chromium

A Case Study: Chromium Vulnerability History

Kayla Davis

Shayde Nofziger

Kayla Nussbaum

Kelly Trainor

Table of Contents

Chapter 1: Domain and Historical Analysis

Product Overview

Description

Chromium describes itself as an "open-source browser project that aims to build a safer, faster, and more stable way for all Internet users to experience the web". Originating back in 2008, Chromium is the base of several web browsers — most notably Google Chrome, but also many other browsers, including Opera (version 15.0 or higher). Chromium is built as a base for browsers geared towards every web user — from developers to the average web surfer. Since it is the base of Chrome, many people use Chromium without even realizing it; as of September 2016, 58.75% of web users use Chrome. The main business objective of Chrome, and by extension Chromium, is to continue to grow their market share and maintain the title of most used browser.

The Chromium project is a large open-source product, consisting of over 14 million source lines of code in 32 languages. It is maintained and constantly updated by its large developer community and releases every 1-2 months. The Google Chrome primary development team relies on Chromium's large development community for assistance in maintaining the code base and all of its features. Volunteer-driven contributions are mainly in the form of testing and bug reporting. In the past year, Chromium has averaged around 700 developers and 6,000 commits per month.

Chromium developers work hard at creating a well documented open source product. There are many documents that help a user get started in developing, testing, reviewing, and committing their code. In order for every user to commit good code, the project enforces a strict policy that every commit to the master code branch must be reviewed and accepted by an owner prior to being integrated with the rest of the system. Even though every commit needs to be reviewed by owners, many bugs and vulnerabilities can sneak into the system. There's evidence to show that the code review system doesn't actually help mitigate vulnerabilities like one might think. In Chromium, files reviewed by more developers are more likely to be vulnerable.

Over the years, many vulnerabilities have been introduced into the system. However, Chromium has still maintained a strong reputation in both its confidentiality with user data and the integrity of the program altogether.

Overview of findings

As the capabilities of web browsers expand, it is important for projects like Chromium to continue to protect against exploits and patch vulnerabilities soon after they are discovered. Even though the project contribution process helps reduce bugs that could crawl into the system, vulnerabilities can still be introduced. According to the issue tracker, there have been 1057 vulnerabilities marked as fixed in the Chromium project, and this paper will discuss four of them. The analysis of the four vulnerabilities below will cover the root cause of the vulnerability, how it was fixed, and recommended engineering practices that could have prevented the vulnerability in the first place.

Product Assets

Chromium has many assets that could potentially be at risk to security attacks. The project is a base for many browsers so there is a substantial risk if an attacker were to compromise the system. One of Chromium's biggest assets is the user data: user profile, user cookies, user preferences, and browser history. It's important to keep these assets secure to maintain the user's trust especially since, most users automatically trust their browsers. Most assume the only danger comes from emails and malicious websites. Not protecting this asset could be detrimental to the project. Another one of Chromium's main assets is the ability to add extensions. The extensions are third party software and while the Chromium team does not maintain the extensions, they are responsible for how they can affect the system. This opens the door for third party software to affect the browser. When not properly handled, extensions could be a potential risk to the system, since many of them access important browser functionality. Allowing extensions these privileges can cause denial of service attacks, cross site scripting, and they can even download malicious software onto a user's machine. In the first vulnerability analysis below, trusting a powerful extension caused a substantial security risk.

The Chromium project enables web applications to provide richer functionality and therefore needs to protect its users against web-based attacks. However, Chromium is just as susceptible to phishing, cross-site scripting and other web-based exploits as any other browser on another platform. A browser is a link between a user's environment and all web services. This information can be invaluable to a hacker. Because of this, it's important that Chromium keeps the browser and its processes secured in a sandbox, not allowing access to the system that's using it. If an attacker could escape this sandbox, it

would be up to the OS to mitigate security risks. Additionally, if a user's browser directory is stored on a server then the server could be compromised and then anything in the cache and history is in the hands of someone else. This isn't as much of a problem for a personal computer but if the device is connected to an enterprise system then that cached data and history can contain sensitive information.

Chromium considers two different kinds of adversaries in their threat model, an opportunistic adversary and a dedicated adversary. An opportunistic adversary is an attacker that does not target any specific user or enterprise for an attack, but rather they deploy attacks that lure users to websites that compromise their machines or to apps that try to gain unwarranted privileges. On the other hand a dedicated adversary may target a user or enterprise for an attack and is willing to steal devices to recover data or credentials. They will do anything that an opportunistic adversary may do.

Example Attacks

A web browser provides three different attack routes: An attack can target the user of a web browser, the web browser program itself, or a web application running within the web browser. With this in mind, we can explore possible vulnerabilities that could be made use of to compromise the security of a web browser application, such as Chromium.

A good web browser tries to protect the user from social engineering attacks, such as phishing. Chromium handles this by displaying warnings for known malicious sites. Without sufficient anti-phishing security, the browser could allow an actor to perform a man-in-the-middle attack and serve an unsuspecting victim a web page other than the one they believe they are accessing. From there, if the victim supplies any sensitive information, the attacker can get their personal data. This compromises user confidentiality, as well as the integrity of website data (the page a user accesses is the one they intended to).

Another approach to attacking a web browser is to use its features against it. Chromium allows for 3rd party applications, known as extensions, to be installed and run in the browser. The goal of this is to allow developers to expand on the functionality the web browser provides. Because these extensions need some level of trust to execute code, the browser must protect against the potential for exploitation of vulnerabilities in them.

Chrome ships with a Flash extension pre installed. If a vulnerability is then found in that version of Flash, an attacker could leverage it and perform code execution on a victim's machine through the Chrome web browser. Among other techniques, Chromium protects against this by running in a "sandboxed process", which would require an attacker to find a second exploit to exit that sandbox in

order to execute code on the machine. An attack in this manner would violate the integrity of the system.

One other common area vulnerable to attack via a web browser is a web application. An attacker can use a plethora of vulnerabilities, such as cross-site request forgery, or cross site scripting to compromise the confidentiality of the system. A malicious actor may attempt to get the web browser to execute Javascript code in another web app's context, potentially exposing sensitive data within that web app to a third party. By wrapping an invisible HTML frame around a real application, an attacker could intercept user input. This attack, clickjacking, would compromise the confidentiality and integrity of the system. Chromium works to defend against these issues by blocking certain cross-site-scripting attacks or by issuing a warning to users.

Vulnerability History

Given the knowledge of Chromium's known assets and vulnerabilities, the team found four interesting vulnerabilities to investigate. These range from high to low severity. See the team's notes below about each vulnerability and a detailed description of its unique history in the Chromium network.

CVE-2013-2879: Issue 252062 - An attacker can sign in a victim to his own account.

