Abstract

Browser extensions are a popular way for a user to customize their online experience. These extensions allow for various types of functionality including but not limited to dynamically changing page content and keeping track of their browsing history. If these functions were compromised, it would be a violation of user privacy. The postMessage mechanism is a potential avenue to exploit extensions if the origin checks guarding the postMessage are unsound. Origin checks are designed to ensure that postMessages are sourced from expected locations. A flawed origin check allows for an attacker to send messages from a URL that matches the check but is not actually from the intended sender. This type of attack has been previously studied on HTML5 websites and this paper extends the postMessage attack vector to extensions.

1 Introduction

Extensions are small programs embedded in the web browser that allow for a customizable experience. Unlike webpages, extensions are not restricted by the same origin policy. This makes sense because useful extensions must be able to interact and communicate with webpages of any origin. In popular browsers such as Chrome or Firefox, users are greeted by a list of required permissions whenever they try to install extensions, and by agreeing to the permissions, users place trust in the extension to not behave maliciously.

Some extensions betray the user’s trust not through malicious intent, but because of a lack of security awareness. The postMessage API allows for extensions in web browsers to communicate directly with web pages. If done incorrectly, this poses a problem because many extensions operate at a higher privilege than webpages. For example, a user may install an extension made by Safe Web Shopping that recommends items to buy based off the user’s prior shopping history in Amazon. As a feature, whenever the user visits the extension maker’s home page at safewebshopping.com, the extension can use postMessage to send a list of Amazon items that the user has recently viewed directly to the website. safewebshopping.com can then use this extra information to enhance the webpage for the user. When information is being transmitted between an extension and webpage like this, the only form of security that can occur is an origin check to confirm that the request is coming from a desired source. These origin checks are easy to get right. Yet somehow, many of these extensions simply fail to write a secure origin check. Going back to our example, it would be unfortunate if the extension was tricked into giving away the user’s Amazon browsing history to maliciouswebshopping.com instead of the extension’s home page. This paper aims to explore this realm of failed origin checks. We are interested in seeing just how many Chrome extensions
on the Chrome Web Store have a postMessage vulnerability and we look forward to figuring out the most creative techniques we can use to bypass origin checks.

2 Related Papers

The Postman Always Rings Twice by Son and Shmatikov [3] analyzes postMessage origin exploits not in extensions, but in HTML5 webpages. To accumulate the vulnerabilities, they create RVScope, an automatic receiver collection tool. RVScope dynamically redefines the event listener so that the code body of the listener is reported whenever the listener registers a message event. The paper finds 14 different classes of origin check failures across a total of 261 hosts. This is out of the 10,121 total hosts they analyzed (2,245 of which have at least one postMessage receiver). With these origin check failures, the authors could induce XSS attacks, cookie disclosure, and even arbitrary reads and writes into local storage. They propose a “light” defense that makes it so only the origin that loaded a third-party frame can send messages to the frame as well as a more restrictive “heavy” defense.

An Evaluation of the Google Chrome Extension Security Architecture by Carlini, Felt, and Wagner [1] is a paper about the security features of Google Chrome. In the paper, they discuss the possibility of browser extensions providing access to private data and credentials for attackers and compromising the security of websites. The authors discuss some security features of the chrome browser that were intended to mitigate these risks. These features include “strong isolation between websites and extensions, privilege separation within an extension and an extension permission system.” This system makes a distinction between content scripts which run in websites and do not have privileges and the other extension code which has higher privileges. This distinction should prevent code from web pages from interacting with the parts of extensions that have privileges. The paper mentions that if a content script is compromised there is not much risk unless the attack continues with messages passed to the rest of the extension. At this point, the higher privileges of the rest of the extension could allow user information to be exposed. Whether the messages come from a compromised content script or from a page that the extension believes to be another page, these same vulnerabilities could be exploited.

A paper by Yang, Huang, and Mendoza entitled Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications [4] covers the vulnerabilities of extensions of postMessage for mobile apps. The paper focuses on implementations that treat the mobile app as a new frame and enable communication with web frames through postMessage. These implementations often remove or ignore origin information which the authors call the Origin Stripping Vulnerability. This vulnerability is very similar to the vulnerabilities that we are attempting to find in this paper. The researchers explain that the risks of this vulnerability include the possibility of an attacker sending arbitrary messages to access internal functionalities and data.

