Weeks of January 24 and February 28

Goals for this Week

1. 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.)
2. Attempt to understand how the Picture in Picture IPC messaging works (source code audit)
3. Research VM-based snapshot fuzzing (or process-based snapshot fuzzing)
4. Grizzly Interfaces
    a. More research into Grizzly interfaces (at first glance, Grizzly interfaces may not help with IPC fuzzing)
    b. Run the test interface
5. Start working on JS fuzzer to fuzz the JSActor methods
    a. Go through potential IPC methods to add to my fuzzer
    b. Add the different messages that I want to fuzz
6. Work on Journal 3
    a. Add the research I've done (JSActor documentation research)
    b. Add all the accomplishments I've done (Having Prompt:Open work and other IPC messages to work)
7. Try to compile and build a basic version of Firefox on MacOS

My Research and What I Learned

Most of my research for the past couple of weeks was regarding Grizzly, VM-based snapshot fuzzing, and how JSActors work internally. Through my research, and talking with my advisor, I realized that Grizzly would not work for me in this case and VM-based snapshot fuzzing is too complicated for me to learn and set up by presentations. In the end, my advisor helped me find another viable approach that will result in a presentable program of some sort. Overall, I had more accomplishments for the past couple of weeks than research.

Grizzly Framework Research

From my prior research, I knew one of the first things I would have to create was an Adapter. As a reminder, an adapter “acts as the interface between Grizzly and the fuzzer” (Schwartzentruber). Essentially, all an Adapter does is tell the Grizzly framework how to take input and pass it to Firefox. According to Schwartzentruber, something called a `TestCase`

needs to be populated which Grizzly will "use to send the data to the browser." However, looking through the examples provided proves that the Grizzly framework may not work for my use case. In addition, a lot of the code seemed complex from having to figure out how to write a fuzzer that works with Grizzly to figuring out how to get Grizzly to send that data to the Firefox internals. Before giving up, I looked into some of the links to fuzzers they linked so I can figure out how I could potentially write my own to work with Grizzly. However, this was where I found my second issue with Grizzly. One of the fuzzers linked was one called Dharma. After looking through how Dharma functions, I concluded that it would not be feasible to learn to use Dharma because it appeared to be its own language that I would have to learn to generate data, especially given the time remaining before presentations. The second fuzzer they linked was one called Avalanche. This fuzzer appeared to also use another language to generate random input. It also appeared that it was useful for generating HTML input to fuzz the HTML parser built into web browsers. The HTML parser is how the browser takes code and displays the webpage developers want it to. Just like any piece of software, the HTML parser is bound to have bugs in it as well. Therefore, people have naturally written a fuzzer to help test the HTML parser. Regardless, the Avalanche fuzzer has no way of helping me since the IPC message layer doesn't use HTML code of any kind. On top of that, I would have to figure out a way to have Grizzly send the data to the IPC layer directly which doesn't seem possible from their example code. Looking at the example code in the Writing an Adapter article the important piece of code appears to be:

```
testcase.add_from_file(gen_file, file_name=testcase.landing_page,
required=True, copy=False)
```

