Analysis of UXSS exploits and mitigations in Chromium

Max Moroz, Sergei Glazunov, Google
{mmoroz,glazunov}@google.com

Abstract

UXSS (*Universal Cross-Site Scripting*) is an attack that exploits client-side vulnerabilities in the browser or browser extensions in order to execute malicious code (usually JavaScript) with access to arbitrary resources (origins). To put it simply:

> *A victim visits a malicious (or hacked / infected) website and an attacker becomes able to read victim's GMail contents, private messages on Facebook, and so on, as well as to perform other actions on behalf of the victim: send emails, upload photos, etc*.

The goal of this research is to analyze vulnerabilities in Chromium leading to UXSS attacks that were reported over 3 years (2014 - 2016), to evaluate potential mitigations that can be implemented in Chromium browser, and to explore the possibilities of new techniques to be used for prevention or detection of vulnerabilities leading to UXSS.

Background

SOP (*Same Origin Policy*) is one of the most important concepts in the web application security model. Basically, it prevents different **origins** from accessing each other's data stored on the client side (i.e. cookies, contents of browser tabs, everything associated with some web application and available on client side).

Origin, as it is defined by the HTML Standard, is a combination of a URI's scheme, host and port. Thus, two different URIs belong to the Same Origin, if they both contain the same scheme (e.g. `https`), host (e.g. `google.com`) and port (e.g. `443`). Otherwise, two URIs belong to different origins and cannot access data of each other by default.

In JavaScript, an **Execution Context** represents the environment in which the code is being executed. The default context is the **Global Execution Context** (GEC), which is usually created when the browser loads a new page. Every GEC has its own set of JS builtins, and every JS object is tied to an execution context. API functions often use the context to perform access checks.

Cross-Site Scripting (XSS) is a client-side code injection attack that allows an attacker to compromise the interactions users have with a vulnerable web application. XSS flaws usually allow an attacker to perform any actions on behalf of a victim user and to access any of their data on the website thus evading the SOP. These vulnerabilities occur when a web application uses an input provided by a user to generate the output without proper validation. Currently, XSS is the most widespread type of web application attacks.

Instead of using flaws in web applications, **Universal Cross-Site Scripting** (UXSS) attacks exploit vulnerabilities in the browser itself or in browser extensions to achieve an XSS condition. As a result, the attacker does not just get access to user session on a single website, but may get access to any page currently opened in the browser, including internal browser pages. Bugs leading to UXSS attacks are among the most significant threats for users of any browser. Chromium Severity Guidelines categorize such bugs as **High Severity Vulnerabilities**.

From an attacker perspective, a UXSS exploit may be almost as valuable as a **Remote Code Execution** (RCE) exploit with the sandbox escape, as UXSS exploits tend to be more reliable and in many cases can satisfy the needs of an attacker, unless the goal is to fully compromise the victim's device.

This research often refers to Chromium's multi-process architecture. See Chromium documentation for more details on this topic. The components which are the most relevant for this research are the following ones:

- **Browser Process** is the main and the most privileged process of the browser

application. That process has privileges equal to the privileges of the user running Chromium on their computer, including access to the file system, network stack, system APIs, and so on. The browser process controls the top-level browser window, user interface, inter-process communication, and does other high level management.
- **Renderer Process** is a less privileged process that is responsible for surfacing web pages to the user and executing JavaScript. Chromium creates multiple renderer processes for different websites opened in the browser. Each process is isolated and runs in the sandbox, therefore a compromise of a renderer process imposes a lower risk compared to a compromise of the browser process.

Blink is the rendering engine used by Chromium.

V8 is the JavaScript and WebAssembly engine used by Chromium.

Until a newer navigation architecture (also known as **PlzNavigate**: https://crbug.com/368813, Design Doc) was deployed (October 2017), an RCE in a renderer process could be trivially turned into a UXSS. The reason being that navigations were initiated by renderer processes whereas a malicious code could bypass cross-origin security checks. With PlzNavigate being effective, the browser process handles all navigations requests and enforces the policy.

Also, it used to be trivial to turn a successful UXSS exploitation into an RCE on Android until December of 2016 (https://crbug.com/664411#c31). Briefly, an attacker could install an arbitrary application on the victim's device without any user interaction or permission. Since then, the user is always prompted to re-authenticate when installing an application through the web flow.

Chrome's Vulnerability Reward Program used to grant equal monetary rewards for UXSS and renderer RCE exploits.

Overview

Chromium browser and its components strongly enforce the Same Origin Policy concept. This is addressed on various levels ranging from the actual SOP implementation in Blink and V8's context security model to new features such as Site Isolation. However, as in any other complex software project, there are mistakes breaking these protections under certain circumstances.

This document presents an overview and analysis of vulnerabilities enabling UXSS attacks that were reported over the years 2014-2016, to draw conclusions about how to mitigate these issues in the future. Note that in that time range neither PlzNavigate nor Site Isolation features were enabled in Chromium.

This research was mostly performed in early 2017. That is why we still refer to Site Isolation as an upcoming enhancement rather than an existing feature.

