Kicking the Rims – A Guide for Securely Writing and Auditing Chrome Extensions

Kicking the Rims – A Guide for Securely Writing and Auditing Chrome Extensions - Matthew Bryant, The Hacker Blog.

- Published: date not stated
- Original: <https://thehackerblog.com/kicking-the-rims-a-guide-for-securely-writing-and-auditing-chrome-extensions/>
- Preserved from: https://thehackerblog.com/kicking-the-rims-a-guide-for-securely-writing-and-auditing-chrome-extensions/ (stored) on 2026-08-16
- 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.

Kicking the Rims – A Guide for Securely Writing and Auditing Chrome Extensions

A Thin Layer of Chrome Extension Security Prior-Art

Chrome extension security and methodologies for auditing Chrome extensions for vulnerabilities appears to be a topic with shockingly little prior art. Especially when compared to other platforms such as Electron, which have had extension research on the topic. Searching the internet for guides and tools for auditing Chrome extensions yields very little, an academic paper written to describe Chrome's extension security model and a 2013 blog post on an example of XSS in an intentionally-vulnerable extension. Other results appear to be out-of-date, such as this Chrome extension fingerprinting guide, which no longer works for new Chrome extensions.

Of course, it's not as if security issues in Chrome extensions haven't been found or are particularly rare. One big example was the case of a Cross-site Scripting (XSS) vulnerability in the Reddit Enhancement Suite (RES) extension that allowed for wormable exploitation. For a good summary of that vulnerability, see this write up on it; the extension had 1.5 million users at the time.

This example is not even a worst case scenario, due to the fact this XSS was in a Content Script of the extension and not in a Background Page (this guide will dive into the differences). In short: vulnerabilities in the Background Pages, the pages with access to all of the privileged extension APIs, are much worse than any regular XSS. It results in the ability to abuse any of the declared API

s of the extension, such as the ability to access all sites as the victim, modify/edit browser bookmarks, history, and more. For example, the Steam Inventory Helper extension suffered from a vulnerability that resulted in arbitrary JavaScript execution in the Background Page context, resulting in the ability to hijack all of the accounts of the victim, on every website they were authenticated to.

Given the incredible popularity of the Chrome browser and its extensions, it seems that this platform is definitely deserving of a closer look into the security pitfalls that can occur. This guide attempts to outline extension security anti-patterns, as well as provide a usable service (tarnish) to aide developers and security researchers in auditing Chrome extensions.

Before diving into security anti-patterns in Chrome extensions it's important to get an understanding of how exactly these extensions are structured. To be upfront and explicit: the developers behind Chrome have put a lot of thought into extension security and insecure anti-patterns. Their architecture makes this very clear as I'll discuss below, and a lot of it is all designed with the core idea of making an environment where developers cannot easily shoot themselves in the foot. In an age where we have platforms such as Electron and NW.js, which seem intent on taking the systemic issue of Cross-site Scripting (XSS) to the desktop and turning it all into Remote Code Execution (RCE) without any safeguards; Chrome's extension environment is a solid foundation on an otherwise shaky landscape. Chrome extensions don't even have the ability to execute arbitrary system commands, but they still take extreme care to ensure that a developer has a hard time doing the wrong thing anyways.

Isolated But Talkative Worlds

A Quick Disclaimer

This section get fairly into the weeds of how Chrome extensions operate. If you're already familiar with this, then you can skip straight to the "Stealing from the Stainless, Security Anti-Patterns in the Extension World" section. Even if you already develop Chrome extensions reading this section is still likely useful as a refresher.

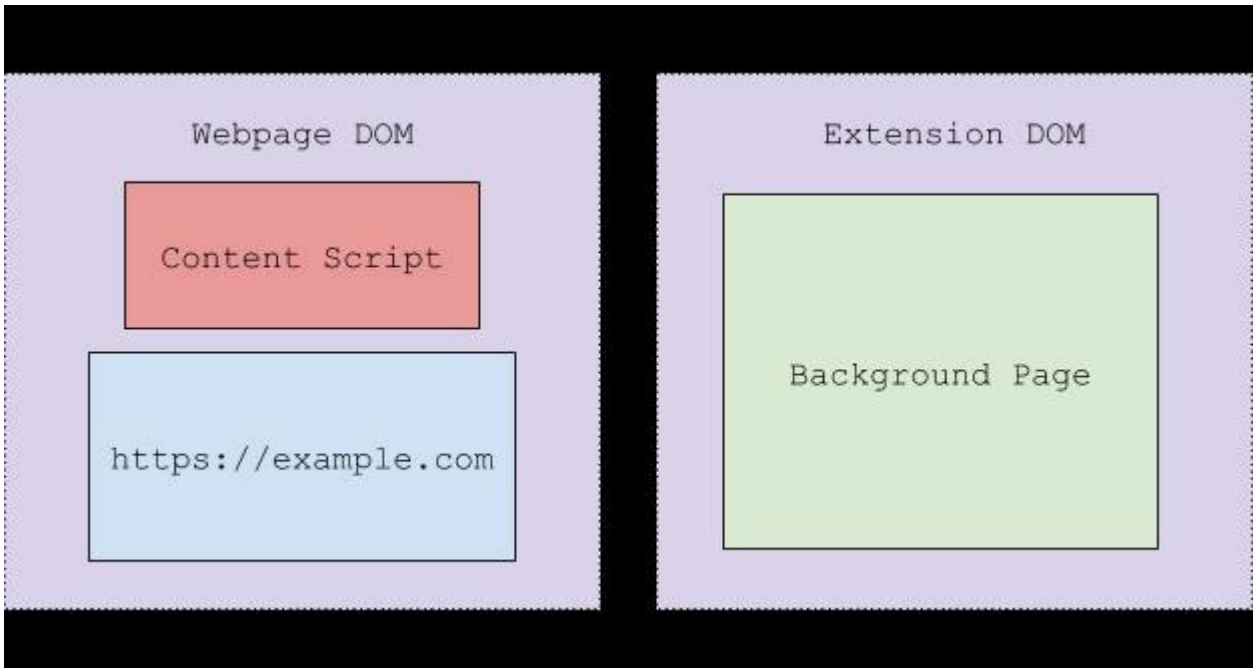
Home is Where the manifest.json Is – The Basic Extension Layout