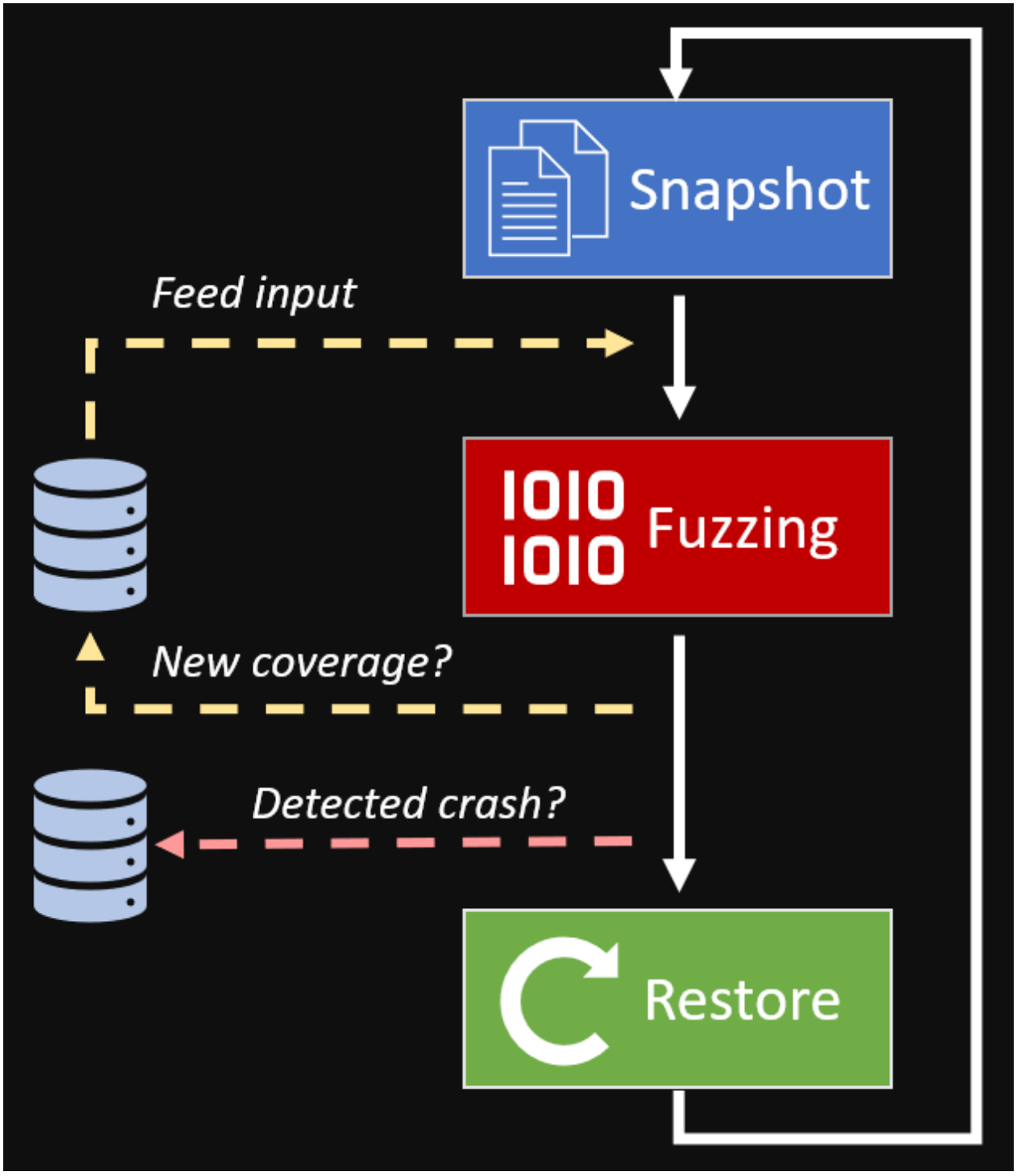
The code takes data from a file and adds it to a landing page. When I originally ran the Grizzly framework with the adapter, it would open and close a page in Firefox rapidly. So looking at this code, it's fairly apparent that Grizzly takes the data, adds it to a regular HTML page. I tried to see if I could add the code to send a JSActor message in a regular JS file and load it from an HTML page. However, I quickly realized that those methods `getActor` and such are only available in a privileged context which is why the Browser Content Toolbox enables me to use that method and send messages. Finally, I came to the conclusion that the Grizzly framework would not fit my needs and I would have to pursue other means. From my initial research into fuzzing and reading about how Mozilla fuzzes the IPC layer, I learned about VM-based snapshot fuzzing. So I decided to research that a little bit to see if that could work for me quickly.

VM-Based Snapshot Fuzzing

Some quick searching led me to an article by Markus Gaasedelen titled "All Your Base Are [Still] Belong To Us." While the article title may not sound related, it is about how someone used a tool called `wtf` to fuzz a game's UDP protocol (I know not professional, but what can I do? The tool seemed like it could help me; in my defense, it does stand for `what the fuzz` and not what you were probably thinking). So I started following along with the tutorial most of which I'm going to cover in the accomplishments section. However, I learned some information about snapshot-based fuzzers that could prove useful or some new information to know. So the way

snapshot-based fuzzers work are they are “seeded a snapshot captured from a live system precisely before executing the code that a researcher is interested in fuzzing” (Gaasedelen). In other words, the snapshot is like taking a picture of an executing process’s memory, variables, and any other information the fuzzer may need to begin fuzzing afterward. After taking this snapshot, the fuzzer would send in the malformed data and would look for a crash. If it finds one, it saves the current test case to disk and resets the emulator (Gaasedelen). In order to get started, one needs to create a snapshot of the process right before the point where Firefox would parse the IPC message. One of my steps was to use WinDbg to debug Firefox and figure out where exactly it parses the IPC message (I couldn’t get this to work as I’ll explain later in the accomplishments section). I did continue reading the article, however, and the next steps after creating a snapshot are to write code to tell the fuzzer, `what the fuzz`, is how to “initialize our snapshot, where it should inject fuzzed testcases for each, and what types of events it can ignore while fuzzing” (Gaasedelen). This code that I would’ve written is called a harness and would have involved me reading and understanding the C++ source code of Firefox’s IPC implementation; something much more complicated than what I’ve been looking at so far. Either way, from first glance, it appeared that VM-based snapshot fuzzing would have been perfect for what I wanted to do. I could have a fuzzer that could send data directly into the Firefox IPC message parser and go from there. Very quickly, here is a photo from the same article by

Gaasedelen which is just a visual depiction of how a VM-based snapshot fuzzer works.