This vulnerability was first reported in a thorough writeup by Andrey Labunets on June 19, 2013. The vulnerability existed because of a chain of bugs that allowed the sign in manger to be tricked into signing in the victim to the attacker's account and execute code with full user privilege. The attacker was able to subvert a CSRF check by a XSS attack on any google.com subdomain to sign in, and then using a bug in the syncing functionality, install an extension and execute malicious code. For reporting this vulnerability Labunets received a reward of $21,500 from the Chromium team.

The Chromium team broke this vulnerability report into two separate issues on June 20, 2013. The team marked issue 252062 as high priority and later labeled it as a fix to the above CVE. The syncing bug that allowed code execution, was recognized in issue 252034 which was later labeled as a fix to CVE-2013-2868.

CVE-2013-2879 was acknowledged as fixed on July 2, 2013. Three different code reviews comprised the fix. The first about displaying a confirmation dialog for untrusted sign ins (revision 208520), the second to stop trusting the sign in process if it navigated to another url (revision 208589), and the third to fix a

regression caused by the second code review (revision 209083). The last revision happened on June 28, 2013, a few days before the fix was formally recognized.

This vulnerability was introduced when the one_click_signin_helper.cc file was first created. The developer who made this file never thought about the case where a user might want to navigate to another url during the sign in process, and thus never checked for it in their code. There were many places where this issue could be found and fixed, in the one_click_signin_helper.cc file there were 134 commits between the introduction of the vulnerability and it's fix. One of the 134 changes exacerbated the issue by allowing a confirmation window for the sync function to be skipped.

These mistakes caused an integrity and confidentiality violation that persisted in the system for 480 days (nearly 16 months). A focus of defence in depth might have completely mitigated this vulnerability, or at least stopped the confirmation mistake that made it worse. In the future developers that worked on this vulnerability should remember that XSS attacks are hard to mitigate. Sometimes, you can mitigate a XSS attack through escaping HTML characters, but that doesn't always solve the issue. In the case of this vulnerability, the attack happened when a parameter wasn't correctly validated by the OAuth 2.0 proxy; it could be tricked to load a javascript url. Developers should know it can be dangerous to always assume that urls for sign in are never tampered with.

CVE-2012-5142: Issue 160803 - Crash in History Navigation

Chromium's history was put at risk when a repudiation vulnerability was found by developer and project member, lcamtuf@google.com, on November 13, 2012. This vulnerability includes a bug which shows that Google Chrome, before 23.0.1271.97, does not properly handle history navigation. This bug allowed the opportunity for remote attackers to execute arbitrary code or cause a denial of service via unspecified vectors. This undoubtedly raises concern among the system because it compromises the availability of the history navigation feature.

Since this was a breach in access, or a denial of service, it was easily identified as a security fault that potential attackers could take advantage of. This vulnerability introduced multiple security faults in Chromium including the following specifications: confidentiality impact (there is information disclosure where system files can be revealed and assessed), integrity impact (there is a compromise in integrity because there is a complete loss of system protection), availability impact (the main resource is shut down meaning that the attacker can make the resource completely unavailable), vulnerability type (denial of service execute of code) —rating this bug with a CVSS Score of 10.0 (the highest). Several developers from chromium jumped on the issue and tried to address this with the best developer for

the job. The were 18 developers (including sherrifbot and bugdroid) who were actively involved in the discussion on the issue: addressing the fix, identifying its potential security risks that it caused, and assisted with merging and completing the fixed issue.

The fix for this vulnerability happened on November 15, 2012 (two days after it was introduced) and was completed by developer Charlie Reis (creis@chromium.org). His fix was part of Chromium's Revision 167856 and involved adding four lines of code that would supply an additional condition to ensure the transient entry is discarded on in-page navigations. Code specifically, Reis required a check for 'pending_entries' and then would ensure the code would continue to execute if true. Although the fix for this vulnerability is relatively minor, it is important to note that several bugs can arise from the depths of misusing the program.

Despite the speed of this vulnerability fix upon discovery, it took some time to make it into the next patch. It wasn't until November 30, 2012 when developer 'bugdroid1' merged the fix in in Merge 1271 and was no longer viewed as a threat to the system. Chromium's system was updated to include this merge and reflected these changes in later versions. Affiliated systems using Chromium, such as openSUSE experienced similar issues with this original bug, as they were not automatically updating to what Chromium was currently running.

On December 12 2012, Matthias Weckbecker of openSUSE reported Bug 794075 for the openSUSE system running on chromium: 23.0.1271.97 and classified this as a high severity in their system. He reports the exact known issue and requests openSUSE to fix this. Since then, openSUSE released an update (openSUSE-SU-2012:1682-1) on December 21, 2012 that fixed several affiliated vulnerabilities including CVE-2012-5142.

What is learned from the mistake made in this vulnerability as well as how others have managed their systems (i.e. openSUSE) involves very well-known software engineering fundamentals. The mistake that was made with this vulnerability arose from insufficient testing. As shown in the commit history, Reis included an extensive test to the main controller where the bug originated. Testing is a very significant engineering practice, and well crafted tests could have prevented an issue like this before it was released. Another valid fundamental that was not exhibited here is automatically keeping a system as up to date as the features it it using. openSUSE does not have automatic updating for their Chromium-reliant components and thus, experienced an issue that was already found and fixed. They allowed this bug to compromise their system for an entire month before discovering it on their own. This found bug in openSUSE (same as found in Chromium, just needed a newer patch) may have been the root action for them to even update the Chromium version they were using.

As developers are tasked day-to-day with fixing many bugs, vulnerabilities, and other faults in code, it is necessary that they realize the importance of lessons learned from vulnerabilities like CVE-2012-5142. Best software practices often include testing as appropriately in depth as needed and optimizing a system to continuously be running on up-to-date software that is also secure. If following these practices, the likelihood of vulnerabilities should be lessened and the confidentiality, integrity, or availability of software will not be compromised.

CVE-2014-3160: Issue 380885 – Cache-based SOP-Bypass for Images

On June 4, 2014, Christian Schneider reported a security vulnerability that would allow a malicious website to bypass Same-Origin Policy restrictions, load any protected image a client has access to, and return that image data back to the attacker's servers. The exploit relied on a vulnerability in Chromium that allowed specially crafted cached SVG images to bypass SOP checks. The vulnerability was fixed in the development code on June 16, 2014, and released as part of a Chrome update on July 14, 2014. Schneider was awarded $1000 for their report and a bonus $1000 for their detailed write-up and Proof of Concept.