Hunting postMessage Vulnerabilities by O’Leary-Steele [2] identifies postMessage vulnerabilities as a culprit in weakening the Same Origin Policy. A tool PMHook (it is very similar to RVScope) is created to efficiently bind onto eventListeners and detect postMessage messages being passed in the browser. The paper mentions that postMessage vulnerabilities are often missed by security scanners and manual review by consultants. The paper further states that there were postMessage vulnerabilities found in many Fortune 500 websites. Due to the presence of these vulnerabilities in the websites of successful companies (presumably with abundant resources), it seems very likely that they can be found in code written by much smaller companies and
individuals as well.

3 What is postMessage?

Normally, scripts on different pages and extensions are not allowed to access each other unless they come from the same origin (protocol, port, and host name). This is known as the “Same Origin Policy”. In HTML5, the postMessage mechanism allows for websites from different origins to communicate with each other, which is a relaxation of the same origin policy [3]. This power can also be extended to extensions. It allows extensions to communicate with other extensions as well as the websites that a user visits. The utility of postMessage is abundantly clear, as the Same Origin Policy restriction can often limit functionality in common tasks such as tracking for ads. Extensions can choose to accept or ignore postMessages. This is often decided by an origin check, which determines where the message came from. Unfortunately, this makes it so that postMessage is only as secure as its origin check.

4 A Toy PostMessage Exploit

To illustrate the postMessage vulnerability, we created a simple example to show what our project is looking for.

The following extension code intends to send history data to texas.edu if the page requests it through a postMessage command. However, the origin check is weak, and naively uses the JavaScript includes function to determine whether the origin matches a certain string. With a carefully crafted origin such as cs.utexas.edu/~alfred one can easily bypass this origin check. There will probably be very few extensions with origin checks this egregious, but this example accentuates just how easy it is to inadvertently introduce security flaws through postMessage. The offending code is in line 5 of content.js.

```javascript
Vulnerable Extension Code content.js:

1 window.addEventListener("message", function(event) {
  2 if (event.source != window) return;
  3 if (event.data.type && event.data.type == "FROM_PAGE") {
    4 if (event.origin.includes("texas.edu")) {
      5 var data = {type: "history", text: historytext};
      6 window.postMessage(data, "*");
    7 }}});
```

```html
Malicious Website cs.utexas.edu/~alfred:

1 <html>
2 <head>
3 <title>secure website</title>
4 <script>
5 window.addEventListener("message", function(event) {
  6 if (event.source != window) return;
  7 if (event.data.type && (event.data.type == "history")) {
    8 document.getElementById("website_message").innerHTML = event.data.text;
  9 }
10 });
11 </script>
12 </head>
13 <body bgcolor="green">
14 <h1> Secure website </h1><hr>
15 <h3 id="website_message"> no malicious activity going on :) </h3>
16 <button type="button" onClick="postMe()">Click Me!</button>
17 </body>
18 </html>
```
5.1 Chrome Extension Collection

The Chrome Web store is a conglomeration of web extensions and web apps, each of them uniquely identified by a different web ID. Given a valid web ID, we can directly download the extension as a .crx file (the compressed file format created by Google) and unzip that file to access all of the extension’s JavaScript code and data. Possessing an extension’s internal code allows for static analysis to find vulnerabilities. One difficulty that came up was finding a list of extension IDs to download. Google does not officially provide such information nor does their web store present extension IDs in an easily digestible format. To solve this problem, we designed an extension which extracted the source code of the user’s current page and collected Chrome extension IDs. The Chrome Web Store dynamically loads in suggested extensions as you scroll further down the page. We manually scrolled through the extension store until the collector extension extracted 7290 extensions. It was possible to use a web browser automation tool such as Selenium to auto-scroll and thus collect even more extensions, but the number 7290 was deemed a sufficient amount of extensions for the scope of this project.

5.2 Static Analysis

For our initial attempt at static analysis, we searched through all of an extension’s JavaScript code and looking for the following two strings:

- `addEventListener("message`
- `.origin`

If an extension had both of those strings in the same JavaScript file, we flagged the extension as having a postMessage listener and worthy of further investigation. Table 1 shows our results. About 9.2% satisfied our condition of having an

5 Methodology

We gather the most popular Google Chrome extensions and run a static code analyzer to scan for origin check failures in the extension JavaScript code. We then manually analyze any flagged extensions and see if they are actually vulnerable due to an origin check exploit.
event listener and some sort of reference to an origin variable. We were cognizant of the fact that a simple pattern match could cause a few false positives and negatives, but this was a good starting point.