Screenshot 1 from Gaasedelen

The biggest obstacle for me now was time since attempting to read C++ code from Firefox's source code which is nearly 8GB would take a long time. It's also important to consider that I will need to understand and write code for a harness, something I've never done. So naturally, it will take me longer. Talking to my advisor further solidified my realization that VM-based snapshot fuzzing wouldn't work for me in this scenario because of my time constraint. However, I would

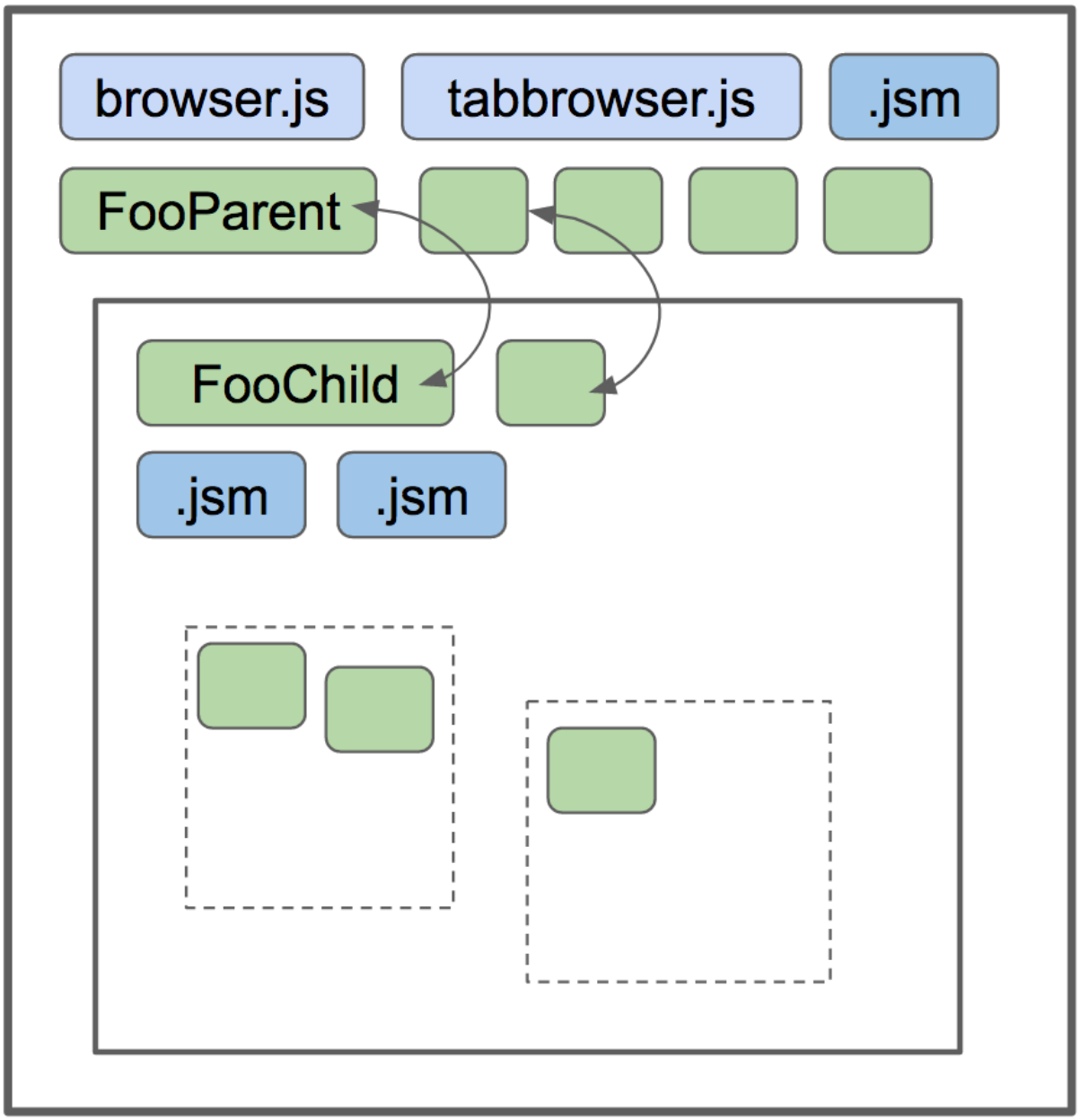
love to keep trying and learn more about VM-based snapshot fuzzing so that I can experiment with it and eventually get it working myself. Either way, the realization, and my advisor agreeing with me set me on the path to explore another avenue: writing my own fuzzer in JS (a language I know) and running it directly against the JSActors themselves. This new idea required me to do some more research into JSActors and read the documentation for them.

JSActors

The biggest source I used for this was from Firefox's Source Documentation itself. Particularly from an article titled "JSActors." The article went into depth about how JSActors are used to communicate between "things-that-may-live-in-a-different-process." According to the article, there are two types "JSProcessActor" and "JSWindowsActor." The former communicates between a child process and its parent while the latter communicates between a frame and its parent. For JSProcessActors, they exist by pair and are instantiated lazily, which means that it will be instantiated when `getActor("MyActor")` is called. Another important piece of information is that the whole pair dies when the child process dies. There could be some interesting bugs here, for example, if Mozilla did not program it properly, it is entirely possible a Use After Free could occur if the child process is not cleaned up properly. I'm not entirely sure how I could test for and identify a bug like that but that could be something to look into in the future. The JSWindowsActor is between a frame and its parent and is what I used when I replicated CVE-2019-11708. The `Prompt:Open` message and `Prompt` JSActor was technically a method for a JSWindowsActor. The article also delves into how Firefox managed IPC messaging before their new Fission version. I don't believe that it'll be of too much importance but some differences between Frame Message Manager and JSWindowActors are:

- A direct channel of communication being instantiated between two pairs in an Actor (which replaces frame scripts)
- A JSWindowActor operates with the following mechanism
    - Every Actor has one counterpart in another process
    - Every Actor inherits a common communications API from a parent class
    - Every Actor has a name ending in either Parent or Child
    - When one JSWindowActor sends a message, the counterpart will receive it.

