Example Security Assessment Report

Author: Collin Berman
Email: cyberman@collinberman.com
Date: 2024-05-08

Summary

This example report was generated by taking an actual report delivered to a customer and removing any reference to the customer name.

xxxxxxxx contracted Cyberman for preliminary security testing of the web application https://app.xxxxxxxx.com and associated API. 8 person-hours of manual testing was conducting looking for vulnerabilities and assessing risk. The application was tested for common security vulnerabilities including the OWASP Top 10, with a focus on user access controls and data confidentiality.

The xxxxxxxx application is protected against many common threats to web applications. For example, the use of "Log in with Google" and "Log in with email" passwordless features eliminate entire classes of vulnerabilities related to password attacks. Moreover, the application is not vulnerable to Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Insecure Direct Object Reference (IDOR), or other vulnerabilities that generally expose customer accounts and data to unauthorized Internet attackers.

In fact, all identified vulnerabilities impacting data confidentiality require some advanced conditions in order to exploit. For example, the only medium severity vulnerability (with a CVSS v4.0 score of 4.8) is data exposed in the local browser cache, which is only accessible locally on a target's machine. The low severity vulnerabilities listed in this report, while exploitable, either only expose limited information or require significant cooperation on the part of a victim user.

As a health care clearinghouse, xxxxxxxx is required under the HIPAA Security Rule to "ensure the confidentiality, integrity, and availability (CIA) of all electronic protected health information (ePHI) [xxxxxxxx] creates, receives, maintains, or transmits" and "protect against any reasonably anticipated threats or hazards to the security or integrity of such information" (45 CFR 164.306). Certain vulnerabilities detailed in this report compromise the confidentiality of some ePHI such as client names. By remediating these vulnerabilities, xxxxxxxx can show it has taken concrete actions to ensure the security of its information.

Testing occurred from 3/18/24 to 3/27/24 and this report represents the security of the application during this time period. Any modifications to the application after this time may affect the vulnerabilities listed here or introduce additional vulnerabilities. In particular, features to support and manage teams of users were not enabled at the time of testing.

Findings

Medium Severity
• Sensitive data cached by browsers

Low Severity
• Cross-site Leaks
• Clickjacking
• Application session cookies shared with 3rd-party
• User email address enumeration
• Login CSRF
• HTTP Content / Missing HSTS
• Session Fixation

Informational
• 14-day session timeout
• Debug log stored in GitHub
• Fragile CSRF protections

Finding Details

Sensitive data cached by browsers (Medium)

CVSS v4:
Base Score: 4.8
Vector String: CVSS:4.0/AV:L/AC:L/AT:N/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description:
• Legitimate use of the web application by authorized users results in the storage of sensitive data in the browsers cache. Cached web responses are stored locally even after the user logs out of the web application.

Impact:
• Attackers with local access to the machine may be able to read recently cached data. The data available in the cache includes ePHI such as patient identifiers and test scores.
• Cache implementations vary across web browsers and operating systems: for example, recent versions of Safari on macOS require administrator access in order to access the local browser cache.

Recommendations:
• The web application should instruct browsers not to cache sensitive information. All HTTP responses containing sensitive information (e.g., https://api.xxxxxxxx.com/api/v1/team-member/reports/list) should include the following response header: `Cache-Control: no-store`.

Steps to Reproduce:
1. Log into the xxxxxxxx web application.
2. If there are no reports for the logged-in user, create a new report.
3. Log out of the web application.
4. In a terminal, navigate to the local cache directory (the cache location varies by web browser and operating system).
5. Run the following shell command to identify sensitive xxxxxxxx data stored in the cache: `grep -R '{"reports":\[{' .`.
6. View the files listed in the output of the previous command. Observe patient identifiers and test scores in the file contents.

Resources:
• CWE-525
• mdn web docs: Cache-Control

Evidence:

(Image removed for client anonymity)

Screenshot Evidence (Chrome)

(Image removed for client anonymity)
Screenshot Evidence (Edge)

(Image removed for client anonymity)
Screenshot Evidence (Firefox)

Cross-site Leaks (Low)

CVSS v4:
Base Score: 2.3
Vector String: CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description:
• Cross-site leaks (XS-leaks) are a type of vulnerability that allows an attacker to infer information about legitimate users. In this case, an attacker can infer whether a user is currently logged into the xxxxxxxx web application if the user browsers to an attacker-controlled website.