The exploit worked by the malicious site first caching an SVG file on the client's browser. This SVG file contained a reference to the protected image using an xlink. The cached image was then loaded into an HTML5 canvas element. Because the image cache does not enforce SOP on embedded images in SVG, the dataURL can be extracted from the canvas and sent to an attacker. This vulnerability is dangerous in that it will load the image data from sites that require authentication (via standard auth or cookie auth). This protected image data can then be viewed by an unauthorized attacker, compromising the confidentiality of the end user. This exploit only worked on images which did not have anti-caching headers. More details on the exploit, including a functional example, are available here.

In order to adhere to the Same-Origin Policy, Chromium requires that all resources be checked to see if they are allowed to be accessed prior to them actually being loaded. The specific Boolean function that handles this is ResourceFetcher::canRequest(...). Inside this function, numerous cases are checked to be able to determine whether the request is cross-origin, and whether or not it should be allowed. Until this exploit was reported and the vulnerability was patched, a check did not exist to prevent all SubResource requests within a cached SVG image.

This exploit was clever, and made use of techniques an engineer can easily overlook. The vulnerability had been in Chromium since this function's creation. There are two ways this exploit could have been prevented: As the exploit is at the browser level, one could argue that it was Chrome's responsibility to check for this violation of Same-Origin Policy, and it's evident that this was overlooked for a long time.

On the flipside, developers of web applications that serve private images can prevent this exploit by including proper anti-caching headers to prohibit the sensitive image from being cached by a client's browser.

CVE-2016-5133 Issue 613626 – Credential Phishing via Transparent Authenticating Proxy Vector

Patch Eudor found a medium level security bug on May 20, 2016. The vulnerability included mishandling origin information during proxy authentication, which would allow man-in-the-middle attackers to spoof a proxy-authentication login prompt or it could trigger incorrect credential storage by modifying the client-server data stream. The main issue was that the browser asks the user for authentication for a proxy. The user would input their credentials for what they thought was a trusted HTTP connection but instead they would send in clear text their credentials to an attacker. This issue violates the privacy and trust of the user and therefore Chromium's confidentiality and integrity was compromised.

If a user were to encounter this issue they would see a pop-up window saying "Proxy Authentication Required" prompting the user for their username and password. Unfortunately the fix couldn't be as simple as tweaking the wording or the UI. Users aren't generally familiar with authentication proxies and if a user is fooled by this type of phishing attempt they could just as easily be fooled by a modified string.

The issue had to be fixed immediately to prevent these phishing attacks. The fix to mitigate this issue involved fixing the origin used in the proxy authentication prompt to use the origin of the proxy server instead of using the target origin. It also involved using the correct origin when saving proxy authentication credentials. To mitigate future issues functionality was added to indicate to the user whether or not the proxy server connection is insecure. An interstitial, or an in between page to connect to the proxy server connection, was made and the omnibox now gets cleared when showing a proxy authentication prompt. Developers should keep in mind that UI design needs to adhere to the user and browsers should tell their users when they are accessing something insecure. It's part of good engineering practices to keep the user from accidentally releasing their personal information through an insecure proxy. There were 3 developers involved in this fix and 10 files were changed across 3 patches. Patch Eudor was rewarded $1,000 for discovering the vulnerability.

Chapter 2: Design Analysis

Architecture Overview

Chromium's architecture breaks down into two main subsystems: a rendering engine and a browser kernel. At a basic level, the rendering engine is responsible for parsing the HTML and CSS received from a webpage, providing an interpreter for Javascript content, and for rendering all of these content sources on the browser. The browser kernel functions are similar to an OS kernel, but instead of managing interaction between low-level system functionality and hardware, Chromium's browser kernel manages low level browser functionality. Its responsibilities include handling multiple tabs (different rendering engine instances), managing browser state, and controlling interaction between the browser and the user's underlying file system. Figure 1 shows the way these two systems connect.

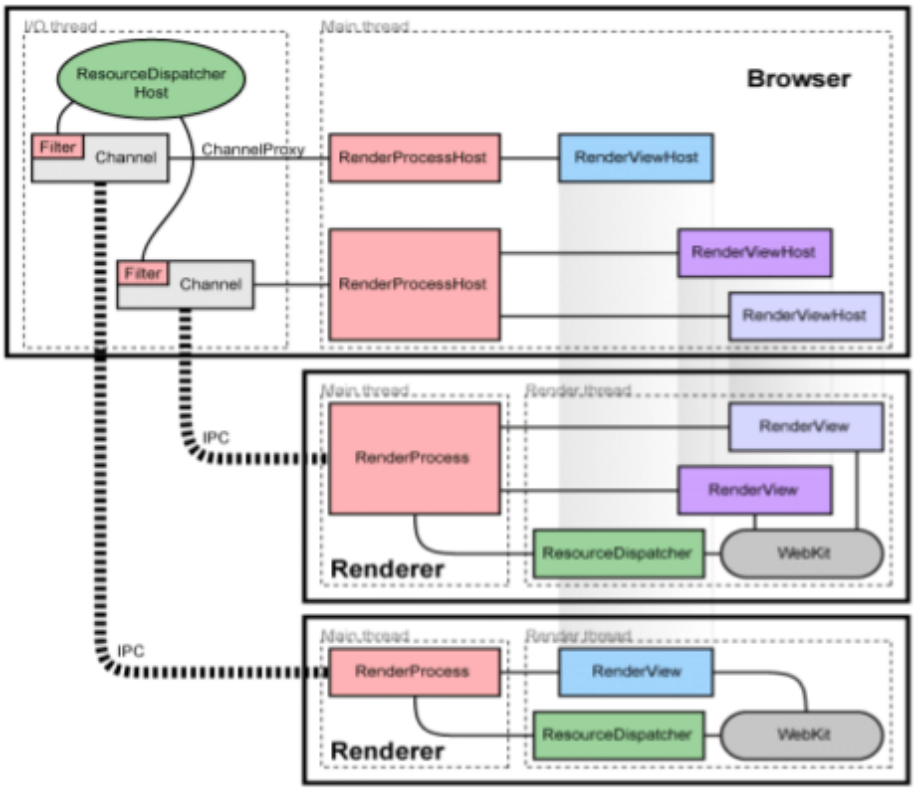

Figure 1: The browser "manages" the UI, tab, and plugin process.
https://www.chromium.org/developers/design-documents/multi-process-architecture#Architectural_overview

Chromium chose to build their product in this way in order to prioritize protecting the user from an attacker reading or writing from the their file system(page 2). Since the browser kernel can interact with the file system, they want to protect if from vulnerabilities elsewhere. They work to secure against a vulnerability in the rendering engine allowing code execution in the browser kernel. While the rendering and browser kernel subsystems are not explicitly security features, they implement security strategies. Each rendering engine instance runs inside a sandbox which bounds the process from escaping into the browser kernel and/or the user's system. It implements this distrustful decomposition, in part, because of the complexity of its features. The render is a complex part of the system that has to deal with many parsing tasks and has historically been the source of many browser vulnerabilities. To mitigate these vulnerabilities, the rendering engine sandbox adheres to two basic security design principles: defense in depth and principle of least privilege. Chromium developers are taught to assume: sandboxed code is malicious code, the sandbox should be fault-tolerant, and that it cannot rely on code emulation for security. To maintain different levels of restrictions in sandboxes, Chromium sets the sandbox configuration in a policy that knows about different levels of restriction.