The file structure of a Chrome extension is actually very simple. A Chrome extension is essentially just a zip folder with a file extension of .crx. The core of the extension is a *manifest.json* file in the root of the folder which specifies the layout, permissions, and other configuration options. To be blunt, understanding the manifest.json format is critical for auditing extensions for security vulnerabilities. All paths of the extension are relative to the base location where the *manifest.json* is located. So if you have an image in the root named *example.jpg*, it would be located at *chrome-extension://[EXTENSION_ID]/example.jpg* (the extension ID is a base32-encoded SHA256 hash of the Chrome extension private key).

The Extension Architecture, Namespace Isolation and the DOM

The design of how Chrome extensions work makes a large difference in how they can be exploited. Much of this is actually outlined in the academic paper I linked to earlier, but I'll dive into it here as well since the paper is a bit dated.

The Chrome extension layout can be best shown when it's visualized:



The above diagram shows the different parts of a Chrome extension. Each colored box is a separate JavaScript variable namespace. Separate variable namespaces mean that if you declared a variable in JavaScript like the following:

```
var test = 2;
```

This variable would be accessible only from its own context (the different colored boxes cannot accessible the variables of each other directly). If this was a variable in the Background Page for example, it would not be accessible from the Content Script or web page. The same goes for the variables declared by the Content Script, they cannot be accessed by either the Background Page, or the web page itself. This sandboxing prevents a rogue web page from interfering with the running Content Script(s) or the extension's Background Page, since it cannot access or change any of their variables or functions.

The Same Origin Policy (SOP) in the Chrome Extension World

This separation makes a lot of sense when you also understand how the Same Origin Policy applies for Chrome extensions. Each Chrome extension has its own origin which is the following format:

```
chrome-extension://[32_CHAR_EXTENSION_ID]
```

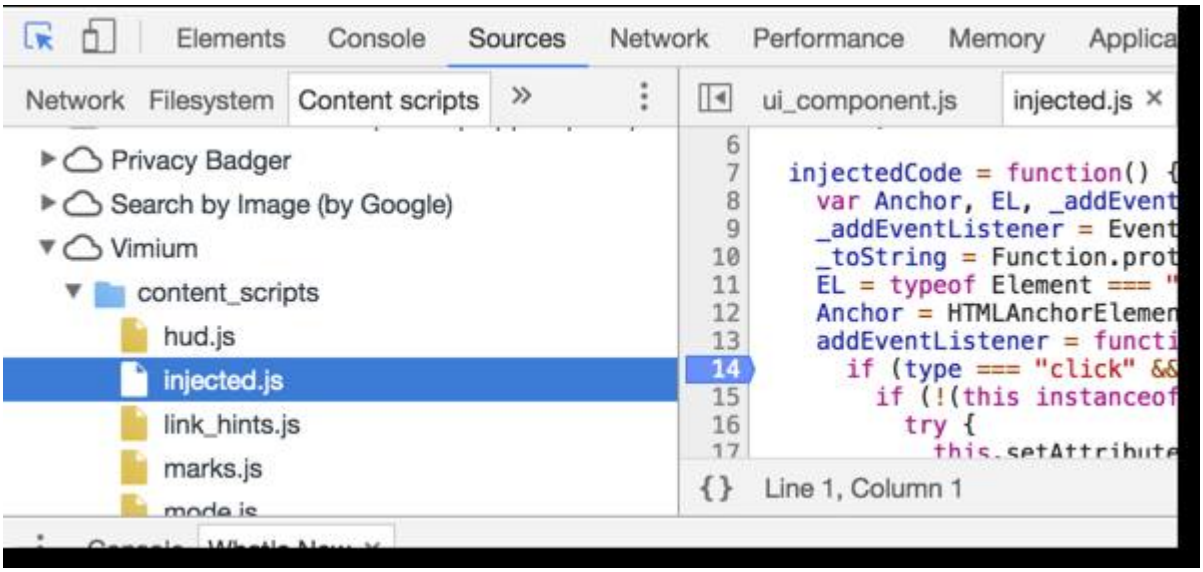
This means that any resource that falls under this origin can be accessed by the Chrome extension API. This origin structure makes sense because all of a Chrome extension's resources will be inside of the *chrome-extension://[32_CHAR_EXTENSION_ID]/* directory. This applies when we're talking about Background Pages and Browser Action pages, all of these execute within the *chrome-extension://[32_CHAR_EXTENSION_ID]* origin. Take the following example:

```
chrome-extension://[32_CHAR_EXTENSION_ID]/index.html
```

```
chrome-extension://[32_CHAR_EXTENSION_ID]/example.html
```

Both of these pages could access both the DOM and the JavaScript namespaces of each other because they have the same origin. Note that this means in the case of an access via an iframe contentWindow or window.opener. The variable namespace of each Background Page is not shared with each other in any global sort of way (except for in the case of multiple Background Page Scripts – which just end up getting globbed into a single Background Page at runtime). You can view and debug a Background Page by enabling Developer Mode in Chrome.

