Weeks of December 14 and January 21

Goals for this Week

1. Retry replicating CVE-2019-11708
2. Read more about firefox internals like their sandboxing methods and their IPC layer.
3. Attempt to follow Firefox's methodology for building their fuzzer.
4. Start taking a look at the source code (Haven't been able to work on this for the past couple of weeks. It will probably be recurring for a couple of weeks since it's also a big codebase)
    a. Understand IPC layer code
    b. Start making a list of things outside sandbox that are reachable by the IPC (filesystem, etc.)
5. Start working on the journal if possible
6. Try to work on Grizzly and attempt to get a basic fuzzer working
7. Finish journal
8. Keep working on Grizzly
    a. Try to define custom interfaces
    b. Test custom interfaces

My Research and What I Learned

For the past three weeks, I researched fuzzing, more internals about Firefox, and even how to get fuzzers working with Firefox. In addition to that, I was able to further my knowledge about how exploitations could work by successfully replicating CVE-2019-11708 (an older sandbox bypass). My research into how Mozilla fuzzes their browser also led me to find other exploits/bugs that fuzzers had found.

Firefox Web Security Checks

One of the aspects I researched was how Firefox internally ensures some common web security checks like the Same-Origin Policy. The same-origin policy attempts to protect websites and users by making sure that a website doesn't use resources from completely different domains. If a website needs to access resources from different domains, it must employ something called Cross-Origin Resource Sharing (CORS). The primary source I consulted was an article by Christoph Kerschbaumer and Frederik Braun titled "Understanding Web Security Checks in Firefox." According to Kerschbaumer and Braun ", all web related security checks within Firefox are evaluated based on the security concept of a `Principal`." This means that every website has some sort of security context which is reflected by a `Principal`. A security context essentially informs Firefox what kind of web page it is and the permissions it has. There are four different types of Principals: `Content-Principal`, `Null-Principal`, `System-Principal`, and `Expanded-Principal`. A `Content-Principal` encapsulates

the origin information of a document and reflects the security context for all web-hosted resources. A `Null-Principal` is used to represent an opaque origin which is for items like iframes, documents loaded with a data: URI, and websites that are not the same origin. A `System-Principal` is used for the browser's user interface, i.e. pages like "about:support"; essentially, a `System-Principal` is used for internal browser pages. Lastly, the `Expanded-Principal` is used for browser extensions (Kerschbaumer and Braun).

There was also a lot more in-depth detail about how Firefox uses the principles and decides whether or not to load a page, however, I decided not to go further. The reason why I began to read this page was that it was linked in the article "Examining JavaScript Inter-Process Communication in Firefox" for background information. The information present in the article will prove extremely useful if I can find a bug that allows me to open custom URLs like in CVE-2019-11708. As a matter of fact, in my last journal, I said that I was unable to get CVE-2019-11708 working. I re-tried under different conditions and was able to successfully replicate it which greatly helped me in understanding how sandbox bypasses would work, especially after analyzing the vulnerable code. As a reminder, the CVE works due to an insufficient vetting of URLs when opening prompts.

```
var RemotePrompt = {
  // Listeners are added in BrowserGlue.jsm
  receiveMessage(message) {
    switch (message.name) {
      case "Prompt:Open":
        if (message.data.uri) {
          this.openModalWindow(message.data, message.target);
        } else {
          this.openTabPrompt(message.data, message.target);
        }
        break;
    }
  },
```

Screenshot 1 from Firefox Browser Source Code

When RemotePrompt is called (the code for it will be clear in a second), it checks if the information it received had a "Prompt:Open" value for the `message.name`. If it had, then it checks if the message's data contained a URI. If a URI did exist, it would just open a new modal window with that URI.

NOTE: A URI stands for Uniform Resource Identifier, and in this context is the same as an URL

To get to that point in code, however, you must send an IPC message from the browser tab to the browser parent. As the code comment in Screenshot 1 says, there is more code in `BrowserGlue.jsm`.

```
mm: {
  "Content:Click": ["ContentClick"],
  "ContentSearch": ["ContentSearch"],
  "FormValidation:ShowPopup": ["FormValidationHandler"],
  "FormValidation:HidePopup": ["FormValidationHandler"],
  "PictureInPicture:Request": ["PictureInPicture"],
  "PictureInPicture:Close": ["PictureInPicture"],
  "Prompt:Open": ["RemotePrompt"],
  "Reader:FaviconRequest": ["ReaderParent"],
  "Reader:UpdateReaderButton": ["ReaderParent"],
  // PLEASE KEEP THIS LIST IN SYNC WITH THE MOBILE LISTENERS IN BrowserCLH.js
  "PasswordManager:findLogins": ["LoginManagerParent"],
  "PasswordManager:findRecipes": ["LoginManagerParent"],
  "PasswordManager:onFormSubmit": ["LoginManagerParent"],
  "PasswordManager:autoCompleteLogins": ["LoginManagerParent"],
  "PasswordManager:removeLogin": ["LoginManagerParent"],
  "PasswordManager:insecureLoginFormPresent": ["LoginManagerParent"],
  "PasswordManager:OpenPreferences": ["LoginManagerParent"],
  // PLEASE KEEP THIS LIST IN SYNC WITH THE MOBILE LISTENERS IN BrowserCLH.js
  "rtcpeer:CancelRequest": ["webrtcUI"],
  "rtcpeer:Request": ["webrtcUI"],
  "webrtc:CancelRequest": ["webrtcUI"],
  "webrtc:Request": ["webrtcUI"],
  "webrtc:StopRecording": ["webrtcUI"],
  "webrtc:UpdateBrowserIndicators": ["webrtcUI"],
},
```

Screenshot 2 from Firefox Browser Source Code

As you can see from the screenshot above, if you run `sendSyncMessage` with `Prompt:Open`, it will call the code from Screenshot 1. Therefore, if an attacker had gained control of a browser tab, then they can send whatever IPC message they want, with whatever data they want. This means that they can also pass in whatever URL they want when sending

```
var RemotePrompt = {
  // Listeners are added in BrowserGlue.jsm
  receiveMessage(message) {
    switch (message.name) {
      case "Prompt:Open":
        if (message.data.uri) {
          this.openModalWindow(message.data, message.target);
        } else {

          this.openTabPrompt(message.data, message.target);



        }
        break;
    }
  },
```

the IPC message. If the attacker controls the URL then they can call whatever malicious code they want the browser parent to execute.