The sandbox is a good step in securing the system against attackers who want to execute malicious code outside their own process. It however, hasn't always been in use for important features. As stated in Chapter One, one of Chromium's assets is the ability to add extensions (plug-ins). Chromium has offered the option for these extensions to follow the principle of least privilege, but historically (page 8) developers have usually requested maximum privileges for their extensions. This means that plug-ins could often be run outside the sandbox with full privileges. Chromium allowed this in order to not hinder extensions from being built, and to maintain compatibility with those pre-existing plug-ins. The problem with this was that attackers could exploit vulnerabilities in those plug-ins to subvert Chromium's architecture and execute malicious code on the browser and possibly the user's system. As an ongoing resolution to this issue, Chromium continues to build out their sandbox functionality to mitigate this problem: by creating a way for the browser to manage sandboxing extensions more appropriately.

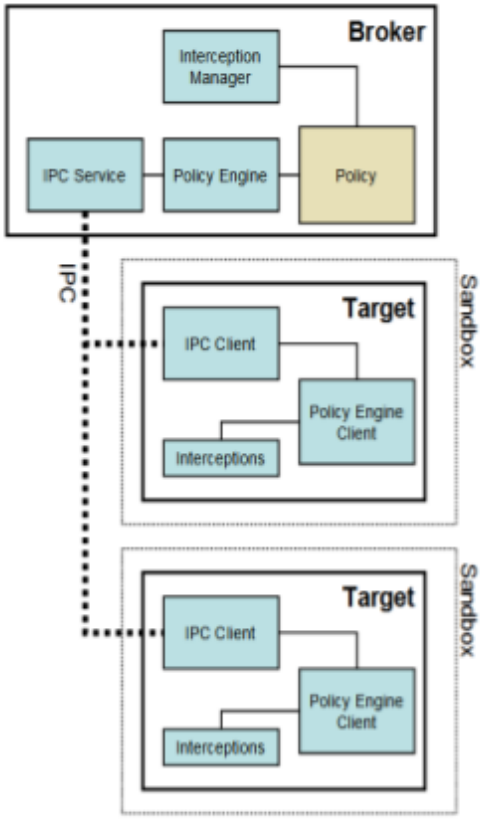

Figure 2: Most plugins will run inside a target process
https://www.chromium.org/developers/design-documents/sandbox

In addition to architecting the system in a way to protect the user from an attacker running malicious code on their system, Chromium implements features that are explicitly for security. Since Chrome 37 (released in August 2014), the WebCrypto API has been enabled by default. Chromium's implementation of the WebCrypto API can be used to perform basic cryptographic operations in web apps and has uses ranging from user authentication to the confidentiality and integrity of communications. Chromium also implements strict transport security, which allows web servers to affirm that browsers should never interact with it through the insecure HTTP protocol. Providing strict security measures impedes exploits from taking place via open protocols and other vulnerable areas in the browser. Chromium has also been able to compile a database of websites identified as malware or phishing sites and uses the Chromium UI to warn users about the site they're trying to navigate to. They have also worked to implement new features in hopes to secure against vulnerabilities like clickjacking, reflective XSS attacks, and CSRF attacks.

Chromium's documentation repeatedly reminds their developers about the importance of security. They work to design the system with defense in depth, and to remind the developer that security is a team responsibility. If a developer makes a security mistake, there are many processes in place to try to find it before it becomes a vulnerability: Chromium audits, regression tests, and fuzzes their code base. Sometimes security mistakes in a subsystem are not caught through these methods, but an active community of bug reporters with a focus on fixing security bugs help mitigate some of the possible issues.

Security bugs that are left unattended can turn into exploitable vulnerabilities which might harm the system in a number of ways. For example, the rendering subsystem is susceptible to availability problems for the user. A denial of service attack caused by the render (or even the browser kernel) can cause the user to need to kill a tab, or sometimes even restart their browser. However, availability failures tend to be a tab specific problem because of the sandbox architecture. Since this is only a local problem, however, Chromium doesn't usually recognize denial of service issues to be security vulnerabilities. Vulnerabilities in the browser kernel can cause both integrity and confidentiality violations outside the subsystem. A viable attack on Chromium's kernel could expose the user data, and any plug-in data that's stored in it. Many of Chromium's plans to change parts of the system architecture work to mitigate these problems. They plan to make changes to make complex parts of the system more simple to reduce the likelihood of adding security bugs and build a better system overall.

Threat Model

Below are several threat models that outline security boundaries for Chromium's most important assets. These include: separation of processes for each of Chromium's tabs, the creation of separate channels through an Inter-Process Communication (IPC) process, and control over extension privileges through sandbox boundaries.