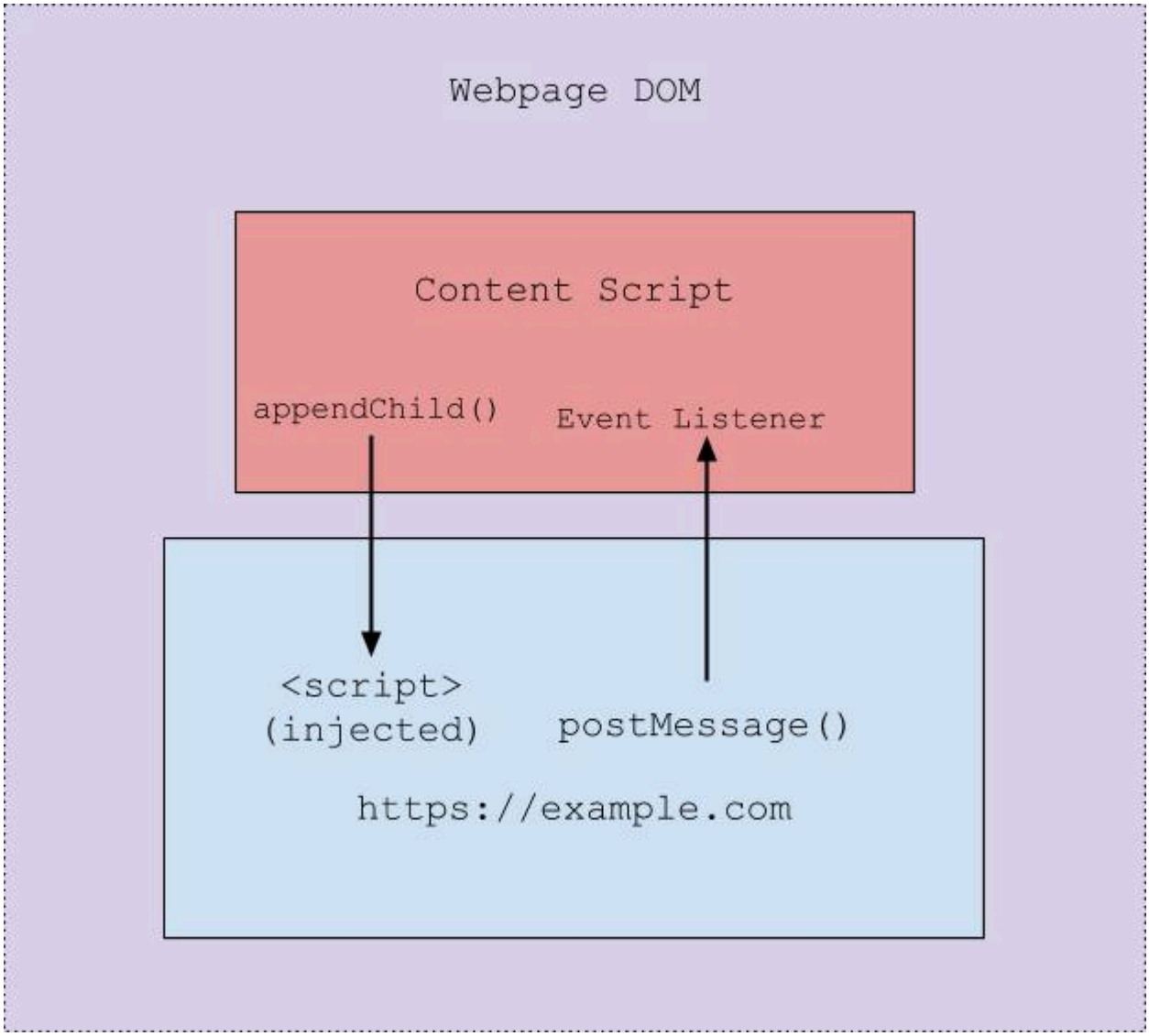
Content Scripts work a little differently, they operate in the origin of the web page they are scoped for. So if you have a Content Script running on *https://example.com* its effective origin is *https://example.com*. This means it can do things like interact with *https://example.com*'s DOM, add event listeners, and perform XMLHTTPRequests to retrieve web pages for this origin. However, it cannot modify the DOM of the corresponding extension's Background Page, because those are different origins. That being said, the Content Script does have slightly more privileges with the ability to message the Background Page, and call some limited Chrome extension APIs. This is a bit of a strange setup because it feels much like your Content Script and your web page are running in separate "pages" due to the namespace isolation, even though they still share a DOM. To view a Content Script in Chrome and debug it, you can pop open the Chrome Developer Tools menu via Options > More Tools > Developer Tools. Once the developer tools are shown, click on the "Sources" tab and click the sub-tab "Content Scripts". Here you can see the running Content Scripts for your various extensions, and can set breakpoints to monitor the execution flow:



Much of my time auditing a Chrome extension is spent in the Chrome developer panel seen above, setting breakpoints and following the execution.