Screenshot 3 From Firefox Browser Source Code

The above code from screenshot 3 shows the older code that was vulnerable. The following screenshot shows the fix that Firefox implemented. They removed the check for the URI in the `Prompt:Open` method and moved it to another function called `openModalWindow` with more checks on the URI.

```
var RemotePrompt = {
  // Listeners are added in BrowserGlue.jsm
  receiveMessage(message) {
    switch (message.name) {
      case "Prompt:Open":
        const COMMON_DIALOG = "chrome://global/content/commonDialog.xul";
        const SELECT_DIALOG = "chrome://global/content/selectDialog.xul";

        if (message.data.tabPrompt) {
          this.openTabPrompt(message.data, message.target);
        } else {
          let uri = (message.data.promptType == "select") ? SELECT_DIALOG : COMMON_DIALOG;
          this.openModalWindow(uri, message.data, message.target);
        }
        break;
    }
  },
```

Screenshot 4 from Firefox Browser Source Code

The importance of all this information is it helped me learn more about the internals of Firefox and what potential bugs in the code could look like. When I start to go through the source code, insufficient vetting of user-controlled arguments is something that I could check. In addition, my reading of the web security checks in Firefox helped me realize that even if an attacker had full access over a URL, they can't force the parent browser to load files from the operating system to gather more information. The reason why is because "Firefox ensure that its URL is not targeting the local file system with a `file://` scheme" when loading a sub-resource (Kerschbaumer and Braun). So understanding web security checks was useful to some extent because if I was to find a vulnerability related to URLs, there is some extra background information that I now know of. In addition to web security checks, I did a lot more research into fuzzing, more specifically fuzzing with Firefox and how Mozilla does it.

Fuzzing

The first thing I did was meet with my advisor, Ned Williamson, on December 20th to see if he could offer any more insight into my project. After talking to him about my attempts to replicate CVE-2019-11708, he helped me to come up with different ideas to find success. Originally, I had tried to exploit it on MacOS which could have been the reason why the exploit

didn't work. He presented me with the idea of trying it on Windows instead and running what 0vercl0ck's GitHub Repository had. 0vercl0ck's GitHub Repository contained a full-chain exploit (i.e. just browsing to a page would result in system access). In addition to that, he also gave me some more details about fuzzing and how to build fuzzers. He told me that when he was working on his fuzzer from Chrome, he built it by basing his code on unit tests and recommended I do the same. In addition, he told me about mutation-based fuzzing and how it is based on the fact that part of the input is going to be fuzzed. Lastly, he recommended I try to follow Mozilla's method of fuzzing Firefox and attempting to get it working locally. Based on those recommendations, I began my research with fuzzing.

The first thing I did was read up on LibFuzzer, so I consulted the documentation. Upon reading the "libFuzzer - a Library, " I understood that it is a way to fuzz isolated code and "is linked with the library and feeds fuzzed inputs to the library via a specific endpoint." This means that when you compile Firefox, you compile the fuzzer with Firefox. This also allows the fuzzer to feed fuzzed inputs into Firefox via a specific endpoint. In addition, the "fuzzer keeps track of which areas of the code are reached and generates mutations on the corpus of input data" ("libFuzzer - a Library"). Some further research showed that the LibFuzzer library appeared to use C/C++ code. While I wasn't able to do more direct research for LibFuzzer, the topic popped up multiple times when researching the fuzzing process at Mozilla.

The first source I read about fuzzing at Mozilla was one by Tyson Smith et al titled "Browser Fuzzing at Mozilla." According to Smith et al., there are different levels of fuzzing:

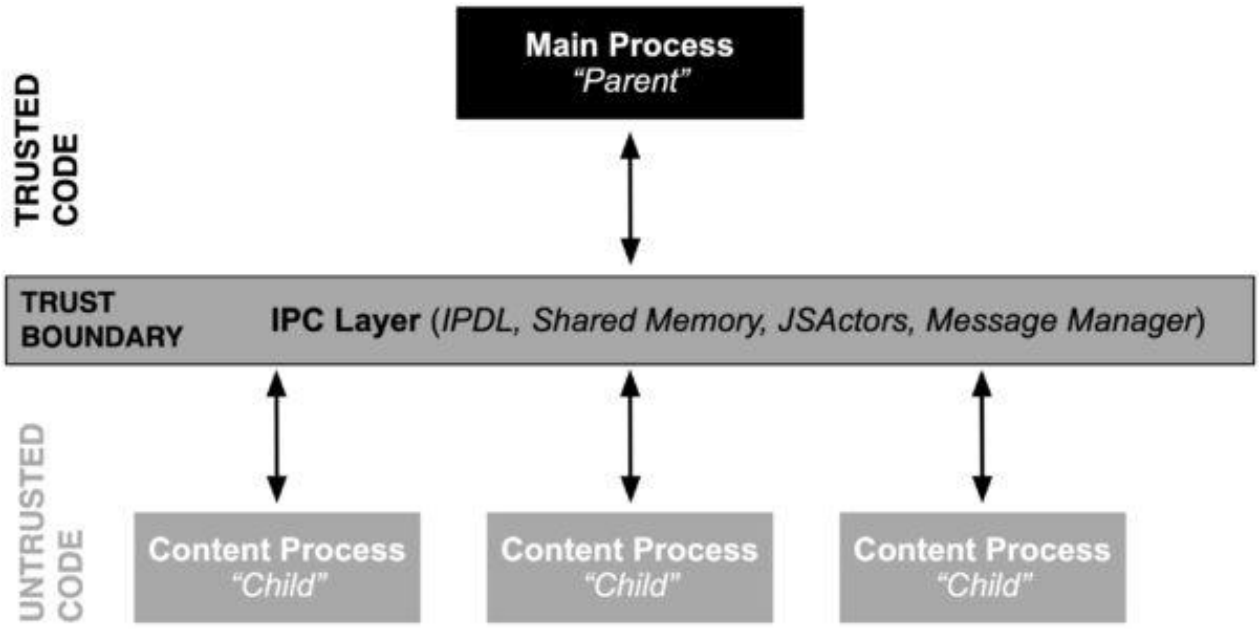
- Fuzzing browser as a whole
- Fuzzing isolated code (with libFuzzer)
- Fuzzing whole components (JS Engine)