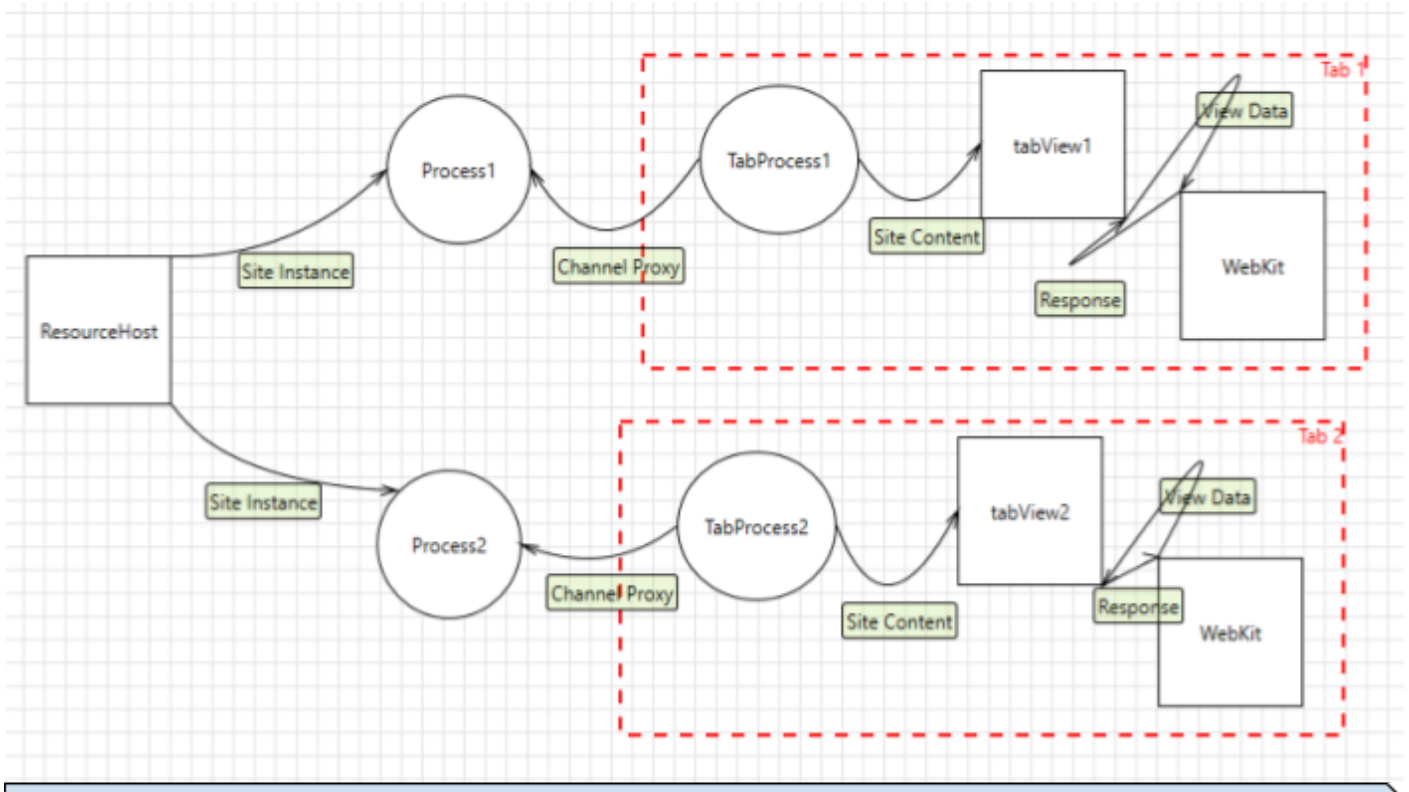

Figure 3 Threat Model of Chromium's Multi-Tab Processes

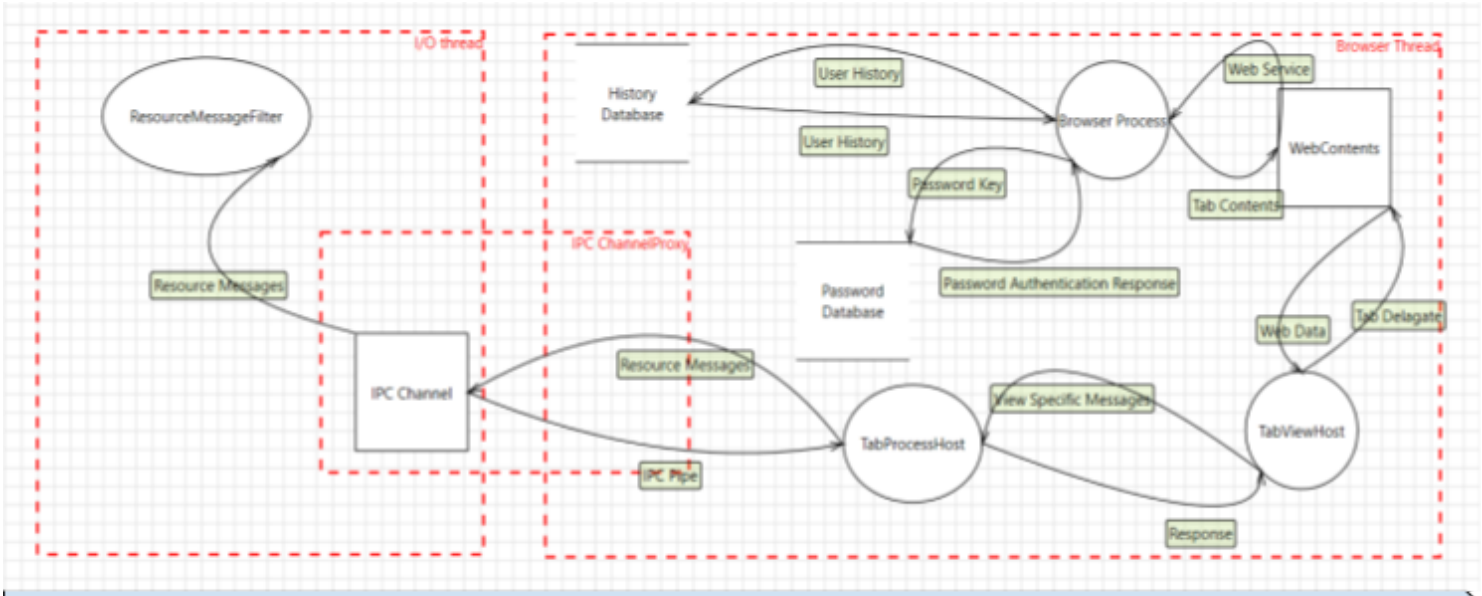

Figure 4 Threat Model to Display Web Content

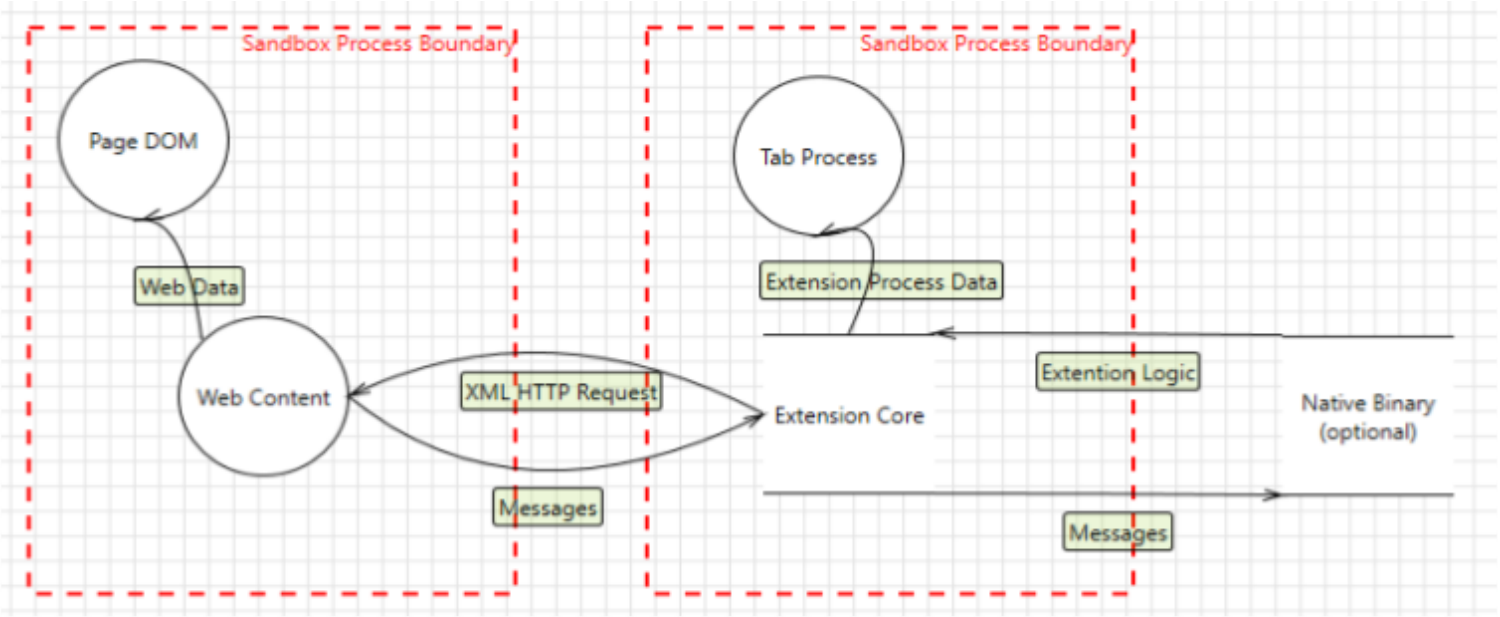

Figure 5 Threat Model Detailing Extensions

Figure 3 highlights how Chromium separates the rendering processes for different tabs. The boundaries around Tab 1 and Tab 2 restrict access from the other processes and the rest of the system. This helps with memory protection and access control.

Figure 4 outlines how web pages are displayed. The I/O Thread contains all of the IPC communication and handles all of the network communication in order to prevent it from interfering with the user interface. The I/O Thread connects the Resource Message Filter to a network layer and extends the boundaries of the user's machine. The IPC Channelproxy boundary acts as a gateway between the I/O Thread and the Browser threat to further secure the IPC pipe.

The Browser Process boundary isolates the Web Content and sensitive user data such as browser history and user passwords. The Web Content represents the contents of a web page, this is very important to the browser system and sets the Browser Process boundary at a higher priority than others. Figure 5 accentuates how the extensions interact with the browser. The Sandbox Process Boundaries limit the overall privileges for each extension and divides the extension into 3 components: content scripts (Web Content), an extension core, and a native binary. For an attacker to gain user privileges they would need to forward malicious input from the content script to the extension core and then from the extension core to the native binary, where the attacker would then need to exploit a vulnerability. Chromium offers different deployment options for extensions depending on the user's environment. User's running OSX or Linux can install extensions via a preferences JSON file (different link) and user's running Windows can install via the Windows registry (different link).

Assets to Threat Model Tracing

These sandboxes described in the architecture overview create a boundary between the web page and the other parts of the browser. Operations that require additional functionality, such as accessing the assets mentioned in chapter one including cookies, the clipboard, and user history require interacting with the browser kernel, which maintains unique privilege information for each sandbox, detailing what it may (and may not) access. By placing this sandbox on the edge of the browser, Chromium intends to decrease the probability of an exploit against the system. Each sandbox has its own rendering engine and set of permissions for the files it may access. If the TabProcess wishes to do more than just alter the state of the DOM, it must communicate through a Channel to the browser kernel API, which determines what actions it may take, and executes those that are allowed.

Due to the sandboxed tabs discussed above, Chromium needs a way to handle input and output from the browser process to the outside world. Network requests, mouse and keyboard inputs, and other external communication messages need to be routed to the correct sandbox to ensure the protection of another asset, user data. Chromium manages this through an Inter-Process Communication (IPC) process. In brief, this process handles the transfer of messages from external inputs to their respective

webpage sandboxes. It ensures that messages aren't sent to unintended TabProcesses, and that user data isn't leaked. Figure 4 shows how these two processes communicate with each other.