Impact:
• Attackers can leverage XS-leaks to identify xxxxxxxx customers as the first stage in an attack before further attempts at exploitation.
* Chromium-based browsers (such as Chrome and Edge) automatically prevent some kinds of XS-leaks by treating cookies as `SameSite=Lax` by default.

Recommendations:
• Set the `xxxxxxxx` session cookie with the `SameSite` attribute set to `Lax` (or `Strict`, however this may negatively impact user experience when navigating to xxxxxxxx from a 3rd-party context by asking users to re-authenticate).
• Set the `Cross-Origin-Opener-Policy` (COOP) HTTP response header with a value of `same-origin` (or `same-origin-allow-popups` if the web application uses the `window.open` function to open another website and communicate with it) on `app.xxxxxxxx.com`. Note that `api.xxxxxxxx.com` already sets the COOP header with a value of `same-site`.
• Set a restrictive `X-Frame-Options` policy and `frame-ancestors` Content Security Policy (CSP) directive as described under the following clickjacking vulnerability.

Steps to Reproduce:
1. Log into the xxxxxxxx web application.
2. In a new browser tab, navigate to a different site, e.g., `https://example.com`.
3. In the new browser tab, open the browser's developer tools and switch to the console pane.
4. Run the following command to open xxxxxxxx in a new tab and maintain a JavaScript reference to the window: `w = window.open('https://app.xxxxxxxx.com/reports')`
5. Run the following command to bring the tab opened in the previous step to a first-party context in order to enable the next step: `w.location = "https://example.com"`
6. Run the following command to count the number of redirects performed by the previous two steps and remember this value for later reference (at the time of testing, this command logged 2): `console.log(w.history.length)`
7. Now, log out of the xxxxxxxx web application.
8. In the browser tab with the developer console open, repeat steps 4 through 6 .
9. Observe that the value logged is now one greater, indicating that an additional redirect back to the login page is performed by the xxxxxxxx application when the user is not logged in (at the time of testing, this value was 3).

Resources:

• XS-Leaks Wiki: Navigations
• Same Site Cookies Explained from Chrome Developer Relations

Evidence:

(Image removed for client anonymity)
Screenshot Evidence (logged in)

(Image removed for client anonymity)
Screenshot Evidence (logged out)

Clickjacking (Low)

CVSS v4:
Base Score: 2.1
Vector String: CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:A/VC:L/VI:L/VA:L/SC:N/SI:N/SA:N

Description:
• Clickjacking attacks embed victim websites and overlay seemingly-innocuous content (e.g., a game) in order to try to induce a victim to perform a series of actions on the victim page.
• For example, if you click in the top left corner of the webpage in the browser and drag to the bottom right, scrolling, you will select most of the text on the page. Tricking a victim user into performing this action in their browser, along with a copy and paste command sequence could compromise the sensitive information within the page's contents.

Impact:
• In the case of the xxxxxxxx web application tested at this time, sensitive data available within the web UI is limited* to report titles, client names, and dates, which qualify as ePHI. The ability for a user to edit or delete their existing reports does not exist in the web UI, such that clickjacking does not impact data integrity or availability.

*Perhaps we can imagine tricking a user into right-clicking a report link and selecting copy. Fortunately, xxxxxxxx restricts Google Docs access by default to only allow authorized users.

Recommendations:
• Set the `X-Frame-Options` HTTP response header with a value of `DENY` or `SAMEORIGIN` on all responses from the web application.
• Set the `Content-Security-Policy` HTTP response header with a value of `frame-ancestors 'none';` or `frame-ancestors: 'self'`.

Steps to Reproduce:
1. Login to https://app.xxxxxxxx.com/login.
2. Browse to https://app.xxxxxxxx.com/reports.
3. Open the browser's Developer Console. Select the Network tab.
4. Refresh the page.
5. Scrolling up in the Network pane, select the request to https://app.xxxxxxxx.com/reports.
6. Scroll down to view the HTTP Response Headers.
7. Observe that there is no `X-Frame-Options` or `Content-Security-Policy` header present.