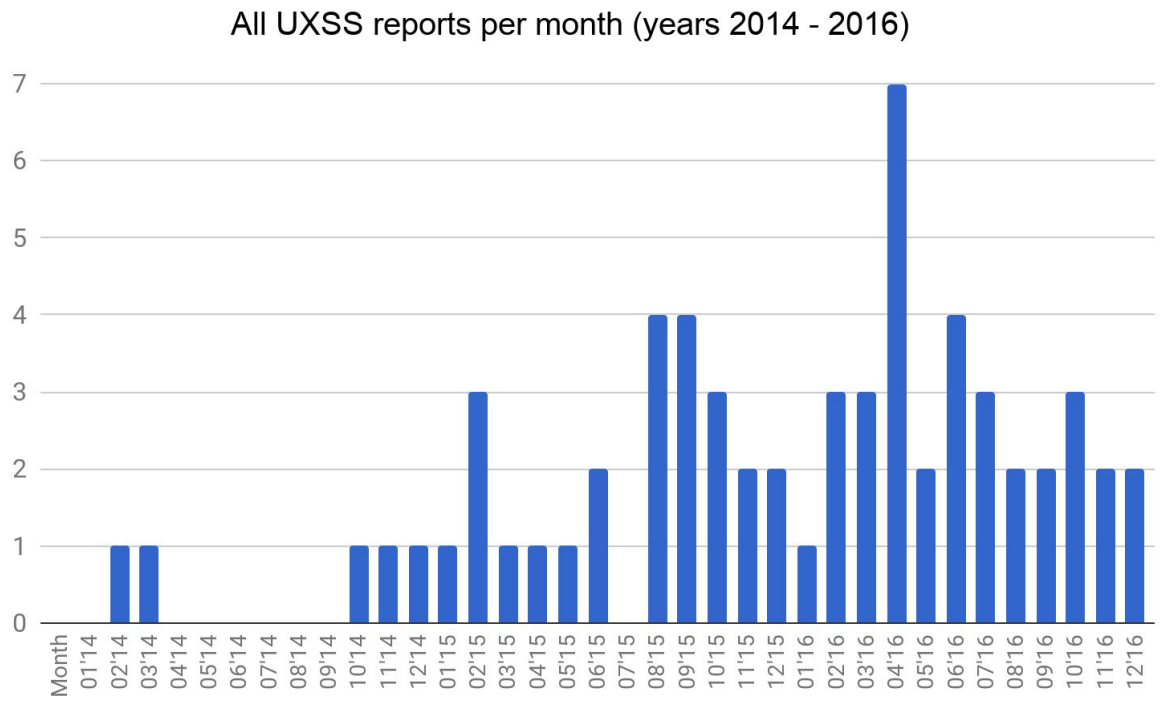
Bug Reports Analyzed

More than a hundred Chromium issues were found using the following queries among the issues reported **in 2014 - 2016**:

- “UXSS”, “XSS”, “Universal XSS”.
- “Cross-Site Scripting”, “Universal Cross-Site Scripting”.
- “SOP Bypass”, “SOP”.
- “Same Origin Policy”, “Same Origin Policy”.
- “Cross-origin”, and some others.

After the first pass and an initial analysis, **63 issues** in the Chromium bug tracker were selected as **valid UXSS reports**. After that, these **63 reports** were tentatively clustered into different classes of vulnerabilities having a common root cause or exploitation pattern. Finally, these classes were refined and all the reports were strictly grouped into **8 classes**. For more detailed information, please see the **Analysis Of Bugs** section.

Distribution Over Time



Analysis Of Bugs

The following classes were defined from the analysis of UXSS exploits reported in Chromium in 2014-2016 years.

Class 1. [16 items] Blink: abusing parser initiated javascript: URI page loads

Description

The biggest vulnerability class in this research.

One of the possible ways to trigger JavaScript execution is to perform javascript: URI navigation. Chromium used to check the origin of the current execution context to determine whether the load should succeed. If there was no active context on the stack, the browser considered the navigation safe. So, if an `<iframe>` element had a cross-origin page loaded in it and was a part of a subtree that wasn't attached to the document, it was possible to force the HTML parser to insert the element to the document and load an arbitrary javascript: URI.

Hardening Measures

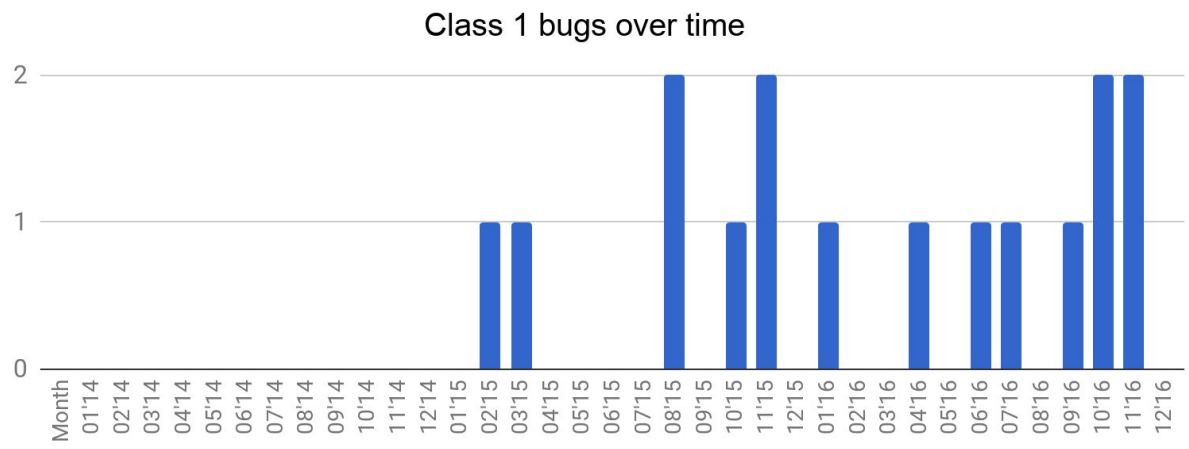
A comment from Daniel Cheng (*dcheng@chromium.org*):

> *We've had a longstanding assumption in the parser that it's always safe to execute a `javascript:` URL if there's no javascript context on the stack, we assume that this is parser triggered. Unfortunately, some of VRP reporters noticed this assumption, and combined this assumption with DOM corruption attacks: the DOM corruption is always a different source, but the goal of the DOM corruption is to trigger this assumption in a situation where it's dangerous. We finally landed general mitigations for this: see https://codereview.chromium.org/2502783004/ and https://codereview.chromium.org/2190523002/.*
>
> *One other thing I forgot to add: we should change `javascript:` navigations to never be synchronous. There are several instances where they are synchronous today, and it's only ever caused trouble.*