Separating the I/O handler from the rendering process ensures that faults in an I/O process do not cause the rendering engine to hang or otherwise affect performance. Additionally, the IPC process reduces the probability of an exploit by preventing messages from being read by TabProcesses other than the one they were intended for. Were this not in place, a maliciously crafted website could take advantage of a bug in the rendering engine and retrieve information from keyboard input, network traffic, or mouse clicks the user makes in a different tab. This special handling of I/O messages reduces the probability of that exploit occurring.

Chromium allows users to expand on the functionality of Chromium through the installation of 3rd party extensions. While Chromium cannot reasonably control the content of an extension, or the specific functionality it may provide, it can protect the other parts of its system by separating extensions into three sandboxed processes: the Content Script, Extension Core, and Native Binary. Chromium gives each of the three processes different permissions and levels of access to the system. Native Binaries are the only process allowed to interact with the host machine with full user privileges, and can only receive messages from the Extension Core. The Extension Core has privileges to the TabProcess, but can only interact with the DOM / web content through XMLHttpRequests and through Content Scripts. Content Scripts may manipulate the DOM of a single web page, but their methods of communication to other processes is limited to sending messages to the Extension Core. These communications are detailed in figure 5.

This extension architecture reduces the probability of an exploit by using the strategy of least privilege. By separating functionality into three different processes with unique permissions, each process is extremely limited in what it can do. The least secure component in this architecture, Content Scripts, do not have the ability to communicate directly with Native Binaries, which are the only process that allows arbitrary code execution. For an attacker to successfully execute code on a user's machine, they must first exploit the Content Script to forward a malicious message to the Extension Core. From there, they must find another exploit in the Extension Core to send *that* malicious message to the Native Binary. From there, the attacker must exploit a vulnerability in the Native Binary to execute code on the host machine. In order for an extension to successfully escape the sandbox, a hacker must be able to exploit vulnerabilities in three separate processes.

Chromium's assets are well organized in its sandboxed architecture. It was engineered from the beginning with security in mind. The threat model above highlights some key pieces of this design.

Through a combination of defense in depth strategies and strict adherence to the principle of least privilege, Chromium makes it extremely difficult for an exploit to break out of its sandbox and gain access to or affect other processes.

Chapter 3: Code Inspection Assessment

Inspection Selection

For the code inspection we chose files that had promising content, had a history of vulnerabilities, and could be related to our product assets. The App_Window.cc file was chosen because it connects two very important product assets, the Web Content that gets displayed in the web browser and any extensions that get attatched to that application window. The StyleElement.cpp file was chosen because it contains sensitive information that could turn into potential vulnerabilities if attacked. The Encryptor.cc file was chosen because it contains encryption algorithyms which interact with sensitive user information, which is one of Chromiums most important assets.

