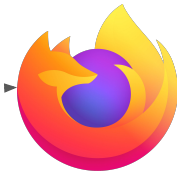
V8 / Chromium / Chrome

Attacking JavaScript Engines in 2022

Samuel Groß (@5aelo), Amy Burnett (@itszn13)

JavaScriptCore / WebKit / Safari

Spidermonkey / Gecko / Firefox



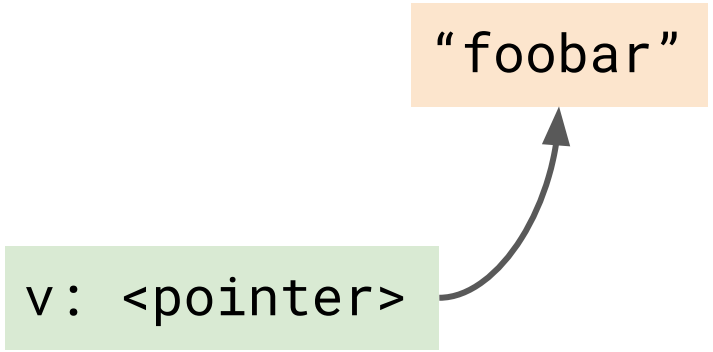
Basic JavaScript

```
let v = 0x1337;
// typeof(v) == “number”
```

```
v: 0x1337
```

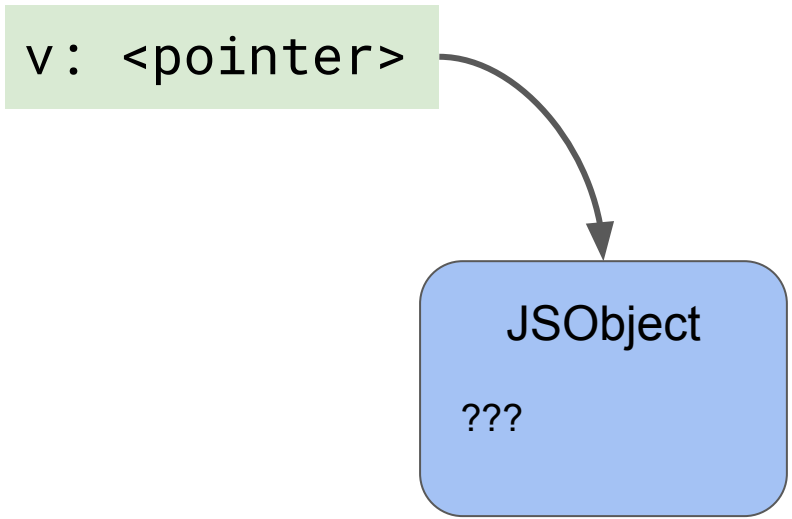
Basic JavaScript

```
let v = 0x1337;
// typeof(v) == “number”
v = "foobar";
// typeof(v) == “string”
```



Basic JavaScript

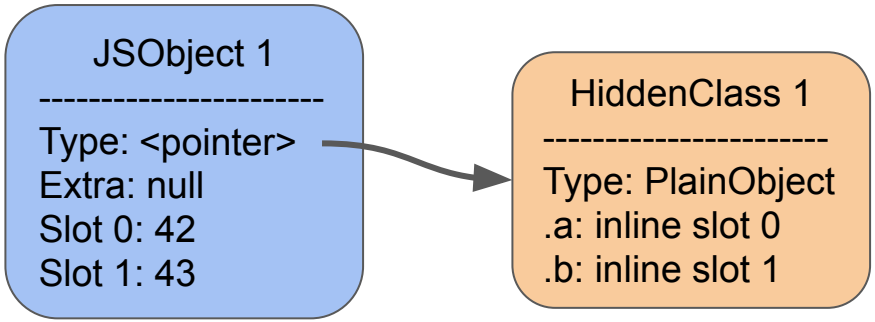
```
let v = 0x1337;
// typeof(v) == “number”
v = "foobar";
// typeof(v) == “string”
v = {a: 42, b: 43};
// typeof(v) == “object”
```



Basic JavaScript

```
let o1 = {a: 42, b: 43};

console.log(o1.a);
```

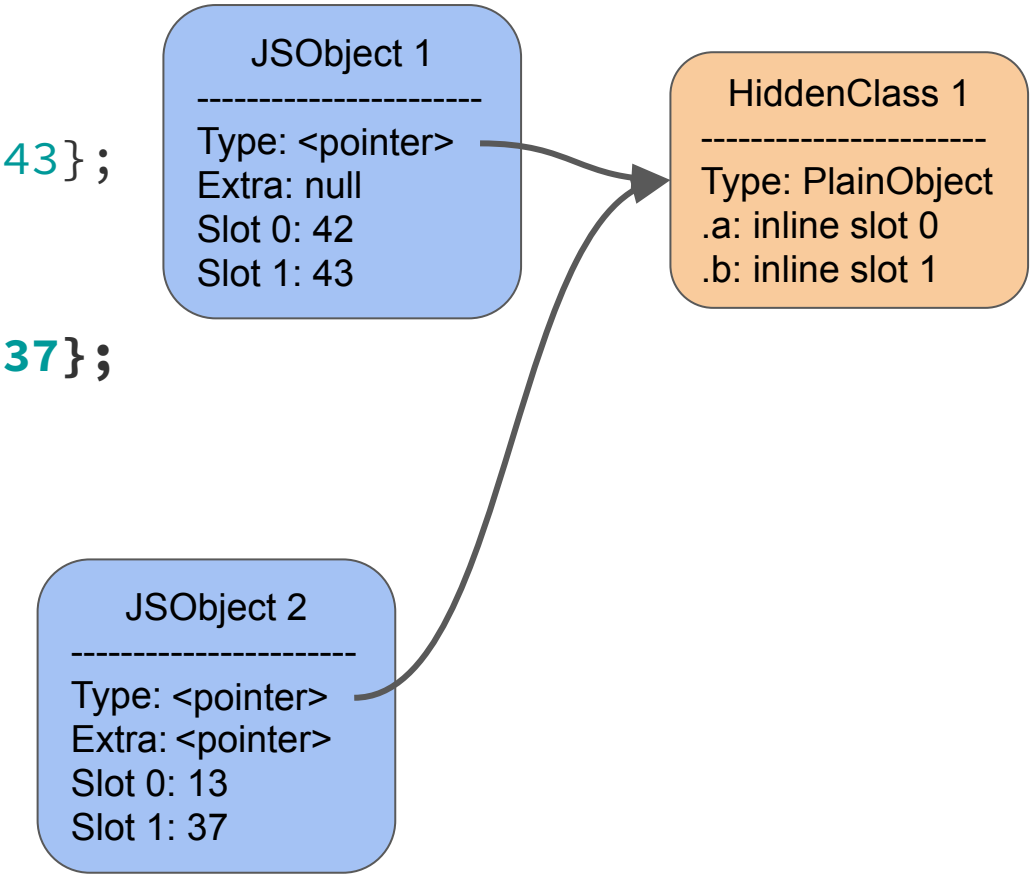


Basic JavaScript

```
let o1 = {a: 42, b: 43};

console.log(o1.a);

let o2 = {a: 13, b: 37};
```



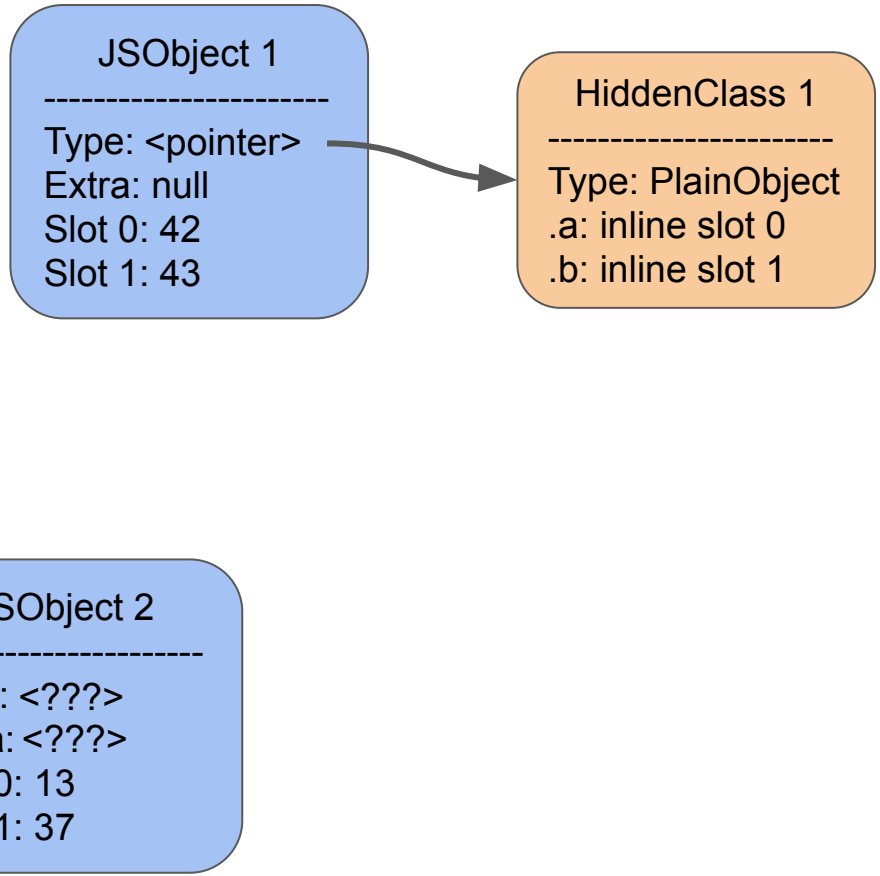
Basic JavaScript

```
let o1 = {a: 42, b: 43};

console.log(o1.a);

let o2 = {a: 13, b: 37};

o2.c = o1;
```



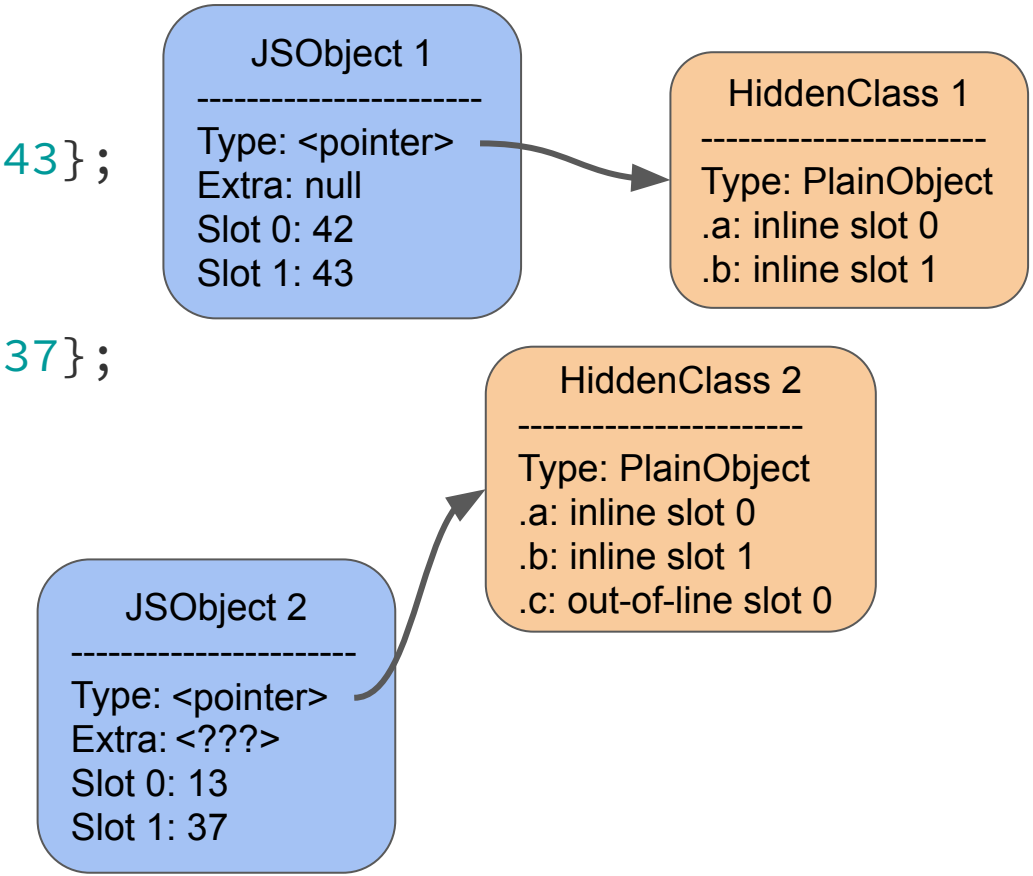
Basic JavaScript

```
let o1 = {a: 42, b: 43};

console.log(o1.a);

let o2 = {a: 13, b: 37};

o2.c = o1;
```