Reports

#	crbug	date	summary
1	456518	7-Feb-2015	HTML parser may leave frame element in an incorrect state

2	464552	5-Mar-2015	Heap-use-after-free in blink::ContainerNode::attach
3	516377	3-Aug-2015	UAF/DOM tree corruption in blink::ContainerNode::parserRemoveChild
4	519558	11-Aug-2015	Security: Universal XSS via ContainerNode::parserInsertBefore
5	541206	8-Oct-2015	Security: Universal XSS using document.adoptNode
6	556724	16-Nov-2015	Security: Universal XSS via persistence of subframes
7	560011	22-Nov-2015	Security: Universal XSS using widget updates in ContainerNode::parserRemoveChild
8	577105	13-Jan-2016	Security: Universal XSS by circumventing the unload event
9	605766	21-Apr-2016	Security: Universal XSS through adopting image elements
10	621362	19-Jun-2016	Security: Universal XSS with Flash calling into JavaScript inside Node::removedFrom
11	630870	24-Jul-2016	Security: Universal XSS by intercepting a UA shadow tree
12	645211	8-Sep-2016	Security: Universal XSS using blink::HTMLMarqueeElement
13	655904	14-Oct-2016	Security: Universal XSS via fullscreen element updates
14	658535	22-Oct-2016	Security: Universal XSS using an <input type="color"> element
15	663476	8-Nov-2016	Security: Universal XSS through removing link elements
16	668552	24-Nov-2016	Security: Universal XSS by polluting private scripts with named properties



Class 2. [6 items] Blink: missing or incorrect usage of cross-origin access checks

Description

This class is relatively straightforward. When performing a sensitive operation, the browser should ensure that the current context (or the current page) has the appropriate permissions. In the bugs listed below the checks were either missing at all or were performed on a wrong context.

Bug 504011 is a good example of the issue. First, an attacker leaks `GetModuleSystem()` function of `V8ContextNativeHandler`. Then, they call that function with a cross-origin window object of another origin. Actually, a cross-origin reference should not be created, but, due to a lack of an access check, that function call returns the requested module object in the context of the other origin.

Bug fixes for this class of bugs are usually small, e.g. for the bug referenced above:

- call a helper function for an access check from the vulnerable function;
- and implement that helper using `BindingSecurity` API.

Hardening Measures

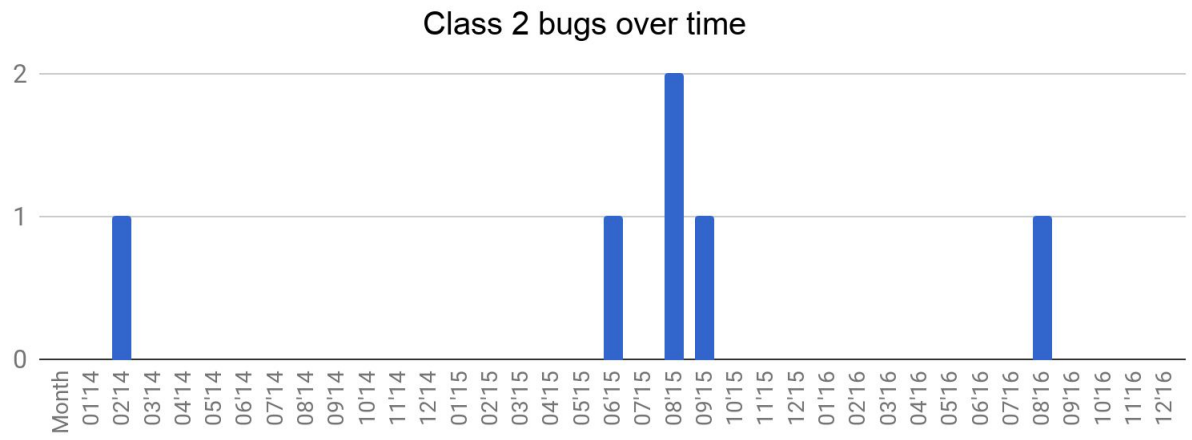
See https://crbug.com/525330: *Null out `DOMWindow::m_frame` as soon as the frame/window is detached*.

Chromium uses objects with different lifetimes to represent a page. For example, a `Frame` object is preserved between navigations, but a new `DOMWindow` is created for every page load. `DOMWindow` objects used to store a reference to the frame even after they had been detached. That led to a number of issues where the access check for a cross-origin frame was performed on a same-origin detached window. This patch made it possible to clear the frame reference.

Reports

#	crbug	date	summary
1	342618	11-Feb-2014	Security: UXSS via dispatchEvent on iframes (subject to some conditions)
2	504011	24-Jun-2015	Security: Cross-origin scripting possible via module system leak
3	522791	20-Aug-2015	Security: Universal XSS using navigator.serviceWorker.ready
4	524074	24-Aug-2015	Security: Universal XSS by loading a javascript: URI from an unloaded

			window
5	529682	9-Sep-2015	Content script is able to eval code in background page of other extension
6	638742	17-Aug-2016	Security: Universal XSS using ThreadDebugger::setMonitorEventsCallback



Class 3. [10 items] Blink and V8: incorrect context used

Description

When creating a new JS wrapper object, Blink methods often have to determine the correct creation context. Bugs in this class occur when the value used as the creation context source is controllable by a user.

Consider bug 632634 as an example. Bindings code for static methods allowed `info.Holder()` (a method of its argument) to be set to an arbitrary value. Then, the creation context of `info.Holder()` could be used to return a `ScriptState` object, which ended up as a cross-origin reference when executed from a static method.

The fix contained two different implementations of previously vulnerable `forHolderObject()` function (`forFunctionObject()` and `forReceiverObject()`) to be used based on whether the calling method or attribute is static or not.