As a first attempt, we manually looked at several of the flagged extensions to see if we could identify vulnerabilities. We looked for vulnerabilities similar to those in The Postman always Rings Twice [3] (e.g. indexOf, match) and tried to exploit them. Several problems arose during our manual analysis. Many of the extensions obfuscated their code such that it was very difficult for us to follow and understand. This obfuscation occurred for approximately half of the extensions we examined, so it effectively halved the number of extensions we could investigate. However, it must be emphasized that obfuscation simply masked the problem. Renaming variable and function names did not remedy any underlying postMessage vulnerabilities. In addition to the obfuscation problem, several extensions had origin checks which simply validated that the postMessage request came from the host website. We could not say that we found an exploit if the extension already allowed the malicious website to post a message (this is simply the functionality of the extension!). There are probably many extensions that don’t do origin checks at all when they should. Such cases are difficult to determine, because it involves understanding the underlying design of a very large number of extensions. As a result of these factors, the extensions that failed to check origin at all were out of scope for our project. We also discovered that while the extensions that were found by this process were vulnerable to our messages, it was difficult to extract information from these extensions. We concluded that these extensions were potentially vulnerable to having their internal functions accessed, but it would be somewhat harder to get useful information from them.

Due to our problems with extracting information from vulnerable extensions, we decided to scan for another factor. We added functionality to our vulnerability scanner that finds postMessage calls that send information out of the extension after the extension receives a message. We accomplished this by examining the function called by the eventListener. We located this function in the JavaScript of the extension and then looked within the function for a postMessage call. The resulting data is in Table 2. The extensions found by this method were much easier to exploit. They sent out information to anything that was listening, so we could easily gain access to the information that was sent after the postMessage calls in our attacker website.

We hoped to further improve our techniques to discover postMessage vulnerabilities. One particular approach we were interested in trying stemmed from the related papers that we read. The Postman Always Rings Twice [3] and Hunting PostMessage Vulnerabilities [2] both dynamically redefine eventListeners to automatically log handler code along with associated messages being passed within the browser. This is a powerful technique, and it could be useful in circumventing the issues caused by obfuscation, but it was too difficult to accomplish in the amount of time that we had.

### 5.3 Static Analysis Results

<table>
<thead>
<tr>
<th>String</th>
<th>count</th>
<th>total</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.origin</td>
<td>669</td>
<td>7290</td>
<td>9.2%</td>
</tr>
<tr>
<td>.origin.indexOf</td>
<td>33</td>
<td>7290</td>
<td>0.5%</td>
</tr>
<tr>
<td>.origin.includes</td>
<td>2</td>
<td>7290</td>
<td>–</td>
</tr>
<tr>
<td>.origin.match</td>
<td>16</td>
<td>7290</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Table 1: No successive postMessage send

For our analysis we used a static scanner which simply pattern matched for specific patterns which were most likely to lead to an origin check exploit. Table 1 reflects the search results when we look for a particular origin check type within a postMessage listener. Table 2 is a little more restrictive (which explains its much lower numbers) although its search has nearly the same criteria as Table...
Table 2: Has successive postMessage send

<table>
<thead>
<tr>
<th>String</th>
<th>count</th>
<th>total</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.origin</td>
<td>30</td>
<td>7290</td>
<td>0.4%</td>
</tr>
<tr>
<td>.origin.indexOf</td>
<td>2</td>
<td>7290</td>
<td>–</td>
</tr>
<tr>
<td>.origin.includes</td>
<td>0</td>
<td>7290</td>
<td>–</td>
</tr>
<tr>
<td>.origin.match</td>
<td>1</td>
<td>7290</td>
<td>–</td>
</tr>
</tbody>
</table>

The most alarming part was not the insecure origin check, but was the information capabilities that an attacker is afforded by the extension after the origin check is bypassed. After the origin check in messaging.js, the Fair Ad Blocker extension sends out another postMessage to respond to the request. With a particularly well crafted initial postMessage by a malicious website, the attacker can force the extension to give specific information about the user. The malicious website can forge the event.data.type of the postMessage that it sends to the Fair Ad Blocker extension, and for each specific case get different information. The case stndz.messages.getAppData allows the attacker’s website to potentially steal the user’s name and email (provided that they supplied this information beforehand to the extension). Perhaps the most security breaking case is stndz.messages.getAppData. Included in the postMessage returned by the extension in such a case is the active tab information of the user. Active tab means that the website can query and detect what other tabs the user is currently browsing, which normally is not supposed to happen under the browser sandbox. This vulnerability is extremely dangerous as it allows the user to be possibly fingerprinted if the attacking website’s tab is kept alive in the background for a significant amount of time. Also, if the user visits any website which stores sensitive information such as session IDs in the URL address, the malicious website can steal those to spoof the user. These scenarios are very plausible, assuming that the attacker can somehow convince or coerce the user to visit the malicious website in the first place. The stndz.messages.getAppData case also exposes the sites that the users whitelisted for ads. This information gives further insight into the user’s

