Weeks of November 8 and December 7

Goals for this Week

1. Update timeline to reflect the change in micro-goal (data collection and analysis to fuzzing tool)
2. Rewatch LiveOverflow YouTube videos on buffer overflows
3. Watch YouTube videos on heap overflows and start reading articles and watching videos about browser internals
4. Research browser internal specifics, specifically the IPC layer and how browsers sandbox their resources and tabs
5. Start researching how to download and set up test browsers
    a. Get a MacOS VM running on my computer
    b. Try to download older browser versions (older Chrome or Firefox browsers that are vulnerable to exploits)
    c. Get the environment set up with possible tools like GDB and Radare2
6. Update website with information about advisors
7. Attempt to replicate an older sandbox exploit (CVE-2019-11708)
8. Read more about Firefox internals

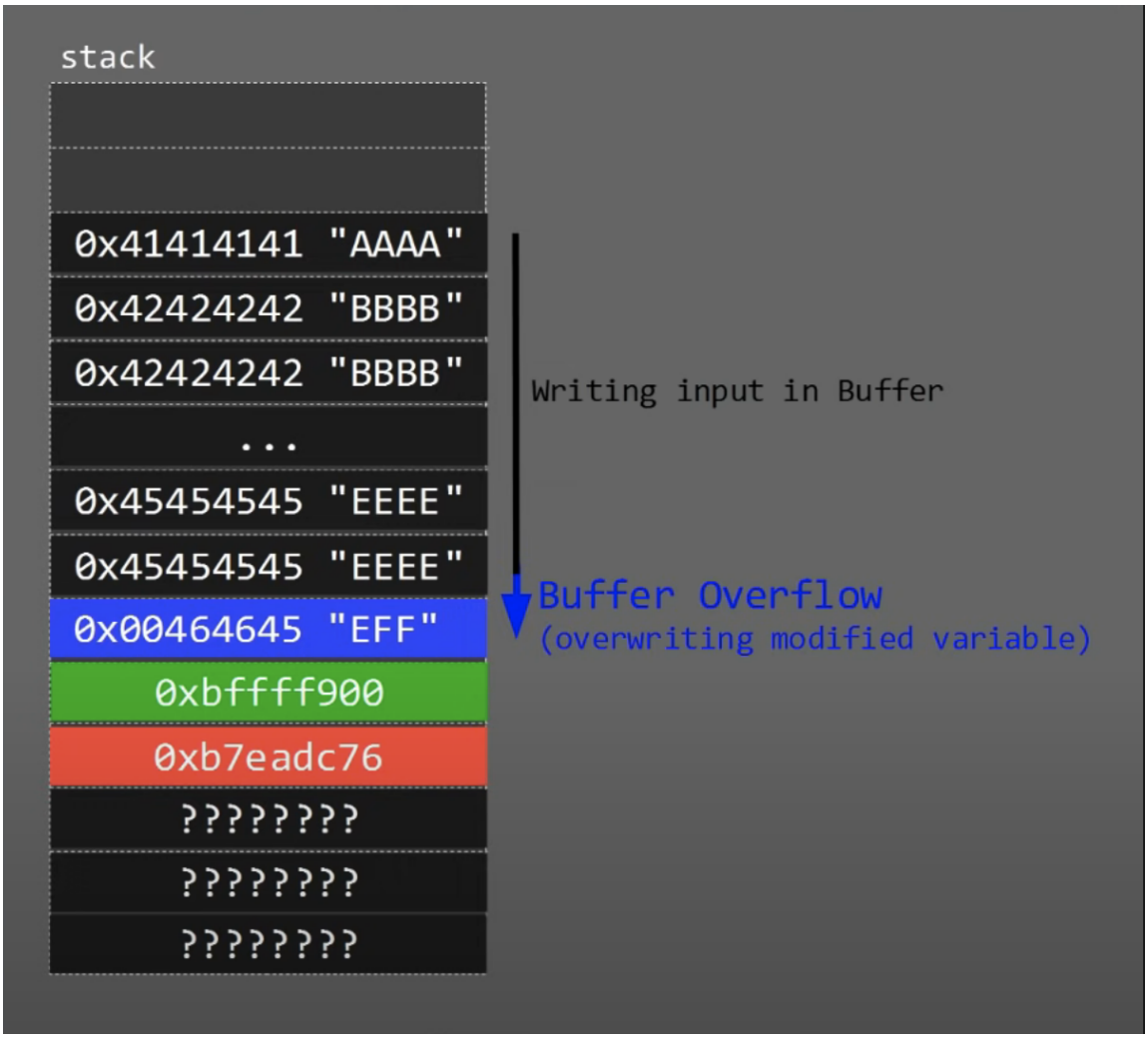
My Research and What I Learned

The first thing I began researching was some fundamentals of program exploitation. From previous experience, I'm aware that there exist memory exploitation techniques called buffer overflows and heap overflows. I reviewed buffer overflows by rewatching tutorials from a YouTuber called LiveOverflow. I also continued with watching videos about heap overflows, which is another form of memory exploitation. Before explaining the research I've done, here is some preliminary information to help establish a foundation.

Program Execution Basics

Usually, when a program runs, the operating system loads said program into the computer memory or RAM. There are a lot of different sections that the program utilizes in RAM. One section is dedicated to holding the code itself, but one very important section is one called the stack. A running process uses the stack to store variables and other runtime data. Many hackers utilize this to attack programs that have insecure coding practices. For example, in the C Programming Language, there exists a function called `gets()`. This function is vulnerable to buffer overflows. `gets()` is used to gather user input, and when calling the function, the programmer needs to specify the "buffer" size. A "buffer" is essentially a place in the stack where the program can store the user input. The security vulnerability in `gets()` stems from the fact that there aren't enough security checks in how much the user inputs and the area in

the stack the function is writing to. Therefore, it is easy for hackers to easily gain arbitrary write access to the memory. Now, most modern programs don't use `gets()` anymore, but the basic premise still exists. Hackers attempt to overwrite some data structure in memory to gain arbitrary write access. Upon gaining arbitrary write access, the hacker can manipulate the program memory and program execution order to call their malicious code or function.



