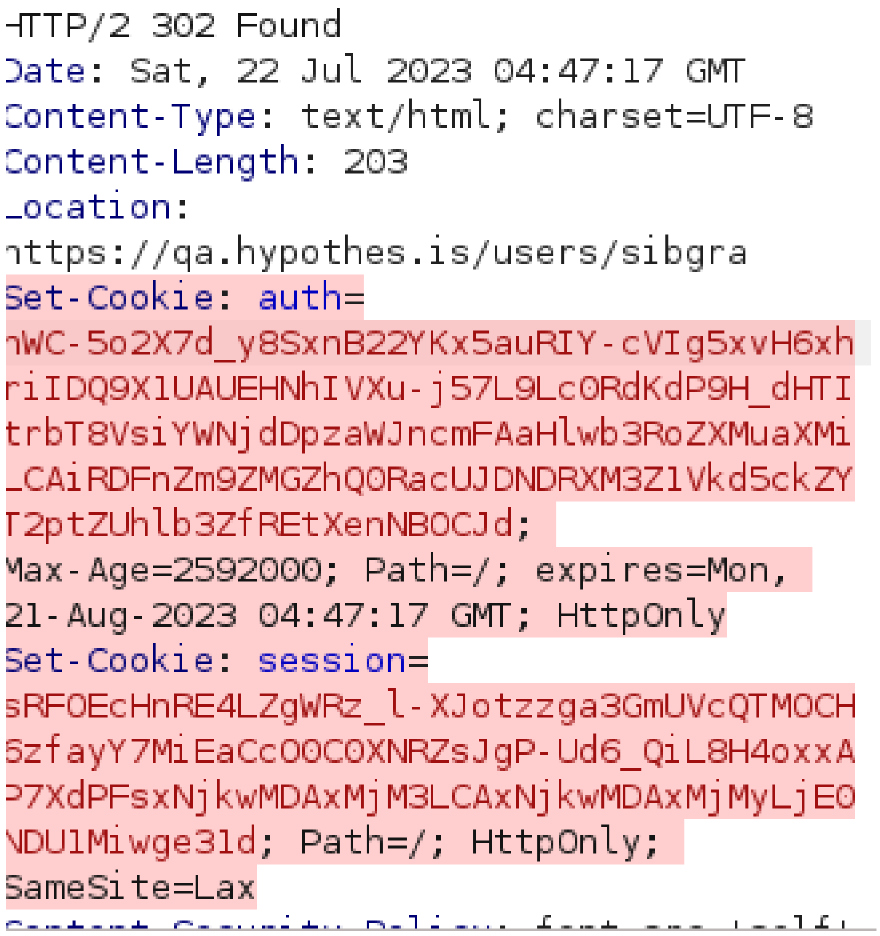
## V-002: Cookies Set Without Secure Flag

| Severity | Remediation | CVSS Score | CVSS Vector |
|---|---|---|---|
| Medium | Resolved | 5.0 | CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:L/I:L/A:L |

### Discussion

*Update: Confirmed a fix in the repo commit history.*

Cookies set during the login process are not set with the *Secure* flag. This means that the browser may send them when making HTTP requests under some specific circumstances, with older browsers being more exposed. The Pyramid Python framework apparently does not set the *secure* flag by default, which may be the root cause of this issue.



```
HTTP/2 302 Found
Date: Sat, 22 Jul 2023 04:47:17 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 203
Location:
https://qa.hypothes.is/users/sibgra
Set-Cookie: auth=
nWC-5o2X7d_y8SxnB22YKx5auRIY-cVIg5xvH6xh
riIDQ9X1UAUEHNhIVXu-j57L9Lc0RdKdP9H_dHTI
trbT8VsiYWNjdDpzaWJncmFAaHlwb3RoZXMuaXMi
LCAiRDFnZm9ZMGZhQ0RacUJDNDRXM3Z1Vkd5ckZY
T2ptZUhlb3ZfREtXenNBOCJd;
Max-Age=2592000; Path=/; expires=Mon,
21-Aug-2023 04:47:17 GMT; HttpOnly
Set-Cookie: session=
sRFOEcHnRE4LZgWRz_l-XJotzzga3GmUVcQTMOCH
6zfayY7MiEaCcO0C0XNRZsJgP-Ud6_QiL8H4oxxA
P7XdPFsxNjkwMDAxMjM3LCAxNjkwMDAxMjMyLjE0
NDU1Miwge31d; Path=/; HttpOnly;
SameSite=Lax
Content-Security-Policy: font-src 'self'
```