Looking back on our reasoning for choosing these files we probably would have chosen differently. The Encryptor and the StyleElement files had promising code inspections but the App_Window file fell short. When choosing files for code inspections it's important to keep in mind that longer files don't always yield any results. It's more important to find files that have a history of vulnerabilities or have any open or recurring issues. Files that work with any product assets or sensitive information should also be considered for code inspections.

Code Inspection Results

This section provides information about several files in Chromium as well as highlighted vulnerabilities and issues that have brought security concerns to attention. These source files are responsible for parsing and corresponding to style sheets, manipulating the display of Chromium browser view windows, and encryption and cipher handling. Each file has their own set of security concerns and analyzed vulnerabilities that have been listed in a chart below each section.

StyleElement.cpp

This file reads a given HTML element and parses its corresponding stylesheet to check it for styles that needed to be updated on the renderer's version of the stylesheet . If there are styles that need to be overridden in order to not inherit from the element's parent, the rendering engine's version of the stylesheet is updated. The file resides in the rendering subsystem, which is an area of the system that

has been a source of many past vulnerabilities. As mentioned in Chapter 2, this subsystem is sandboxed so malicious attackers can not reach the assets outlined in Chapter 1. This means the only asset this source code touches is the integrity of the displayed HTML styling. If a developer made a mistake in this file, the way webpages render styles could look different than intended. If a developer allowed write access to the Document Object Model from this file, bigger problems, like XSS attacks could arise.

Line Number	Severity	Description	Issues/Bugs	Fix?
79-83	High	Denial of Service via application crash or other possible unspecified impact through crafted JS code that triggers tree mutation	CVE-2014-1743 Use-after-free vulnerability in the StyleElement::removedFromDocument function in core/dom/StyleElement.cpp in Blink, as used in Google Chrome before 35.0.1916.114. Compromises the availability and integrity of the system.	Fix is supplied by adding an additional check to see if the user is registered as a candidate. If not, they assert a warning. High security label → $3,000 Bounty Awarded for fixing bug
149-157	High	Multiple unspecified vulnerabilities in Google Chrome before 27.0.1453.93 allow attackers to cause a denial of service or possibly have other impact via unknown vectors.	CVE-2013-2836 Various fixes from internal audits, fuzzing and other initiatives. Has been connected to many other issues including the above vulnerability. Affects the function `StyleElement::clearSheet / ElementRuleCollector::collectMatchingRulesForList`	This bug was attached to several issues and one CVE. It was found detected by AddressSanitize. This bug is tracking several already fixed issues.

App_Window.cc

The App_Window.cc file contains the functionality responsible for manipulating and displaying the Chromium browser view windows. It contains functions for initializing new browser windows, entering full-screen mode, and determining / saving the current state of the Chromium application. The primary asset that this code is responsible for is the window GUI state. It must determine whether certain actions are allowed to be taken under certain conditions, such as when a window may enter full-screen mode. Additionally, as each browser window / tab runs in its own process, App_window is responsible

for ensuring that its process is terminated when the main application is closed. App_window lies within the extensions subsystem, providing an API for extensions to use to manipulate the browser and get information on its state.

There are two "TODO" sections in this codebase that indicate potential security concerns. The first is that there are no tests associated with entering full screen mode on MacOS. This is an area of concern because Chromium has acknowledged bugs and vulnerabilities in this functionality in the past. If the code continues to be left untested, they could miss potential issues in the system.

The other area for potential vulnerabilities in this code has to do with when Chromium believes it is valid to open a "file chooser" window. A developer reported a bug showing that file chooser windows would crash if they were selected from a "panel" window type. The fix for this was simply to check if the window was a panel and if it was, don't attempt to open the file chooser. With this fix, the team explicitly stated that there was more work left to do, but the issue was left at that and hasn't been re-visited since September 23, 2015.

Line Number	Severity	Description	Issues/Bugs	Fix?
920-926	Low	If a panel window is created and the user accessess a page that's a sandbox page or otherwise and then tries to click on an input file button the page doesn't do what it's intended to do. The file selection dialog should appear but instead nothing happens. This functionality worked before in the Chrome 23.x release version. This issue has been open for 3 years.	Panel windows do not work with input type file	The issue should be re-evaluated by the developers to see if it's worth fixing for the current version.
632-642	Low	This functionality has not been fully tested because the developers cannot recreate a 'user gesture'. A user gesture is a touch event that can either be a Press, Move, Release, or Cancel.	ExtentionFullscreenAccessPass test fails due to no user gesture	Properly simulate a 'user gesture' to fully test out this code

Encryptor.cc

This file contains Chromium's encryptor implementation. It has functions built for block cipher encryption. The functions in this file implement the encryption (plaintext to ciphertext) and decryption (ciphertext to plaintext) functionality using an AES key. The file also implements a 128-bit counter for use in AES-CTR encryption. The code for this file lives in the browser kernel subsystem with other crypto functionality that Chromium added. Anything that is encrypted with this encryptor is affected by this source code; that means assets like user passwords and cookies are affected by the file. If a developer made a mistake in this part of the system the Encryptor could be at risk. Breaking encryption of user data could be detrimental for user confidentiality.

Line Number	Severity	Description	Issues/Bugs	Fix?
33-41	Medium	Direct Privilege escalation by using an OpenSSL utility file	On destruction this class will cleanup the ctx, and also clear the OpenSSL. This Could affect the integrity of the program because it does a hard reset of the the OpenSSL	Unknown and Unidentified as an Issue
66-70	Low	Possible improper handle of an overflow	Comments and code reveal that if there is an overflow, they increment the value that is most significant at the point in the system. Does not reveal what is classified as "most significant" and is unknown if the system is aware of this behavior	Unknown and Unidentified as an Issue
69-76	Low	The header file (encryptor.h) specifies a warning of how to use the decryption authentication. The logic for this is unclear and would be used incorrectly had not been warned via warning	// WARNING: In CBC mode, Decrypt() returns false if it detects the padding in the decrypted plaintext is wrong. Padding errors can result from tampered ciphertext or a wrong decryption key. But successful decryption does not imply the authenticity of the data. The caller of Decrypt() must either authenticate the ciphertext before decrypting it, or take care to not report decryption failure. Otherwise it could inadvertently be used as a padding oracle to attack the	Unknown and Unidentified as an Issue

			cryptosystem. `bool Decrypt(const base::StringPiece& ciphertext, std::string* plaintext);`	
72-73	Medium	Incorrect Return Type for boolean value Encryptor	Comments reveal that the devs left this file with tasks to still complete: "//*TODO*(hclam): Return false if counter value overflows." The code snippet actually returns `true` & provides a false positive in the system. This could become corrupt or abused by the way the system is handling the boolean (misuse of the intent of the system)	Unknown and Unidentified as an Issue
138-174	High	Unused private functions have been removed. This is concerning with availability.	Issue 2218903002: These functions were standalone functions that were not commented out during production. This fix happened 4 months ago so there is potential that there is a use case that has not been hit where these two functions are actually being used by the system in some way.	Removed functions with minimal testing to see affected components. Resolved commit message (Closed)

Project-Specific Checklist

Domain Specific Concerns

- **XSS (client and server side)**: Being a web browser, Chromium is in danger of these attacks. These sorts of attacks can often be used to bypass checks and maybe even authentication.
- **Failing to protect network traffic and information leakage** : As a web browser, Chromium needs to protect the data it interacts with.
- **When parsing input, be mindful of it**: The rendering engine is where most of the parsing happens and is also a source for many of the vulnerabilities in the system.