Here is a fairly simple diagram of how JSWindowActors work from the article "JSActors":



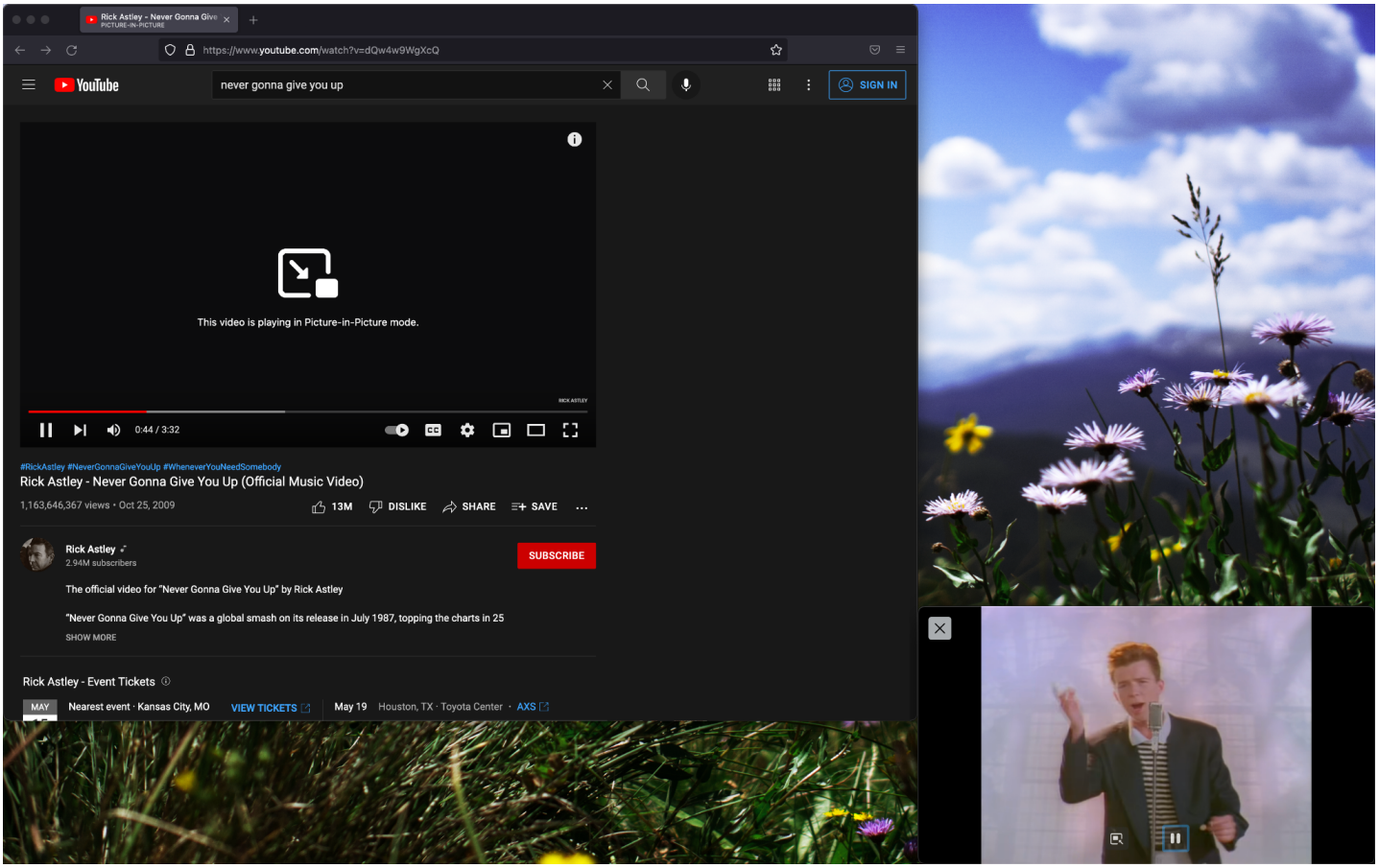
Screenshot 2 from "JSActors"

As you can see from the screenshot above, the inner rectangle is a frame with the outer rectangle representing its parent. The `jsm` files are what contains the actual code for what to do when a Child or Parent receives a message. As you can see, in this case, the Actor is called Foo since there is a FooParent and FooChild. The FooChild JSActor is the counterpart for FooParent. Some other crucial information I found from the documentation was that I can view all registered JSActors in two files: `BrowserGlue.jsm` and `ActorManagerParent.jsm`. So I decided to look through the list to see if I can identify potentially interesting functions or methods to test later. It was also one of the things that my advisor recommended: trying to identify potential IPC functions that can lead to the outside of the system of increasing privileges

to execute code on the computer itself. For example, going through the source code file helped me find that there were IPC messages for `Pdfjs`, `ScreenshotsComponent`, `FormValidation`, `PictureInPicture`, `PopupBlocking`, etc. I figured for things like `PictureInPicture` since it has to open up a new window within the operating system, it has to execute privileged calls, so maybe there's a way to have code execute that way. In addition with `PopupBlocking`, there may be a way to prevent it from working properly, then open a malicious pop-up and pop a shell. These were all ideas, however, and further research and source code audit would be necessary. I did begin researching Picture In Picture however and began to find some information which I will explain in the Accomplishments section.