• http://espn.nfl.standsapp.org
• http://espn.nfl.localhost.com
• http://espn.nfl.standsapp.org.XXXXXX.com
• http://cs.utexas.localhost.edu
• http://lgblnidahcdcdiepkckcfdfhknnjh.edu

Vulnerable Origin Check messaging.js:

```javascript
1 if (!event.origin.match(/^(http(s)\:\//|localhost|\.|(\.|localhost)\.|(\.|localhost)\.|stndz.com)(:\d*)?$/i))
2 return;
```

The above code is the insecure origin check which was discovered by our static scanner. The origin check appears to be pattern match using a Regex string, but the specified Regex is not particularly robust nor secure. There are several URL origins (many of which would be plausible and unsuspicious to a normal user!) which easily bypasses this origin check. We enumerate a few of these below:

- http://espn.nfl.standsapp.org
- http://espn.nfl.localhost.com

5.4 Vulnerability Spotted in the Wild

We have found an extension with a postMessage origin check vulnerability. The offending extension is Fair Ad Blocker which can be downloaded at https://chrome.google.com/webstore/detail/fair-adblocker/lgblnfidahcdcdiepkckcfdfhknnjh. This extension is an ad blocker which is focused on blocking the most annoying and disruptive advertisements while allowing users to dictate the number of ads they would consider fair.

Vulnerable Origin Check messaging.js:

```javascript
1 if (!event.origin.match(/^(http(s)\:\//|localhost|\.|(\.|localhost)\.|(\.|localhost)\.|stndz.com)(:\d*)?$/i))
2 return;
```

The above code is the insecure origin check which was discovered by our static scanner. The origin check appears to be pattern match using a Regex string, but the specified Regex is not particularly robust nor secure. There are several URL origins (many of which would be plausible and unsuspicious to a normal user!) which easily bypasses this origin check. We enumerate a few of these below:

- http://espn.nfl.standsapp.org
- http://espn.nfl.localhost.com
- http://espn.nfl.standsapp.org.XXXXXX.com
- http://cs.utexas.localhost.edu
- http://lgblnidahcdcdiepkckcfdfhknnjh.edu

The most alarming part was not the insecure origin check, but was the information capabilities that an attacker is afforded by the extension after the origin check is bypassed. After the origin check in messaging.js, the Fair Ad Blocker extension sends out another postMessage to respond to the request. With a particularly well crafted initial postMessage by a malicious website, the attacker can force the extension to give specific information about the user. The malicious website can forge the event.data.type of the postMessage that it sends to the Fair Ad Blocker extension, and for each specific case get different information. The case stndz.messages.getAppData allows the attacker’s website to potentially steal the user’s name and email (provided that they supplied this information beforehand to the extension). Perhaps the most security breaking case is stndz.messages.getAppData. Included in the postMessage returned by the extension in such a case is the active tab information of the user. Active tab means that the website can query and detect what other tabs the user is currently browsing, which normally is not supposed to happen under the browser sandbox. This vulnerability is extremely dangerous as it allows the user to be possibly fingerprinted if the attacking website’s tab is kept alive in the background for a significant amount of time. Also, if the user visits any website which stores sensitive information such as session IDs in the URL address, the malicious website can steal those to spoof the user. These scenarios are very plausible, assuming that the attacker can somehow convince or coerce the user to visit the malicious website in the first place. The stndz.messages.getAppData case also exposes the sites that the users whitelisted for ads. This information gives further insight into the user’s
browsing habits, because sites on a whitelist are probably those that the user visits frequently.

Information Leaking Code `messaging.js`:

```javascript
switch (event.data.type) {
  case 'check-stands-request':
    window.postMessage({ type: 'check-stands-response' }, '*');
    break;
  case stndz.messages.updateUser:
    sendMessageToBackground(event.data, function(result) {
      window.postMessage({ type: 'update-user-response', requestId: event.data.requestId, result: result }, '*');
    });
    break;
  case stndz.messages.getUserData:
    sendMessageToBackground(event.data, function(userData) {
      console.log(JSON.stringify(userData));
      window.postMessage({ type: stndz.messages.getUserData + '-response', userData: userData }, '*');
    });
    break;
  case stndz.messages.getAppData:
    sendMessageToBackground(event.data, function(stats) {
      window.postMessage({ type: stndz.messages.getAppData + '-response', stats: stats }, '*');
    });
    break;
  ...
}
```