Table I
THE SECURITY FLAWS FOUND AND MISSED IN THE CHROMIUM PROJECT BY MODERN CODE REVIEW, MANUALLY CLASSIFIED ACCORDING TO THE TAXONOMY OF HOWARD *et al.* [23]

Security flaw	Found by code review	Missed by code review
XSS (client side)	15 (21%)	12 (10%)
C++ catastrophes	8 (11%)	33 (29%)
Buffer overruns	6 (8%)	18 (16%)
Too much privilege	5 (7%)	5 (4%)
Information leakage	5 (7%)	1 (1%)
Race conditions	4 (6%)	5 (4%)
Format String problems	4 (6%)	3 (3%)
Catching all exceptions	3 (4%)	0 (0%)
Failing to protect network traffic	2 (3%)	4 (3%)
Integer overflows	2 (3%)	3 (3%)
Use of Magic URLs, predictable cookies	1 (1%)	1 (1%)
Use of weak password-based systems	1 (1%)	1 (1%)
Command injection	0 (0%)	7 (6%)
XSS (server side)	0 (0%)	1 (1%)
Other	15 (21%)	21 (18%)

Figure 6: Security Flaws
http://sback.it/publications/scam2016.pdf

Coding Mistakes

- C++ Catastrophes: Chromium is written primarily in C and C++ , so it is in danger of these sorts of language specific issues:
    - Failure to initialize a pointer
    - Failure to reset pointers on deletion
    - Bad copy, constructor, destructor, and assignment functions
    - Array new and delete mismatches
    - Ignoring compiler warning settings
- **Integer overflows**: This vulnerability is important to watch because, if combined with a malloc, it can cause buffer overflows.
- **Buffer overflows**: Data shows that this security flaw is often missed in code reviews, but it needs to be checked since most of the system is written in C or C++. This mistake can cause availability problems in the browser.
- **Command injection**: This vulnerability has been in the OWASP top ten most critical web security risks and is also often missed in Chromium code reviews.
- **Properly handling exceptions:** Resources need to be properly disposed of in the exceptions.

Design Concerns

- **Too much privilege**: Developers need to be careful of what permissions they run processes at. If this is broken, the sandboxing of the system might be bypassed. This can cause an opening for arbitrary code execution vulnerabilities
- **Validate design and implementation of new features in the rendering engine** to make sure that the sandbox is not broken. Remember, this is important; it has the highest reward payout for bug reports.
- **Validate the design in concurrent parts of the system:** This is important to mitigate the issue of race conditions.

Past Vulnerabilities

The checklist mentioned above outlines many of the design, domain, and coding mistakes that cause the issues seen in Figure 7. If prioritization was needed for the checklist, it might be advantageous to focus on the types of security mistakes that cause the most vulnerabilities. This means that developers should be aware of handling exceptions, race conditions, and buffer and integer overflows. Focusing on these issues could remove the two biggest causes of vulnerabilities: denial of service and overflows.

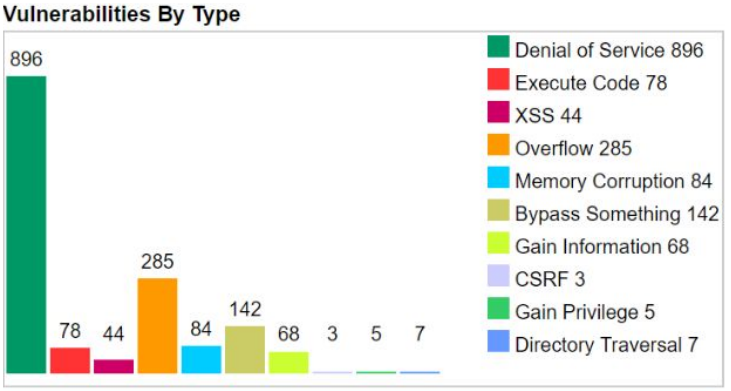

Figure 7 Type Distribution of All Vulnerabilities
http://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224

Availability Concerns

As seen in Figure 7, denial of service attacks are the biggest cause of vulnerabilities in the system. Handling the issues mentioned in Past Vulnerabilities should work to mitigate many of the availability issues. It's important to keep in mind that Chromium does not consider minor (local) availability issues as vulnerabilities, and thus it might not be as important to mitigate some of these concerns for the Chromium team.

Inspection Summary

Our inspection of the two files - app_window.cc and encryptor.cc - revealed some potential areas of concern for the Chromium project. Many of the code files contain “TODOS” or other comments indicating that certain functionality has not been implemented. Additionally, many of these comments center around known issues in the system, and reference open bugs reports. While Chromium does a good job of identifying issues, it is apparent they do not always get around to implementing the changes to fix them. An example of this is the App_Window.cc file, wherein a bug report from 2015 indicated an issue with a file chooser dialog. A temporary fix was applied, with a more permanent fix mentioned - only in the code comment - as a TODO. In the year since this was created, no visible effort has been made to implement the permanent fix.

An analysis of the StyleElement.cpp file revealed that the only asset that gets touched is the HTML styling. However, if the style sheets were written in a way to access the Document Object Model then they could be liable for Cross Site Scripting. This was mentioned in this CVE issue. The fix involves an authorization check and a $3,000 bounty was awarded for fixing this bug. It is important to fully inspect all files, while style sheets aren’t normally prone to security vulnerabilities an issue like this one could have put the system’s availability and integrity at risk.

Overall we believe that Chromium has a strong process for handling security and vulnerabilities in their design, as well with their contribution requirements in the open source community. Where Chromium could improve is in their communication after discovering root causes of vulnerabilities and identifying the necessary fixes. The team has a pattern of implementing quick / temporary fixes that address the problem at hand, but are still not considered to be the “best fix”. When this happens, developers often provide comments in the code detailing what a permanent fix would look like. Unfortunately, many of these comments are rarely revisited in the future, and many remain unimplemented.