Crossing the Barriers with Injection and Message Passing

However, despite the separation of namespaces there is still plenty of room for the Chrome extension to do its work. For example, say the Content Script needs to retrieve the value of a variable defined in the web page's namespace. While it cannot access the web page's namespace directly it can access the web page's DOM and inject a new script tag (<script>) into it. This injected script tag would then be executing in the web page's namespace and would have access to its variables. Upon retrieving the variable, the injected script can then pass the value back to the Content Script via postMessage(). This is what is shown by having the Content Script and the web page inside of the parent "Web page DOM" box, both have access to the web page's DOM but do not have access to eachothers namespace. The following illustration demonstrates the flow of grabbing a variable from the web page and passing it back to the Content Script:



One of the the most important things to understand in the Chrome extension auditing process is how these isolated worlds work together. The answer is (mainly) via message passing. In Chrome extensions there are a few different ways to pass messages around, such as chrome.runtime.send

Message() for messaging the Background Pages from Content Script(s), or window.addEventListener() for passing messages from web pages to Content Script(s). From a security perspective, whenever a message is passed from a lower privileged context to a higher privileged context (for example a message from a Content Script to the Background page) there exists a possible avenue for exploitation via unexpected input. It's also important to ensure that messaging is scoped properly as to not send out sensitive data to unintended origins. In the case of postMessage(), this means not using a wildcard "*" to send messages, since any webpages with a reference to the window could potentially listen in for them.

As if the namespace isolation was not enough, there is another layer of protection around Chrome extension resources. Files in a Chrome extension, by default, are not able to be iframed, sourced, or otherwise included into regular web pages on the Internet. So *https://example.com* could not iframe *chrome-extension://pflahkdjlekaeehbenhpkpipgkbbdbbo/test.html* for example. However, Chrome's extension API does allow you to loosen this restriction by declaring these resources via the web_accessible_resources directive in the extension's manifest. This is useful in cases where you want to allow a regular web page to include JavaScript, images, or other resources from your extension. The downside to this from a security perspective is that any HTML pages that are set with this flag can now be attacked via clickjacking, passing malicious input into location.hash, unexpected messaging via postMessage(), or by iframing and running background pages in an unexpected order. For this reason it is generally dangerous for developers to wildcard a large number of their resources with this directive. Additionally, if a Chrome extension exposes any resources via this directive, an arbitrary web page on the Internet can use these sourceable resources to fingerprint that a user is running a particular extension. This has plenty of uses both in exploitation and in general web tracking.

Background Pages and Content Security Policy

The background page is the most privileged of the various worlds as it has the ability call all of the Chrome extension APIs declared in the extension's manifest. These APIs are the core of what makes extensions powerful, with the ability to do things like manage cookies, bookmarks, history, downloads, proxy settings, etc. Given the powerful nature of these pages, Chrome requires that developers have a Content Security Policy (CSP) declared with certain minimal requirements. The Chrome extension documentation states the policy is the following by default:

```
script-src 'self'; object-src 'self'
```

That's effectively true, though to be pedantic it's worth mentioning the default policy is actually the following (you can verify this yourself using the Network panel of Chrome's developer tools):

```
script-src 'self' blob: filesystem: chrome-extension-resource:; object-src 'self' blob: filesystem:;
```

The above policy is slightly more lax to allow for some general JavaScript operations, such as creating blob: URIs and interacting with the filesystem. Of course, often developers see this default policy and are annoyed by just how strict it is. This annoyance often results with developers

attempting to loosen the CSP as much as possible in order to just “have it work”. The Chrome team foresaw this danger and added in additional requirements to prevent developers from making their CSP too loose. For this reason there is no way to allow for the ‘unsafe-inline’ source in a Chrome extension CSP (hold for <script>s with nonces). This means that a developer can never make use of inline JavaScript execution similar to the following:

```
Name: <input onfocus=”example()” name=”test” />
...
<a href=”javascript:example()”>Click to start</a>
...
<script>alert(“Welcome!”)</script>
```

While this can be painful to many developers who are used to this style of web development, its security advantages cannot be understated. In doing this, Chrome has made it much harder for developers to write Cross-site Scripting (XSS) vulnerabilities in their Background Pages. From my experience of auditing quite a few Chrome extensions this has been the only mitigating factor in an otherwise completely exploitable vulnerability. Additionally, it’s worth mentioning that this requirement often forces developers to write their extensions in a more clean way, since they have to separate their views and their core logic.

However, you can still make many of the other common mistakes you see with CSP. Developers can (and often do) add ‘unsafe-eval’ to their CSP and often wildcard CDNs, and other sources which anyone can upload and source scripts from. These things often allow for attackers to bypass all of the protections of the CSP.

Stealing from the Stainless, Security Anti-Patterns in the Extension World

Content Scripts Obey No Man...or CSP

With all of the talk around Content Security Policy (CSP) requirements for Background Pages one might get the idea that Cross-site Scripting (XSS) is dead in Chrome extensions. This is not at all the case, it has just been weighted over to the Content Script side of the extension. Content Scripts do not have to obey the CSP declared for the extension, they also don’t have to obey the CSP of the web page they’re executing under (unless they inject a <script> into the web page’s DOM). So if https://example.com has a CSP of the following:

```
script-src 'self'; object-src 'self'
```

This is completely irrelevant to the JavaScript executing in the Content Script context. It can, for example, make use of eval() all it wants, despite this being forbidden by the web page it runs in the origin of. This very often translates into developers creating Chrome extensions with Content Scripts that actually introduce new vulnerabilities into popular websites, which would otherwise be

safe. The RES XSS vulnerability that was mentioned earlier is a great example of this occurring. Despite Reddit not having a Cross-site Scripting (XSS) vulnerability, RES introduced one into the site for those with the extension installed. It did this by taking a user-controllable image title and injecting it back as HTML unsafely into Reddit's web page DOM. The result was Reddit was vulnerable to a zero-click XSS for all users of the RES extension.