I believed that I would be able to fuzz isolated code with libFuzzer and fuzz the IPC system itself specifically, however, later research found that would not be feasible. Mozilla uses sanitizers like "`AddressSanitizer`, `ThreadSanitizer`, and `UndefinedBehaviorSanitizer`, and debug builds that enable assertions and other runtime checks" (Smith et al.). Sanitizers help to find any memory errors, for example, the `AddressSanitizer` is "a fast memory error detector that detects use-after-free and out-of-bound bugs in C/C++ programs" ("Address Sanitizer"). Hence, using different sanitizers will help identify various bugs. It is important to note that while all the tools provide different lenses for finding different bugs, some tools are incompatible with each other, or require a custom Firefox build (compiling Firefox from the source code but with different compiler options) (Smith et al.). Some tools like code coverage and libFuzzer require build instrumentation. Build instrumentation is just a way of measuring the browser's performance and locating errors. Smith et al. also state that different "operating system and architecture combination requires a unique build and may only support a subset of these tools." Therefore, further research will need to be done on which tools are supported on macOS builds. Since there are so many different operating systems and different options for compiling Firefox, Mozilla developed a tool called `fuzzfetch` to reduce the complexity. `Fuzzfetch` will "download and unpack the specific required build and supports downloading specific revisions" (Smith et al.). This tool will be useful later down the line when I attempt to run a general fuzzer against the latest macOS Firefox build. Mozilla uses a framework called `Grizzly` for fuzzers that target the browser. According to Smith et al. "`Grizzly` manages and runs test cases and monitors for results" and is extensible by extending interfaces like Target or

Adapter (Smith). Another tool that Mozilla also uses is called the Orion monorepo which is used to encapsulate the configuration required to run each fuzzer (Smither et al.). The article continues to discuss the next steps after finding a bug. While the steps and tools in that phase will be useful, they are only useful to me if I find a bug. So, I decided to omit my research in that area here as it is not pertinent yet. While "Browser Fuzzing at Mozilla" was great for general knowledge about how Mozilla fuzzes their browser, I needed more specific information on how to fuzz the IPC layer. I was able to find an article called "Effectively Fuzzing the IPC Layer in Firefox" by Cristoph Kerschbaumer and Christian Holler. The article provided more information on the IPC layer itself as well as recommendations for fuzzing the IPC layer.

More IPC Layer Information

When Firefox launches, it internally spawns one privileged process, which launches and coordinates activities of multiple content processes (Kerschbaumer and Holler). This separation allows Firefox to be more secure since it can distinguish less trustworthy code into processes that have reduced access. The IPC layer is in charge of the delegation of operations between the content and parent process (Kerschbaumer and Holler). The layer spans several security boundaries in Firefox with the most obvious being the PARENT <-> CONTENT process boundary. The content process is unprivileged, sandboxed, and often considered compromised (Kerschbaumer and Holler). The parent process has "full acess to the host machine" and any violation of this boundary "will result in a sandbox escape" (Kerschbaumer and Holler).



Screenshot 5 from "Effectively Fuzzing the IPC Layer in Firefox"

As screenshot 5 highlights, the parent process is the only trusted code and the content processes are untrusted. The IPC Layer is the boundary between these two and utilizes the communication methods in parenthesis. `IPDL` Protocols, `Shared Memory`, and `JSActors` are

the three main communication methods with `Message Manager` being an obsolete method. `JSActors` are built on top of `IPDL` and hence are `IPDL` aware (Kerschbaumer and Holler).

IPDL

`IPDL` stands for Inter-process-communication Protocol Definition Language and is a Mozilla-specific language. According to Kerschbaumer and Holler, it is used to define the mechanism to pass messages between processes in C++ code. The `Content` protocol is the primary protocol that governs the communication between parent and child process in Firefox (Kerschbaumer and Holler). The protocol definition is very hierarchic, and several sub-protocols can be seen in the file `PContent.ipdl`. Each protocol has methods for starting each sub-protocol and provides access to the methods inside the newly-started sub-protocol. Another important file is `PBackground.ipdl` which allows for background communication. This structure means "there is not one flat attack surface containing all interesting methods" (Kerschbaumer and Holler). Therefore a fuzzing process that increases the number of message round trips to reach a certain sub-protocol will be useful in surfacing a vulnerability. Since `JSActors` are built on top of `IPDL`, having an understanding of how the `IPDL` works could be beneficial in understanding `JSActors`. Taking a glance at the source code for the `IPDL` protocols shows that the code has a lot of different `struct`s for the various different protocols. I may have to take a more detailed look at the source code later.