### Impact Analysis

This issue is mitigated by the following:

- Modern browsers will not send credentials by default unless the originating request is from a page that matches the origin
- The application server sends a *Strict-Transport-Security* header with a max-age of 6 months and instructs the browser to include subdomains

This reduces the opportunities for exploitation, but does not eliminate them.

### Remediation Recommendations

Add the secure flag to cookies that are set following a successful login. For applications that use the Pyramid framework, this is documented here:

- pyramid.session

### Additional Information

N/A

## V-003: Stale AWS Access Keys

| Severity | Remediation | CVSS Score | CVSS Vector |
|---|---|---|---|
| Medium | Resolved | 5.0 | CVSS:3.0/AV:N/AC:H/PR:L/UI:N/S:U/C:L/I:L/A:L |

### Discussion

*Update: This was confirmed resolved in a more recent cloud security scan.*

Two AWS access keys were identified as not having been used in the past 90 days:

- lms-via
- pg-snapshot-writer

### Impact Analysis

Stale access keys can introduce risk if they are disclosed, which may have a higher probability due to their not being used yet still remaining valid, usable credentials. Occasionally such credentials remain in source code or deployment scripts for an indefinite period of time, unnoticed. Long-lived unused credentials are also potentially indicative of weak or ad-hoc management of privileged access. However, the risk is entirely based on circumstances that were not apparent at the time of testing. Their being unused may not be the case, or the credentials may not have meaningful permissions.

### Remediation Recommendations

Investigate the purpose and need for these access keys and then reevaluate whether or not they should remain active.

### Additional Information

N/A

## V-004: Potential OAuth Authentication Bypass

| Severity | Remediation | CVSS Score | CVSS Vector |
|---|---|---|---|
| Low | Resolved | 5.0 | CVSS:3.0/AV:N/AC:H/PR:N/UI:R/S:U/C:L/I:L/A:L |

### Discussion

*Update: This issue was corrected in the source code. The function now explicitly fails in the edge case where this vulnerability may be present.*

*Note: Strict CVSS3 rating results in a score in the medium range, however Subgraph has downgraded this to low given the low probability of exploitability.*

There is a *potential* theoretical vulnerability in the implementation of OAuth client authentication.

The potentially problematic code is located here:

```
provided_secret = request.client_secret
if request.client_secret is None:
    # hmac.compare_digest raises when one value is `None`
    provided_secret = ""

if not hmac.compare_digest(client.secret, provided_secret):
    return False

request.client = Client(client)
return True
```

The above code is from a function that authenticates an OAuth client. The snippet covers the case where *client_secret* evaluates (for some reason) to *None*; in that case the empty string value is assigned. The reason for this is because *hmac.compare_digest()* will throw an exception if one of the two values being compared is *None*. From a security perspective, this codes creates a theoretical possibility that, if the client_secret is *None*, then authenticated messages could be forged as the secret falls back to a known, hardcoded value. The right approach may be to fail the validation if the value of the *client_secret* is *None*. This wasn't tested or reproduced, so it is not know how or when, if ever, this issue could manifest. It may only be possible in circumstances where there is major human error.

### Impact Analysis

This is a low risk issue. In order for this to be exploitable, the *client_secret* must evaluate to the Python constant *None*. The circumstances in which this could occur are not fully understood, and may never practically exist, though, if there are any such circumstances, it is possible that an adversary could forge e.g. launch requests.

### Remediation Recommendations

The risk of this function should be more deeply understood, and, if warranted, more robust error handling for this case should be implemented.

### 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