Attacking JavaScript Engines in 2022

Samuel Groß (@5aelo), Amy Burnett (@itszn13)
let v = 0x1337;
// typeof(v) == "number"

v: 0x1337
Basic JavaScript

```javascript
let v = 0x1337;
// typeof(v) == “number”
v = "foobar";
// typeof(v) == “string”
```
let v = 0x1337;
// typeof(v) == "number"
v = "foobar";
// typeof(v) == "string"
v = {a: 42, b: 43};
// typeof(v) == "object"
let o1 = {a: 42, b: 43};
console.log(o1.a);
Basic JavaScript

```javascript
let o1 = {a: 42, b: 43};
console.log(o1.a);
let o2 = {a: 13, b: 37};
```
Basic JavaScript

```javascript
let o1 = {a: 42, b: 43};
console.log(o1.a);
let o2 = {a: 13, b: 37};
o2.c = o1;
```
Basic JavaScript

```javascript
let o1 = {a: 42, b: 43};
console.log(o1.a);
let o2 = {a: 13, b: 37};
o2.c = o1;
```
Basic JavaScript

```javascript
let o1 = {a: 42, b: 43};
console.log(o1.a);
let o2 = {a: 13, b: 37};
o2.c = o1;
```
let o1 = {a: 42, b: 43};
console.log(o1.a);
let o2 = {a: 13, b: 37};
o2.c = o1;
delete o2.a;
function main() {
    console.log("Hello World!");
} main();
```javascript
function main() {
    console.log("Hello World!");
} main();
```
function main() {
    console.log("Hello World!");
} main();

Bytecode Compiler

Interpreter

Runtime
(objects, globals, constructors, functions, methods, …)

JIT Compiler(s)

Bytecode
LdaGlobal
Star1
LdaNamedProperty
Star0

Wasm Compiler(s)

Garbage Collector (GC)
function main() {
    console.log("Hello World!");
} main();

Interpreter

Wasm Compiler(s)

Runtime
(objects, globals, constructors, functions, methods, …)

Runtime State

JIT Compiler(s)

Bytecode

LdaGlobal
Star1
LdaNamedProperty
Star0

Bytecode Compiler

Garbage Collector (GC)

Wasm Compiler(s)
Interpreter

Wasm Compiler(s)

Runtime
(objects, globals, constructors, functions, methods, ...)

Garbage Collector (GC)

JIT Compiler(s)

Bytecode Compiler

Bytecode
LdaGlobal
Star1
LdaNamedProperty
Star0

Runtime State
Interpreter

Wasm Compiler(s)

Runtime
(objects, globals, constructors, functions, methods, …)

Runtime State

JIT Compiler(s)

Bytecode Compiler

Garbage Collector (GC)

Machine Code

add x3, x28, x3
ldr x4, [x26, #376]
cmp w3, w4
b.eq #x+0x23c
ldur w5, [x3, #1]

Wasm Machine Code

LdaGlobal Star1

LdaNamedProperty Star0
Interpreter

Wasm Compiler(s)

Runtime
(objects, globals, constructors, functions, methods, …)

Runtime State

Garbage Collector (GC)

JIT Compiler(s)

Bytecode Compiler

Machine Code

add x3, x28, x3
ldr x4, [x26, #376]
cmp w3, w4
b.eq #+0x23c
ldur w5, [x3, #-1]
Interpreter

Wasm Compiler(s)

Runtime
(objects, globals, constructors, functions, methods, …)

Runtime
(objects, globals, constructors, functions, methods, …)

Bytecode Compiler

JIT Compiler(s)

Bytecode
LdaGlobal
Start
LdaNamedProperty
Start

Machine Code
add x3, x28, x3
1dr x4, [x26, #376]
cmp w3, w4
b.eq #+0x23c
1dur w5, [x3, #-1]

Wasm Compiler(s)