Shared Memory

As the name implies, Shared Memories (`Shmems`) are the main objects to share memory across processes. According to Kerschbaumer and Holler, the easiest way to find them in code is to search for `AllocShmem` and `AllocUnsafeShmem` or search all files ending in '`.ipdl`.' Kerschbaumer and Holler continue to give an example of a `Shmem` object being used in the context of `PContent::InvokeDragSession`. If you remember, the `PContent` is the `Content IPDL` protocol, and the InvokeDragSession is called to transfer drag/drop images, files, etc. In addition to Shmem, there is a ShareMemoryBasic object which is lower-level and does not use IPDL. There was a lot more information on Shared Memory objects but I felt that most of the information was not pertinent to me. I wanted to focus more on JSActors.

JSActors

Kerschbaumer and Holler state that JSActors are the “preferred way to perform IPC in JavaScript.” The `JSProcessActor` provides a communication channel between a child process and its parent. The `JSWindowActor` provides a communication channel between a frame and its parent. `sendAsyncMessage()` and `sendQuery()` are two methods for sending messages. `receiveMessage()` is for receiving messages. While all the theoretical knowledge is great, I still need to find a way to practice it practically or find practical uses for it. Kerschbaumer and Holler go on to explain Message Manager but since that is an obsolete method, I decided not to spend too much time researching that or include it here.

Fuzzing the IPC Layer

As a reminder, there were three ways to fuzz: fuzzing the browser as a whole, fuzzing isolated code, and fuzzing whole components. According to Kerschbaumer and Holler, isolating the components “has not been successful” for fuzzing the IPC layer. This is due to interesting classes, such as the `ContentParent` IPC endpoint, having very complex runtime contracts with their surrounding environment. In the end, all that means is that using the classes in isolation “results in a lot of false-positive crashes” (Kerschbaumer and Holler). Therefore, they recommend that a system testing approach is the only viable solution for comprehensive IPC testing. One potential approach they provide is process- or VM-based snapshot fuzzing. In order to VM-based snapshot fuzzing, you could "start Firefox with a new tab, navigate to a dummy page, then perform a snapshot of the parent" (Kerschbaumer and Holler). A snapshot is like a picture of the process's memory and executing code. After taking a snapshot, you could replace the regular child messages with coverage-guided fuzzing. The snapshot approach would also allow for resetting the parent to a defined state without suffering a performance bottleneck of restarting the process (Kerschbaumer and Holler). In lieu of all this, there are a couple of requirements for VM-based snapshot fuzzing:

- Callback when the process is ready for fuzzing
    - A callback is essentially a function that will inform the fuzzer when things are "ready" to start intercepting and fuzzing communication
    - An example that Kerschbaumer and Holler give is the point when a new tab becomes ready for browsing might be a good entry point for PARENT <-> CONTENT child fuzzing
    - One way to intercept the message exactly at this point might be to check for a `STATE_STOP` message with the expected URI
- Callback for when communication sockets are created
    - On Linux, Firefox uses UNIX sockets for IPC communication
    - macOS is based on UNIX, and so is Linux, which makes me think Firefox on macOS could also use UNIX sockets for IPC communication. I still need to find exact information on how IPC communication occurs in macOS
    - Kerschbaumer and Holler also provide a method to check for intercepting the creation of sockets in the parent: `ChannelImpl::CreatePipe`

My plan of attack now with this information is to attempt to use the Grizzly Fuzzing Framework to fuzz the IPC layer. I still need to figure out a way to get the Grizzly Fuzzing Framework to work with VM-based snapshot fuzzing or will have to find another tool to assist me.

Miscellaneous Information

A lot of the resources I’ve looked at would keep bringing up the idea of coverage-guided fuzzing and black-box fuzzing. So, I decided to research what the difference between the two was. I was able to find the documentation for another fuzzing framework called ClusterFuzz which had an article on coverage guided fuzzing vs. black-box fuzzing. According to “Coverage Guided vs Blackbox Fuzzing,” coverage-guided fuzzing uses libFuzzer and is also known as Greybox fuzzing. Coverage-guided fuzzing uses "program instrumentation [(tools to measure a program's performance)] to trace the code coverage reached by each input fed to a fuzz target"

("Coverage Guided"). Essentially this means that the fuzzer provides a program with some random input and checks to see which parts of the code the input reached. "Coverage Guided vs Blackbox Fuzzing" also recommends that coverage-guided fuzzing is most effective and works best when the target is self-contained, deterministic, and can execute a dozen or more times per second. Conversely, a black-box fuzzer "generates inputs for a target program without knowledge of its internal behavior or implementation" ("Coverage Guided"). A black-box fuzzer may generate inputs from scratch, or rely on a static corpus of valid input files to base mutations on. A corpus is just a large database of information. One important distinction to note is that in coverage-guided fuzzing, the corpus does grow because the fuzzer will be able to dynamically create new inputs based on where the input reaches in the code and so on. In black-box fuzzing, the corpus does not grow and remains static. "Coverage Guided vs Blackbox Fuzzing" also states that black-box fuzzing works well when the target is large, not deterministic for the same input, slow, and input format is complicated or highly structured. Since a browser is a very large target, I believe that black-box fuzzing will be the best for fuzzing Firefox. My research into the Grizzly Fuzzing Framework revealed that black-box fuzzing may indeed be the way to go.

According to Tyson Smith in his article titled "Grizzly Browser Fuzzing Framework," Firefox built the framework to help with fuzzing and automate fuzzing. It was designed to allow fuzzer developers to focus on writing fuzzers and not worry about the overhead of creating tools and scripts to run them. According to Smith, the Grizzly framework is extensible by "extending the Target or Adapter interface[s]." To be more specific, Targets are used to add support for specific browsers, and Adapters are used to add support for fuzzers (Smith). This is where I will have to do more research. I believe that Adapters will help me fuzz the IPC layer and will hopefully help me find a bug. The framework was also built to primarily support black-box fuzzers (Smith). As a result, Grizzly operates without knowledge of the internals of Firefox. Therefore, it might even be easier for me to get started with using Grizzly without needing to fully understand the internals of Firefox.