Resources:
• OWASP Clickjacking Defense Cheat Sheet
• mdn web docs: X-Frame-Options
• mdn web docs: Content-Security-Policy/frame-ancestors

Evidence:

(Image removed for client anonymity)
Screenshot Evidence (Safari)

(Image removed for client anonymity)
Screenshot Evidence (HTTP Response Headers)

Application session cookies shared with 3rd-party (Low)

CVSS v4:
Base Score: 2.1
Vector String: CVSS:4.0/AV:A/AC:L/AT:P/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description:
• xxxxxxxx users' session cookies -- sensitive pieces of data that are used to authorize user actions -- are inadvertently shared with the third party Softr. This happens because `api.xxxxxxxx.com` sets the cookie with the attribute `domain=.xxxxxxxx.com` set for permissive sharing with subdomains of `xxxxxxxx.com`. When a logged-in user then browses to `xxxxxxxx.com` (or `www.xxxxxxxx.com`), their browser automatically sends the session cookie in the HTTP request.

Impact:
• With a user's session cookie, an attacker is authorized to perform any action on behalf of the user. At the time of testing, sensitive actions are limited to listing report titles, client names, and dates, which are considered ePHI.
• The types of attackers who might obtain session cookies via this vulnerability could include IT staff at Softr, malicious insiders or hackers within Softr, or general Internet users after a Softr data breach. It is worth noting that Softr does not appear to be HIPAA compliant.
• Because session cookies expire after 14 days, the window of opportunity for an attacker to exploit this vulnerability is limited. This vulnerability does not allow for mass exploitation of all users, but only those that have logged in within the last 14 days.

Recommendations:
• Setting a cookie's `domain` attribute instructs web browsers to include the cookie in requests to any subdomain of the domain it was set on. This can be useful functionality for creating applications made up of multiple services with different domains. However, care should be taken to avoid selecting a domain that has untrusted subdomains.
• Rearranging subdomains can remediate the vulnerability. For example, moving the API from `api.xxxxxxxx.com` to `api.app.xxxxxxxx.com` would allow for setting `domain=.app.xxxxxxxx.com` on the session cookie in order to avoid sharing the cookie with `xxxxxxxx.com`.
• This vulnerability can be completely remediated by restricting the session cookie so that it is not shared with other subdomains. This is done by not setting the `domain` attribute. This approach requires re-architecting the application so that `app.xxxxxxxx.com` only serves static content, that is, its behavior does not depend on whether users have a valid session cookie.

Steps to Reproduce:
1. Log into https://app.xxxxxxxx.com/login
2. Browse to https://xxxxxxxx.com, which is hosted by Softr.
3. Open the browser's developer console and switch to the Network tab.
4. Refresh the page.
5. Click the first request in the list (you may have to scroll up) for www.xxxxxxxx.com.
6. Scroll down in the Headers pane to view the request headers.
7. Observe that the `Cookie` header contains the `xxxxxxxx` session cookie.

Resources:
• mdn web docs: Using HTTP cookies/domain attribute

Evidence:

(Image removed for client anonymity)
Screenshot Evidence

User email address enumeration (Low)

CVSS v4:
Base Score: 0
Vector String: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N

Description:
• The API method `/api/v1/public/get-user-by-slug` allows unauthenticated users to check whether a certain user `slug` is registered on the xxxxxxxx application. By default, a user's `slug` is a sanitized form of the user's email address, so that an attacker can likely guess a user's `slug`. Using the API without authentication does not require a CAPTCHA.

Impact:
• An attacker may be able to exploit this vulnerability in order to identify valid users to target further.

Recommendations:
• Require authentication for this API. Add authorization checks so that a user is only allowed to retrieve data about themself or other users in their team. Retrieving data about arbitrary users should be an action reserved for administrator roles.
• Add rate limiting for this API endpoint if it needs to be called by unauthenticated clients.

Steps to Reproduce:
• Run the following Bash command in a terminal:

```
curl -X POST -H 'Content-Type: application/json' -d '{"slug":"collinberman-1-a-2-gmail-com"}' 'https://api.xxxxxxxx.com/api/v1/public/get-user-by-slug'
```

Evidence:

(Image removed for client anonymity)
Screenshot Evidence

Login CSRF (Low)

CVSS v4:
Base Score: 2.0
Vector String: CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description:
• Login CSRF is a type of attack where the attacker tricks a victim into logging in to the attackers account. This could be accomplished, for example, by sending the victim a phishing email with the attacker's login link. If the victim believes they are legitimately logged into their own account, they may use the attacker's account to create a new report. The attacker would then be able to download the report and be authorized as an editor for the Google Docs report.