When studying this reality, an anti-pattern begins to emerge. Developers writing Content Scripts have a strong chance of shooting themselves (and their users) in the foot if they perform unsafe DOM operations using user-controlled input. Extension developers should take extreme care to ensure they are not performing unsafe DOM operations using calls such as innerHTML, html(), or setting href (javascript: URIs) to name a few examples.

The Web Page DOM Cannot Be Trusted

Another common anti-pattern that Chrome extension developers will fall into is that they will trust content provided from an external web page. This manifests in multiple forms such as trusting data in the DOM, events fired from event listeners, or messages sent from the web page to a Content Script.

In the case of the DOM, an attacker can modify the layout to any format they'd like in order to exploit the Content Script's use of it. It should also be noted that any case where sensitive data is put into the DOM is accessible by attackers via a malicious web page, or via an XSS vulnerability in a trusted web page. For example, the Grammarly Chrome extension made this mistake when they put sensitive authentication tokens in the DOM of all webpages, allowing for a malicious webpage to simply extract it from the DOM and use it to authenticate to their API. This is perhaps more common when an extension is putting some sort of UI into a web page for a user to view. Often times in these UI elements developers will put sensitive information which should not be let outside of the Chrome extension at all. Even worse, often these elements are later queried again by the Content Script and use to perform trusted operations. These patterns allow for an attacker to swoop in the middle of these actions and modify the DOM elements to contain unexpected input.

JavaScript DOM Events Must Be Verified

Event listeners, one of the primary channels between Content Scripts and the web page DOM are also subject to exploitation by attackers. These are especially deceptive because developers expect events to be generated purely from user actions and not synthetically created by an attacker. Take a script such as the following:

```
// Create an element to inject
var injected_last_key_element = document.createElement( "div" );
injected_last_key_element.innerHTML = 'Last character typed: <div id="last_keypress_div"></div>';
// Inject this box into the page
document.body.appendChild(
   injected_last_key_element
);

// Listen for keydown event and show it's value in the previously-injected div
document.body.addEventListener( "keydown", function( keyevent ) {
   document.getElementById( "last_keypress_div" ).innerHTML = keyevent.code;
});
```

The above code will listen for keydown events and will display the last key code value inside of a injected into the document's body. So if you pressed the "a" key on your keyboard, the string "KeyA" would appear inside of this injected . The following is a screenshot from MDN documentation on KeyboardEvent:

[[image: keyboard-event-code]](https://thehackerblog.com/wp-content/uploads/2018/06/keyboard-event-code.png)The page on KeyboardEvent.code itself states the following:

"*The KeyboardEvent.code property represents a physical key on the keyboard (as opposed to the character generated by pressing the key). In other words, this property returns a value which isn't altered by keyboard layout or the state of the modifier keys.*"

Reading this documentation, as a developer you might think that XSS is not possible here. After all, this will only display the last key code pressed by the user's keyboard, and the documentation says the property is "Read Only"! How could that be used to cause XSS? Even if you could send synthetic events, it would just be the predefined key codes, and the would be rewritten on each event, right?

However, you'd be completely wrong, this is actually easily exploitable. The following code demonstrates code which would result in an XSS:

```
// Generate a synthetic key event
function generate_fake_key_event(target_element, key_event_type, custom_properties) {
  var new_event = document.createEvent(
    "KeyboardEvent",
  );
  for (var property_keyname in custom_properties)
    if (custom_properties.hasOwnProperty(property_keyname)) {
      // Chromium Hack
      Object.defineProperty(new_event, property_keyname, {
        get: function() {
          return custom_properties[property_keyname];
        }
      });
      new_event.initKeyboardEvent(
        key_event_type,
        true,
        true,
        document.defaultView,
        false,
        false,
        false,
        false,
        0,
        0
      );
      new_event[property_keyname] = custom_properties[property_keyname];
    }
  }
  target_element.dispatchEvent(
    new_event
  );
}
// Send a keydown with a code of <img src=x onerror=alert('XSS') />
generate_fake_key_event(
  document.body,
  "keydown",
  {
    "code": "<img src=x onerror=alert('XSS') />",
  }
)
```

The above code results in the XSS firing, and an alert with the text “XSS” to be displayed:

The above example demonstrates a few important pitfalls of trusting arbitrary events:

- Events can be generated without any user interaction.
- Even if an event states something similar to “Read Only” in the documentation, this is purely to state that an authentically-created event can’t have this property modified after being generated. This doesn’t apply at all to synthetically-generated events.
- Synthetically-generated events don’t even have to follow the expected format for event property values. Even though the documentation says the “codes” are in a predefined format, this is not enforced in the case of synthetically-generated events. So an attacker can specify ** instead of something like KeyA.