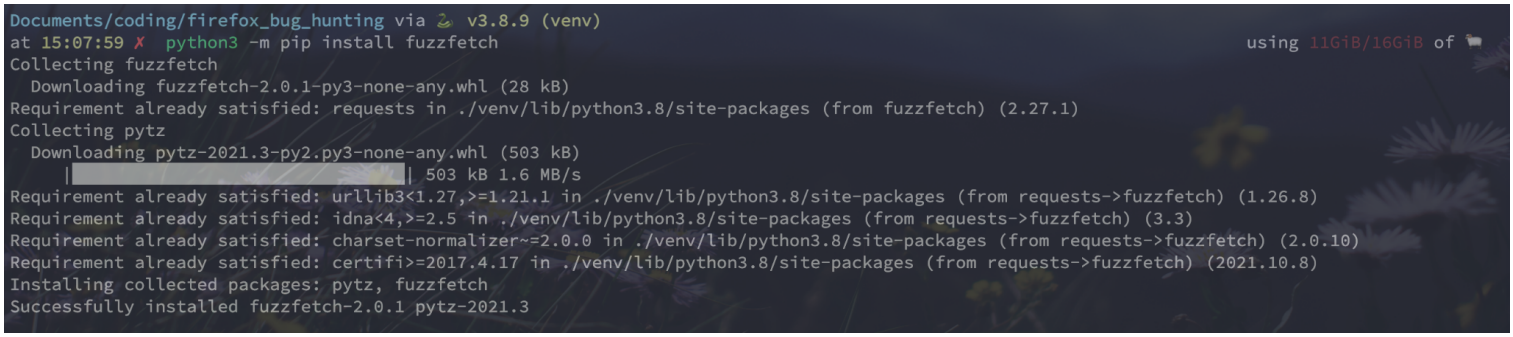
Conclusion

Although this journal doesn't contain as much research as the previous journal, the research I conducted for this journal is still very useful and will help me with fuzzing for the next couple weeks. My research helped me understand even more about fuzzing. However, there was a lot of progress in the accomplishments I've had.

Accomplishments

- My biggest accomplishment for the past couple weeks was being able to successfully replicate CVE-2019-11708
- Earlier I had tried to replicate it on macOS but this time I used a Windows installation
- I also had tried to use the general Firefox browser version, but the GitHub repository I was looking at recommended using a specially compiled Firefox browser that he had also uploaded.

- I also replicated the exploit two times. The first time, I ran the full chain exploit to see if the exploit in the GitHub repository would even work. The second time, I manually did the sandbox escape bypass to see if I understood how it works.
    - In the screen recording videos that I have linked, the first thing I do is start a web server using a Python3 module
    - The web server is just a way for me to host files and to mimic what an attacker would do.
    - After hosting the web server, I could access the files from the browser just like I would when browsing to a website
- In both videos, we know that exploit works because just browsing to a website opens up the calculator app on Windows. The browser should not be able to open the calculator app just by running some JavaScript code.
- The below link is me running the full chain exploit on Firefox
    - https://youtu.be/PlCGoBg4faE
- The below link is me attempting to manually execute the sandbox bypass. You can see me entering the JavaScript code necessary to send an IPC message asking to open a prompt. The commands I used should be explained in the previous journal
    - https://youtu.be/5siv9wPwOrk
- The next biggest accomplishment I achieved was being able to get a version of Firefox that I could fuzz and attempt to fuzz it using Grizzly
- I followed along with the Grizzly Getting Started Guide at https://github.com/MozillaSecurity/grizzly/wiki/Getting-Started
- The first thing I did was install the Grizzly framework with `python3 -m pip install grizzly-framework`
- Similarly, I also installed the `fuzzfetch` tool using a similar command



- After installing `fuzzfetch`, I downloaded a build of Firefox that was made for fuzzing with `python3 -m fuzzfetch --asan --name firefox --fuzzing -o $CODE/browsers/`
    - In the above, `$CODE` is a variable that contains my current working directory (the folder path to where I am currently in the terminal)