A number of issues in this class abused JS exception creation. Bug 453979 is a good example:

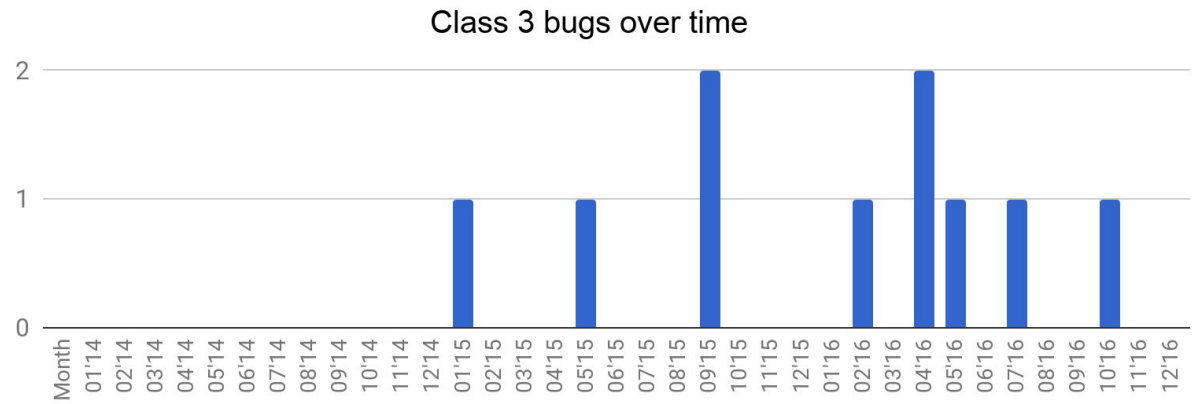
> *When a DOM method throws an exception, the creation context for the exception object is inherited from the object the method is called on even if it's from a different origin. The created object doesn't have any access checks so an attacker can use it to obtain a reference to e.g. the function constructor.*

Fixes for these issues typically include sanitization of a creation context when throwing an exception. Another fix was landed to convert cross-site exceptions into security errors.

Lastly, 583445 is another notable example. In that case the browser didn't update the execution context after navigation, so the new page would run JavaScript in the context of the previous one.

Reports

#	crbug	date	summary
1	453979	30-Jan-2015	Security: UXSS in V8 with exception object
2	494640	31-May-2015	Security: Universal XSS using IDBKeyRange static methods
3	530301	10-Sep-2015	Security: Universal XSS using stack overflow exceptions
4	531891	15-Sep-2015	Security: Universal XSS using exceptions thrown from Object.observe
5	583445	2-Feb-2016	Universal XSS in DocumentLoader::createWriterFor
6	605910	22-Apr-2016	Security: Universal XSS using iterables
7	607483	28-Apr-2016	Security: Universal XSS converting IDL array/sequence values
8	616225	31-May-2016	Security: Universal XSS in V8Console::memoryGetterCallback
9	632634	29-Jul-2016	Security: Universal XSS with static methods and ScriptState::forHolderObject
10	656274	15-Oct-2016	Security: Cross-origin object leak via fetch



Class 4. [11 items] Navigation: isNavigationAllowed() bypass, missing or bypassed ScriptForbiddenScope, etc

Description

This is the most recent class of vulnerabilities. 7 of 8 bugs here were reported in 2016 (see the chart below).

The class combines bugs that have used weaknesses in the page navigation logic to break one of two invariants. The first one is that it's impossible to perform a synchronous cross-origin page load. It could lead to UXSS because of the TOCTOU issue related to loading `javascript:` URIs in `<iframe>` elements. The second one is that two documents cannot be attached to the same `Frame` object simultaneously. If one of the documents is same-origin, and the other is cross-origin, the attacker can use the former to modify the latter while the `Frame` acts as a proxy.

Bug 616907 is a good example:

> *This is an architectural problem with the implementation of `ScopedPageLoadDeferrer`. Basically, it works by marking pages as deferred at the time a deferrer is instantiated. The issue is that pages created past this point don't defer loads by default. An attacker can move an iframe across the deferral boundary, which allows synchronous cross-origin navigations in unexpected circumstances.*

And the fix was to disable opening new pages while the deferrer is instantiated.

Another notable case is bug 600182:

> *When a `ScopedPageLoadDeferrer` is destroyed, the deferring state is updated on the associated pages and loaders. If any history load was set aside during the event loop the deferrer has been protecting, it's processed during the update without checking if navigation is allowed on the frame.*
> *This opens an avenue for an attacker to bypass the `FrameNavigationDisabler`.*

The fix against this one was to move `isNavigationAllowed()` check to main entry point for loads.

Hardening Measures

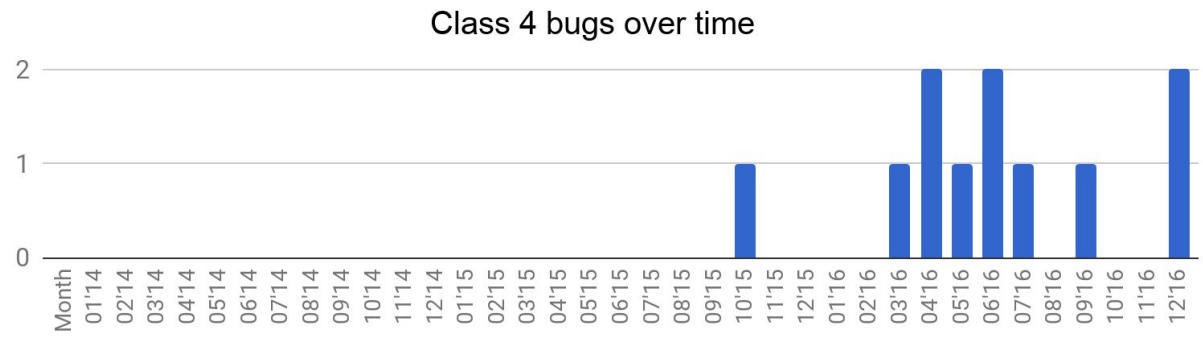
See https://crbug.com/629431: *Security: extension system must respect the page load deferrer*.