PictureInPicture Actor

So one of the biggest things I spent my time on for the source code auditing was the PictureInPicture Actor. So to begin, PictureInPicture (PiP) is something Firefox has for when you're watching a video and you want a small window of it to show up on other screens as well. Here are a couple pictures of what PictureInPicture looks like.

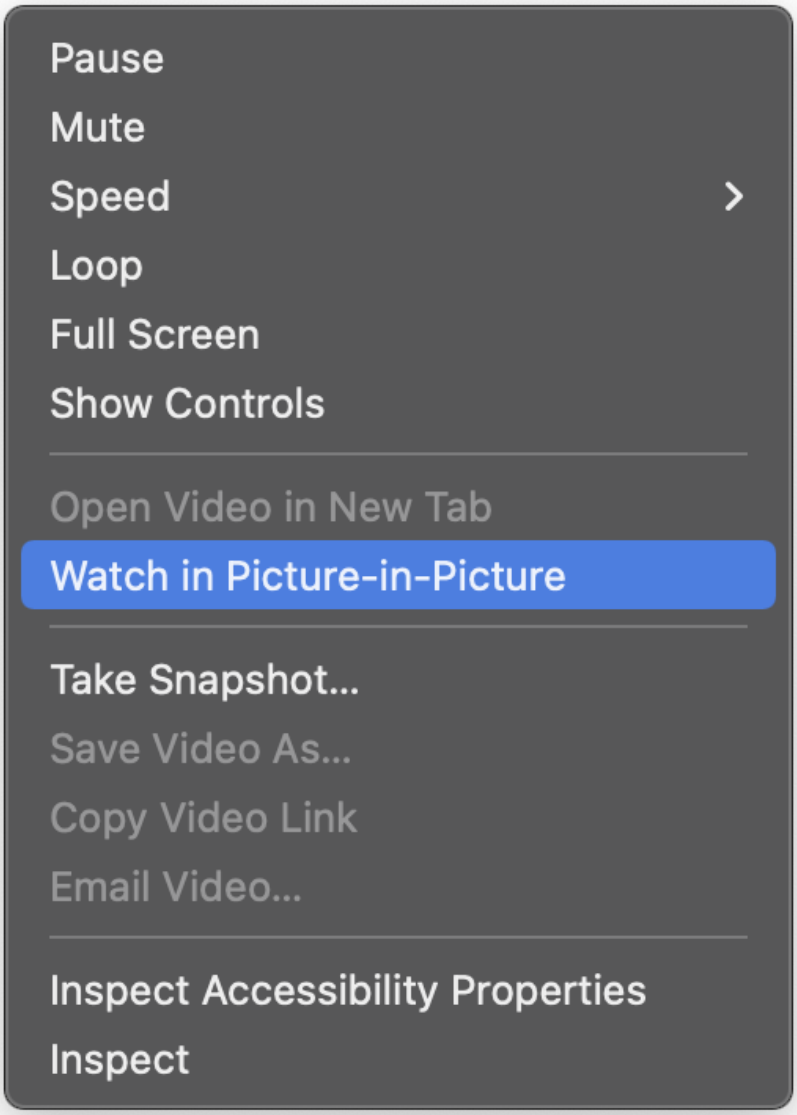


Screenshot 3 of my desktop

As you can see, the main window is to the left and it's on YouTube but the video itself is popped out and playing in a smaller window in the corner. Here's a close up of the smaller window:

Screenshot 4 of PiP

When taking screenshots on my computer, I have options to take a screenshot of my entire screen, a certain region I select, **or an application window.** The reason why that last point is important is because that's what I used to take Screenshot 4. This method I used to take the screenshot means that the operating system treats that mini frame as a brand new video. Essentially, since the operating system creates a new window and has to put a video there, I thought maybe this would be a great place to find a sandbox bypass. It makes sense since Firefox has to go to the parent actor and ask the operating system to create a brand new window. Child processes shouldn't be able to create new videos and load content into them because then when someone goes to a website it can just open a brand new window and execute some code. Regardless, I started going through the code to see if I could understand how it works. Before delving into the code, one thing to know is that to enter Picture In Picture mode, you need to enter the context menu and select the option. The context menu looks like this:



Screenshot 5 of the Context Menu

This is important to know for how the Picture In Picture IPC mechanism works. I started by going through the code for the `PictureInPicture.jsm`. The part of the code I started with was the following:

```
class PictureInPictureLauncherParent extends JSWindowActorParent {
  receiveMessage(aMessage) {
    switch (aMessage.name) {
      case "PictureInPicture:Request": {
        let videoData = aMessage.data;
        PictureInPicture.handlePictureInPictureRequest(this.manager,
```

```
videoData);
        break;
      }
    }
  }
}
```

This part was important because based on my previous analysis of the `PromptParent.jsm`, I knew that the part where the Parent or Chiild checks for a new message is where there is a `receiveMessage` function followed by a `switch` statement. A `switch` statement is like an `if` statement on steroids. Based on the contents of the passed in data, in this case `aMessage.name`, it will execute certain code. For example, in the code above, when the Parent actor receives a message with the name “PictureInPicture:Request” it will execute the code underneath. It will take the data passed with the message and send it to another function as video data. Now, that video data portion seems very interesting because technically, the attacker could control its contents. So I decided to see if I can figure out what the `handlePictureInPictureRequest` function does do. Note, there were other `receiveMessage` functions with `switch` statements but I focused on this one for now because it appeared to be what was called when initially creating the window. Here is the code for that function:

```
  gCurrentPlayerCount += 1;

  Services.telemetry.scalarSetMaximum(
    "pictureinpicture.most_concurrent_players",
    gCurrentPlayerCount
  );

  let browser = wgp.browsingContext.top.embedderElement;
  let parentWin = browser.ownerGlobal;

  let win = await this.openPipWindow(parentWin, videoData);
  win.setIsPlayingState(videoData.playing);
  win.setIsMutedState(videoData.isMuted);
```

The above code appears to be the core of the function and is what actually creates the PiP window. So from here, I could discern that `videoData` is an object with two attributes `playing` and `isMuted`. However, it calls yet another function called `openPipWindow` and passes `videoData` to it as well. Going into that function reveals that it takes other information passed in through `videoData` and created the PiP window itself.

```
async openPipWindow(parentWin, videoData) {
    let { top, left, width, height } = this.fitToScreen(parentWin,
videoData);

    let features =
      `${PLAYER_FEATURES},top=${top},left=${left},` +
      `outerWidth=${width},outerHeight=${height}`;

    let pipWindow = Services.ww.openWindow(
      parentWin,
      PLAYER_URI,
      null,
      features,
      null
    );

    if (Services.appinfo.OS == "WINNT") {
      WindowsUIUtils.setWindowIconNoData(pipWindow);
    }

    return new Promise(resolve => {
      pipWindow.addEventListener(
        "load",
        () => {
          resolve(pipWindow);
        },
        { once: true }
      );
    });
  },
```

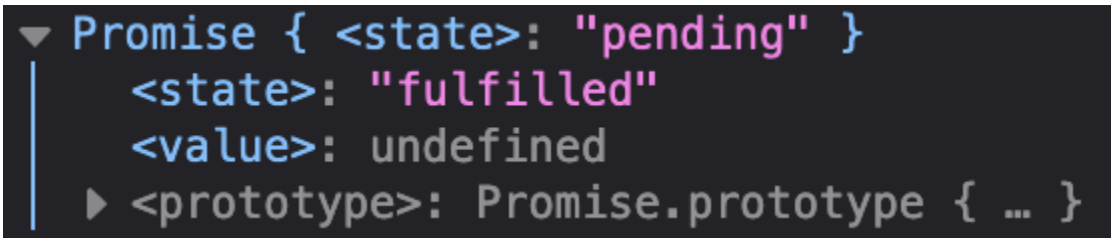
As you can see, it takes parts called `width`, and `height` from the `videoData` variable and passes it to the `openWindow` function. Looking at the comments above specifies that videoData should contain those two attributes. I wasn't sure where the `PLAYER_URI` was coming from or how to tell Firefox to display a YouTube video or whatever the case is. So, I decided to just try and execute code in the Browser Content Toolbox to see if I could figure out how to create a PictureInPicture window. I started with the below which just got the actor for the PictureInPicture and so then I could send the "PictureInPicture:Request" message.

```
var actor =
window.top.content.windowGlobalChild.getActor("PictureInPicture")
```

Sending that message was my next step, along with what I understood `videoData` had to contain. So I sent the following message and data

```
actor.sendQuery("PictureInPicture:Request", {videoData: {playing: true,
isMuted: false, videoHeight: 200, videoWidth: 400}})
```

However, when I sent this message, I got back a `Promise` with a value of `undefined`.



Screenshot 6 of an `undefined Promise`

A Promise is what JS returns after an asynchronous operation. When code is executing in a process it usually happens line by line so if there is a line of code that takes a long time to execute, the entire program will stall until that finishes. To bypass that issue, an asynchronous operation will send that line of code to another thread to keep executing while the current program continues its execution. JSActors makes use of this and from the above screenshot, you can see that the value was `undefined`. However, when it was successful it essentially returned the data I sent plus a couple of other changes which I remember from my testing with `PromptParent`. It was at this point that I got a little stuck and decided to attempt to debug Firefox itself using the built in debugger to see how a video enters PiP mode from the context menu. Without going into too much detail about how that exactly works, the high level overview is that there is another actor called the ContextMenu actor. So when you click on the “Watch in Picture-in-Picture” in the context menu as you can see from screenshot 5. When you click on that option, it sends a message to the PictureInPictureChild Actor. The structure of PictureInPictureChild is slightly different so I’ll have to spend more time understanding that code. The function I identified above is then called from the PictureInPictureChild actor. I’m not going into further detail mainly because I’m not sure if this will remain applicable for me as I keep writing a fuzzing tool and such.