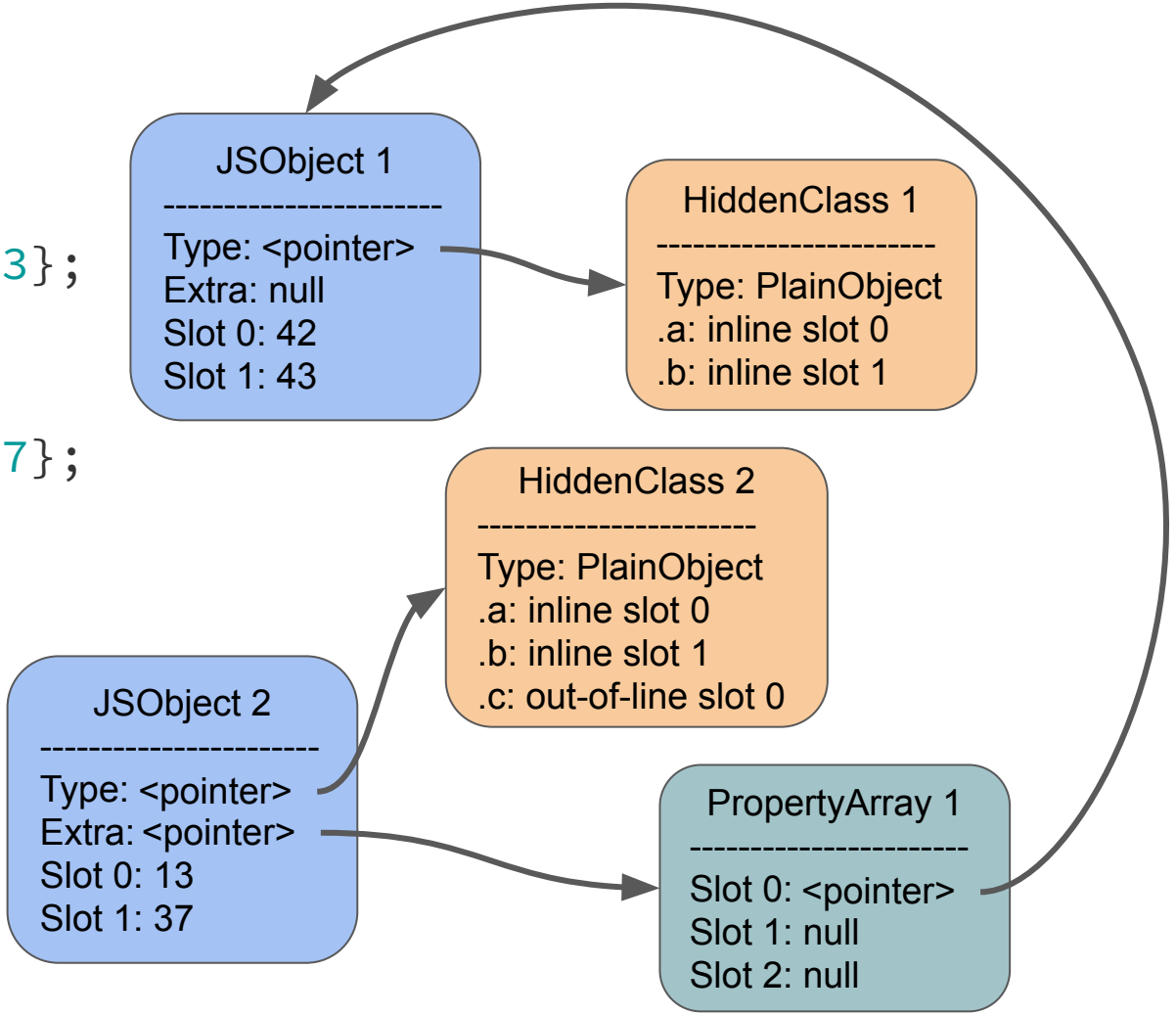
Basic JavaScript

```
let o1 = {a: 42, b: 43};

console.log(o1.a);

let o2 = {a: 13, b: 37};

o2.c = o1;
```



Basic JavaScript

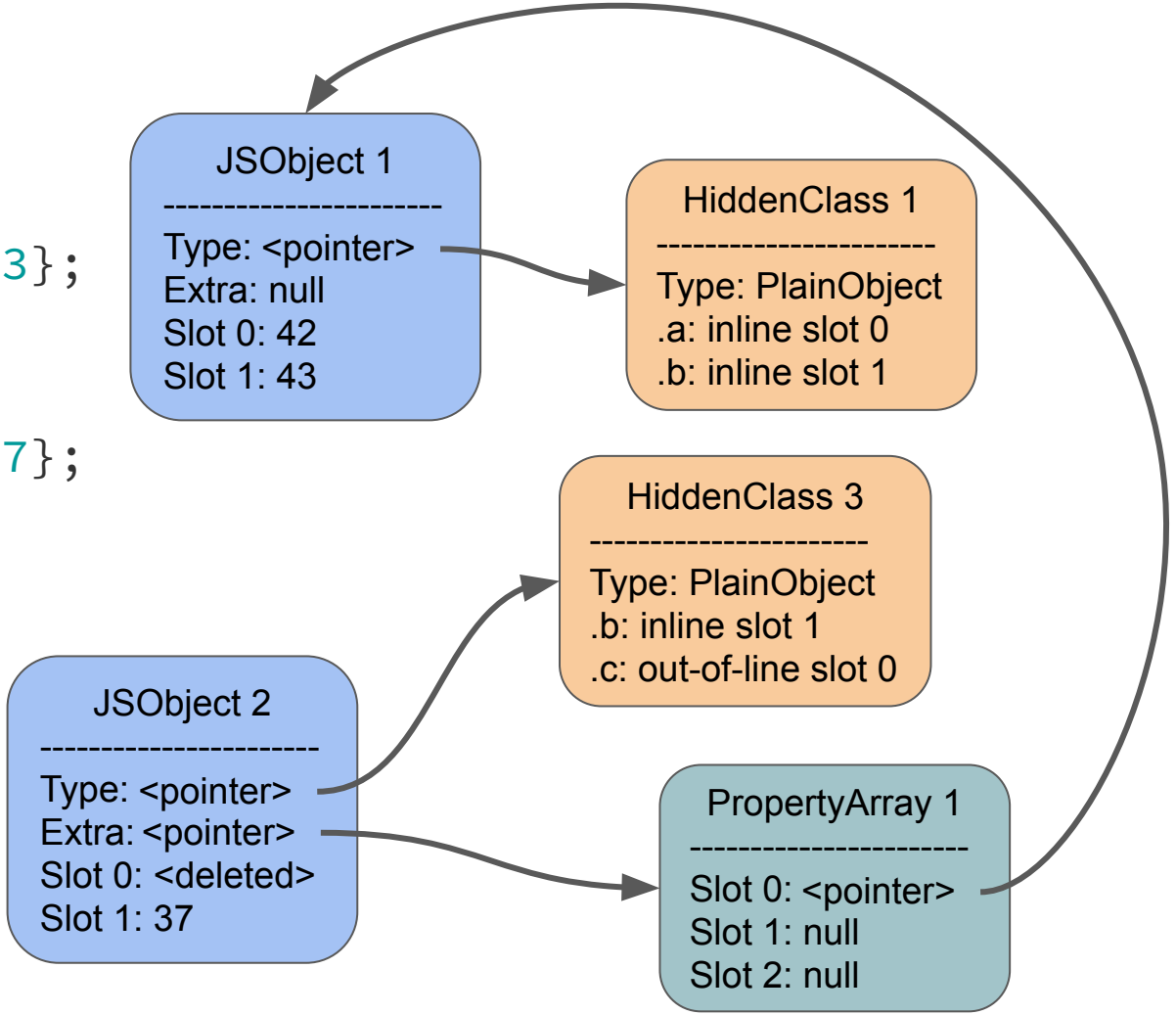
```
let o1 = {a: 42, b: 43};

console.log(o1.a);

let o2 = {a: 13, b: 37};

o2.c = o1;

delete o2.a;
```



Interpreter

Bytecode Compiler

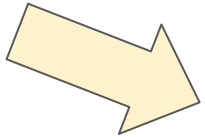
Runtime
(objects, globals, constructors, functions, methods, …)

JIT Compiler(s)

Garbage Collector (GC)

Wasm Compiler(s)

```
function main() {
    console.log("Hello World!");
}
main();
```