Garbage Collector (GC)
JIT Compilation
A (Hypothetical) JIT Optimization Example

```javascript
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);

function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}
```
Example: “Training” the JIT

```javascript
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);

function set(p, v) {
    if (p.x < 0 || p.x >= W ||
        p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}

// "Train" the JIT
for (let i = 0; i < 10000; i++) {
    set({x: 1, y: 2}, 3);
}
```
Example: Bytecode Parsing

```javascript
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);
function set(p, v) {
  if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
    throw "invalid point";
  }
  bmp[p.x * W + p.y] = v;
}
x1 = LoadProperty p, 'x'
GotoIf .throwException, x1 < 0
x2 = LoadProperty p, 'x'
GotoIf .throwException, x2 >= 64
```
Example: Speculation + Lowering

```javascript
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);
function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}
```

```javascript
CheckType p, ObjType1
x1 = LoadField p, +8
GotoIf .throwException, x1 < 0
CheckType p, ObjType1
x2 = LoadField p, +8
GotoIf .throwException, x2 >= 64
```
Example: Speculation + Lowering

```javascript
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);
function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}
```

```javascript
CheckType p, ObjType1
x1 = LoadField p, +8
GotoIf .throwException, x1 < 0
CheckType p, ObjType1
x2 = LoadField p, +8
GotoIf .throwException, x2 >= 64
```
Example: Redundancy Elimination

const W = 64, H = 64;
const bmp = new Uint8Array(W * H);

function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}

CheckType p, ObjType1
x1 = LoadField p, +8
GotoIf .throwException, x1 < 0
CheckType p, ObjType1
x2 = LoadField p, +8
GotoIf .throwException, x1 >= 64
Example: Bytecode Parsing

```
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);
function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}
```
Example: Constant Folding + Lowering

```
const W = 64, H = 64;
const bmp = new Uint8Array(W * H);
function set(p, v) {
    if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
        throw "invalid point";
    }
    bmp[p.x * W + p.y] = v;
}
i1 = IntegerMul x, 64
i2 = IntegerAdd i1, y
CheckBounds i2, 4096
CheckType v, Uint8
StoreUint8Array bmp, i2, v
```
Example: Range Analysis + Bounds Check Elimination

const W = 64, H = 64; // x = Range [0, 64)
const bmp = new Uint8Array(W * H); // y = Range [0, 64)
function set(p, v) { // i1 = Range [0, 4033)
i1 = IntegerMul x, 64 // i2 = Range [0, 4096)
i2 = IntegerAdd i1, y
if (p.x < 0 || p.x >= W || // i2 = Range [0, 4096)
p.y < 0 || p.y >= H) {
    throw "invalid point";
}
    bmp[p.x * W + p.y] = v;

    CheckBounds i2, 4096
    ...
Example: Final JIT IR Code

const W = 64, H = 64;
const bmp = new Uint8Array(W * H);

function set(p, v) {
  if (p.x < 0 || p.x >= W || p.y < 0 || p.y >= H) {
    throw "invalid point";
  }
  bmp[p.x * W + p.y] = v;
}

CheckType p, ObjType1
x = LoadField p, +8
y = LoadField p, +16
GotoIf .throwException x < 0 || ...
i1 = IntegerMul x, 64
i2 = IntegerAdd i1, y
CheckType v, Uint8
StoreUint8Array bmp, i2, v
JIT Compilation (simplified)

- **Unoptimized Bytecode**
  - Feedback from Past Executions

- **Speculation**
  - (insert type checks based on feedback)

- **Optimization**
  - (mostly remove unnecessary stuff, but also, e.g., move things out of loops, …)

- **Lowering**
  - (convert higher-level IR to lower-level IR, ultimately to machine code)

- **Static Analysis of Input Code**
- **Runtime State of Various Objects**

- **Optimized Machine Code**

---
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => e == 3, 42);
// a == [0, 1, 2, 42, 4, 5];
```

**Description**

The `findIndex()` method executes the `callbackFn` function once for every index in the array until it finds the one where `callbackFn` returns a truthy value.
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => e == 3, 42);
// a == [0, 1, 2, 42, 4, 5];
```

```
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => e == 3, 42);
// a == [0, 1, 2, 42, 4, 5];
```
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => e == 3, 42);
// a == [0, 1, 2, 42, 4, 5];
```
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}
let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => false, 42);
```

**Return value**

The index of the first element in the array that passes the test. Otherwise, -1.
function replace(a, cond, v) {
  let i = a.findIndex(cond);
  a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => false, 42);