Several bugs in this category relied on using a nested event loop to perform page loads. After the load has been completed, the exploit has to continue its execution, however, it's not possible

to schedule a JavaScript callback inside a nested loop with a regular timeout or promise. The fix addressed a problem in the extension API which allowed an attacker to bypass the restriction.

Reports

#	crbug	date	summary
1	546545	22-Oct-2015	Security: Universal XSS using plugin objects
2	597532	24-Mar-2016	Security: Universal XSS using a FrameNavigationDisabler bypass
3	600182	3-Apr-2016	Security: Universal XSS using deferred history loads
4	601706	8-Apr-2016	Security: Universal XSS using a flaw in the load deferral logic
5	613266	19-May-2016	Security: Universal XSS via reentrancy in FrameLoader::startLoad
6	616907	2-Jun-2016	Security: Universal XSS using a ScopedPageLoadDeferrer bypass
7	617495	6-Jun-2016	Security: Universal XSS via same document navigations
8	628942	17-Jul-2016	Security: Universal XSS with ScopedPageLoadDeferrer and RemoteFrame
9	646610	13-Sep-2016	Security: Universal XSS using OOPIF
10	671102	5-Dec-2016	Security: Universal XSS through bypassing ScopedPageSuspender with closing windows
11	673170	11-Dec-2016	Security: Universal XSS using late widget updates



Class 5. [8 items] Extensions API: leak of a function or an object and use of an arbitrary or a hijacked createContext

Description

The extensions system has access to some JavaScript internals and also provides a public API to the user context. Due to different mistakes in the API implementation, there were a few tricks

used to abuse extensions API.

The main idea of all bugs in this class is to leak an internal object. For example, bug 590275:

> *`RequireForJsInner` calls `GetProperty` on the internal object that is used to store exported functions for the module system. A getter function defined on `Object.prototype` could leak that object.*

Then, the leaked object is being used to gain a cross-origin access, for example:

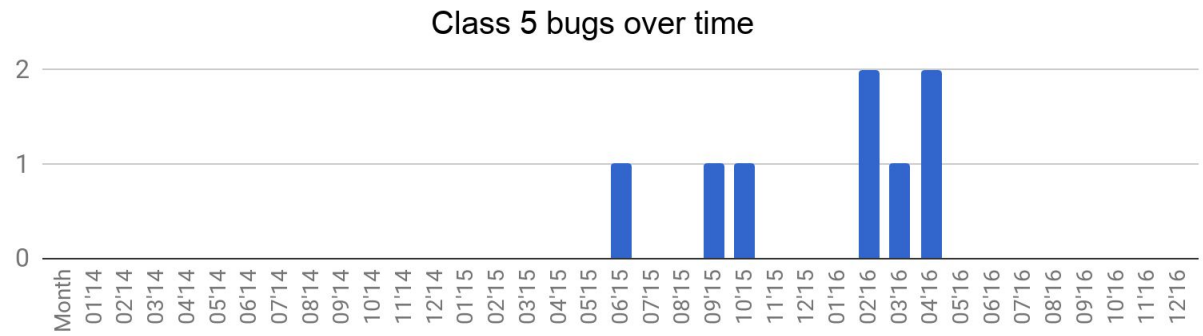
- via native functions such as user_gestures.RunWithUserGesture in bug 590118;
- by abusing `SendRequestNatives::GetGlobal` to obtain a victim windows object in bug 546677.

There were different fixes applied against different bugs. Notable examples are:

- Stop using the given `creationContext` in public APIs.
- Harden against bindings interception.

Reports

#	crbug	date	summary
1	497507	6-Jun-2015	Security: Cross-origin scripting possible via native functions
2	534923	22-Sep-2015	Security: Universal XSS via the unload_event module
3	546677	22-Oct-2015	Universal XSS with SendRequestNatives::GetGlobal
4	590118	26-Feb-2016	Security: Universal XSS using an intercepted native function
5	590275	26-Feb-2016	Internal object leak in ModuleSystem::RequireForJsInner => Universal XSS
6	598165	26-Mar-2016	Security: Universal XSS via the interception of |Binding| with Object.prototype.create
7	601073	6-Apr-2016	Security: Universal XSS in extension bindings
8	604901	19-Apr-2016	Security: Persistent UXSS via SchemaRegistry



Class 6. [3 items] V8: missing or incorrect usage of access check

Description

This class is similar to class 2, the only difference is that the missed checks should be on the V8 side.

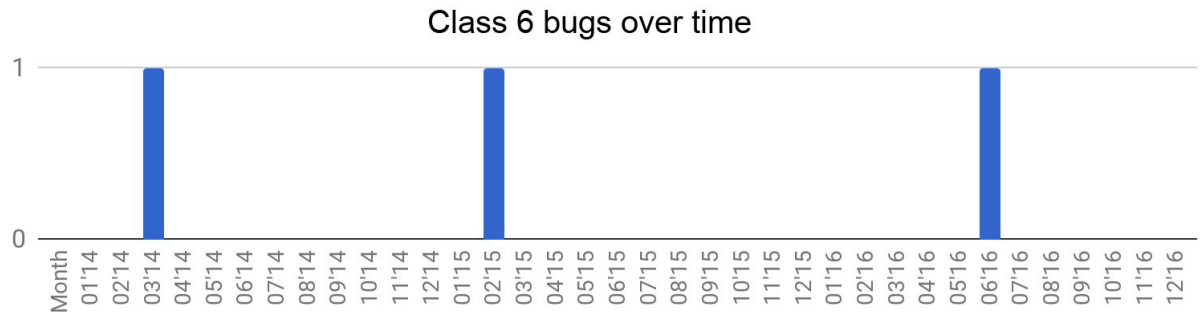
Consider bug 354123 as an example:

> *The current implementation of `Object.setPrototypeOf` doesn't have any security checks. To exploit this as a UXSS an attacker could replace the victim's window prototype with an object which has default methods/property accessors redefined to leak an object from the victim's JS context.*

The fix for this issue was pretty small. Required access check calls were added.