Bytecode Compiler

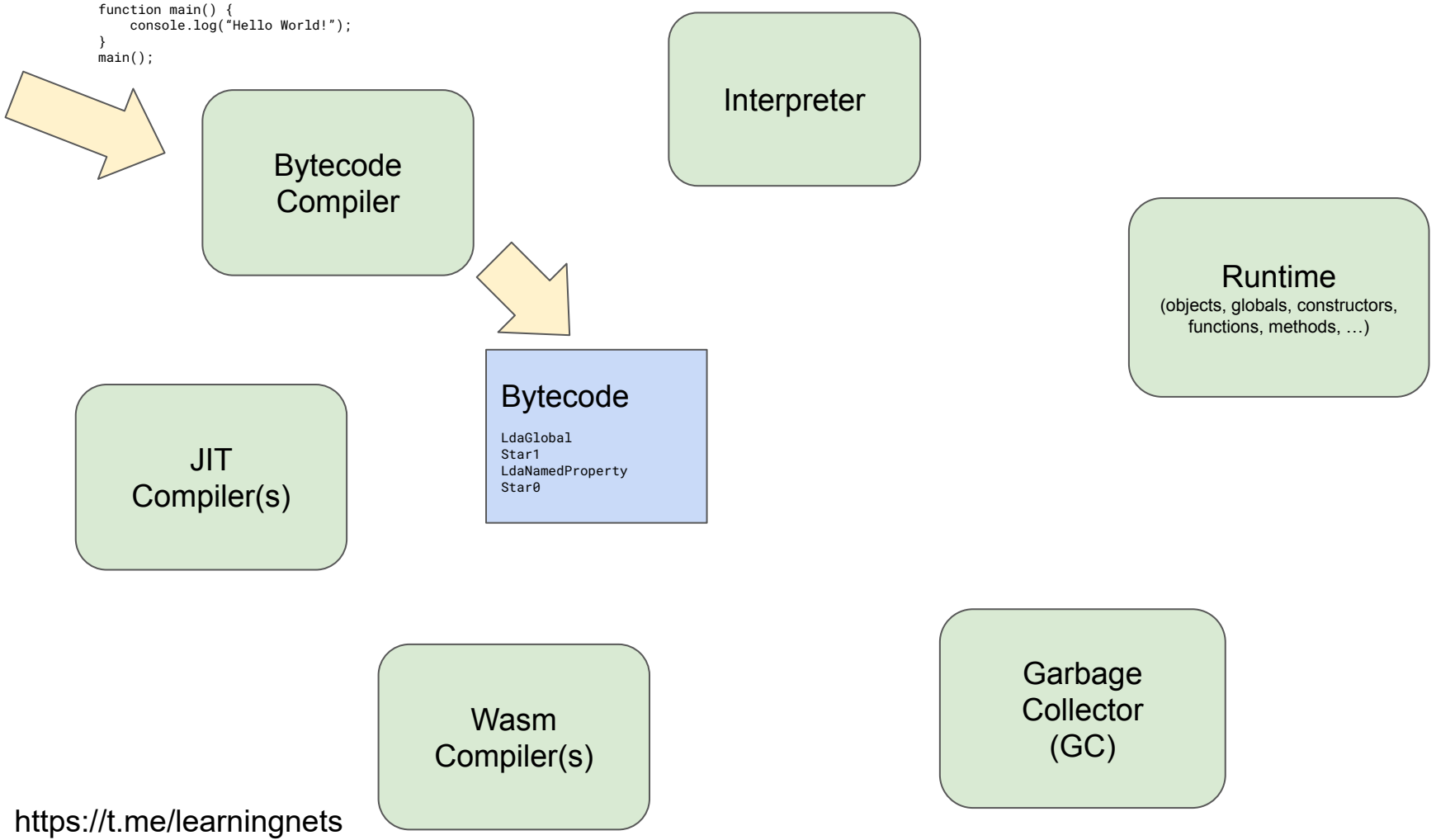
Interpreter

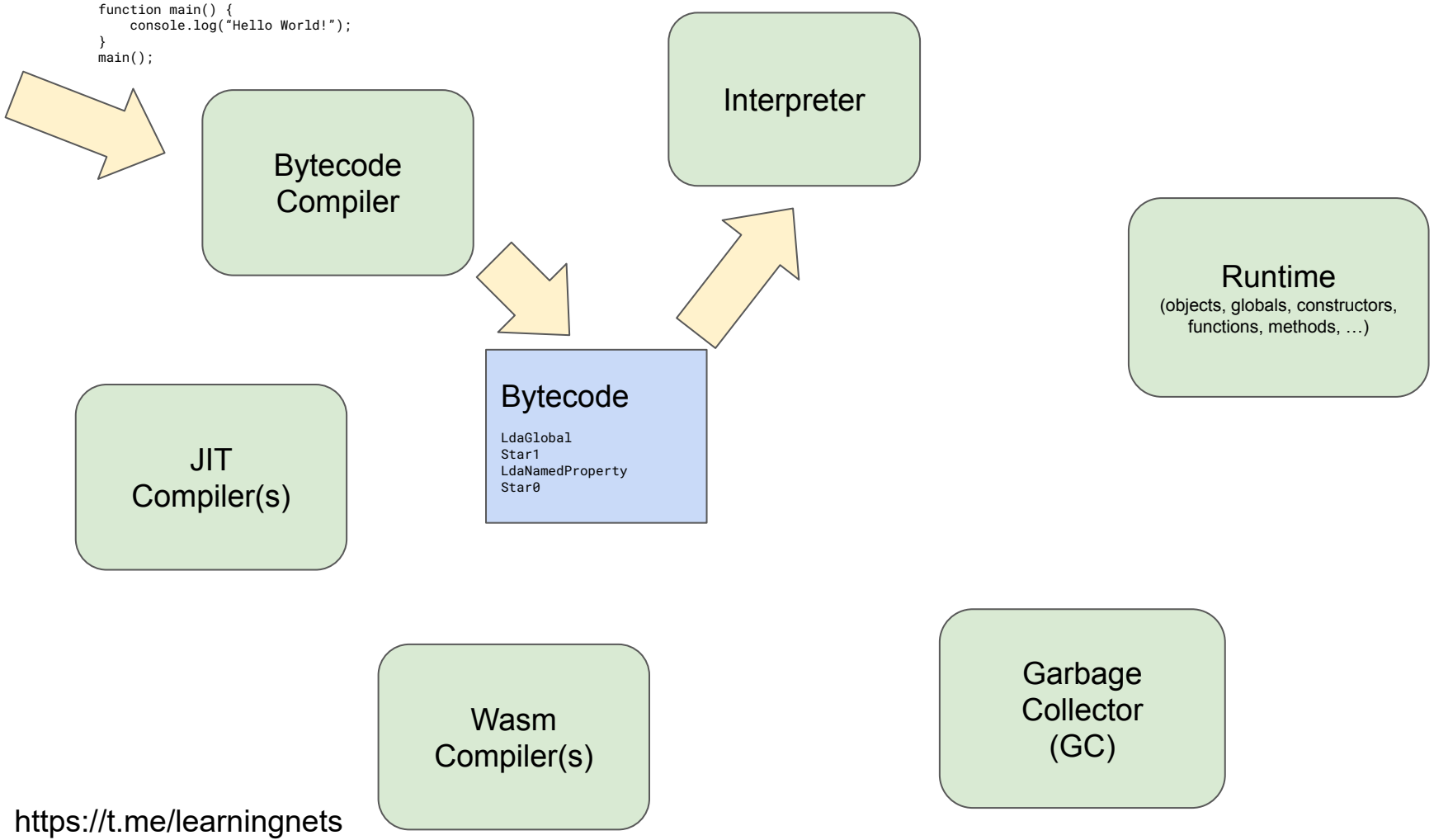
Runtime
(objects, globals, constructors, functions, methods, …)

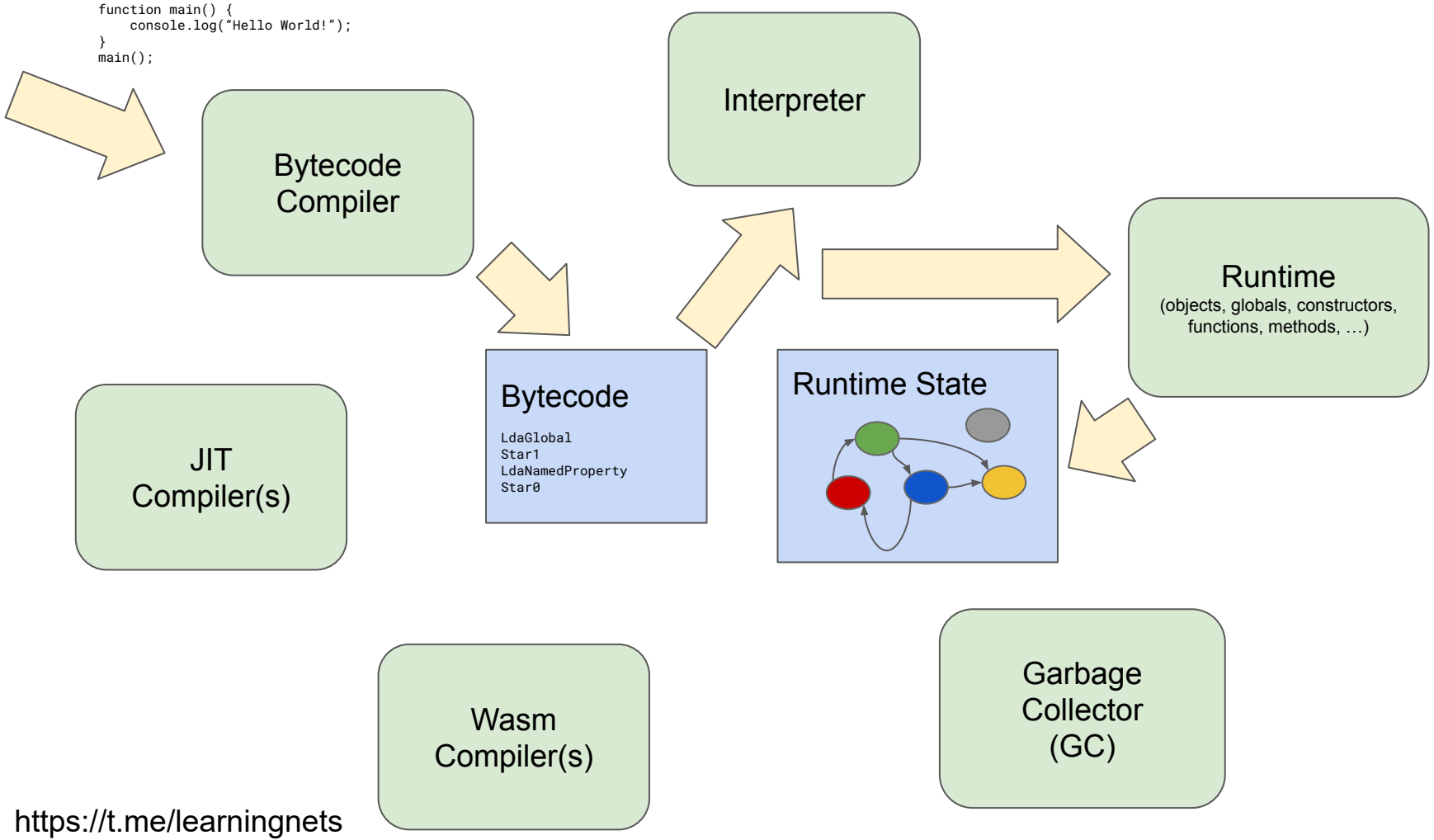
JIT Compiler(s)

Wasm Compiler(s)

Garbage Collector (GC)







Interpreter

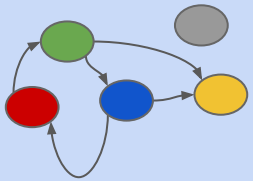
Bytecode Compiler

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

Bytecode

```
LdaGlobal
Star1
LdaNamedProperty
Star0
```

Runtime State



JIT Compiler(s)

Wasm Compiler(s)

Garbage Collector (GC)

Bytecode Compiler

Interpreter

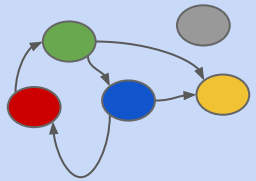
Runtime
(objects, globals, constructors, functions, methods, …)

JIT Compiler(s)

Machine Code

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

Runtime State



Wasm Compiler(s)

Garbage Collector (GC)

Interpreter

Bytecode Compiler

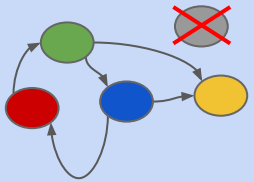
Runtime
(objects, globals, constructors, functions, methods, …)

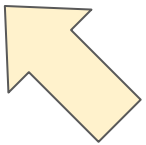
JIT Compiler(s)