This all sounds very painful, but luckily there is a simple check that can be done to verify that an event is actually user-generated. User generated events have their isTrusted property set to true, whereas script-generated events have the isTrusted property set to false. Using this check we can verify if an event was actually created by a user or is simply synthetic:

```
// Listen for keydown event and show its value in the previously-injected div
document.body.addEventListener( "keydown", function( keyevent ) {
  if( keyevent.isTrusted ) {
    document.getElementById( "last_keypress_div" ).innerHTML = keyevent.code;
  }
});
```

Now an attacker cannot send completely mangled synthetic events to us. All events we process will have to be triggered originally by the end user. While the isTrusted property is not commonly used, it is essential in the case of Content Scripts processing web page events.

This example is pretty contrived because if you have an attacker creating arbitrary events on your page you are likely already owned at the web page level, so an XSS in the web page caused by this is circular. A real world example of this would be the Background Page doing something unsafe with the event data retrieved by a Content Script, but for simplicity-sake we’ve kept it to just a demonstration of JavaScript event spoofing in a regular webpage.

Messages Sent From Web Pages Cannot Be Trusted

Yet another common pattern in Chrome extensions is the use of JavaScript’s postMessage() for message passing to call privileged Chrome extension APIs in the Background Page. Often a message is issued from a web page, received by the Content Script via an event listener, and is

then relayed to the Background Page to call some privileged Chrome APIs. This creates a direct bridge where any web page can make use of privileged Chrome extension APIs, and results in some nasty things if the developer is not checking the origin of the message.

Often, even if a developer is checking the origin of the received messages, they will implement a check similar to the following:

```
window.addEventListener( "message", function( received_message ) {
   // Check to make sure this a sub-domain of our site
   if( received_message.origin.indexOf( ".trusteddomain.com" ) !== -1 ) {
      process_message( received_message );
   }
}, false);
```

Or, if they are a fan of regex they will do something similar to the below examples:

```
received_message.match(/https:\/\/www\.trusteddomain\.com/i)
...
received_message.match(/https?:\/\/www.trusteddomain.com$/i)
...
```

Sadly all of these checks are bypassable, the following demonstrates bypasses for each:

```
// Bypassed with https://a.trusteddomain.com.attacker.com
received_message.origin.indexOf( ".trusteddomain.com" ) !== -1

// Also bypassed with https://a.trusteddomain.com.attacker.com
received_message.match(/https?:\/\/www\.trusteddomain\.com/i)

// Bypassed with https://wwwatrusteddomain.com
received_message.match(/https?:\/\/www.trusteddomain.com$/i)
```

This is almost certainly because the origin property of the received message is a string of the site's origin and not an object with parsed-out origin parts. Parsing URLs in general is well known to be pitfall-prone problem, so handing everyone a string and asking them to check it themselves is naturally going to lend itself to these issues. Perhaps in future web standards a native origin verification function could be added to give developers a more assured way to validate this behavior.

To safely validate origins of messages it is recommended to have a static list of trustable HTTPS origins. Ensuring HTTPS is important because without it, an extension can be vulnerable to man in the middle attacks. The following is an example of secure code to do this:

```
var trusted_origins = [
   "https://trusteddomain.com",
   "https://www.trusteddomain.com",
   "https://extension.trusteddomain.com"
];

if( trusted_origins.includes( received_message.origin ) ) {
   // We can trust this message came from the right place
   process_message( received_message );
}
```

This keeps the surface area small and leaves little room for error when checking to see if an origin is trusted.

Of course, many developers would prefer to simply whitelist all subdomains of their main domain so that they will not have to change the source code to add new hostnames. While I'd recommend against doing this (for reasons discussed below), safe code for doing so can be seen below:

```
// Pull out the message origin
var message_origin = received_message.origin;

// Specify your trusted domain
var base_domain = "trusteddomain.com";

if( message_origin.startsWith( "https://" ) && ( message_origin.edsWith( "."[b] + base_domain ) || message_origin === "
   // Message is HTTPS and a sub-domain, trust it.
   process_message( received_message );
}
```

The above code is straightforward, it simply checks to ensure that an origin is both HTTPS and either a sub-domain of the trusted base-domain, or just the base-domain itself.

One final, but equally important thing to remember to check is the message event source. Inbound messages have a source property which is a reference to the window which sent it. Most of the time a Content Script is making the assumption that the sender of the message is the same window which the Content Script is running on. For this reason it's important that the following simple check be added as well:

```
// Check to make sure the sender is the window we expect.
// If it's not, return immediately.
if( received_message.source !== window ) {
   return;
}
```

The above check ensures that the window which sent the message is the same one the Content Script is running on. If it is not, the script returns instantly instead of processing the message. A common mistake developers will make is to not check the source of the message and perform DOM manipulations based off the content of the received message. This means that another page could potentially send a malicious message to a target site to force an insecure DOM operation via Content Script resulting in XSS. While there are edge cases where you wouldn't want the same source window as the Content Script, the vast majority should implement this simple check.

The King Shouldn't Live Outside the Castle Walls

Now that I've shown you safe code for ensuring that something is a subdomain of your trusted domain, I want to take a moment to discuss surface area.

Often enough, you will see extensions that allow privileged Chrome extension API calls only from subdomains of a domain owned by the Chrome extension owner. This pattern is often seen as useful for developers because they can more easily change their website's code to make different calls to the Chrome extension's APIs instead of having to update the extension itself. Once you're operating on this idea, often the next one is to simply allow privileged API calls from any subdomain of your trusted base domain.