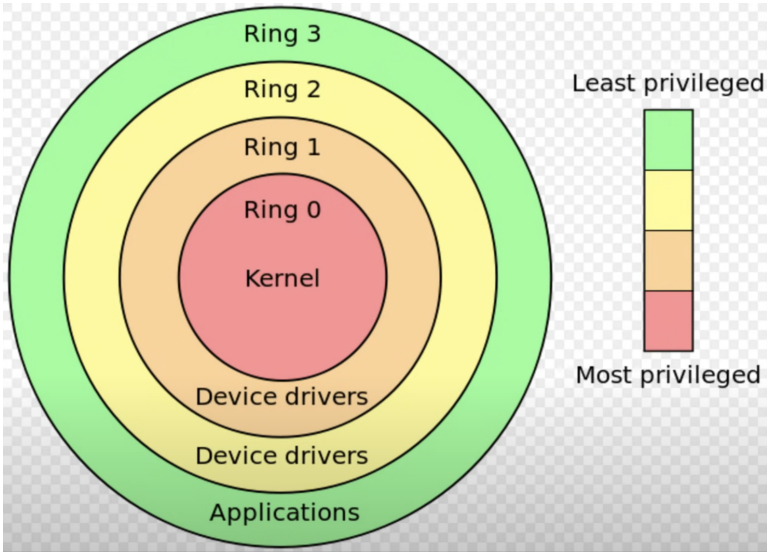
Screenshot 1 from "First Stack Buffer Overflow"

The above picture demonstrates a fairly simple model of the stack. The very first entry, the "AAAA," is the start of a buffer that the program had created. Ideally, the buffer should end well before the blue box. However, since the program was using `gets()`, the attacker was able to overwrite the contents of the blue box. The following green and red boxes contain what are called addresses. These addresses point to other parts in the memory or stack and help the program decide what code to run next, or where to look for certain information. Addresses are how the computer knows where certain functions and variables are located in memory. Now, attackers, since they have arbitrary write access, can control these "pointers" (since they point to different areas of memory) and have the program execute whatever code they want. Now,

while an attacker might have control over a process, they might not be able to do whatever they want because of something called privilege levels.

Permissions and Privilege Basics

At most offices and schools, the user account one uses for daily tasks allows them to do basic tasks like create files and run certain applications. However, if one wants to install a new program, they need something called administrator privileges. Processes also follow a similar ideology. Processes have different levels of privileges which determine what things they can access. The picture to the left is a general overview of how operating systems separate privilege levels. At an operating system's core, something called the kernel runs, which requires the most privilege, but also once the bootup sequence has completed, almost nothing runs with "Ring 0" privileges. Most device drivers have an extremely high privilege level, considering they need to be able to talk directly to the hardware. Applications have the lowest privilege level, but there are layers to the application level as well. The idea of administrators versus standard users has a huge effect at the application level. Generally speaking, when a process is run, the process inherits the same privileges as the user that called it. So for example, if a standard user started a web browser, the browser would have the same privileges as the standard user. Similarly, if a process is run as an administrator, then the process also inherits the privileges of that administrator. This is where things can get dangerous. If a vulnerable application, say an application that relies on the `gets()` function, was executed as administrator and an attacker compromised the process. This would mean the attacker now has administrator privileges on the computer system. This idea of privileges is also applied to browsers, which will become significant for my goals, and will be explained soon.



Screenshot 2 from "Syscalls, Kernel Vs. User Mode"

Additional Memory Background Information

In addition to the stack, computers, and more specifically programs, utilize something else called the heap. Before explaining what the heap is, here are a couple more details about the stack. The stack is just an area with a memory area at the bottom and is what we call a Last In First Out (LiFo) data structure ("First Stack Buffer Overflow to Modify Variable"). The stack, and LiFo, is similar to a plate stack at a buffet. The restaurant staff puts new plates on the top and they are the first to go. The stack is exactly like that in computers; referring back to

Screenshot 1, the “AAAA” was the last thing for the program to add to the stack, and therefore, will be the first thing to exit the stack when the program, or function, ends.

One other important piece of information to note: the stack is also called the local stack because a new one is “created” every time a function is called. In a program, there could exist two types of variables, a way of storing data: local variables and global variables. Local variables only exist within the function it was created in. For example, most C programs have a `main` function that contains the code to run when the program is run. However, programmers can create other functions, and in those, declare their own variables. The newly created variables can only be accessed and exist within the functions, i.e. you can’t access the said variable from the `main` function. The local stack is used to store these variables. The local stack is also automatically allocated, or created, by the computer every time a function is called, which is what the computer does when a program starts since technically it enters a new function: `main` in the case of a C program (“5.6 Heap Memory”).

Heap memory is also called “dynamic memory” and is used to store different types of objects since variables in the heap can be resized (“5.6 Heap Memory”; Martin). Now in most programs, the stack is used to store regular variables which can contain numbers (integers), strings, decimals (floats), and characters. In addition, data structures called arrays are also stored on the stack (arrays are essentially a way of storing lots of the same data under a variable). However, in C, there is something called `structs`. A struct is essentially one big object which contains lots of other data.

```
struct data {
    char name[64];
}
```

The above code snippet is an example of what a struct looks like in C. These structs are stored in the heap rather than the stuck because the size of it can vary at runtime and isn’t exactly set. One extremely crucial difference between the heap and the stack is, the heap is created manually by the programmer which means they must ALSO de-allocate the heap (“5.6 Heap Memory”; Martin). Not doing so will result in memory leaks and can be exploited later. In C and C++, dynamic memory is allocated using a function called `malloc`. So to create a `data` struct (the struct from the above code snippet) in memory, you would use the following code:

```
struct data *d;
d = malloc(sizeof(struct data));
```

The first line `struct data *d;` creates a new variable that will store a pointer, essentially it will contain an address that points to a different location in memory where the heap itself is. The next line actually uses the `malloc` function, which takes in an argument of the size of the struct, and then returns the address where that space in memory is. Now, the reason why the heap is dynamic is that the programmer can later increase the size of the heap if they want to store a larger struct. Most large programs, especially browsers, will rely on the heap heavily which is

why I had to make sure I understood it, and how potential exploits work against it. One last thing about heaps is that, like stacks, it is just another place of memory that a program uses. The heap can be exploited when programmers don't properly free the heap memory, that is de-allocate anything they created in the heap, leading to attacks called use-after-free. In addition, another form of memory exploitation called heap overflows can occur, the specifics of which will be covered later.