Machine Code

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

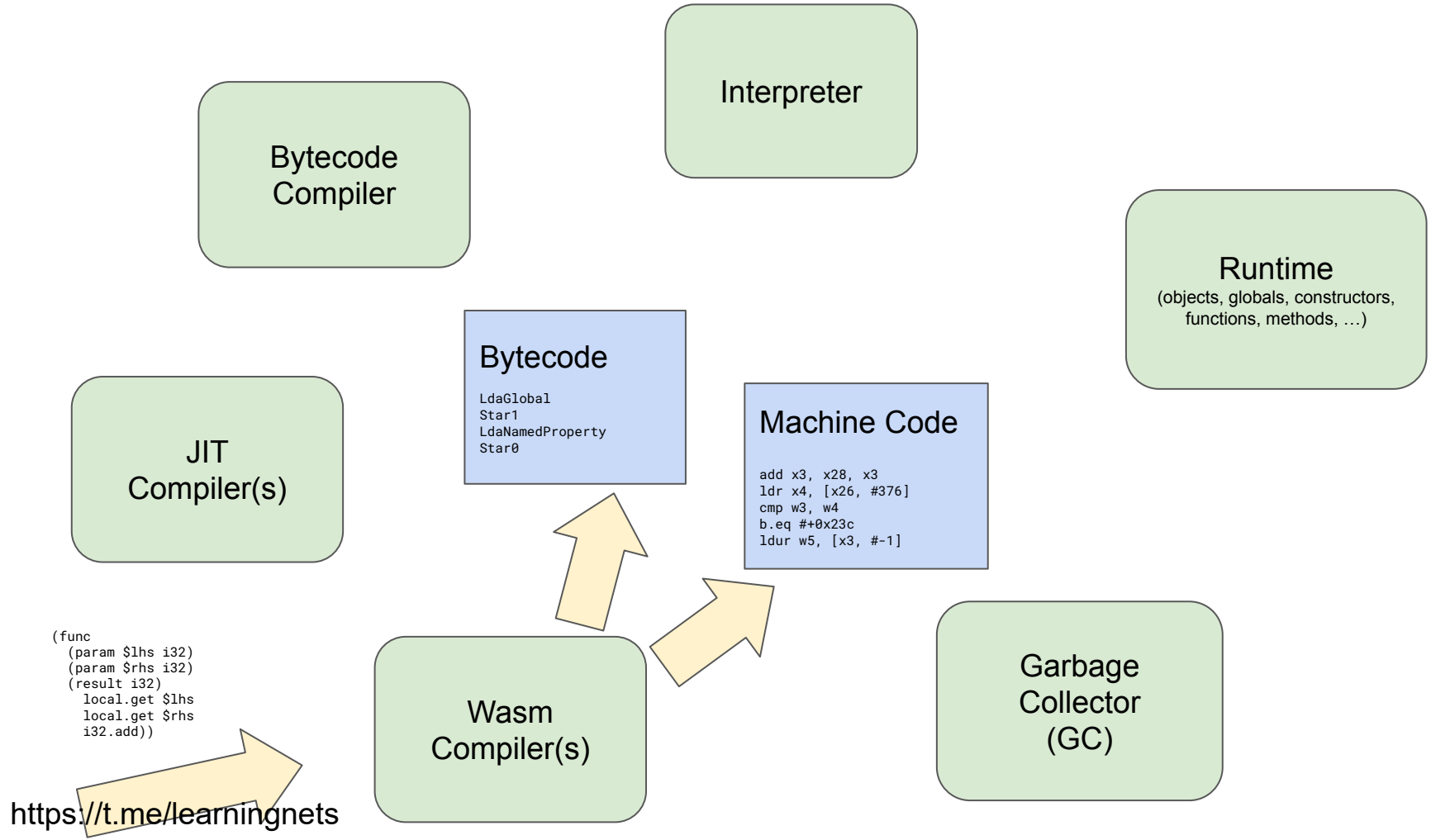
Runtime State





Garbage Collector (GC)

Wasm Compiler(s)



JIT Compilation

A (Hypothetical) JIT Optimization Example

```
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

```
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

```
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

```
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, x2 >= 64
```

Example: Speculation + 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;
}
```

```
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;
}
```

```
W = LoadGlobal ‘W’
i1 = Mul x, W
i2 = Add i1, y
bmp = LoadGlobal ‘bmp’
StoreElememt bmp, i2, 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;
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;
}
```

```
// x = Range [0, 64)
// y = Range [0, 64)
i1 = IntegerMul x, 64
// i1 = Range [0, 4033)
i2 = IntegerAdd i1, y
// i2 = Range [0, 4096)
~~CheckBounds i2, 4096~~
...
```

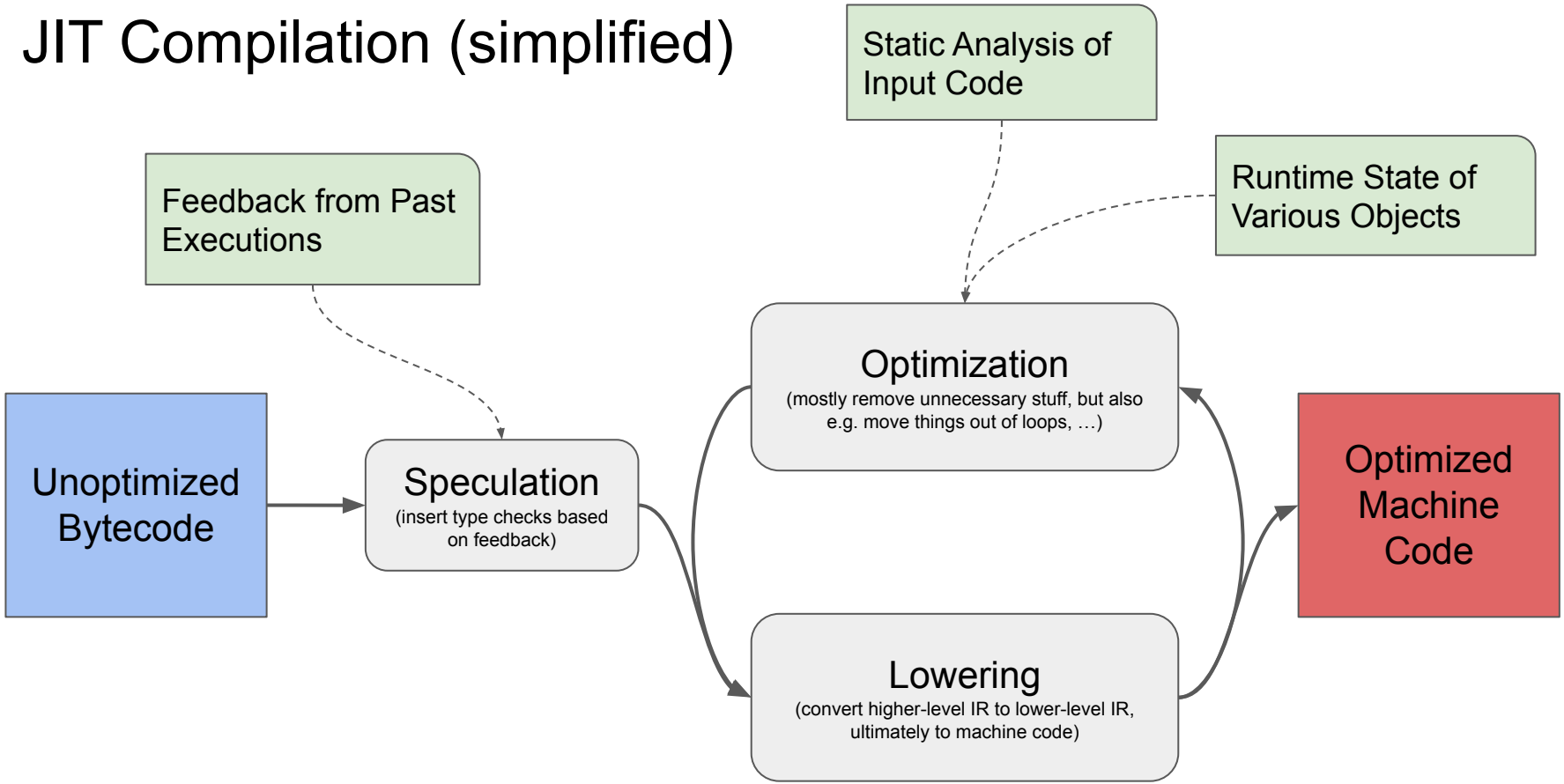
Example: Final JIT IR Code

```
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)



A (Hypothetical) JIT Bug Example

```
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

```
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];
```

```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
CheckBounds a, i
StoreArray a, i, v
```

A (Hypothetical) JIT Bug Example

```
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];
```

```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [0, a.length - 1)
~~CheckBounds a, i~~
StoreArray a, i, v
```

A (Hypothetical) JIT Bug Example

```
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);
```

```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [0, a.length - 1)
~~CheckBounds a, i~~
StoreArray a, i, v
```

Return value

The index of the first element in the array that passes the test. Otherwise, `-1`.

A (Hypothetical) JIT Bug Example

```
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);
```

```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [0, a.length - 1)
CheckBounds a, i
StoreArray a, i, v
```

Return value

The index of the first element in the array that passes the test. Otherwise, `-1`.

A (Hypothetical) JIT Bug Example

```
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
```

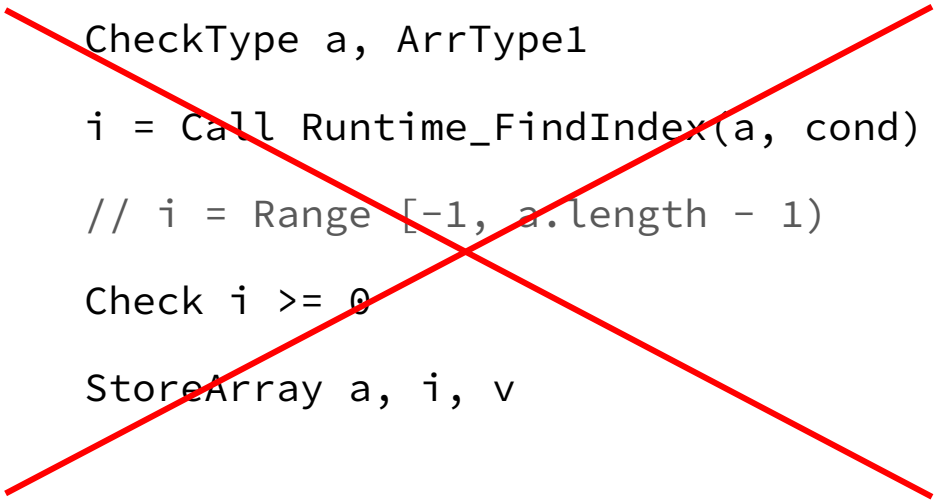
A (Hypothetical) JIT Bug Example

```
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);
```

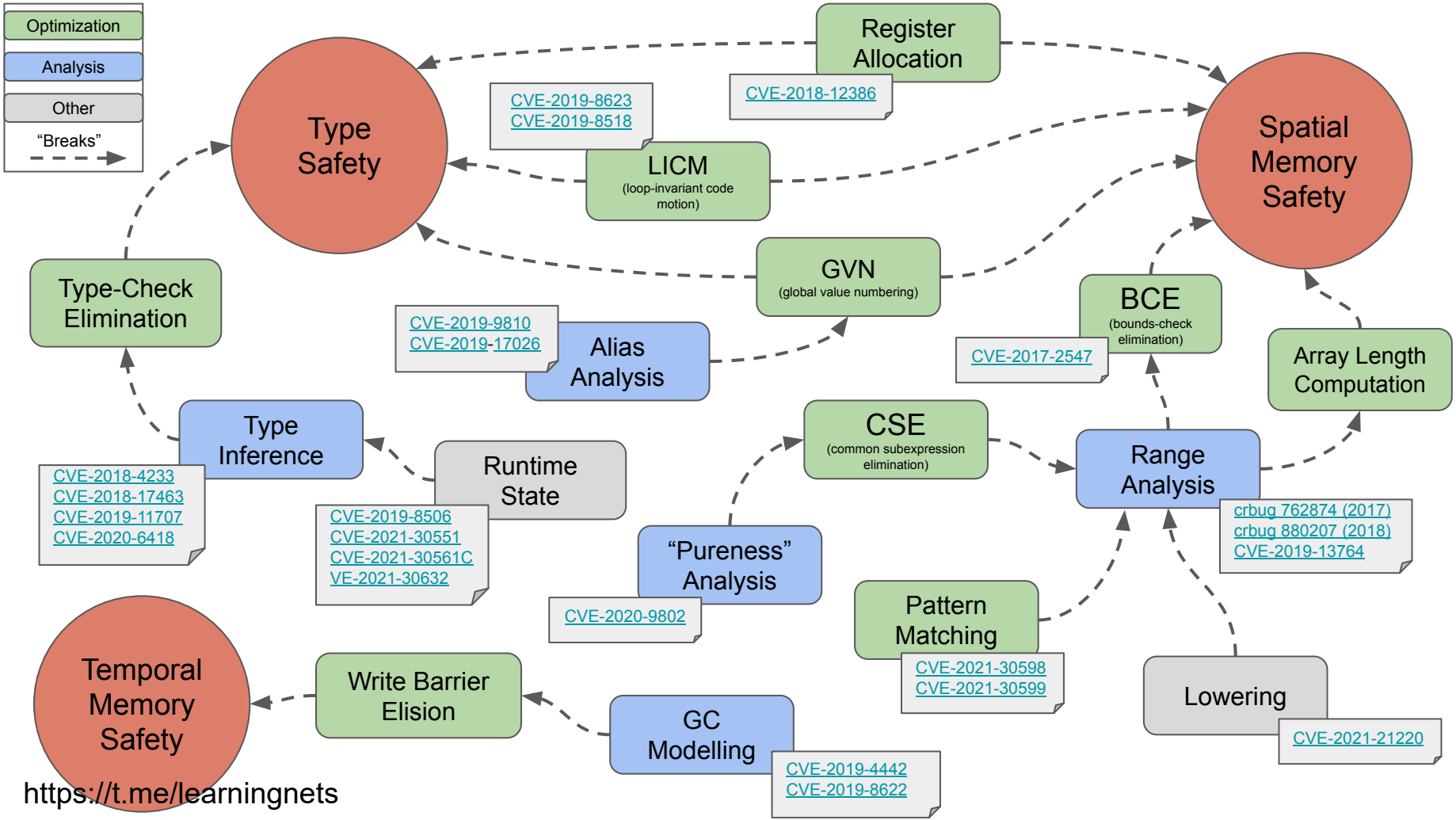
```
CheckType a, ArrType1
i = Call Runtime_FindIndex(a, cond)
// i = Range [-1, a.length - 1)
Check i >= 0
StoreArray a, i, v
```