Impact:
• After successful exploitation, an attacker would be able to access the reports that were created while the victim was logged in to the attacker's account.

Recommendations:
• A session cookie is set in a user's browser when they initiate the "Log in with email" process. When a user later logs in with the emailed link, validate that their browser has the correct session cookie. This helps ensure that the user logging in is using the same browser that was used to initiate the "Log in with email" process.
• Consider displaying a warning to users that log in with an different IP address or from a different geographical region than was used to initiate the "Log in with email" process.
• Consider selecting a user's default profile picture from a set of multiple options, so that a user is more likely to notice if they are logged into someone else's account.

Steps to Reproduce:
1. Browse to http://app.xxxxxxxx.com/login.
2. Initiate the "Log in with email" process with your user's email address.
3. Open the link that was sent to your email on a different device (e.g., your phone).
4. Observe that you are successfully logged in on the second device.

HTTP Content / Missing HSTS (Low)

CVSS v4:
Base Score: 2.1
Vector String: CVSS:4.0/AV:A/AC:H/AT:P/PR:N/UI:P/VC:L/VI:L/VA:L/SC:N/SI:N/SA:N

Description:
• The xxxxxxxx web application serves content over unencrypted HTTP, in particular the login page at http://app.xxxxxxxx.com/login is accessible without TLS/SSL encryption.

Impact:
• Accessing the web application over unencrypted HTTP can allow attackers on the same network access to sensitive data, for example, user email addresses.
• Attackers that are able to insert themselves in the network communication path may be able to alter the application's response to the user. For example, an attacker could add a password field to the login page to try to trick the user into sharing additional sensitive information.
• Such an attacker could generate a fake response to trick the user into thinking they are successfully logged in so that they will add their next report over the unencrypted HTTP channel. This impacts the integrity of report data, as well as confidentiality.

Recommendations:
• Redirect the user to use HTTPS whenever unencrypted HTTP is used to request application content.
Set an appropriate `Strict-Transport-Security` (HSTS) HTTP response header in all HTTP responses to instruct users' browsers not to access the application without encrypted HTTPS. Consider setting this header on the apex domain xxxxxxxx.com as well in order to protect against an attack during the first time a user logs in to the application. The following example value can be used for the HSTS header: `Strict-Transport-Security: max-age=63072000; includeSubDomains`

Steps to Reproduce:
1. Browse to http://app.xxxxxxxx.com/login.
2. Observe that the browser's address bar indicates you are using an insecure connection.
3. If you are redirected to an HTTPS connection, check whether your browser may be enforcing HTTPS for all connections.

Resources:
• mdn web docs: Strict-Transport-Security

Evidence:

(Image removed for client anonymity)
Screenshot Evidence

Session Fixation (Low)

CVSS v4:
Base Score: 2.1
Vector String: CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

Description:
• Session fixation is a type of attack that requires an attacker to somehow set a cookie in a victim's browser. By setting a victim's session cookie to a valid session value obtained from the application, an attacker can piggyback on the victim's authentication process. When the victim logs into the application, the attacker is also logged in as the victim by using the same cookie value.
• Setting a cookie in a victim's browser is not possible by default and requires exploiting another vulnerability. For example, a cross-site scripting (XSS) vulnerability in www.xxxxxxxx.com or another subdomain could be leveraged to successfully set a cookie of the attacker's choosing in victim browsers.

Impact:
• Attackers may be able to circumvent the "Log in with email" authentication process. This requires the presence of another vulnerability in another xxxxxxxx system, e.g., www.xxxxxxxx.com.
• The "Log in with Google" authentication process is not vulnerable, as the `/oauth2callback` endpoint sets a new session cookie.

Recommendations:
• Set a new `xxxxxxxx` session cookie in the user's browser when the log in, even if they already have a valid session cookie.