Conclusion

While my research for the past couple of weeks may not have been extensive, it was instrumental in helping me realize how my current plan of attack would not work. That realization also helped encourage me to reach out to my advisor and devise another plan that would potentially work. I will discuss that plan in my accomplishments as creating a new plan was something I had done and not researched. My research has also shown me new types of tools that I can spend time learning later on even after the capstone class ends.

Accomplishments

- I had a lot of varied accomplishments with the biggest one being my new plan of attack for writing a fuzzing tool that I can showcase at presentation.

- After talking to my advisor, he believed that trying to learn how to do VM-based snapshot fuzzing would take me too long in the time I had until presentations. Once I showed him how I was able to interact with JSActors themselves, he suggested that I write my own fuzzing library in JavaScript interacting with the JSActors. So, I would be sending random data as the data to various different IPC messages and see if anything would crash
-

```
Components.utils.import("resource://gre/modules/Console.jsm");

function generate_input(type) {
    if (type == "string") {

    }
}

let ipcJSActors = {
    "ClickHandler" : "Content:Click",
    // seems to need some other kind of privilege "ContentSearch": "",
    "FormValidation": "FormValidation:ShowPopup", // SOME OF THE
PARAMETERS; it doesn't fully work though
    //actor.sendQuery("FormValidation:ShowPopup", {position: "after_start",
screenRect: {width: 100, left: 150, top: 250, height: 100}, message:
"Testing"})
    "Pdfjs": "PDFJS:Parent:saveURL",
    //actor.sendQuery("PDFJS:Parent:saveURL", {blobURL:
"file:///Users/abhinavvemulapalli/Downloads/2020_1040Form.pdf", filename:
"testing.pdf"})
    //"Plugin": "RequestPlugins",
    "Prompt": "Prompt:Open",
    "ScreenshotsComponent": "",
    "WebRTC": "",
    "LoginManager": "",
    "PictureInPicture": "",
    "PopupBlocking": "",
    "Printing": "",
}

actor = tabs[0].content.windowGlobalChild.getActor("Prompt");

prom = await actor.sendQuery("Prompt:Open", {
    promptType: "dialog",
    title: "👻",
    modalType: 1,
```

```
    promptPrincipal: null,
    inPermutUnload: false,
    _remoteID: "id-lol"
});
```

  - In the above code, the first line is just importing a library so I can output information when I'm debugging to ensure what I'm writing is working properly
  - I also started creating a function to generate input that would be random. It would return data depending on what kind of data I would need. So far I have something for strings but it doesn't generate random data yet.
  - Underneath is a dictionary mapping the JSActors I hope to test with the messages those actors have. It's not fully completed but I plan on completing it soon.
  - Some of the messages also have comments reminding me what kind of data a message takes. I got that by just manually testing each message and reading the code for each one. The reason why I didn't include all of them in my research is because I'm not sure if they will pan out. I had high hopes for PictureInPicture and it's also the one I've spent the most time on.
  - Underneath is the very familiar code of getting an Actor and sending a message.
  - The `await` keyword before the `actor.sendQuery` is how I can get the value of the Promise and store it in the `prom` variable.
  - So far, the code above doesn't do any fuzzing, but I have plans to expand upon its functionality such as finishing the random input generator itself and continuing to understand other JSActor Messages and how to send them.
- Before I actually got started on that, I spent a lot of time trying to follow the Ret2Systems blog and get VM-base snapshot fuzzing working.
- In order to do so, I had to install WinDbg so that I could debug Firefox
  - Before explaining further, let me explain what WinDbg is. WinDbg is a debugger which essentially means its a way of finding running through your code and seeing the state of the application to find the cause of any crashes.
  - In this case, I was hoping to set a breakpoint in the process right before it starts parsing the IPC message so that I can identify where in the source code it is. If I knew where in the source code that part was, I could better understand how to create a harness and create a snapshot for the VM-based snapshot fuzzing.
- Firefox themselves had some documentation on how to debug Firefox and I would follow along.
- One of the first problems I ran into was the debugger and Firefox itself would hang every time I started WinDbg and attached it to the process
  - To fix it, I had to run a command called `sxi av` in order for Firefox to launch normally.
  - Afterwards, I followed along to help get the symbol and source server set up because I believed those two will help me view the source code of the process once I hit a breakpoint which would have allowed me to pinpoint the exact file

and location of IPC message parser code.