Memory Corruption Attacks

The very first attack I learned how to do was stack overflows, also known as buffer overflows from "First Stack Buffer Overflow to Modify Variable" and "First Exploit! Buffer Overflow with Shellcode." Looking back at Screenshot 1, an attacker could overwrite the buffer and overwrite data later in the stack. After a function finishes executing, the local stack needs to get destroyed and program execution needs to return back to where it started. Programs do this by storing the original address the execution started at the lowest part of a stack. So, when a function finishes, the program gets to that last stack entry and then jumps to that address to continue program execution. In Screenshot 1, the red box is an example of what a "return address" could look like. So, the whole premise of a buffer overflow is that an attacker can overflow the buffer and overwrite the "return address" which would allow the attacker to execute whatever code they want. Attackers do this by using something called shellcode, essentially code written in Assembly (an extremely low-level language; even lower than C) ("First Exploit! Buffer Overflow with Shellcode"). Certain shellcodes can result in a shell where attackers can enter whatever commands they want and control the computer. Once again, the permissions of the process running determine exactly what commands an attacker can run in the shell.

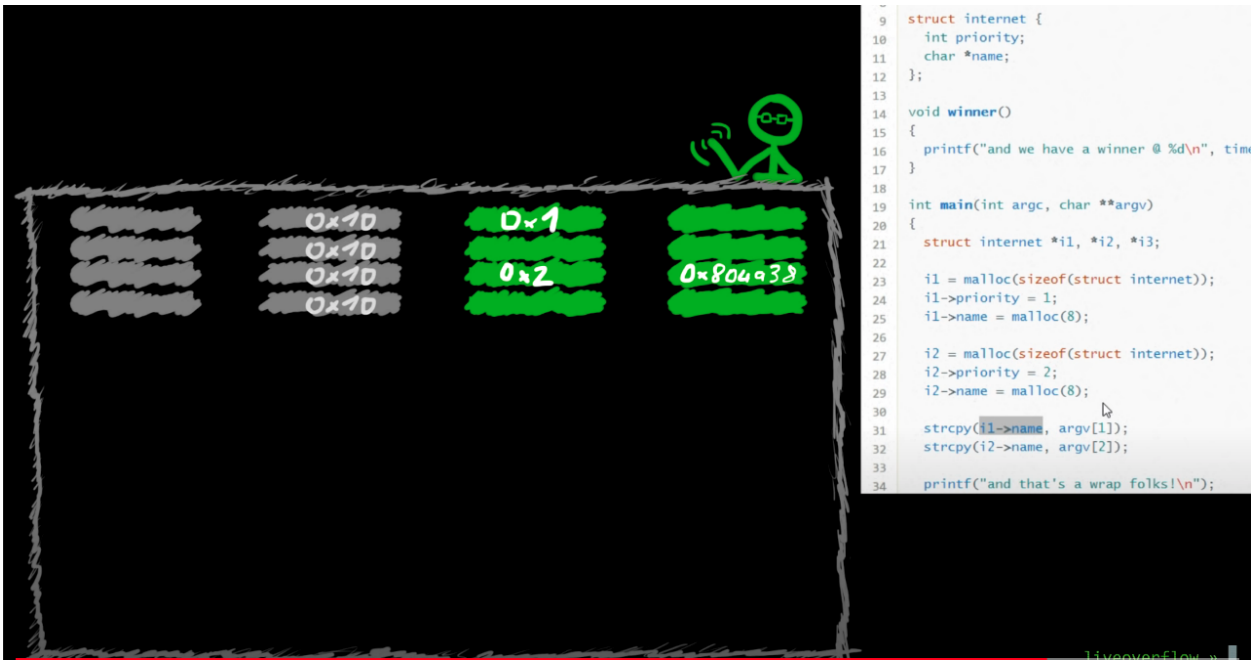
There are many techniques for getting shellcode to running from using something called NOP slides to even redirecting the program elsewhere. For example, most binaries (compiled code, otherwise known as programs) are shipped with libraries compiled with them or use a system library. A library is just code that contains functions that other programmers can use. Now the specifics of how one can utilize this aren't entirely necessary, but there is a common library called `libc` on most Linux systems. `libc` contains many functions, one of which is `system`. The `system` function can be used to execute commands on the system, so another potential attack vector is to replace the return address with the address of the `system` function. The attacker would also have to format the stack so that the `system` function will work properly, but if done correctly, the vulnerable program will then redirect its code to the `system` function and continue from there ("Doing ret2libc with a Buffer Overflow"). This is another way to gain code execution on computers.

NOTE: Code execution is a bit of a misnomer. Code execution just means the attacker has ways of running commands on a system, which also does mean executing code.

However, the above attack is only possible if the library is statically linked. Statically linked just means that when the binary was compiled, the library was also compiled and included in the binary. Libraries can also be dynamically linked, which means binaries don't contain the library, and subsequently, the functions present in the library. This also means that the library must be

present on the computer when the binary is being run. While the exact specifics aren't necessary, the binary knows where certain functions are in the library due to something called the Procedure Linkage Table (PLT). The PLT is useful because when the binary is compiled, the compiler (a program that converts the C code into machine code, something the computer can understand and execute) doesn't know the addresses of the function in memory. So the compiler creates a dummy function to "replace" the original function being called. The PLT then creates a mapping between the dummy function and the actual address in the library. There is also the Global Offset Table (GOT) which allows programs to run independently of where the code is actually stored. It's another table that maps between the base address of the program and where the code is during runtime. It's a fairly complex idea that stems from an idea called ASLR or Adress Space Layout Randomization, a modern security practice to mitigate the above buffer overflows. The attack with static libraries is possible because the address of the `system` function in the statically linked library `libc` is constant every time the program runs. Therefore, the exploit can just contain the address itself and it will always work. Modern systems however randomize these addresses so it's harder for the attacker to exploit. The GOT allows the program to still work even when these addresses are randomized and different from when it was compiled. With the GOT we can either have arbitrary write or arbitrary read. In this scenario, an arbitrary write will allow us to write anywhere in memory, overwrite offset in GOT, and have the program execute our own code. Arbitrary read means we can leak a value from the memory of the process (for example with use-after-free, another exploit that I will cover) ("Global Offset Table (GOT) and Procedure Linkage Table (PLT)"). The videos I watched to learn about those were about another form of exploit called format string exploits. While those exploits won't be useful to me specifically, the idea behind GOT and PLT will be useful later down the line, I think.

In addition to stack overflows, I also learned about heap overflows which mainly attack the application's usage of the heap. Similar to buffer overflows, when data structures are stored in the heap, it is possible to overflow the data in the structure.