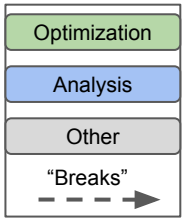
A (Hypothetical) JIT Bug Example

```
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
```

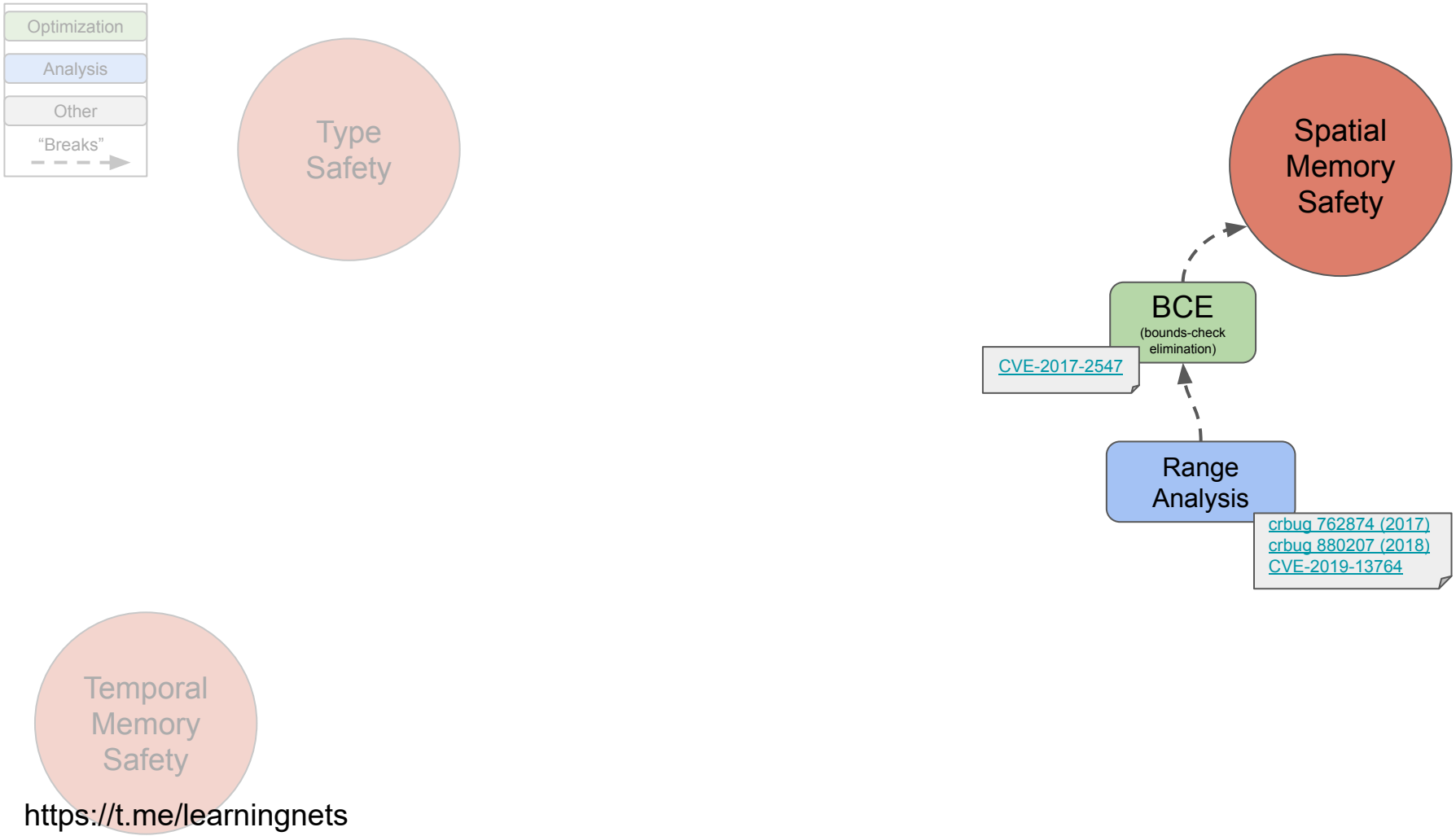


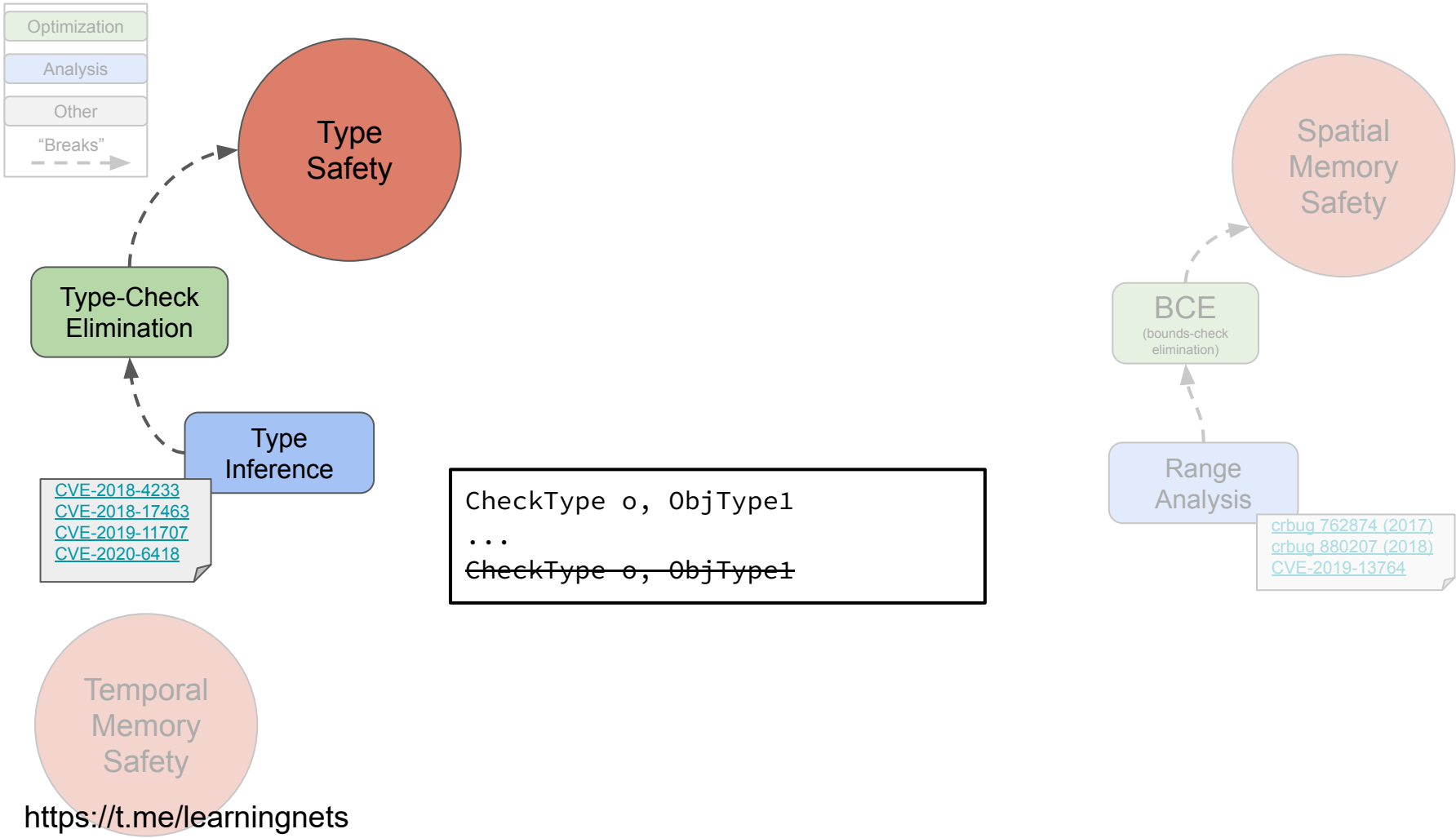


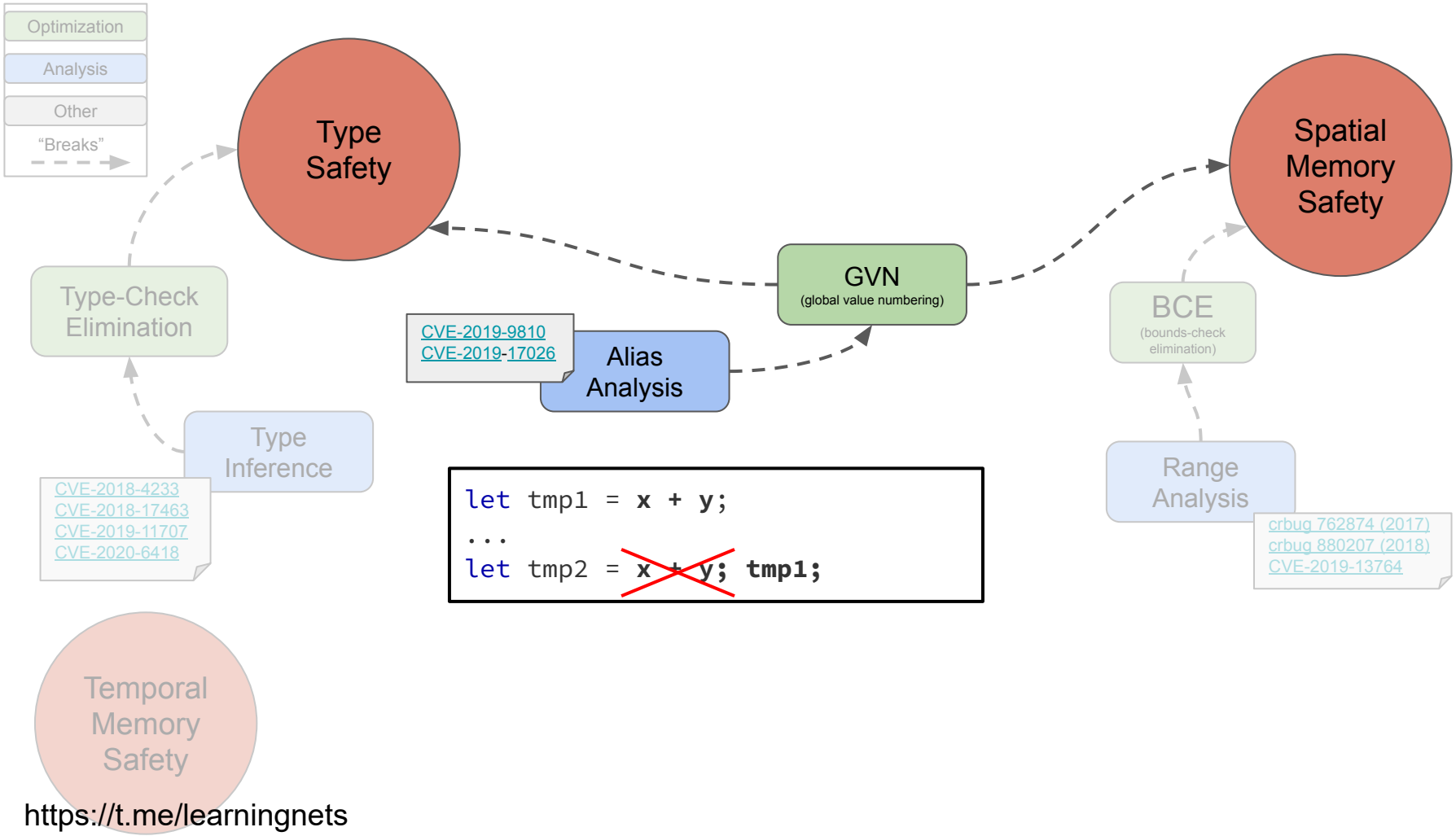
Type Safety

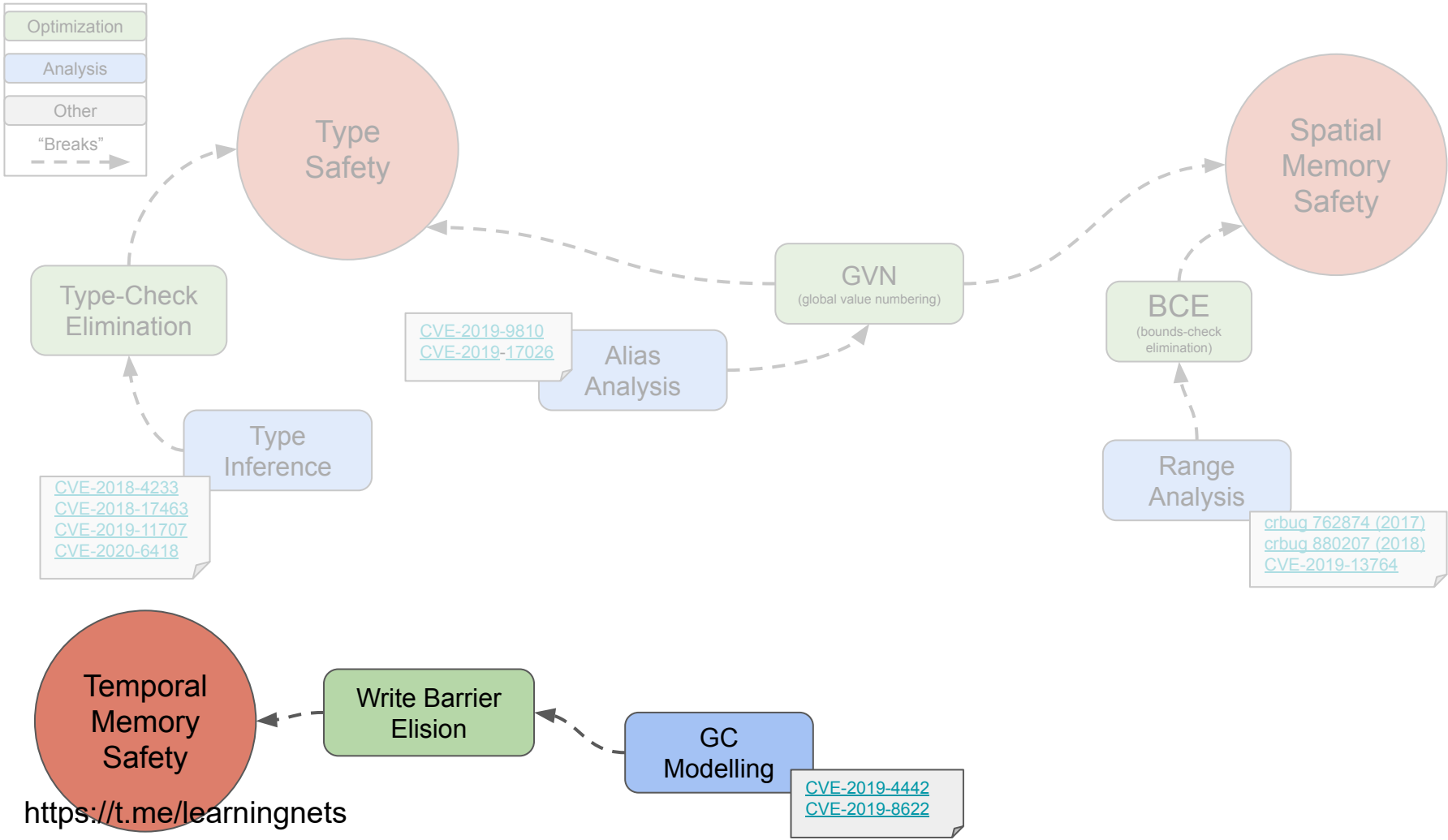
Spatial Memory Safety

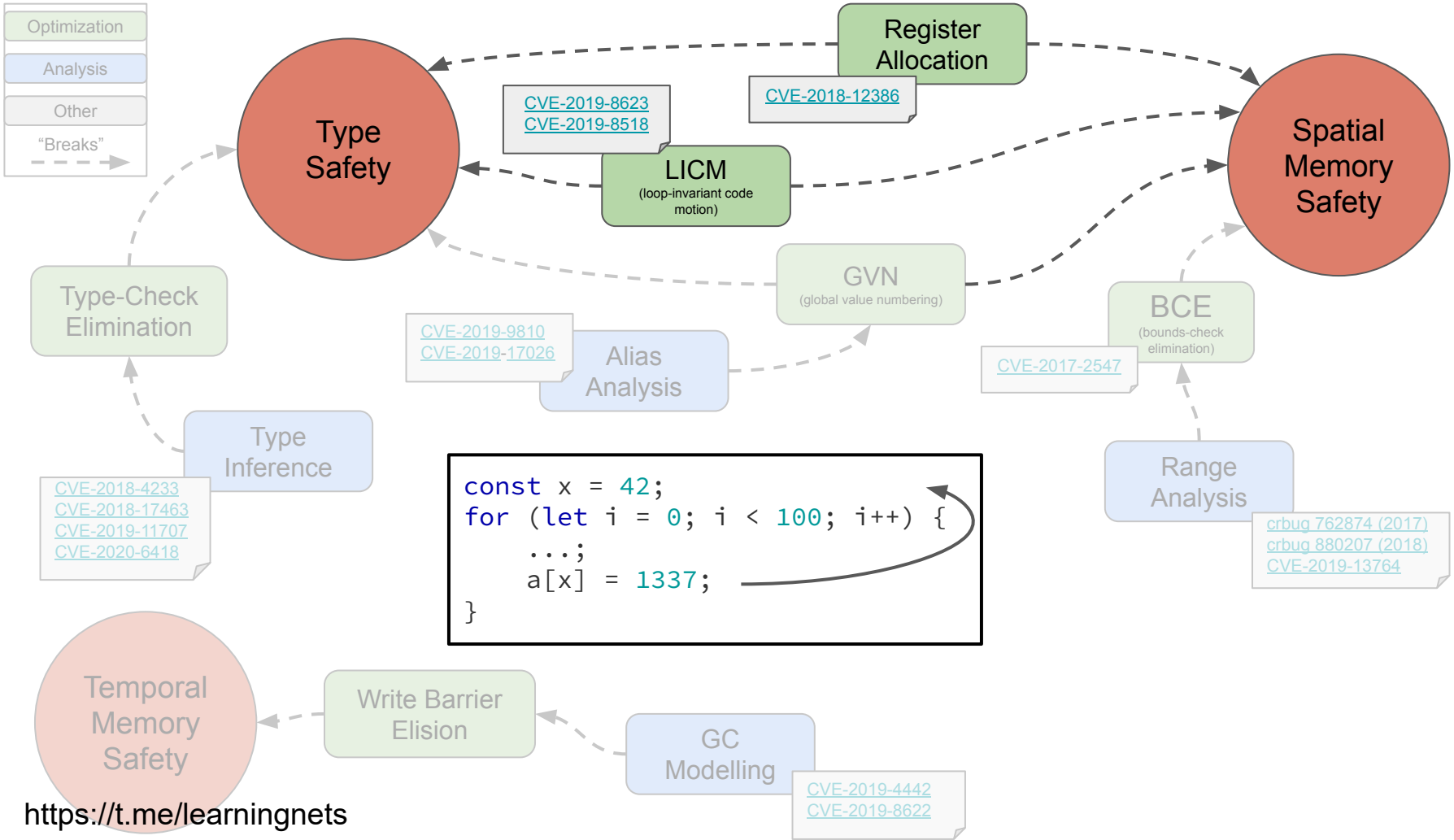
Temporal Memory Safety