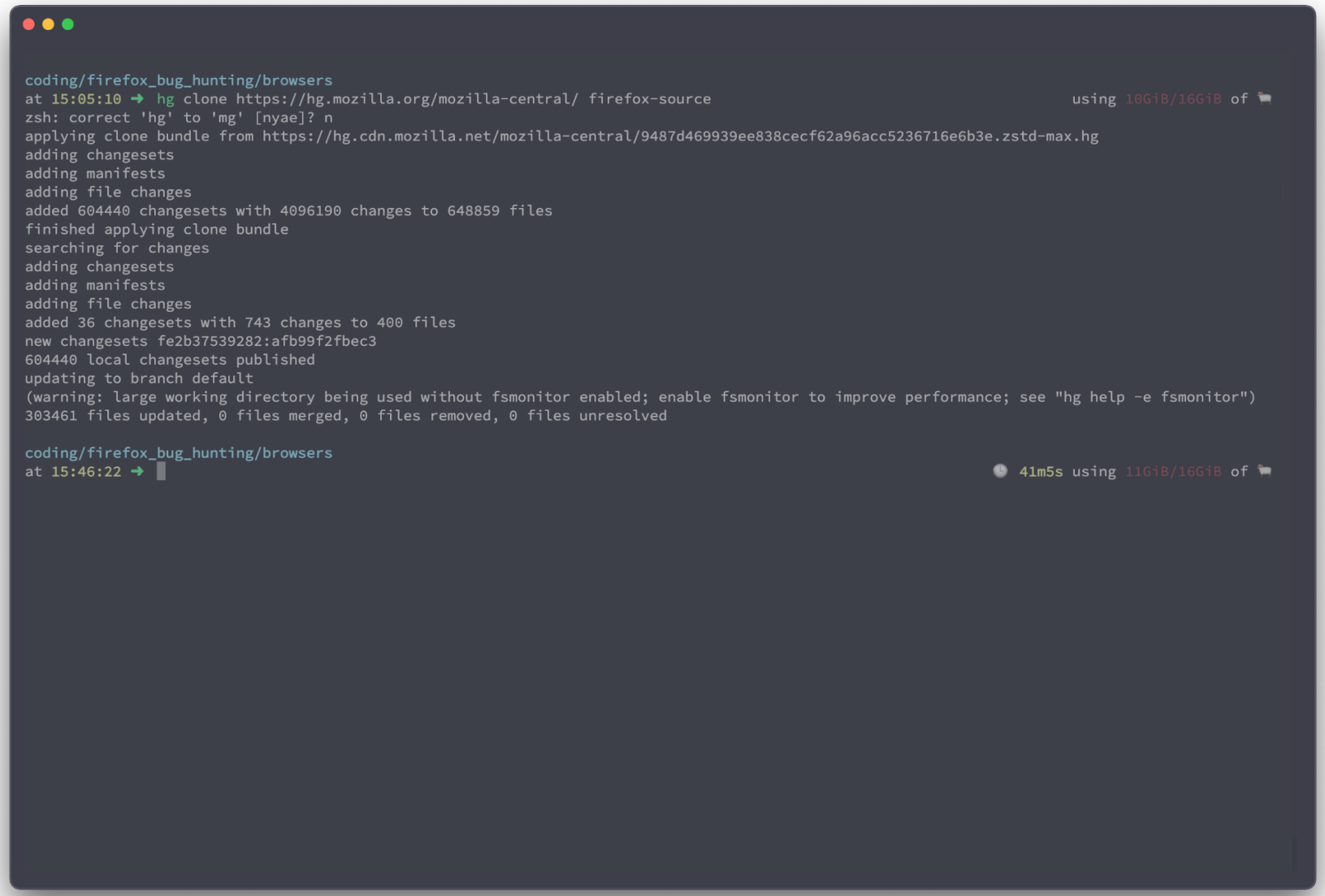
```
Documents/coding/firefox_bug_hunting via 🐍 v3.8.9 (venv)
at 15:12:17 ➜ python3 -m fuzzfetch --asan --name firefox --fuzzing -o $CODE/browsers                    using 11GiB/16GiB of 🐏
[2022-01-11 15:12:39] Identified task: https://firefox-ci-tc.services.mozilla.com/api/index/v1/task/gecko.v2.mozilla-central.latest.firefox.macosx64-fuzzing-asan-opt
[2022-01-11 15:12:39] > Task ID: BqusNjODRVWfuw9UeZzdnQ
[2022-01-11 15:12:39] > Rank: 1641936775
[2022-01-11 15:12:39] > Changeset: 8f317a43f9f592d052af4d0922397cf9cb5c0c30
[2022-01-11 15:12:39] > Build ID: 20220111213255
[2022-01-11 15:12:39] > Downloading: https://firefox-ci-tc.services.mozilla.com/api/queue/v1/task/BqusNjODRVWfuw9UeZzdnQ/artifacts/public/build/target.dmg (175.64MiB total)
[2022-01-11 15:13:09] .. still downloading (80.8%, 4.96MB/s)
[2022-01-11 15:13:16] .. downloaded (4.99MB/s)
[2022-01-11 15:13:16] .. extracting
[2022-01-11 15:14:01] Extracted into /Users/abhinavvemulapalli/Documents/coding/firefox_bug_hunting/browsers/firefox
```

- I was then able to run Grizzly with the “no-op” adapter which “is meant to be an example and does not really do much” (“Grizzly Getting Started”)
    - Even though it does not really do much, getting it working meant that I was successfull in downloading the tools and running it properly

```
Documents/coding/firefox_bug_hunting via 🐍 v3.8.9 (venv)
at 15:20:04 ➜ python3 -m grizzly $CODE/browsers/firefox/Nightly.app/Contents/MacOS/firefox no-op                    using 10GiB/16GiB of 🐏
[2022-01-11 15:20:08] Starting Grizzly (25376)
[2022-01-11 15:20:08] Ignoring: timeout, log-limit
[2022-01-11 15:20:08] Using test time limit: 30s, timeout: 45s
[2022-01-11 15:20:08] Results will be stored in '/Users/abhinavvemulapalli/Documents/coding/firefox_bug_hunting/results'
[2022-01-11 15:20:13] I0001-R00
[2022-01-11 15:20:15] I0002-R00
[2022-01-11 15:20:15] I0004-R00
[2022-01-11 15:20:17] I0008-R00
[2022-01-11 15:20:19] I0016-R00
```

- Lastly, I also began to attempt to compile my on Firefox version that I could fuzz if necessary
- I started doing this just to be safe and compiling can also take a very long time.
- I followed the steps recommended by Firefox’s Source Documentation at https://firefox-source-docs.mozilla.org/tools/sanitizer/asan.html#what-is-address-sanitizer
- I used the above link with the Adress Sanitizer because that is what Grizzly recommended for a fuzzing friendly build. I believed it would be best for me to use something similar to make sure Grizzly works as intended
- To begin, I had to download the Firefox source code using a tool called Mercurial
    - The instructions at https://firefox-source-docs.mozilla.org/contributing/vcs/mercurial.html#mercurial-overview helped me download the source code