Screenshot 3 from "The Heap: What does malloc() Do"

The screenshot above is a drawing of the heap of some code on the right. The general premise of the program is that it has a `struct` called `internet`, which contains a number called `priority` and a string called `name` (`char *`). The program then creates and allocates space for two `internet` structs in the heap. The green boxes are the actual data in the struct. So the first green box contains whatever value the `priority` variable has. The second green box contains whatever value the `name` variable has. Now the program is using a function called `strcpy` which just copies a string from one variable (`argv[1]` and `argv[2]`) into another. The function itself is vulnerable to buffer overflows just like `gets`. So in this example, the first input passed into the program gets copied into the first `internet` struct, namely into its `name` variable. So if the attacker were to pass in enough characters to overwrite the next two green boxes and the box with "0x2" and "0x804938," the attacker could then overwrite an entry in the GOT. This is possible because the next line follows whatever address is in the "0x804938" box and then writes the user's second input at that location. This means an attacker could overwrite the "0x804938" box with an address pointing to some entry in the GOT table. Then, the attacker can put another function's address, or an address pointing to shellcode, at that entry in the GOT table ("The Heap: How to Exploit a Heap Overflow").

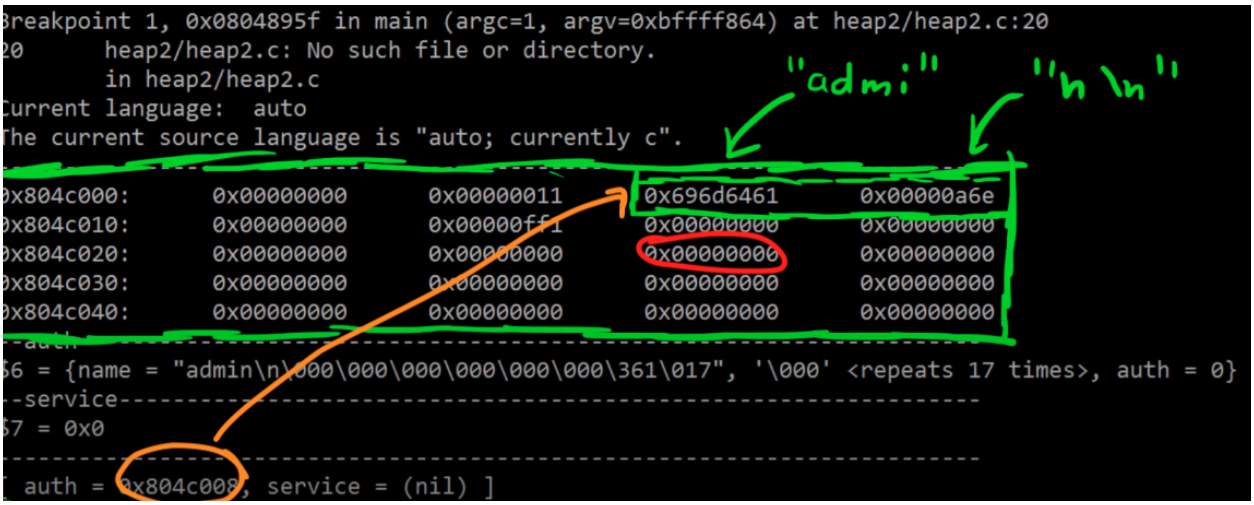
Another form of exploiting the heap is something called use-after-free. Since the heap is controlled by the programmer and not the operating system, the programmer needs to clear the heap properly. In "The Heap: How Do Use-After-Free Exploits Work?" the vulnerable program has an `auth` struct that contains a string called `name` and a number called `auth`. Now the point of the program was to somehow change the value of the `auth` variable when the program never configured it.

```
struct auth {
   char name[32];
   int auth;
};
```

The program also lets the user do other things if they choose a menu option called "service." All that happens is the `strdup` function is called which duplicates a string, places it in the heap, and returns the address pointing to that location (hopefully this will make more sense with a visual of the memory layout soon). So the attack vector is as follows:

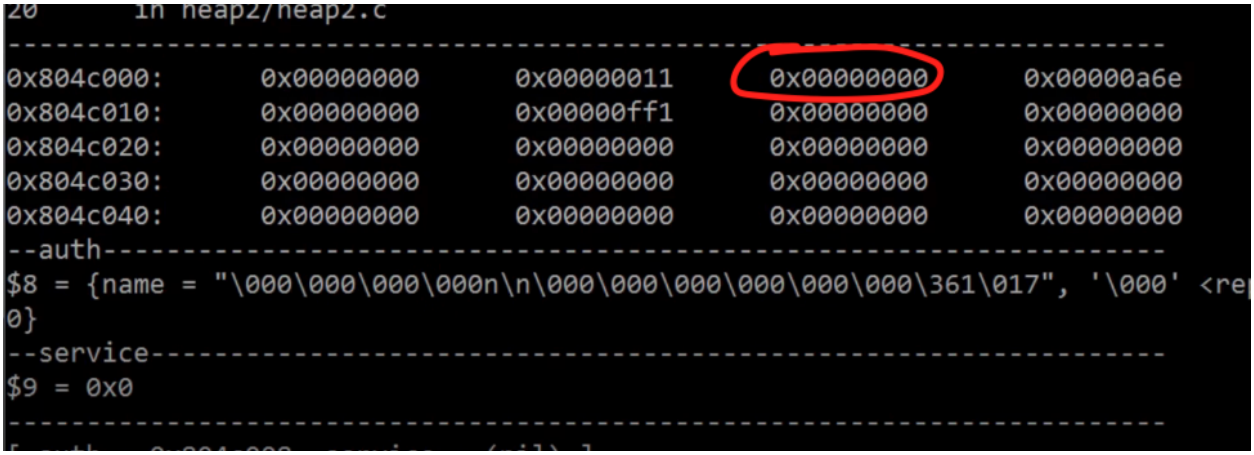
1. Create an `auth` struct
2. Free the `auth` struct
3. Duplicate a string onto the heap
4. Overflow it to overwrite the `auth` variable inside the `auth` struct.

The code for this program was written horribly which causes weird size differences in the memory but the idea is still the same.



Screenshot 4 from "The Heap: How Do Use-After-Free Exploits Work?"

Running the program and then authenticating with the word "admin" results in the heap layout above. The text "admin" is present in the first two "boxes" after "0x00000011." Due to the size of the `name` variable in the `auth` struct, the whole string goes up until the third line; So the `auth` variable itself is essentially the third row and third column (circled in red).