Reports

#	crbug	date	summary
1	354123	19-Mar-2014	UXSS with Object.setPrototypeOf
2	455961	6-Feb-2015	Cross origin access with window.name and Object.getOwnPropertyDescriptor
3	619166	10-Jun-2016	Universal XSS with global proxies, interceptors, and synchronous page loads



Class 7: [3 items] Flash-specific issues

Description

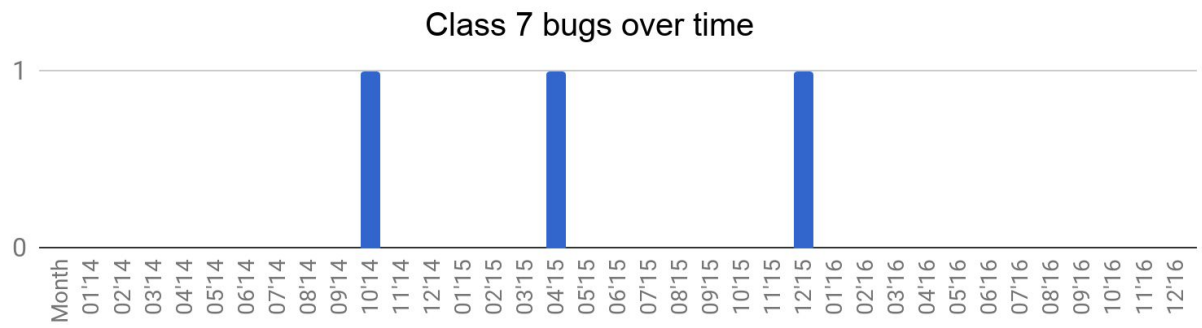
The Flash plugin has its own implementation of the SOP policy. This class combines missing security checks in the plugin itself and issues in the code responsible for Flash support in

Chromium.

Most of the bugs in this class were fixed by Adobe. The fix for 569496 was to suppress page loads inside `PPB_Flash_MessageLoop`.

Reports

#	crbug	date	summary
1	425280	20-Oct-2014	Security: Flash Cross Domain Policy Bypass by Using File Upload and Redirection - only in Chrome
2	481639	27-Apr-2015	Security: Boundless Tunes - universal SOP bypass through ActionSctipt's Sound object
3	569496	14-Dec-2015	Security: Universal XSS using Flash message loop



Class 8. [6 items] Custom issues: external dependencies, custom modes (e.g. Design mode, DevTools), plugins (e.g. Pepper), special resource types

Description

This class contains reports exploiting different parts of Chromium codebase without any common pattern. Each of these bugs can be moved to a singleton class, but putting them into a single bucket of custom issues seems slightly more reasonable.

Due to the different nature of the vulnerabilities, the fixes were very different as well.

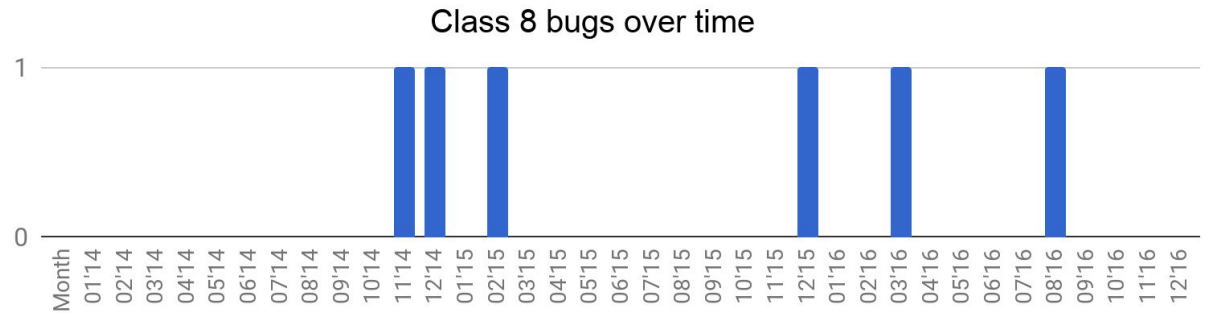
One pattern that is common for two issues in this class (429542 and 594383) is that both vulnerabilities were caused by a special handling of the `file://` scheme. The fixes were the following:

- Make "`file:`" to be an effectively unique origin.

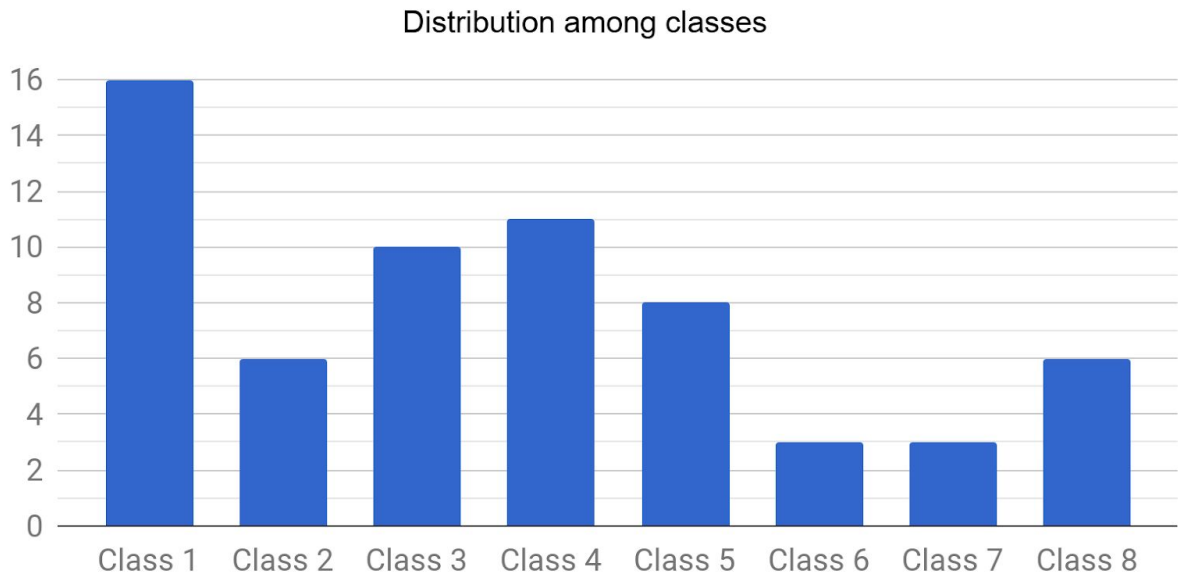
- Apply `WebSettings` before initializing the main frame.