The index of the first element in the array that passes the test. Otherwise, -1.
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}
```

```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [-1, a.length - 1)
Check i >= 0
StoreArray a, i, v
```
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => {
    a.length = 0; return true;
}, 42);
```

CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [-1, a.length - 1)
Check i >= 0
StoreArray a, i, v
A (Hypothetical) JIT Bug Example

```javascript
function replace(a, cond, v) {
    let i = a.findIndex(cond);
    a[i] = v;
}

let a = [0, 1, 2, 3, 4, 5];
replace(a, (e) => {  
a.length = 0; return true;
}, 42);
```

CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [-1, a.length - 1)
Check i >= 0
StoreArray a, i, v
Optimization
Analysis
Other
“Breaks”

Type Safety

Spatial Memory Safety

Temporal Memory Safety

“Breaks”
Type Safety

Type-Check Elimination

Type Inference

Temporal Memory Safety

BCE (bounds-check elimination)

Spatial Memory Safety

Range Analysis

Optimization

Analysis

Other

“Breaks”

CVE-2018-4233
CVE-2018-17463
CVE-2019-11707
CVE-2020-6418

 CVE-2017-762874
 CVE-2019-13764

CheckType o, ObjType1...
CheckType o, ObjType1
let tmp1 = x + y;
...
let tmp2 = x + y; tmp1;
const x = 42;
for (let i = 0; i < 100; i++) {
  ...
  a[x] = 1337;
}

---

```
const x = 42;
for (let i = 0; i < 100; i++) {
  ...
  a[x] = 1337;
}```
Exploitation
- Choose (arbitrary) victim array
- Choose (arbitrary) OOB index
- Choose read or write access
- Trigger bug to corrupt memory

Exploitation
- Choose (arbitrary) victim type
- Choose (arbitrary) target type
- Choose (arbitrary) operation
- Trigger bug to confuse objects

Exploitation
- Choose (arbitrary) victim type
- Choose (arbitrary) replacement type
- Trigger bug and GC to cause UaF
JS Outside JIT
• Plenty of complexity elsewhere
• Few bug patterns, many “1-off” bugs
Exploitation & Mitigations
Exploit Flow  

Circa 2016

Bug

Create Fake JOBJECT

Corrupt Existing JOBJECT

Arbitrary Memory Read & Write

JIT Overwrite “Shellcode” Sandbox Escape
What About Classical Mitigations?

- ASLR: Usually easy to construct a leak via type confusion or OOB (second bug not required)
- DEP/NX: JIT provides easy ways to map shellcode
- Stack Cookies: Most JS bugs are heap based

Can OOB read and write with same bug

| Victim Array | 0 | 1 | 2 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Other Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
"Modern" Mitigations

When most people think of modern mitigations they think of Control Flow Integrity (CFI)

Armv8.3+: Pointer Authentication (PAC) and Branch Target Identification (BTI)

Intel: Shadow Stack and Control-flow Enforcement Technology (CET)

Windows: Control Flow Guard (CFG)
"Modern" Mitigations - Not Quite There Yet

When most people think of modern mitigations they think of Control Flow Integrity (CFI)

Armv8.3+: Pointer Authentication (PAC) and Branch Target Identification (BTI)

Intel: Shadow Stack and Control Flow Enforcement Technology (CET)

Windows: Control Flow Guard (CFG)

JSC supports PAC
V8 does not yet have full support for CET, CFG, or PAC
**Pointer Authentication**

Newer iOS devices and M1 Macbooks benefit from Armv8's Pointer Authentication

- PAC*: signs the pointer, writes cryptographic signature to upper bits
- AUT*: verifies the pointer

 Mostly used to protect code pointers, but may be used for data as well

```
0x00007fc75ae25b20          0xa9b6414141414141
           ↑               ↑
0xa9b67fc75ae25b20          0x8000414141414141
```
Bypassing Pointer Authentication

PAC bypasses can be considered similar to bugs; ie patched quickly if disclosed

Example Bypass Methods

- Pointer Forgery: Writable memory which later gets signed [ref]
- Swap or use signed pointers which lack context

```assembly
ADRP   X16, #_pow_ptr_3@PAGE
LDR    X16, [X16,#_pow_ptr_3@PAGEOFF]
PACIZA X16