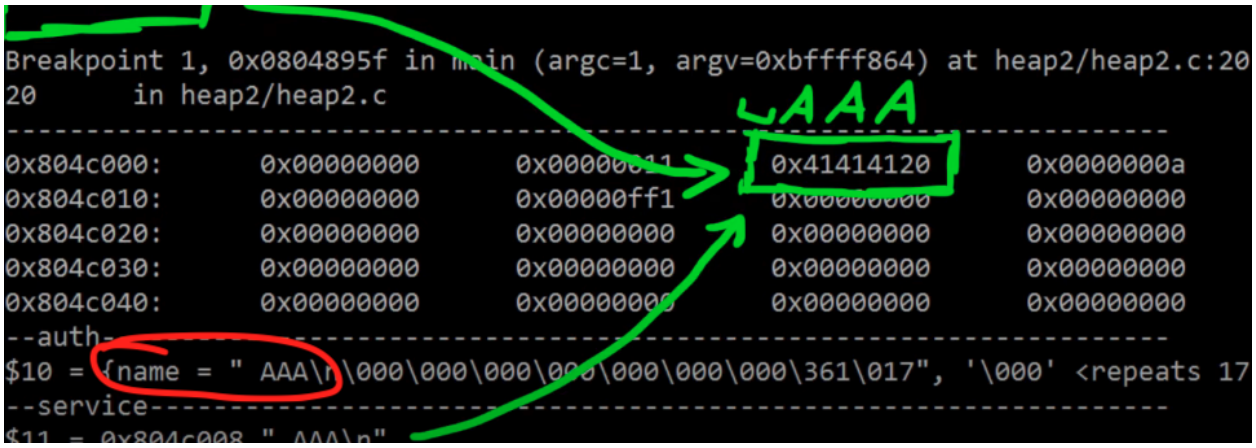


Screenshot 5 from "The Heap: How Do Use-After-Free Exploits Work?"

After freeing the `auth` struct, the heap layout would look like the above. So while the heap itself no longer contains anything related to the auth struct, the variable which originally created the space in the heap to store it, still contains an address that points to the location circled in red.

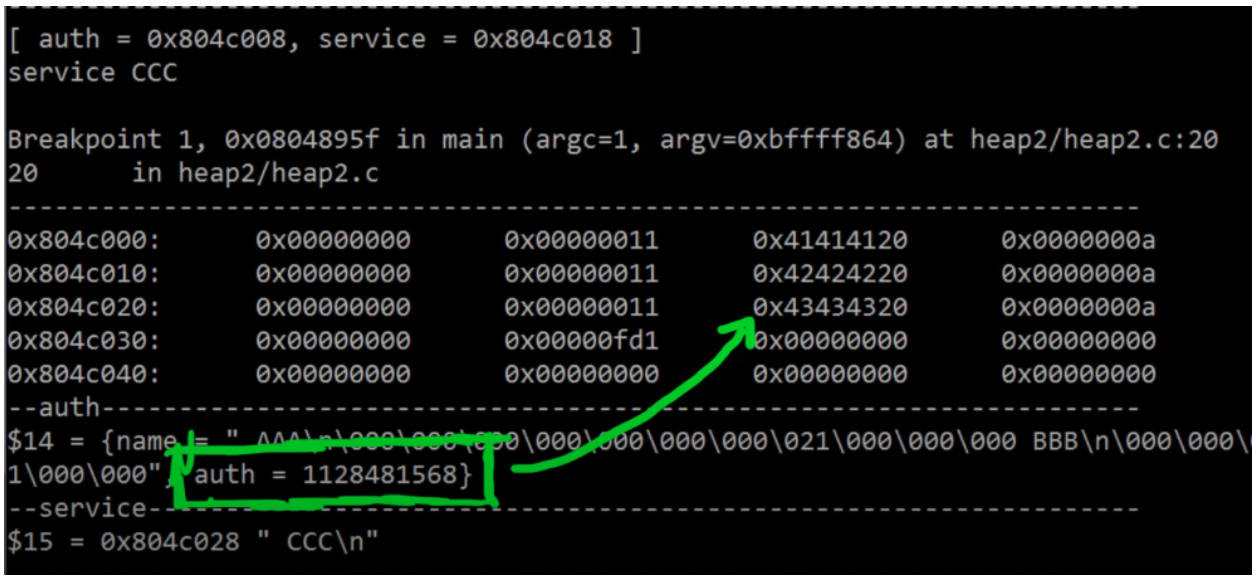
```
auth_ptr = malloc(sizeof(auth));
```

So, `auth_ptr`, when `malloc` is originally called, contains the address 0x804c008 (the location circled in red). Therefore, when the program later attempts to access the `auth` variable inside of the `auth` struct, after it has been freed, it will still access whatever data was present, starting at 0x804c008.



Screenshot 6 from "The Heap: How Do Use-After-Free Exploits Work?"

Before getting to the point mentioned, we still need to do the third step: "duplicate a string onto the heap." The above screenshot demonstrates what the heap looks like after duplicating a string. If you notice, the `name` variable in the `auth` struct now contains the characters "AAA" even though we freed the `auth` struct.



Screenshot 7 from "The Heap: How Do Use-After-Free Exploits Work?"

Continuing to duplicate strings shows that eventually, the auth variable is overwritten with characters. So when the code later tries to actually authenticate the user, it thinks the user is authenticated even though there is no `auth` struct.

```
if(strncmp(line, "login", 5) == 0) {
    if(auth_ptr->auth) {
        printf("you have logged in already!\n");
    } else {
        printf("please enter your password\n");
    }
}
```

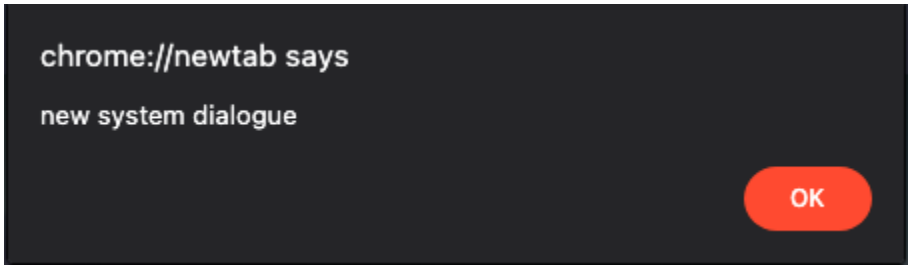
Here is the code that does what I just explained. When the user tries to “login,” the program will attempt to access the struct at 0x804c008 and get the value of the `auth` variable. That is essentially what a use-after-free exploit is; a struct was created in the heap and was later freed, but a variable still contains a pointer to where the struct was and the code continues to use that variable. Just learning memory exploitation techniques isn’t enough, however, because most modern browsers aren’t susceptible to basic buffer overflows like the ones I’ve learned about.