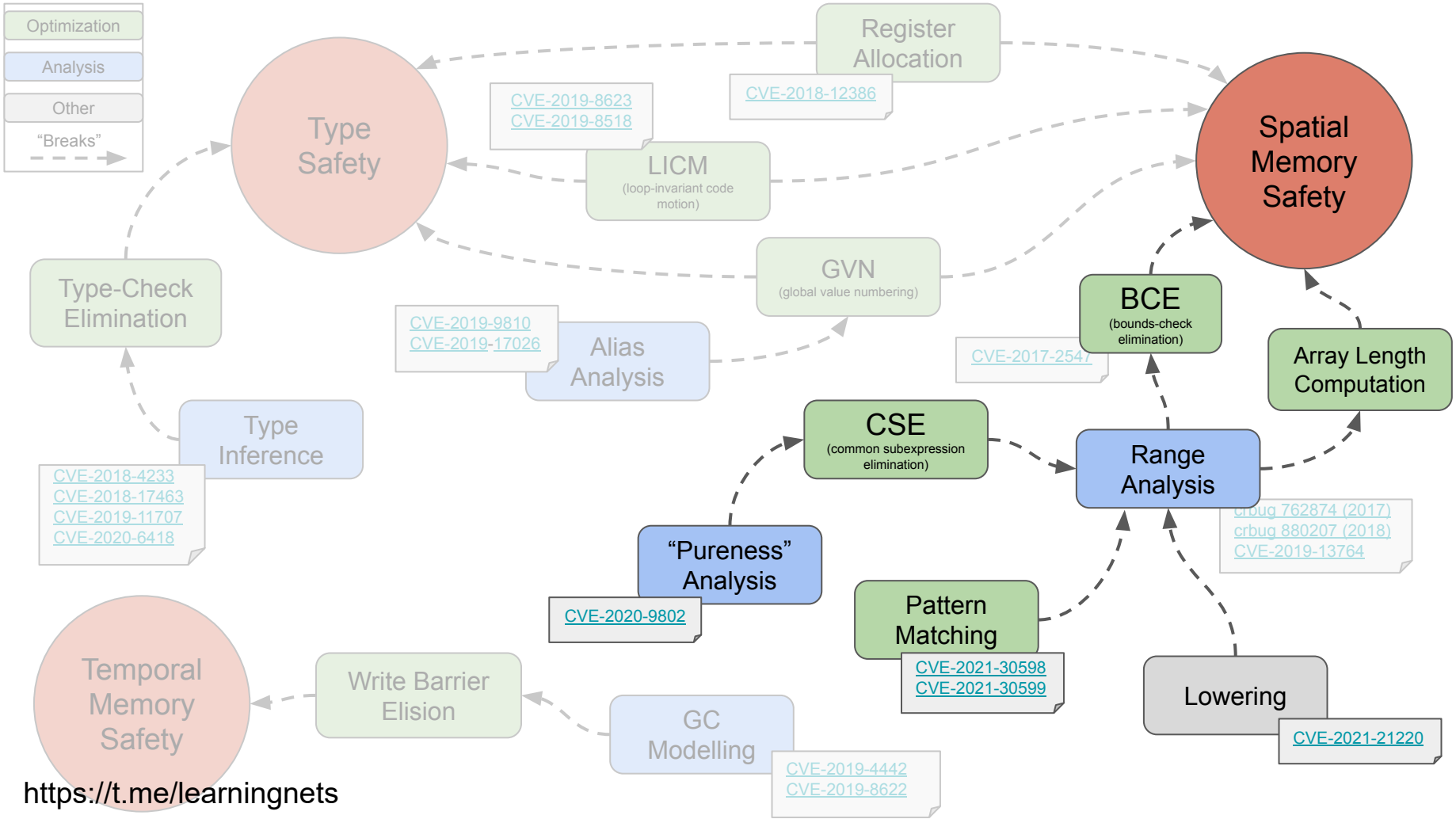


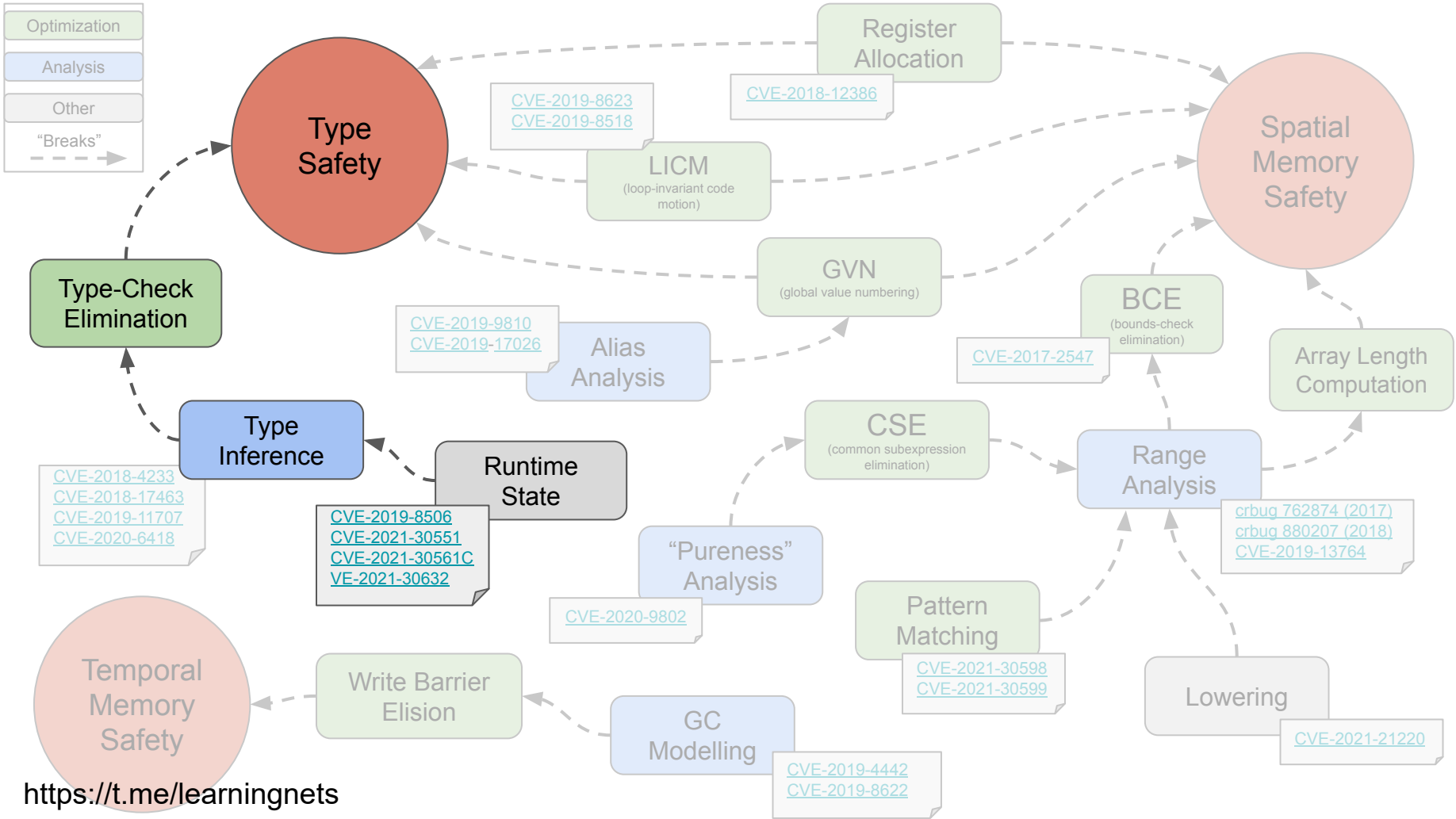


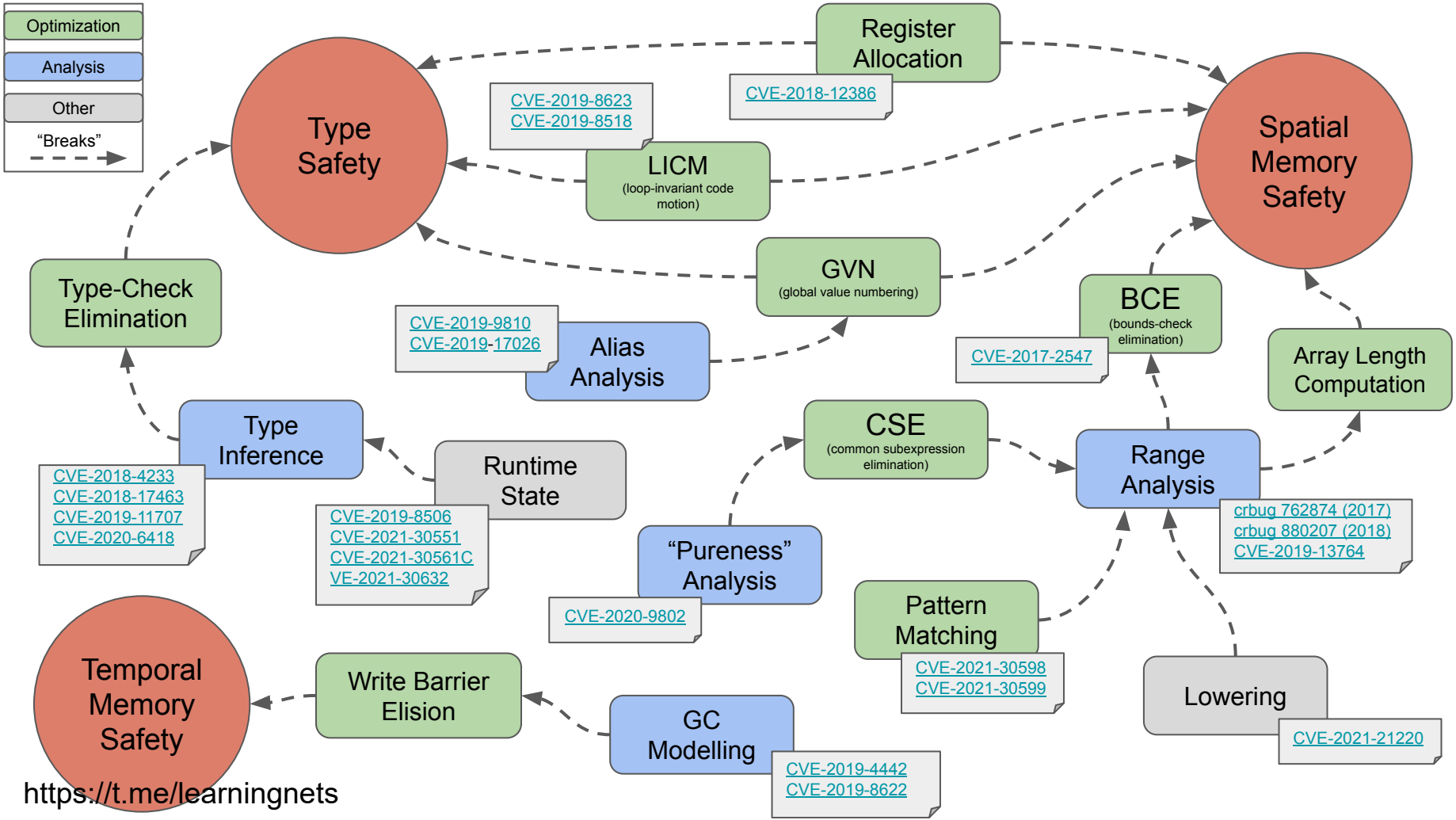


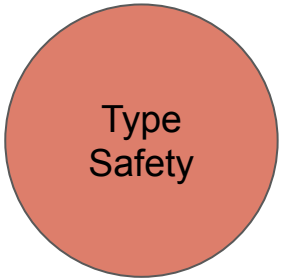






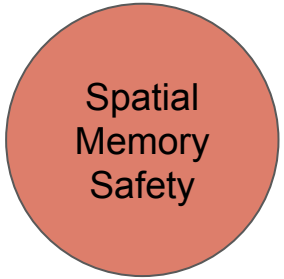






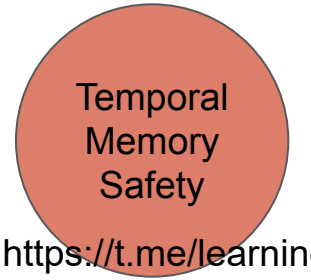
Exploitation

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



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) replacement type
- Trigger bug and GC to cause UaF