Reports

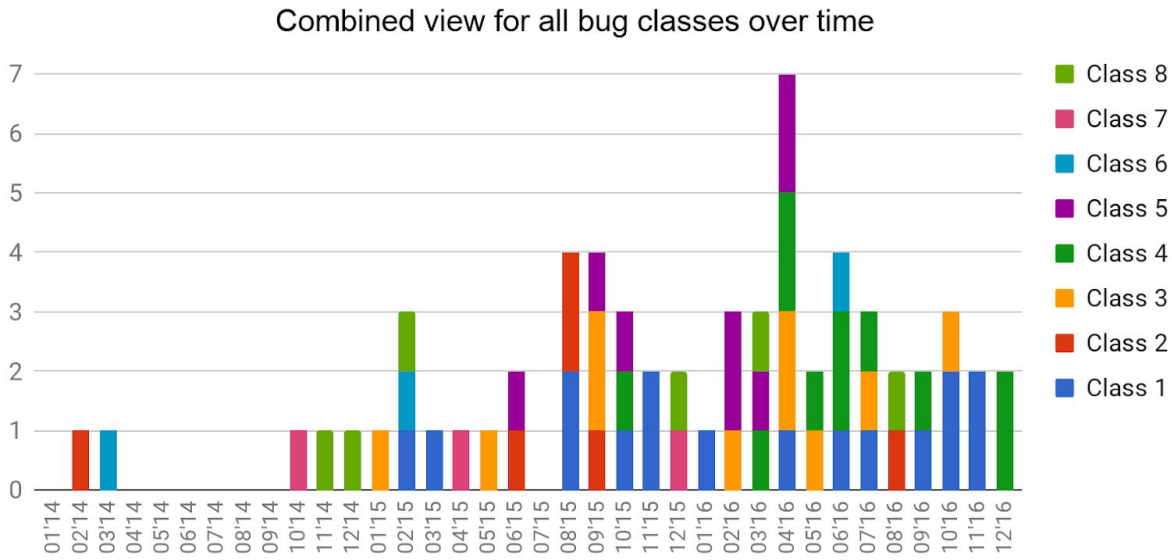
#	crbug	date	summary
1	429542	2-Nov-2014	Security: file-to-file SOP bypass on Linux via /proc/self/fd/
2	444927	23-Dec-2014	Security: Inherited designMode and cross-window drag-n-drop allow to modify a cross-origin iframe's DOM
3	462843	28-Feb-2015	Security: UXSS in AuthenticatorHelper
4	569955	15-Dec-2015	Security: Universal XSS by using fullscreen API
5	594383	13-Mar-2016	Security: UXSS via window.open() via file:// pages
6	637594	14-Aug-2016	Security: Universal XSS using DevTools



Distribution Of Reports Among Different Classes



Combined View Over Time



UXSS In Other Browsers

Chromium is not unique to have vulnerabilities leading to UXSS. Every major browser has had such bugs in the past. Let's consider a few examples and compare them to the categories defined for Chromium.

Safari

Chromium and Safari used to share the same rendering engine (WebKit), so a lot of older bugs listed in this paper affect Safari as well. One of the more recent examples is https://bugs.chromium.org/p/project-zero/issues/detail?id=1068. Even though it's a bug in the bindings for JavaScriptCore, Safari's JavaScript engine which is different from Chromium's V8, it exactly matches the class 3. One of the arguments was used to determine the creation context for an exception object. An attacker could pass a cross-origin object as the argument to obtain a cross-origin exception which exposed the other origin's `Function` constructor.

The fix modified the function to obtain the context directly through an additional argument.

Edge

CVE-2017-0002 demonstrates a bug in dealing with `about:blank` pages. Such pages are special in a way that it's not possible to derive their origin from the URL, instead they inherit the origin of either an opener or a parent page. Implementing this behaviour incorrectly might lead to violations of SOP. In this particular case the problem was that an `about:blank` page that didn't inherit the origin (i.e. had an empty one) was able to access any other `about:blank` page. This vulnerability is actually almost identical to an old Chromium bug 89453. As for classification, it falls under class 8.

Unfortunately, there is no public link to the fix.

Firefox

In one notable vulnerability in Firefox, incorrect handling of special characters allowed an attacker to spoof the page URL. Even though the internal code would still be able to determine the origin correctly from the spoofed URL, the Flash plugin treated the spoofed part as the actual domain. So, the attacker could run ActionScript code in the context of the victim page. This bug belongs to class 7.

The fix was to improve URL validation.

Takeaway

As one can see, the bug classes introduced in this document apply to other browsers as well, moreover, some of the aforementioned bugs are identical for different browsers. That leads to the conclusion that this paper might also be useful for reasoning about UXSS defenses in browsers other than Chromium.