Understanding Browser Internals

When speaking about browsers, there are two main components that many people aim to exploit: the JavaScript engine and the sandbox. The JavaScript engine, as one could expect, is in charge of interpreting and running the JavaScript code that almost every website seems to have. The sandbox on the other hand aims to separate privilege levels, the reason for which will be apparent soon. For the most part, most browsers implement their JavaScript engine in C/C++, and most of the vulnerabilities in the JavaScript engine are memory corruption vulnerabilities. My reasoning for learning memory corruption vulnerabilities, more specifically use-after-free, was so I can attempt to find a vulnerability in the JavaScript engine. Before sandboxing was implemented in browsers, if an attacker found a vulnerability in the JavaScript engine, and gained arbitrary code execution, they “have full access over the process” (“What is a Browser Security Sandbox?!”). Since the browser is technically a privileged process that can access the filesystem and more, if the browser process is compromised, that means the attacker has full control over the system. Nowadays, there is a parent process that can do “everything that a normal process can do,” and every website runs in a “child process” (“What is a Browser Security Sandbox?!”). The child process is unprivileged and cannot access the filesystem or more privileged actions like that. In a way, it is similar to the difference between the administrator and standard user. The parent process can do things like creating files, calling other applications, etc; things the child process can’t do. This means that even if there is a vulnerability in the JS engine, the computer will still be safe since the attacker is “boxed” off from accessing other system resources. Each operating system provides different ways to sandbox a process. I was able to find resources and documentation to understand how macOS sandboxes a process, however, I have not been able to read those sources yet.

There are issues with sandboxing a website process and giving it fewer permissions. Websites are usually the processes that need to create files on the computer (cookies) or even open up a new system dialogue (like alert windows as seen in Screenshot 8). So to be able to

do things like that, many browsers implement a system called Inter-Process Communication (IPC).



Screenshot 8

The IPC layer is a communication protocol where the child process can request the parent process to do something, like write a cookie to the file system, and the parent process will do that ("What is a Browser Security Sandbox?!"). The IPC essentially separates privileges and the IPC in JavaScript is used in various parts of the user interface (Braun). Therefore, I need to and aim to, find a vulnerability in the browser sandbox implementation. According to LiveOverflow in "What is a Browser Security Sandbox?!," there are three ways to do so:

1) Attack the OS sandbox feature
2) Attack the IPC layer
3) Find a logic bug in the IPC communication layer

The first one is attacking the operating system itself rather than the browser, which is not what I'm trying to do: I'm attempting to find a browser vulnerability. The second option is feasible to do and would involve reading the C++ source code and fuzzing it to potentially find a memory corruption vulnerability. The third option seems the best for me since it doesn't entirely require me to understand C++ or even memory corruption exploits since the latter can be extremely complicated and difficult to master in a couple of months.

The next step, therefore, was to understand how the IPC works in Firefox, which I did so by reading a blog by Firefox themselves about their IPC layer in Firefox. According to Braun, the two most common patterns for using IPC from JavaScript are JSActors and MessageManagers. JSActors are the preferred method for JS code to communicate between processes, with one implementation in the child process and the counterpart in the parent. In addition, there is a separate parent instance for every pair so it can associate a message with either a specific content window (JSWindowActors - maybe to open a modal, like in Screenshot 8) or a child process (JSProcessActors - maybe some native user interface pieces like a dropdown select) (Braun). A child instance has two ways of sending messages: a one-off message with `sendAsyncMessage("someMessage", value)`, or if it needs a response, it can send a query with `sendQuery("someMessage", value)`. Parent instances then implement a `receiveMessage(msg)` function to handle all incoming messages. MessageManagers, on the other hand, is a bit outdated. Firefox is actually undergoing a transition into a new security architecture to achieve site isolation, moving from a "process per tab" to a "process per site." The transition, called Project Fission, started and is most probably in effect, considering the article was written in April 2021. The MessageManagers system was in use prior to the architecture change in Project Fission. Each tab would have multiple message managers and the JavaScript in both processes would register message listeners using the

`addMessageListener` methods and would send messages with `sendAsyncMessage`. To help keep track of messages, the message names are usually prefixed with the components they are used in. The `receiveMessage` function is where the untrusted information flows from the child into the parent process and "should therefore be the focus of additional scrutiny" (Braun). Following along with the blog, I was able to configure a Firefox browser to emulate a compromised child process and to explore the sandbox layer, and directly interface with the IPC layer. I even attempted to replicate an older sandbox bypass vulnerability, which failed; both of which I will tackle in the accomplishments section. In addition to reading about the IPC layer, I began reading about how Firefox does web security checks but was unable to finish reading the resource yet.