JS Outside JIT

Bytecode Compiler

CVE-2022-0102

Interpreter

CVE-2021-30517
CVE-2021-38001

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

CVE-2021-1789
CVE-2021-21225
CVE-2021-38003

- Plenty of complexity elsewhere
- Few bug patterns, many “1-off” bugs

JIT Compiler(s)

See prev. slides :)

Wasm Compiler(s)

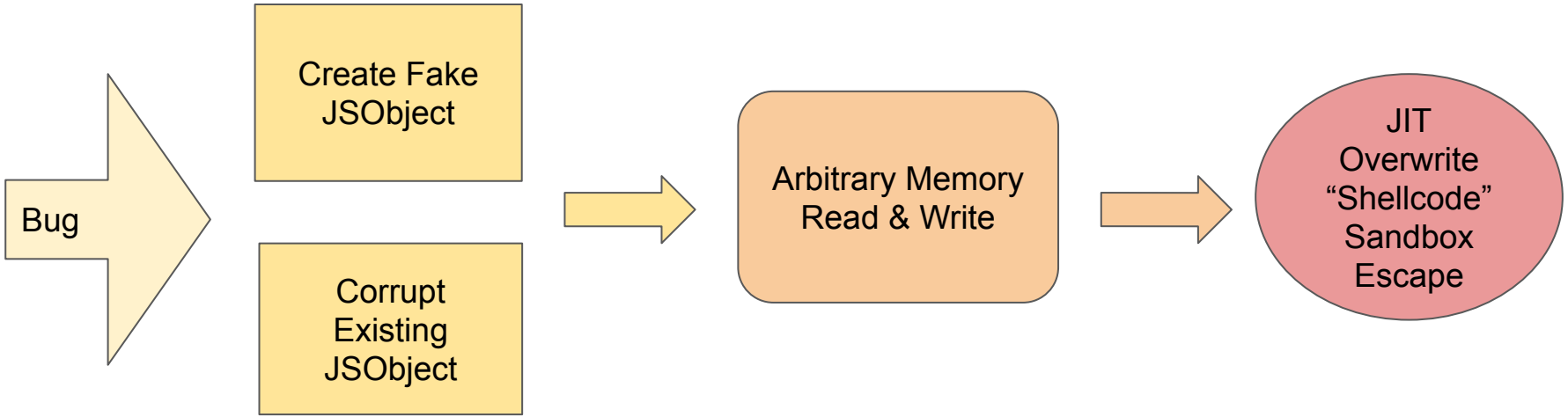
CVE-2021-30734

Garbage Collector (GC)