6 Limitations

Although vulnerable origin checks are easy to create, actually leveraging them into exploits is very challenging. These sorts of attacks can only make gadgets out of code that is present in the postMessage listener after the failed origin check. This greatly limits the power of malicious website, and generally does not allow for arbitrary execution of code. In the Fair Ad Blocker example that was discussed above, there was a postMessage listener which listened for a particular `executeScriptOnTab` event and allowed for arbitrary scripts to run on any tab in the browser. Given this capability, an attacker can wreak havoc on a victim’s browser. Fortunately, this listener was not accessible from the scope of the postMessage listener of the malicious website, and thus could not be exploited. Even the information that can be extracted from the extension is limited and subject to how the extension was initially designed. Many extensions that had bad origin checks simply did not provide a channel that could funnel out information to the malicious website. This manifests clearly in Tables 1 and 2 in Section 5.3. Without a successive postMessage send after the failed origin check, it may be impossible for the malicious website to steal sensitive information even if it is there for the taking. The worst that the attacker can usually do in this scenario is to mess with the internal operations of the extension so that it doesn’t behave as intended. Although this is a security concern, it isn’t as dangerous as information leakage.

7 Defenses

Bad origin checks can completely compromise extensions. Depending on the capabilities that the creators expose to messages, attackers can access data, use certain functionality in an extension, or even run arbitrary code. These attacks are completely avoidable by taking the proper precautions. Developers have the largest role in preventing these vulnerabilities. They must work to make ori-
gin checks secure by taking time to ensure that they are only exposing their extension to messages from valid sources. Origin checks that use \texttt{indexOf}, \texttt{match}, and \texttt{includes} are very unlikely to be secure. Developers should take the time to understand exactly which sources should be able to send messages to their extension, and whether their origin checks allow any other sources to send messages. Developers can also work to reduce \texttt{postMessage} usage when it is not necessary to accomplish the extension’s function. For example, allowing user data to be sent through \texttt{postMessage} to your website is unlikely to be necessary when you can just have the user enter their username or other information normally. Developers should also ensure that no scripts are being run that can be affected in any way by \texttt{postMessage} calls. Allowing arbitrary scripts to be run is one of the worst outcomes that extensions can expose to attackers. Finally, obfuscating an extension’s code can make it much harder for attackers to make use of a bad origin check. If the attacker is not specifically targeting the extension, they may just move on to an extension that is easier to exploit.

Users cannot prevent these attacks but they should be aware of the information or capabilities they allow extensions to access. In general, this knowledge will protect them from malicious extensions, but it can also mitigate the damage from attackers who target extensions. If users are more aware of the security risks, they can avoid certain attacks. They should realize that any of the permissions that an extension requests can potentially be exploited.

8 Future Work

There are several areas where work could be done in the future. For example, the same type of study could be done to find vulnerabilities in Firefox extensions. There could also be a way to perform dynamic checking for origin vulnerabilities. This could be a way to avoid the issues associated with finding ways to exploit obfuscated code. It may also be possible that dynamic checking would expose vulnerabilities that our static solution did not find. Another possible avenue for future work would be to make an online tool to break origin checks. This might be useful to developers who would like to know whether their origin checks are vulnerable. It might also work to raise awareness about the ways that origin checks are broken. We also plan on notifying Standsapp, the company behind the Fair Ad Blocker extension, about the vulnerability in their extension so it can take steps to protect its users.

9 Conclusion

Vulnerabilities due to bad origin checks are relatively common and can be used to expose user information and other functionality to an attacker. In this paper, we have proven that bad origin checks exist in currently available and widely used browser extensions. Through our static analysis, we found hundreds of extensions that are potentially vulnerable (although not all of those extensions will be vulnerable). We have explained several ways that these bad origin checks can be subverted. We have also shown that, once bad origin checks are subverted, the vulnerable extensions can be exploited to gain information about users. We exploited a bad origin check in an ad blocking extension used by over 2,000,000 users in order to gain access to user data and browsing history. Vulnerabilities like this are not uncommon and it is reasonable to think that there maybe exploits in use currently by attackers. As a result, it seems that bad origin checks are an issue that extension developers and extension marketplaces should address. These \texttt{postMessage} vulnerabilities are relatively easy to find and only slightly harder to fix. These vulnerabilities do not need to exist, but as long as there is no systematic way to remove all of these vulnerabilities, individual developers will keep enacting poor origin checks. If a systematic solution cannot be achieved, it would be useful to at least raise awareness about this vulnerability so
that developers can take the time to fix their origin checks.

10 Availability

The code for our project can be found at the publicly open repository. It includes code for the toy exploit, the exploit of the Fair Ad Blocker, the static analysis scanner, and the Chrome extension ID scraper.

https://github.com/alfredzhong0/cs380s-project

References