Understanding How to Fuzz

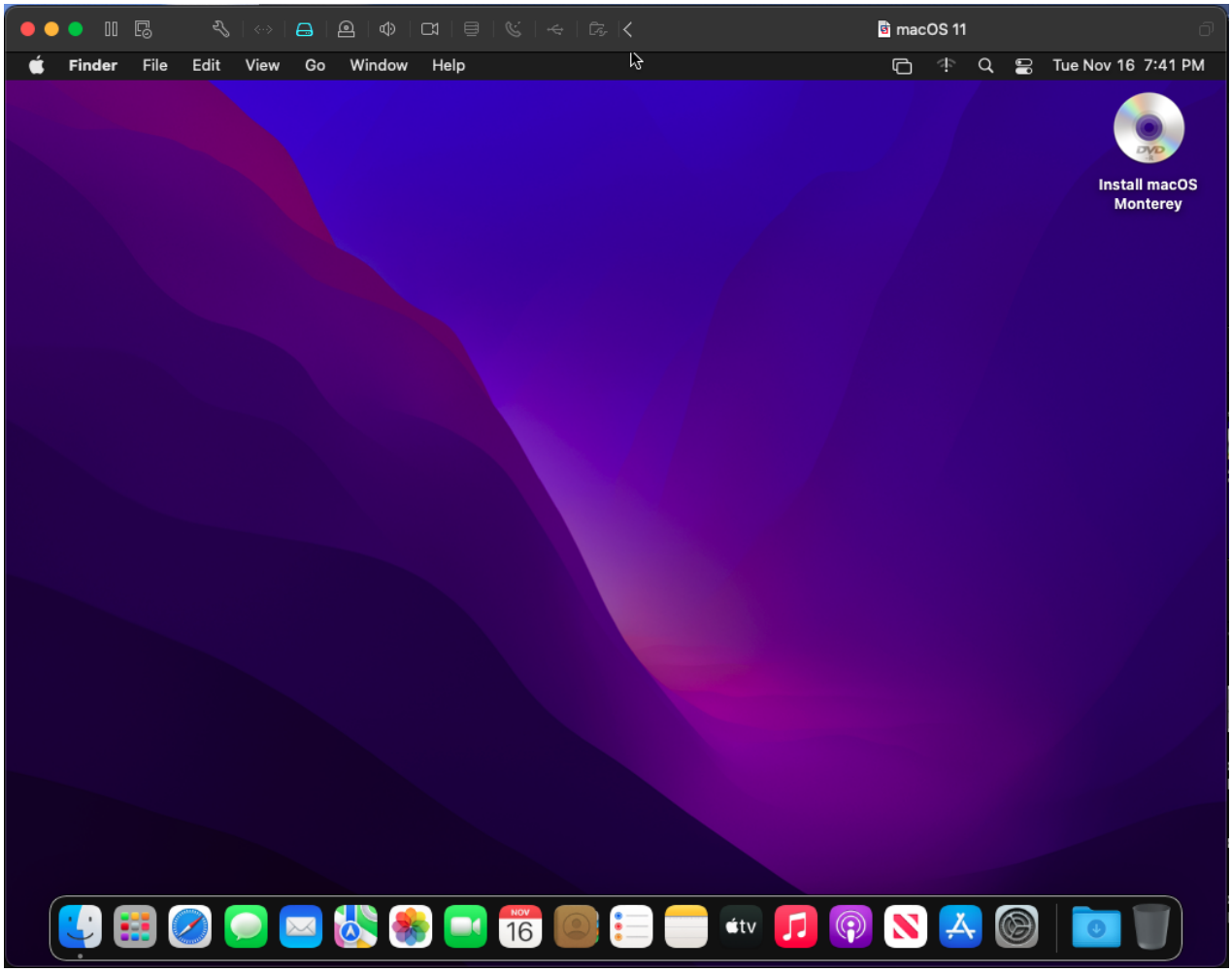
My advisor, Ned Williamson, sent over one of his talks on how he was able to fuzz the Chrome IPC layer. I watched it in hopes of getting a basic understanding of how to fuzz and finding more resources on fuzzing the Firefox IPC layer. One of the biggest recommendations Williamson had was to enumerate previous bug reports, something I incorporated into my timeline but need to spend more time on. Williamson recommended also attempting to find the bug myself without taking a look at the description of the bug to start building my skills of finding being able to find bugs.

Williamson also describes fuzzing as "randomly mutating bytes in current input corpus" and passing them to the browser, or small portions of the browser IPC layer. The biggest recommendation, and resource, I've gotten from watching the talk was about a library called libFuzzer, and implementation designed for fuzzing APIs. From libFuzzer's documentation itself, libFuzzer is a way to fuzz isolated code by keeping track of which areas of the code the fuzzed input reaches, and generates mutations on the corpus of input data. Reaching out to my advisor and asking for some clarifying questions helped me understand that when he was attempting to use libFuzzer, he compiled the portion of Chrome he was testing and linked the libFuzzer library with that portion to feed the fuzzed inputs to the library via a specific fuzzing endpoint. I had to ask clarifying questions because, in the talk, he did say to pick a self-contained chunk of code, play with the API until you figure out how to interact with it in an interesting way, mock around annoying bits (encryption, net, etc.), and then use a fuzzer. I didn't fully understand exactly how he was using libFuzzer which is why I asked him about that. The rest of the talk was dedicated to how the IPC is implemented in Chrome which was not pertinent to me. One last piece of useful information I did find, however, was that the Chromium (Chrome) source code should have an example fuzzer (src/net/quic/quic_stream_factory_fuzzer.cc in the Chromium source code).

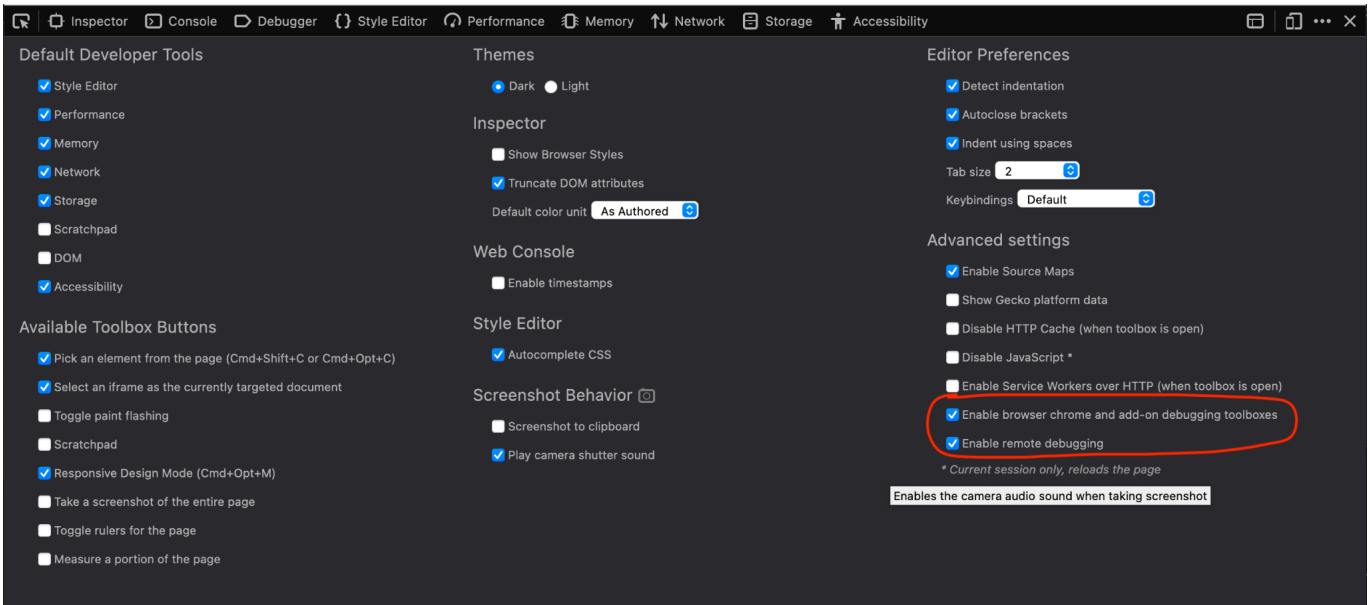
Conclusion

Most of my research for the past month was related to obtaining background information on some of the biggest topics I intend on doing this year through all my steps of attempting to find a browser exploit.