Writable Mem
0x41414141

0x66c0000414141
```

Additionally, V8 currently supports PAC, but not in JITed code [ref]
"Scripted" Code Execution

If you can't get arbitrary asm code, you may be able to call existing functionality.

Build control flow with manipulated calls / actions made from JavaScript.

Required sandbox escape functionality usually already exists!

Good Example: ObjectiveC Selector Calls [ref][ref]

corrupted_web_obj.do_action()
The Rise Of Data-Only Attacks

On PAC devices and as CFI rolls out, shellcode/rop exec is becoming harder…

However, this is usually not the endgame of a JS exploit

Exploits may attempt to attack cross-process data integrity / confidentiality

- Corrupt IPC data / messages / state to exploit a sandbox bug
- Read sensitive data stored within the process itself

These attacks do not rely on code exec, only memory read and write
Exploitation Tricks: Winning Races With Linked Lists

A lot of data attacks become races: Either

- You complete the write in time
- You smash some other data and crash…

We can abuse linked list structures to stall this race [ref]

Corrupt link list to form a cycle
Wait until program hits loop
Perform write to target data
Repair linked list and allow thread to resume
Attacking Cross-Origin

We have control of all the data in the compromised process

- Force the process to load sensitive data
- Inject JavaScript into other website -> hijack session
- Abuse persistent data features in other websites [ref]
Mitigating Cross Origin Attacks

Chrome and Firefox have enabled "Site Isolation"

- Iframes are in separate processes
- Requests and access enforced by the network IPC
Multiple Endgames

Bug

Create Fake JSObject

Corrupt Existing JSObject

“Shellcode” Sandbox Escape

“Scripted” Sandbox Escape

Data-Only Sandbox Escape

Renderer Data Exfiltration

Arbitrary Memory Read & Write
Mitigating Arbitrary Read / Write

Arbitrary read/write is a very powerful primitive

Thus, vendors are creating mitigations to make it more difficult

Pointer Caging - Code restricts pointers to specific regions of memory
JavaScriptCore's GigaCage

GigaCage prevents pointers being used to corrupt sensitive memory

class JSArrayBufferView {
    using VectorPtr = CagedPtr<Gigacage::Primitive, void, tagCagedPtr>;
    VectorPtr m_vector;
}

CagedPtr forces all pointer accesses to remain in a specific "GigaCage" region
JavaScriptCore's GigaCage
Is GigaCage Effective?

Required to protect a "vulnerable" pointer:

- Explicit caged typing of the pointer
- Correct uncaging implementation when accessing (such as in the JIT)

There are a lot of objects and a lot of pointers

- Attackers just need to find single uncaged pointer they can r/w from
- This is made easier by faking object state
Is GigaCage Effective?

Current easiest method: make a fake JSArray...

Slightly limited R/W, but allows corrupting more complex structures elsewhere.
Moving Towards A Heap Sandbox

Attackers will continue to find objects with corruptible pointers

Why not constrain the entire JS Heap?

- JavaScript manages many "external pointers" to browser memory
Moving Towards A Heap Sandbox

Attackers will continue to find objects with corruptible pointers

Why not constrain the entire JS Heap?

- JavaScript manages many "external pointers" to browser memory

  Solution: Hold these pointers outside the heap and reference with index #
Future V8 Heap Sandbox

All JS objects confined to sandbox memory
All other sensitive memory is outside:
- External pointers (and type) in table
- JIT compiler structures and code
- Any reference to other memory

Exploit now relies on unsound behavior of external objects and code it has handles to (similar to a sandbox escape...)
Exploit Flow Today

- Bug
  - Fake Own JSObject
  - Corrupt Existing JSObject

Arbitrary Memory Read & Write

- “Shellcode” Sandbox Escape
- “Scripted” Sandbox Escape
- Data-Only Sandbox Escape
- Renderer Data Exfiltration

Limited Memory Read & Write
What Have We Learned

Fewer bug classes, instead more “1-off” bugs, more complex JIT bugs

No significant changes to “early” exploitation phase (Same primitives available)

Current mitigations are not fully effective or applied evenly

Future mitigations seem more promising! (But still not bulletproof)