This behavior is also easy to fall into when using the externally_connectable directive. This directive states which origins can send messages to the extension's Background Page. Even in the official documentation the example provided is the following which allows for all subdomains of a base domain over either HTTP or HTTPS to send messages:

```
"externally_connectable": {
  "matches": ["*://*.example.com/*"]
}
```

All of that seems very reasonable from a development point of view. However, what the developer has effectively done is move the barrier of security from the strongly-locked-down Background Page, to any sub-domain of their domain. This means that an attacker could completely hijack these privileged calls if they have arbitrary JavaScript execution on any subdomain of the trusted domain. There are a number of ways that this can occur:

- A web page on any subdomain is vulnerable to a Cross-site Scripting (XSS) vulnerability.
- An attacker is able to get an HTML file uploaded to any subdomain containing malicious code.
- Any subdomain is vulnerable to a takeover vulnerability, due to it being pointed to an old IP address, unallocatedcloudresources, a CNAME to an expired domain name, etc.

It takes only one slip up in any of these subdomains to result in the Chrome extension becoming vulnerable. Compare this to all of the protections such as CSP, and web navigation blocking given to Chrome extension Background Pages and the stark contrast becomes clear. Why play a game rigged in the attacker's favor?

A good example of this problem can be seen in a critical vulnerability I found in the ZenMate VPN Chrome extension (at the time of this writing it has ~3.5 million users). The following is an excerpt

from their (previously) vulnerable manifest.json:

```
...trimmed for brevity...
{
  "js": [
    "scripts/page_api.js"
  ],
  "matches": [
    "*://*.zenmate.com/*",
    "*://*.zenmate.ae/*",
    "*://*.zenmate.ma/*",
    "*://*.zenmate.dk/*",
    "*://*.zenmate.at/*",
    "*://*.zenmate.ch/*",
    "*://*.zenmate.de/*",
    "*://*.zenmate.li/*",
    "*://*.zenmate.ca/*",
    "*://*.zenmate.co.uk/*",
    "*://*.zenmate.ie/*",
    "*://*.zenmate.co.nz/*",
    "*://*.zenmate.com.ar/*",
    "*://*.zenmate.cl/*",
    "*://*.zenmate.co/*",
    "*://*.zenmate.es/*",
    "*://*.zenmate.mx/*",
    "*://*.zenmate.com.pa/*",
    "*://*.zenmate.com.pe/*",
    "*://*.zenmate.com.ve/*",
    "*://*.zenmate.fi/*",
    "*://*.zenmate.fr/*",
    "*://*.zenmate.co.il/*",
    "*://*.zenmate.in/*",
    "*://*.zenmate.hu/*",
    "*://*.zenmate.co.id/*",
    "*://*.zenmate.is/*",
    "*://*.zenmate.it/*",
    "*://*.zenmate.jp/*",
    "*://*.zenmate.kr/*",
    "*://*.zenmate.lu/*",
    "*://*.zenmate.lt/*",
    "*://*.zenmate.lv/*",
    "*://*.zenmate.my/*",
    "*://*.zenmate.be/*",
    "*://*.zenmate.nl/*",
    "*://*.zenmate.pl/*",
    "*://*.zenmate.com.br/*",
```

```
    "*://*.zenmate.pt/*",
    "*://*.zenmate.ro/*",
    "*://*.zenmate.com.ru/*",
    "*://*.zenmate.se/*",
    "*://*.zenmate.sg/*",
    "*://*.zenmate.com.ph/*",
    "*://*.zenmate.com.tr/*",
    "*://*.zenmate.pk/*",
    "*://*.zenmate.vn/*",
    "*://*.zenmate.hk/*"
  ],
  "run_at": "document_start"
}
...trimmed for brevity...
```

The Content Script *page_api.js* allows for privileged API calls to the extension in order to do things like retrieve user information, whitelist sites so they're not proxied, and toggling whether or not the user is connected to the VPN. Given the above list of dozens of domains, there is a lot of surface area to potentially exploit. In order to hijack this extension we'd need an XSS on any sub-domain, on any of these dozens of domains.

However, it turns out we didn't need even that. One of the domains, zenmate.li, was expired and open for registration. After buying it and setting up a website for it, all that was needed to extract user information was to run the following payload on this whitelisted domain:

```
// Make call to Content Script to get all user data
__zm.getData(function(results) {
    console.log(
        results
    );
});

// Turn off VPN
__zm.toggle(false);
```

With this payload we can retrieve all the user's information (their authentication tokens, email address, etc), and can completely de-anonymize them effectively bypassing all the protections of the extension. While the vendor's response was prompt and this issue is now fixed this demonstrates exactly the type of critical problems that can occur when you move control of the extension outward to multiple websites and expand the surface area. For a full write-up please see this post which goes further into details.

Generally Sane Parsing of URLs

There are other situations where a developer might find themselves parsing a given URL to see if it's from a trustable origin. This can occur through usage of APIs such as chrome.tabs.get() which returns a Tab object full of metadata on the queried tab. Developers will often attempt to parse the url property of this object via a regular expression to see if it's a site that they trust. As we saw before, parsing URLs is tricky business and is very hard to get right. The core of this problem is that the code you write to pull out a URL's origin could differ from how Chrome does it internally, resulting in a bypass.