- The first thing I did was configure WinDbg to use the symbol server which is a prerequisite for the source server

```
0:001> .sympath SRV*c:\symbols*http://symbols.mozilla.org/firefox;SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*http://symbols.mozilla.org/firefox;SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Expanded Symbol search path is: srv*c:\symbols*http://symbols.mozilla.org/firefox;srv*c:\symbols*http://msdl.microsoft.com/download/symbols

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       SRV*c:\symbols*http://symbols.mozilla.org/firefox
Deferred                                       SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

- Once the symbol server was loaded, I had to tell WinDbg where to put the cache files for the symbols on the local computer

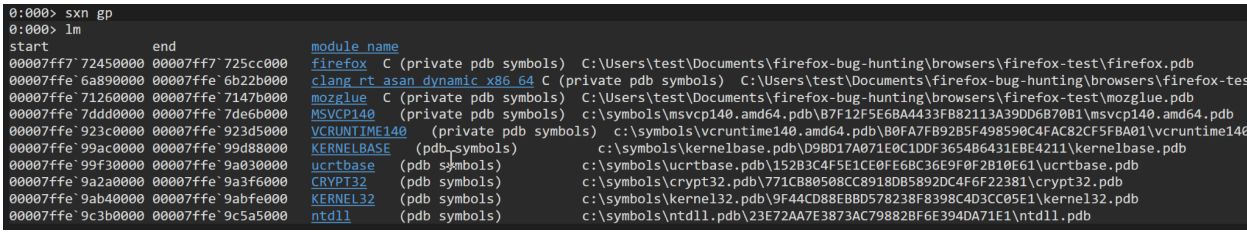
```
0:001> .symfix+ c:\symbols
0:001> .reload /f
Reloading current modules
..*** WARNING: Unable to verify checksum for firefox.exe
.*** WARNING: Unable to verify checksum for C:\Users\test\Documents\firefox-bug-hunting\browsers\firefox-test\clang_rt.asan_dynamic-x86_64.dll
.*** WARNING: Unable to verify checksum for C:\Users\test\Documents\firefox-bug-hunting\browsers\firefox-test\mozglue.dll
.

Press ctrl-c (cdb, kd, ntsd) or ctrl-break (windbg) to abort symbol loads that take too long.
Run !sym noisy before .reload to track down problems loading symbols.

...............
```

- The next step was to configure WinDbg to examine child processes and some other things recommended by Mozilla

```
0:000> .childdbg 1
Processes created by the current process will be debugged
0:000> .tlist
   0n0 System Process
   0n4 System
 0n100 Registry
 0n336 smss.exe
 0n452 csrss.exe
 0n560 wininit.exe
 0n568 csrss.exe
 0n640 services.exe
 0n668 winlogon.exe
 0n724 lsass.exe
 0n848 svchost.exe
 0n880 fontdrvhost.exe
 0n876 fontdrvhost.exe
 0n968 svchost.exe
0n1020 svchost.exe
 0n440 dwm.exe
0n1056 svchost.exe
0n1064 svchost.exe
```



  - Once I had that setup, I had to enter `.srcfix` to enable the source server
- I even had to install Visual Studio 2022 so that I had the necessary tools to build `what the fuzz`.
- However, after all of that work, I still couldn't figure out how to set a breakpoint near the IPC message parser and start looking at the source code from there.
- This was the point where I reached out to my advisor in hopes they could help me get VM-based snapshot fuzzing working on Firefox and on my machine
- The last thing I had accomplished this time was a successful compilation of Firefox.
  - The reason why this is good is because in my initial testing, Firefox would automatically replace bad data in an IPC message with valid data.
  - When I brought that point up to my advisor, he recommended that I patch the Firefox I have so it doesn't do that
  - The first step to doing that was compiling my own version of Firefox so that I could eventually patch Firefox and compile it
- In order to do that, I removed all the extra options I had before to configure Firefox for fuzzing and just compiled a basic, stock Firefox image with the `./mach build` command.

Reflection on Goals and Timeline

I've been able to meet a lot of the goals I've set for myself with the exception of writing my own fuzzing tool. I haven't been able to finish my random data generator function and finish adding all different actors and messages. Especially with TSA and midterms, I haven't been able to work on the project too much so I feel like I'm falling behind. However, since I have recently changed plans and now it is completely different, my timeline has become a bit more specific and I will have to change it a bit. I'm on the last stretch in my timeline and my two biggest items to finish are source code auditing and writing a fuzzer. Both of which go hand in hand since the fuzzer I'm writing will have to know how to send the data with the IPC message. So in order to write a fuzzer that will send the proper data and fuzz the IPC messages properly, I need to read the source code to ensure I send the right data. One reason why this fuzzer would be cool to have at my presentation is because I believe this would be the first fuzzer/fuzzing library built to fuzz the JSActor method of sending IPC messages. It is also entirely plausible that I find a bug through my source code audit rather than the fuzzer I'm writing. Regardless, I may feel behind but I think I'm on track to have a fuzzer built for presentation. I may not be able to present a fuzzer that has successfully found a bug but I should be able to present a tool that I have written myself which could potentially find a bug. I will also have a ton of new knowledge to present to people which was one of my main goals to begin with: learn how browsers and fuzzers work and things like that. So I would consider that aspect of my project a success. Overall, I think with

the help of my advisor and the biweekly meetings we've setup, I'll finish the fuzzer fairly soon and will hopefully be on my way to finding a bug.