Steps to Reproduce:
1. Open a fresh browser window in Incognito mode. This is to ensure there is no stale session. This browser will be called the attacker's browser.
2. Start the "Log in with email" process for any valid email, starting a new session.
3. View the `xxxxxxxx` session cookie from the browser's developer console. Cookies are viewed in the Application tab in Chromium-based browsers and in the Storage tab in Firefox.
4. Copy the value of the `xxxxxxxx` cookie.
5. Open your other, non-Incognito mode browser. This will be known as the victim browser.
6. Browse to https://www.xxxxxxxx.com in the victim browser. Open the developer console to the cookie editor, as in Step 3. This is simulating the impact of a potential cross-site scripting vulnerability on another subdomain.
6. If there is an existing cookie named `xxxxxxxx`, edit its value to the value copied in Step 4. Otherwise, create a new cookie with the name `xxxxxxxx` and value from Step 4. Ensure the cookie's `domain` attribute is set to `.xxxxxxxx.com`.
7. In the same browser, browse to https://app.xxxxxxxx.com/. Start the "Log in with email" process with the victim's email address.
8. Complete the login process by opening the login link in the victim's browser.
9. Refresh the xxxxxxxx login page in the attacker's browser. Observe that you are logged in as the victim.

Resources:
• https://owasp.org/www-community/attacks/Session_fixation

14-day session timeout (Informational)

CVSS v4:
Base Score: 0

Description:
• Users are automatically logged out of the PugWork application after 14 days. Although this satisfies the HIPAA Security Rule's requirement for Automatic Logoff, 14 days is longer than recommended.

Impact:
• If a user does not manually logout, their session and associated xxxxxxxx session cookie will continue to be valid for 14 days.
• Leaving a workstation unattended presents a risk that can be mitigated by shortening the Automatic Logoff time.
• If an attacker compromises a session cookie, they can continue to use the session cookie to impersonate a valid user for the remainder of the 14 days.

Recommendations:
• Consider shortening the logoff timer to 30 minutes or allowing customers to set their own Automatic Logoff time.

Steps to Reproduce:
1. Log into https://app.xxxxxxxx.com/login.
2. Observe that you are still able to view and add reports over the next 14 days.

Resources:
• HIPAA Security Rule: Automatic logoff

Debug log stored in GitHub (Informational)

CVSS v4:
Base Score: 0

Description:
• While not a security issue in this case, storing logs in GitHub is not a best practice. Logs do not need version control and storing them in GitHub makes the repository more complex and harder to manage.

Impact:
• No sensitive data was exposed in this case.

Recommendations:
• Consider removing debug logs from GitHub.
• Consider a dedicated logging solution like Cloud Logging.

Steps to Reproduce:
1. Browse to https://github.com/xxxxxxxx/xxxxxxxx-saas/blob/main/saas/python-lambda/npm-debug.log

Fragile CSRF protections (Informational)

CVSS v4:
Base Score: 0

Description:
• Cross-Site Request Forgery (CSRF) attacks rely on POST requests sent from an attacker controlled website to the victim website. The xxxxxxxx web application is implemented to use a form of POST request that is not allowed to be sent cross-origin by the Same Origin Policy (SOP), effectively preventing CSRF attacks at the time of testing.

Impact:
• The xxxxxxxx web application is not vulnerable to CSRF attacks at the time of testing.

Recommendations:
• As a best practice, implement CSRF tokens, a more robust CSRF defense. Since the application is stateful, the Synchronizer Token Pattern may be used (and has popular implementations in npm).
• Set the `xxxxxxxx` session cookie to have the attribute `SameSite=Lax`. Note that this does not defend against CSRF attacks launched via a vulnerability on another `xxxxxxxx.com` subdomain (e.g., Softr-hosted `www.xxxxxxxx.com`).
• Consider adding additional lightweight CSRF mitigations similar to `Content-Type` validation, such as a Double Cookie pattern. As another simple example, add and enforce an additional required HTTP request header for all API requests, e.g., `X-CSRF-Protection=1`. These types of protection mechanisms rely on the boundary of the SOP, in particular, what qualifies as a CORS request. In general, requiring additional HTTP request headers that are not included in form requests can act as a CSRF mitigation. While not bullet-proof, adding additional layer of defense-in-depth can reduce the likelihood of a future CSRF vulnerability.

Resources:
• OWASP: Cross Site Request Forgery

TOOLS USED

Name	Description
Burp Suite	Web security testing toolkit https://portswigger.net/burp
Nmap	Network scanner used to discover hosts and ports https://nmap.org
Nikto	Website scanner used to discover common vulnerabilities https://github.com/sullo/nikto
SSLScan	A tool to enumerate supported cipher suites https://github.com/rbsec/sslscan
Dradis Framework	Security reporting framework http://dradis.com