A clean way to sanely pull an origin out of a given URL is to use the URL() constructor:

```
// Input URL
var input_url = "https://trusted-domain.com/example-url/?red=blue#yellow";

// Safely check to see if a URL matches a target origin
function check_origin_match( input_url, target_origin ) {
   var parsed_url = new URL(
      input_url
   );
   return ( parsed_url.origin === target_origin );
}

if( check_origin_match( input_url, "https://trusted-domain.com" ) ) {
   // URL is from a trusted origin!
} else {
   // Not from a trusted origin.
}
```

The above code is special because it shifts the work to Chrome's URL parser to retrieve the origin from a given URL. This ensures that your code and Chrome's code agree on the actual origin of a given URL. Instead of using the *check_origin_match()* function above, you can just pass a URL to the URL constructor yourself and check the origin property using some of the provided code above. This should address cases where you need to securely check a URL's origin due to it not being already parsed out for you. The final URL object also contains useful parsed-out fields for hash, hostname, URL path, parameters, and more.

Clickjacking & Careful Use of web_accessible_resources

The web_accessible_resources directive denotes which resources such as extension pages, images, and JavaScript can be embedded by arbitrary websites. As outlined earlier, by default, arbitrary web pages cannot embed extension pages in iframes, or source them via script or stylesheet tags. Example usage of this directive can be seen below:

```
{
  ...trimmed for brevity...
  "web_accessible_resources": [
    "images/*.png",
    "style/double-rainbow.css",
    "script/double-rainbow.js",
    "script/main.js",
    "templates/*"
  ],
  ...trimmed for brevity...
}
```

As can be seen from the above example, not only can you specify specific resources but you can also wildcard a folder of resources as well. However, occasionally developers will experience an issue with Chrome blocking their ability to embed extension resources and will do something like the following:

```
{
  ...trimmed for brevity...
  "web_accessible_resources": [
    "*",
  ],
  ...trimmed for brevity...
}
```

This is a completely valid policy and essentially means that all Chrome extension resources can now be embedded in third party websites. The problem is that this turns into a clickjacking vulnerability when your extension contains pages which perform privileged actions and also fall under your web_accessible_resources policy. For a good example of an extension vulnerability caused by clickjacking see this post about a UXSS in Steam Inventory Helper.

Automating the Auditing Process With tarnish

Due to the unique structure of Chrome extensions, I decided to write a service to help developers and security researchers audit Chrome extensions for security vulnerabilities. This tool, which I've named tarnish, has the following features:

- Pulls any Chrome extension from a provided Chrome webstore link.
- **manifest.json** *viewer*: simply displays a JSON-prettified version of the extension's manifest.
- **Fingerprint Analysis**: Detection of web_accessible_resources and automatic generation of Chrome extension fingerprinting JavaScript.
- **Potential Clickjacking Analysis**: Detection of extension HTML pages with the web_accessible_resources directive set. These are potentially vulnerable to clickjacking depending on the

purpose of the pages.

- **Permission Warning(s) viewer**: which shows a list of all the Chrome permission prompt warnings which will be displayed upon a user attempting to install the extension.
- **Dangerous Function(s)**: shows the location of dangerous functions which could potentially be exploited by an attacker (e.g. functions such as innerHTML, chrome.tabs.executeScript).
- **Entry Point(s)**: shows where the extension takes in user/external input. This is useful for understanding an extension's surface area and looking for potential points to send maliciously-crafted data to the extension.
- Both the Dangerous Function(s) and Entry Point(s) scanners have the following for their generated alerts:
- Relevant code snippet and line that caused the alert.
- Description of the issue.
- A "View File" button to view the full source file containing the code.
- The path of the alerted file.
- The full Chrome extension URI of the alerted file.
- The type of file it is, such as a Background Page script, Content Script, Browser Action, etc.
- If the vulnerable line is in a JavaScript file, the paths of all of the pages where it is included as well as these page's type, and web_accessible_resource status.
- **Content Security Policy (CSP) analyzer and bypass checker**: This will point out weaknesses in your extension's CSP and will also illuminate any potential ways to bypass your CSP due to whitelisted CDNs, etc.
- **Known Vulnerable Libraries**: This uses Retire.js to check for any usage of known-vulnerable JavaScript libraries.
- Download extension and formatted versions.
- Download the original extension.
- Download a beautified version of the extension (auto prettified HTML and JavaScript).
- Automatic caching of scan results, running an extension scan will take a good amount of time the first time you run it. However the second time, assuming the extension hasn't been updated, will be almost instant due to the results being cached.
- Linkable Report URLs, easily link someone else to an extension report generated by tarnish.

All of these features have been created to automate annoying repetitive actions I've had to undertake while auditing various Chrome extensions. If you have suggestions or bugs in any of the functionality of the service, please feel free to reach out to me and I'll look into it.

Click here to try out the tarnish Chrome extension analyzer.

Matthew Bryant (mandatory)

Security researcher who needs to sleep more. Opinions expressed are solely my own and do not express the views or opinions of my employer.

"Zero-Days" Without Incident - Compromising Angular via Expired npm Publisher Email Domains

NOTE: *If you're just looking for the high level points, see the"The TL;DR Summary & High-LevelPoints

Video Downloader and Video Downloader Plus Chrome Extension Hijack Exploit - UXSS via CSP Bypass (~15.5 Million Affected)

Published on February 22, 2019

Steam, Fire, and Paste – A Story of UXSS via DOM-XSS & Clickjacking in Steam Inventory Helper

Published on June 07, 2018