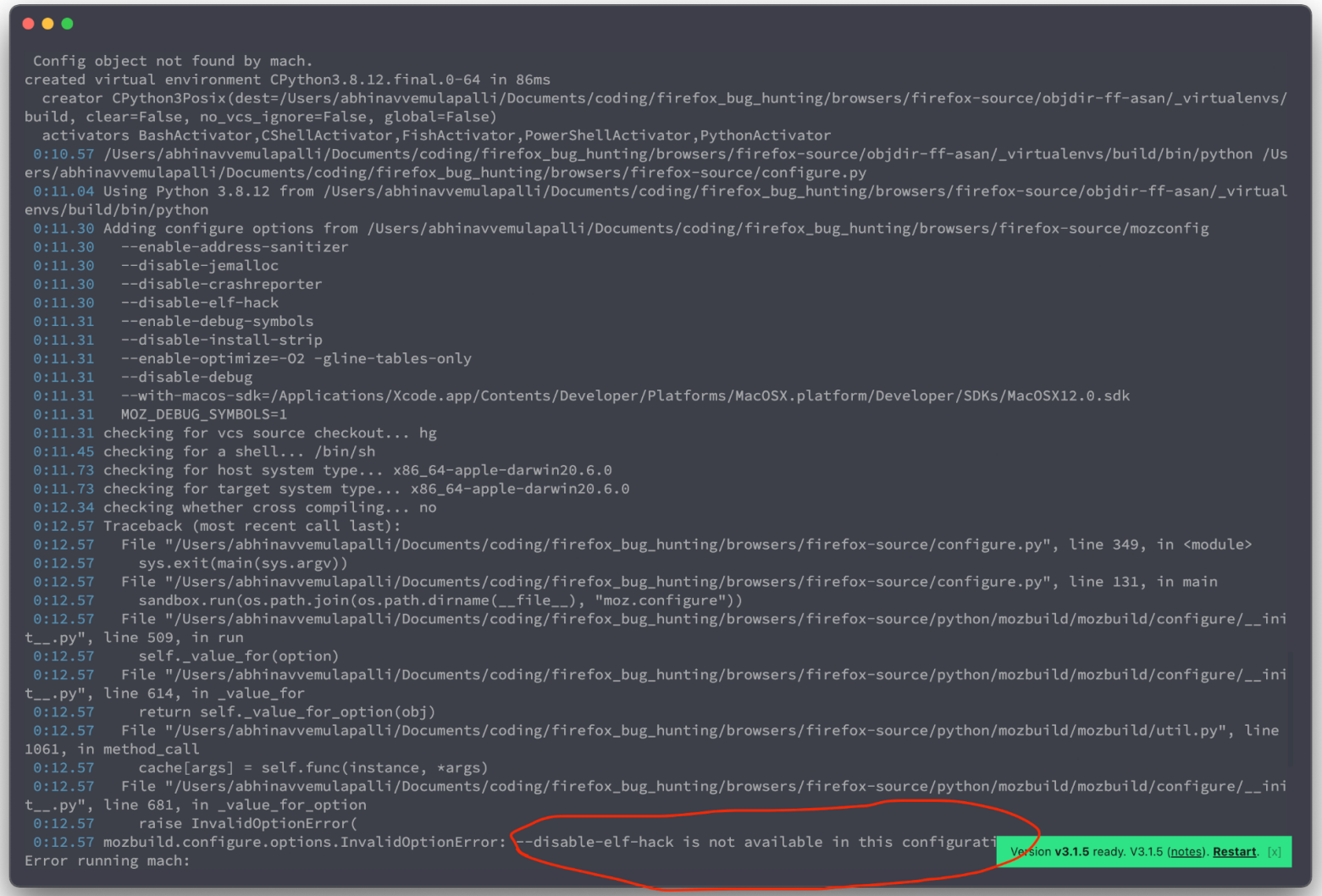
- Mercurial is a piece of software called a Version Control System (VCS). It is similar to `git` in that it helps multiple developers collaborate on one project and have version history implemented to help organize the code.