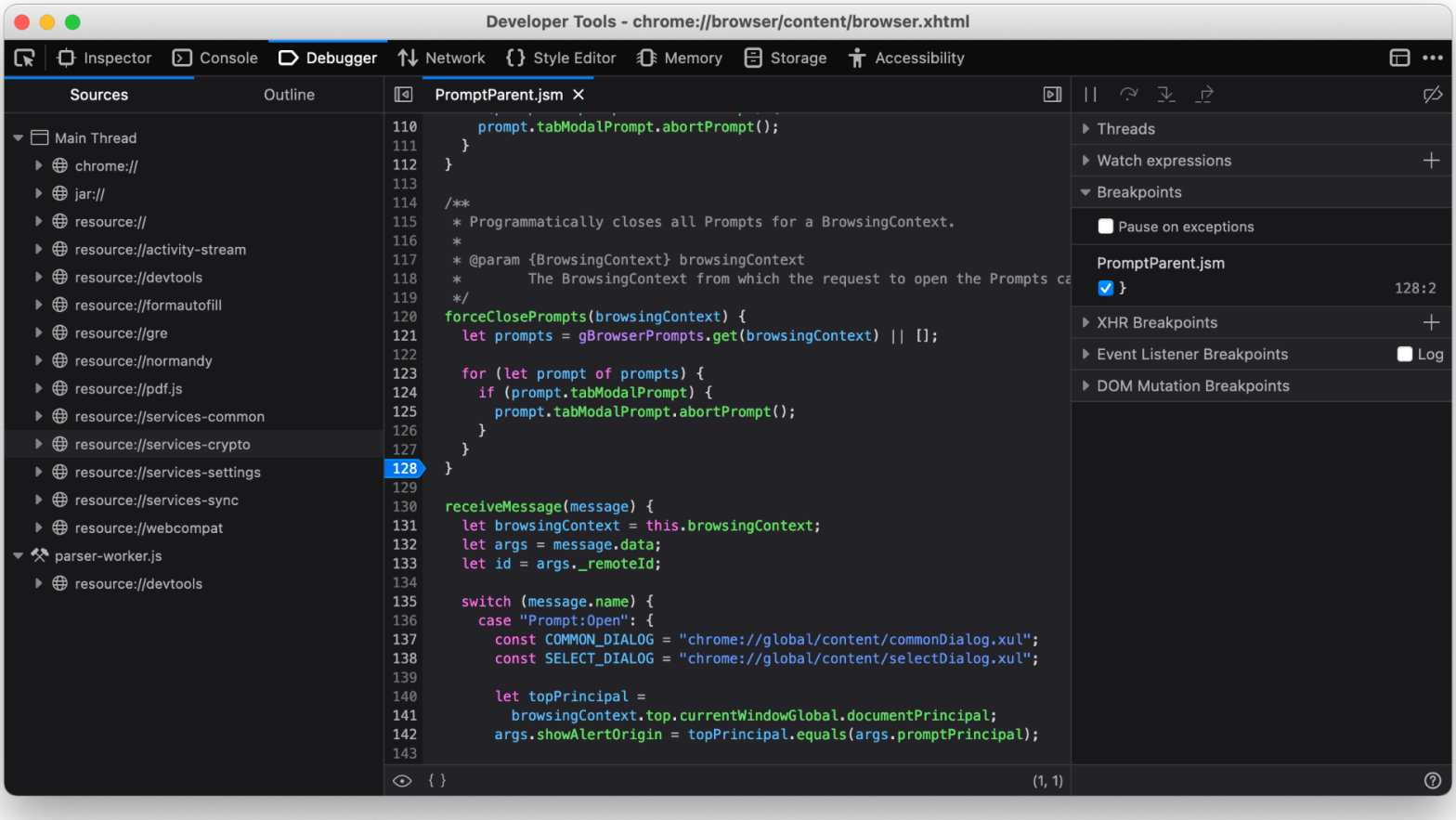
Accomplishments



- Was able to install and setup macOS VM on my computer
- Installed an older version of Firefox and turned on various settings like remote debugging and played around with Browser Content Toolbox and Browser Toolbox to experiment with the IPC layer



Had to turn on settings circled in red to get needed tools to test



Using the Browser Toolbox to view the code for a parent JSActor when trying to replicate an older bug

- Attempted to replicate bug for CVE-2019-11708 but was unable to. The bug was related to the Prompt handler shown in the screenshot above. The Prompt handler is responsible for showing the alert dialogues like those in Screenshot 8. The CVE had to do with the fact that an attacker could have the browser open a URL in the privileged parent process by passing in the URL to the above `Prompt:Open` handler. The attacker could then pass a malicious URL and gain access. This CVE was an example of a logic bug; there was insufficient vetting – ensuring the safety of process supplied arguments – of passed in parameters that led the privileged parent process to open a URL that could be malicious.
    - **NOTE:** CVE is just a security bug that was officially reported and contains lots of information about the severity of it and how it works.

Reflection on Goals and Timeline

I'm very behind on my timeline. I was expecting to have done a lot of research by now and been comfortable with testing Firefox and writing a simple fuzzing tool; I gravely underestimated the time it would take me, and as a result, will need to spend the majority of my winter break catching up and making sure I'm ready to start testing Firefox when we're back, if not already have started testing. According to my timeline, I should be reading the source code for Firefox and understanding the JS engine and sandboxing method. However, I am still a little more than halfway through my research phase as I have completed learning about some of the basics, but need to look more into Firefox-specific implementations. So, I'm close to being at the step of reading the Firefox source code, but I still have a ways to go. I need to read more about fuzzing, more specifically libFuzzer, and how to use that, as well as other security implementations by Firefox, like their web security checks. Looking back at my timeline, I believe I forgot to update the dates for some of my goals relating to the fuzzer tool which means I'm behind on that. So, that is something I need to update and be more realistic with. The biggest obstacle I've faced so far is trying to get the older CVE to work. While I was able to get an older version of Firefox installed, I attempted to find an exploit online and try to replicate it myself but was unable to do so. The process was valuable, however, because it prompted me to reach out to my advisor again and see how I would know if I found an exploitable bug or not, to which he replied, that it depends on experience and that he was willing to call with me and explain and walkthrough some bugs, potentially. So even though I am behind, I know clearly the direction I want to head in and have plans to meet with my advisor early into winter break to make sure I'm extremely productive and can catch up over winter break.