Potential Mitigations And Countermeasures

This section presents several ideas that are intended to either detect or mitigate bugs leading to UXSS, as well as the caveats that come with these methods. These ideas are not tied to any particular classes described above and aim to be rather generic approaches.

DataFlowSanitizer Instrumentation

DataFlowSanitizer is a memory tool for a generalised dynamic data flow analysis. DFSan API allows to mark any byte in memory with a tag. DFSan will propagate the tags as the data is copied around. For some operations (e.g. addition), the tags will be joined to form new combined tags if the source operands have different tags. Then, at any point of the program execution, we may query which tags are attached to particular bytes of memory.

In theory, this approach might have allowed to ultimately track which origins the objects are associated with, and perform access checks based on the DFSan tags assigned. However, this approach seemed to be too low level for that kind of bugs. It would not be possible to assign proper tags for memory bytes inside basic types such as `WTF::String` or `WTF::Vector`, as objects of those types are not aware of their associated origins or even associated frames.

Origin Sanitization Via DOM Wrappers

Another proposal that has been discussed previously is called “Per-origin Memory Protection”. A brief summary of that proposal is to associate origins with DOM wrappers and hook all entry points from V8 to Blink (and probably some entry points from Blink to V8).

As it was discussed in the proposal document, protecting a wrapper access is not sufficient to prevent UXSS exploits. There are many ways to exploit a UXSS without going through wrappers. Due to that, it might not be a great idea, though anyway it should be helpful for preventing many vulnerabilities leading to UXSS.

Fuzzing For UXSS

Several bug reports (such as https://crbug.com/497507 and https://crbug.com/504011) use the following approach to demonstrate a UXSS exploitation. A harmless **parent.html** page embeds **child.html** page, which performs the exploitation and eventually gets access to the parent page. The access is then demonstrated by modifying the background color of the parent page.

Assuming that there is a possibility that the cross-origin access can be obtained by executing a JavaScript code produced by a fuzzer, we can approach the fuzzing in the following way.

1) The fuzzing process operates on a pair of html files:
    - **parent.html** is a harness with a constant content that looks roughly as follows:

```
<!doctype html>
<title>Parent</title>
<body>
<script>
...
var savedState = deepCopy(window);
setTimeout(() => verifyState(window, savedState), TIMEOUT);
</script>
<iframe sandbox="allow-scripts" src="child.html"></iframe>
</body>
```

    - **child.html** contains generated JavaScript code and performs arbitrary API method calls and DOM tree manipulations. It's possible to use existing fuzzers to generate such files.

2) The `deepCopy` method creates a snapshot of the current state of JavaScript's `window` object. Since most DOM objects usually have JS wrappers that are reachable from `window`, they are also included in the snapshot.

3) After the code in **child.html** had some time to run, `verifyState` traverses through the object tree and compares it to the snapshot. A mismatch would likely indicate an SOP violation.

Another idea how to use fuzz testing to discover SOP bypasses might be applied on top of the existing fuzzers. The proposal is to verify `document.origin` at the end of every test case generated by existing fuzzers. A bug should be reported, if the `origin` value is either:

- not equal to `http://localhost:8000` for test cases served via HTTP;
- not equal to `null` for test cases opened directly from the disk.

However, as it was described above in the document, the majority of the known SOP bypasses

are logical bugs rather than memory corruption ones. Due to that, an automated generation of payloads triggering SOP bypasses would be much harder than generation of payloads triggering memory corruption errors via random manipulations done with different JavaScript objects.

Another interesting point (which explains the lack of UXSS obtained via memory corruption), is that a Code Execution exploit in the renderer process is essentially equivalent to a UXSS. The reason being that a Code Execution makes it possible for an attacker to overwrite the security checks and obtain a cross-origin access. [*Note:* this is not applicable after PlzNagivate launch].

Site Isolation

The vast majority of the vulnerabilities analyzed during the research were using cross-site frames. In such case, different origins would normally share the same renderer process. Because of that, there is a large attack surface available for SOP bypassing:

- Logical bugs in Blink, Bindings, V8.
- Extensions and other APIs.
- Memory corruption bugs leading to RCE.

Site Isolation project aims to add support for a **site-per-process** policy that ensures all renderer processes contain documents from at most one website. In early 2017, when this research was originally conducted, Site Isolation looked very promising and valuable for mitigating UXSS attacks.

Now that Site Isolation is available [*Note:* Site Isolation was deployed in desktop Chromium in the middle of 2018], a compromised renderer is not able to access another site without bypassing the sandbox restrictions, which significantly elevates the cost of an attack, as the sandbox attack surface is much smaller compared to the attack surface described above.

It's worth noting that Site Isolation still has some limitations. First of all, the Site Isolation design document has a strict definition for a site: a page's site includes the scheme and registered domain name, including the public suffix, but ignoring subdomains, port, or path. That doesn't exactly match the definition of an origin, otherwise some browser features (e.g. document.domain modification) would be extremely difficult to implement. Therefore Site Isolation will not protect from UXSS when an attacker is able to run JavaScript within the same second-level domain and scheme as the victim.

Secondly, several web features like `<img>` and `<script>` historically allow cross-origin requests. Chromium uses Cross-Origin Resource Blocking (CORB) to prevent using them as a way to leak sensitive cross-origin data. However, CORB uses content type sniffing, which might yield incorrect results in some corner cases, and only protects JSON, XML and HTML data, so it will not stop the attacker from stealing JS, CSS and media files.

Conclusion

Site Isolation is the most promising countermeasure against UXSS attacks. Not only because it is the most viable of the ideas considered above in the document, but mainly because it will break the vast majority of UXSS exploits which rely on the availability of cross-origin frames in the same process. After deploying Site Isolation in Chromium, the cost of developing a UXSS exploit will be very close to the cost of developing a full RCE exploit, including the sandbox escape, which is considered to be the most complicated and expensive component of an exploit chain.