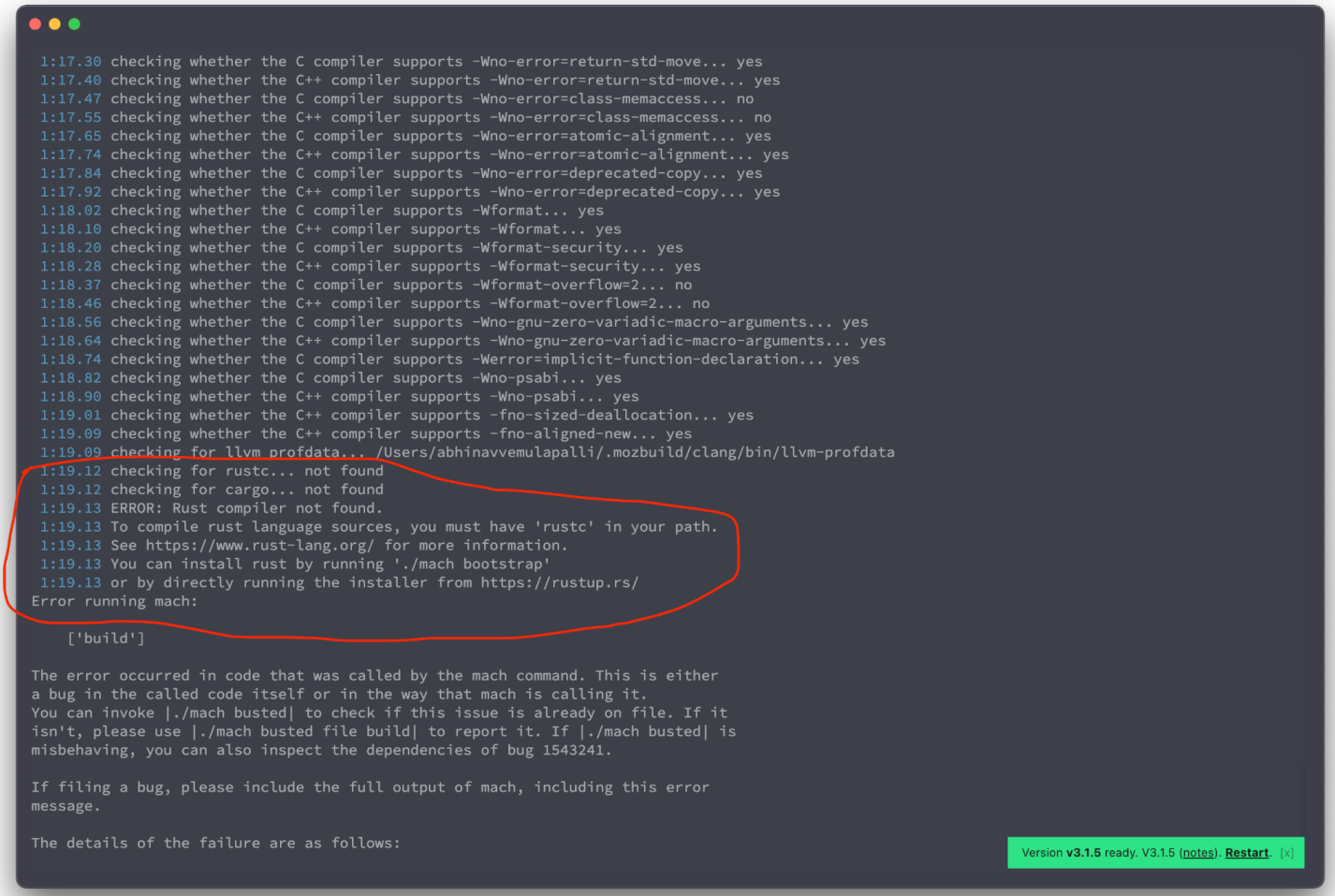
- The next step was to follow the steps at the `AddressSanitizer` documentation was to build a `mozconfig` file to tell the computer how to compile Firefox. Here is the `mozconfig` file I originally tried to use:

```
firefox_bug_hunting/browsers/firefox-source via  via  v15.3.0 via  v3.8.12 via
at 14:32:55 ➜ cat mozconfig                                                    using 11GiB/16GiB of
# Combined .mozconfig file for ASan on Linux+Mac

mk_add_options MOZ_OBJDIR=@TOPSRCDIR@/objdir-ff-asan

# Enable ASan specific code and build workarounds
ac_add_options --enable-address-sanitizer

# These three are required by ASan
ac_add_options --disable-jemalloc
ac_add_options --disable-crashreporter
ac_add_options --disable-elf-hack

# Keep symbols to symbolize ASan traces later
export MOZ_DEBUG_SYMBOLS=1
ac_add_options --enable-debug-symbols
ac_add_options --disable-install-strip

# Settings for an opt build (preferred)
# The -gline-tables-only ensures that all the necessary debug information for ASan
# is present, but the rest is stripped so the resulting binaries are smaller.
ac_add_options --enable-optimize="-O2 -gline-tables-only"
ac_add_options --disable-debug

# Settings for a debug+opt build
#ac_add_options --enable-optimize
#ac_add_options --enable-debug

# MacOSX only: Uncomment and adjust this path to match your SDK
ac_add_options --with-macos-sdk=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12.0.sdk

firefox_bug_hunting/browsers/firefox-source via  via  v15.3.0 via  v3.8.12 via
at 14:32:57 ➜                                                                  using 11GiB/16GiB of

Version v3.1.5 ready. V3.1.5 (notes). Restart. [x]
```

- When I tried to compile with those options, however, I got an error stating the “disable-elf-hack option was not available in this configuration.



- So, I commented that line in the mozconfig file so the compiler would not try to use that option. I re-compiled but this time I ran into an error stating Rust was not installed. Rust is another programming language, and apparently Firefox depends on having it installed, so I went to https://rustup.rs/ and installed Rust.



- Alas, even that was not enough and I kept running into some generic error that I was unable to comprehend.
- However, looking closer at the AddressSanitizer documentation page, I realized that the Firefox source code comes with some `mozconfig` templates. I have not been able to analyze them yet and attempt to use them, but that will be one of my future goals.

- In the above screenshot, you can see my current working directory and how the Firefox source code comes with some `mozconfig` templates, with the "`debug-fuzzing`" and "`nightly-fuzzing-asan`" `mozconfig` files looking very interesting.

Reflection on Goals and Timeline

I believe that I am on track with my timeline now, maybe slightly behind but still on track. After talking to my advisors, they believed that I was still on track, if not ahead. Regardless, I still tried to catch up on my research with fuzzing and even start to implement fuzzing tools over break and did so successfully. At this point in the timeline, the only thing I am still behind on is taking a look at the source code for the IPC layer and understanding how the sandboxing works. The reason why this would be important, and is something my advisor Ned Williamson

recommended, would be to build a list of things that I can access outside the operating system with the IPC. For example, file system access is an extremely interesting vector because if I was able to find a sandbox escape that way, I could have some amount of control over the files on the user's computer. So, if I could build a list of things that is outside of the sandbox that is reachable by the IPC. If I can compile a list, I could look over it with my advisor and try to identify interesting attack vectors. Other next steps would also include compiling Firefox and attempting to learn about Adapters in the Grizzly framework and getting started with constructing my own Adapter. In addition, I am caught up with background research and fuzzing research and need to get started on my tool soon, which I already have. Overall, I'm fairly happy with the place I'm in with regards to the project but it still feels like I have a lot to do.