CVE-2021-37975

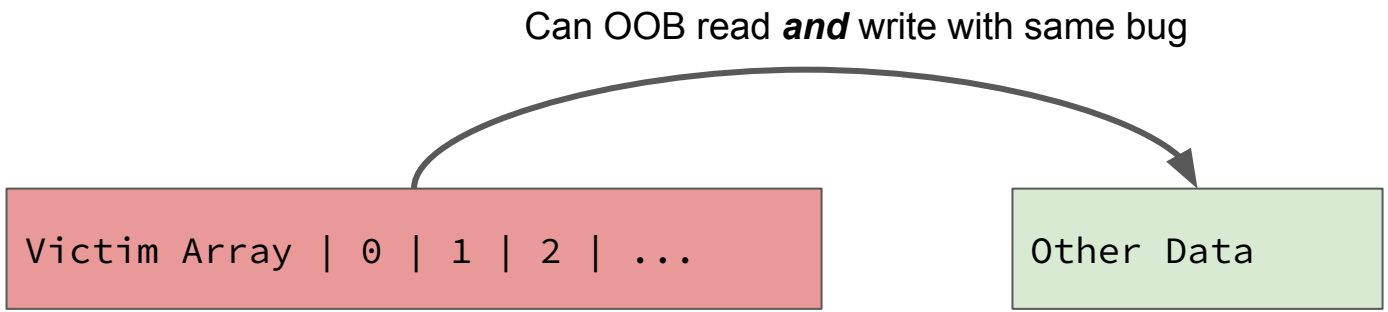
Exploitation & Mitigations

Exploit Flow Circa 2016



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



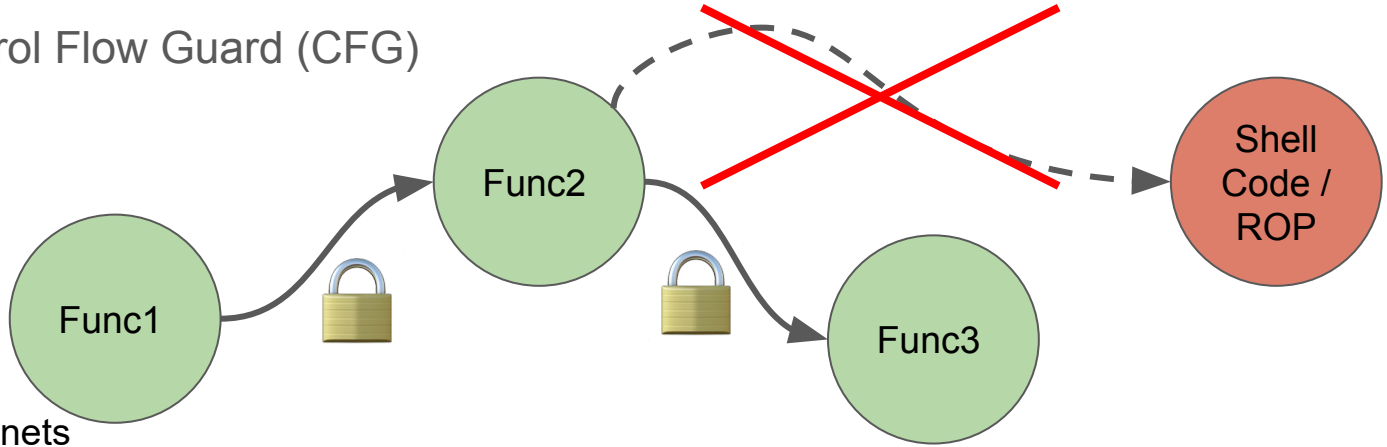
"Modern" Mitigations

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

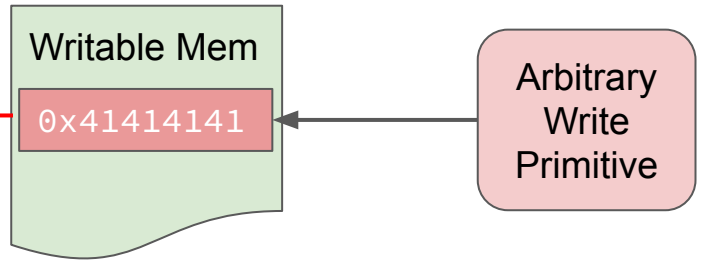
Bypassing Pointer Authentication

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

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



Additionally, V8 currently supports PAC, but not in JITed code [ref]

"Scripted" Code Execution

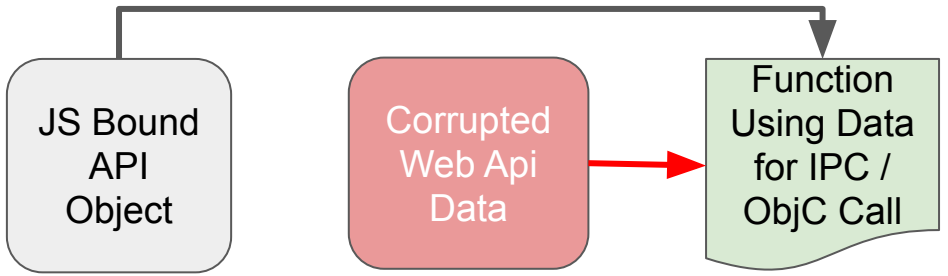
If you can't get arbitrary asm code, you may be able to call existing functionality

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 **integerty** / **confidentiality**

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

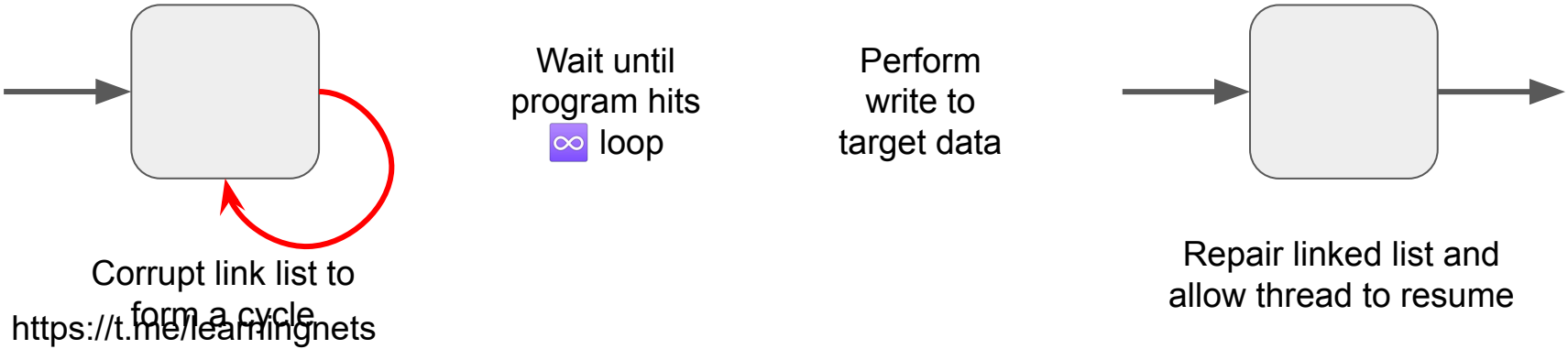
These attacks do not rely on code exec, **only memory read and write**

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…

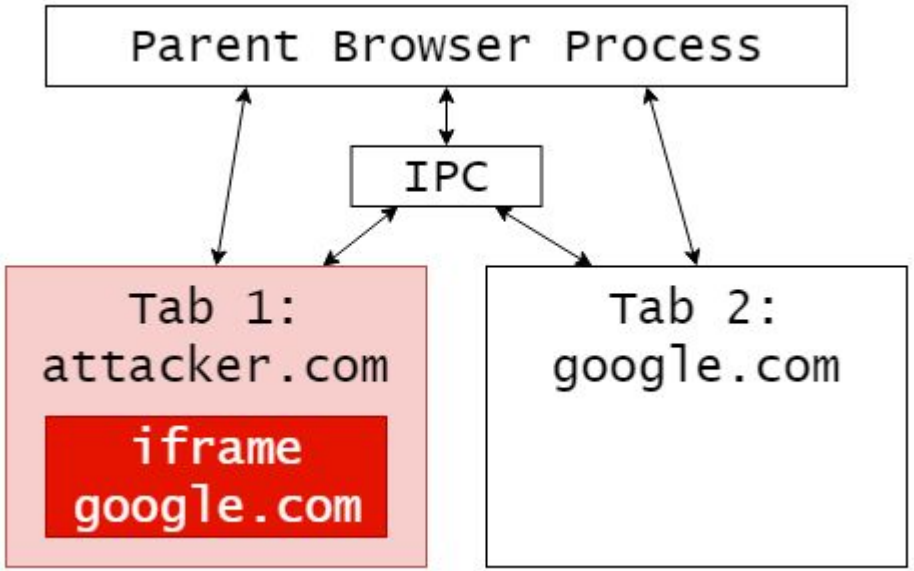
We can abuse linked list structures to stall this race [ref]



Attacking Cross-Origin

We have control of all the data in the compromised process

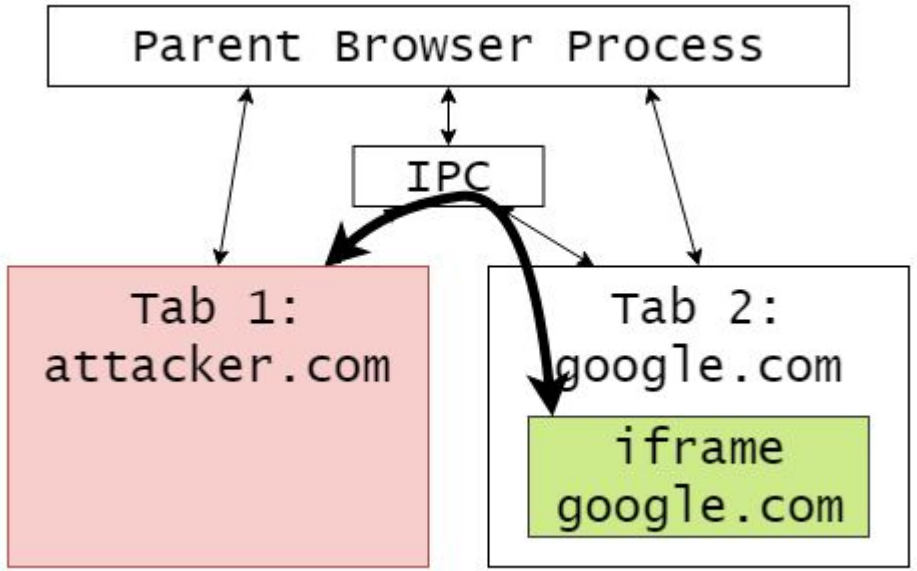
- Force the process to load sensitive data
- Inject JavaScript into other website -> hijack session
- Abuse persistent data features in other websites [ref]



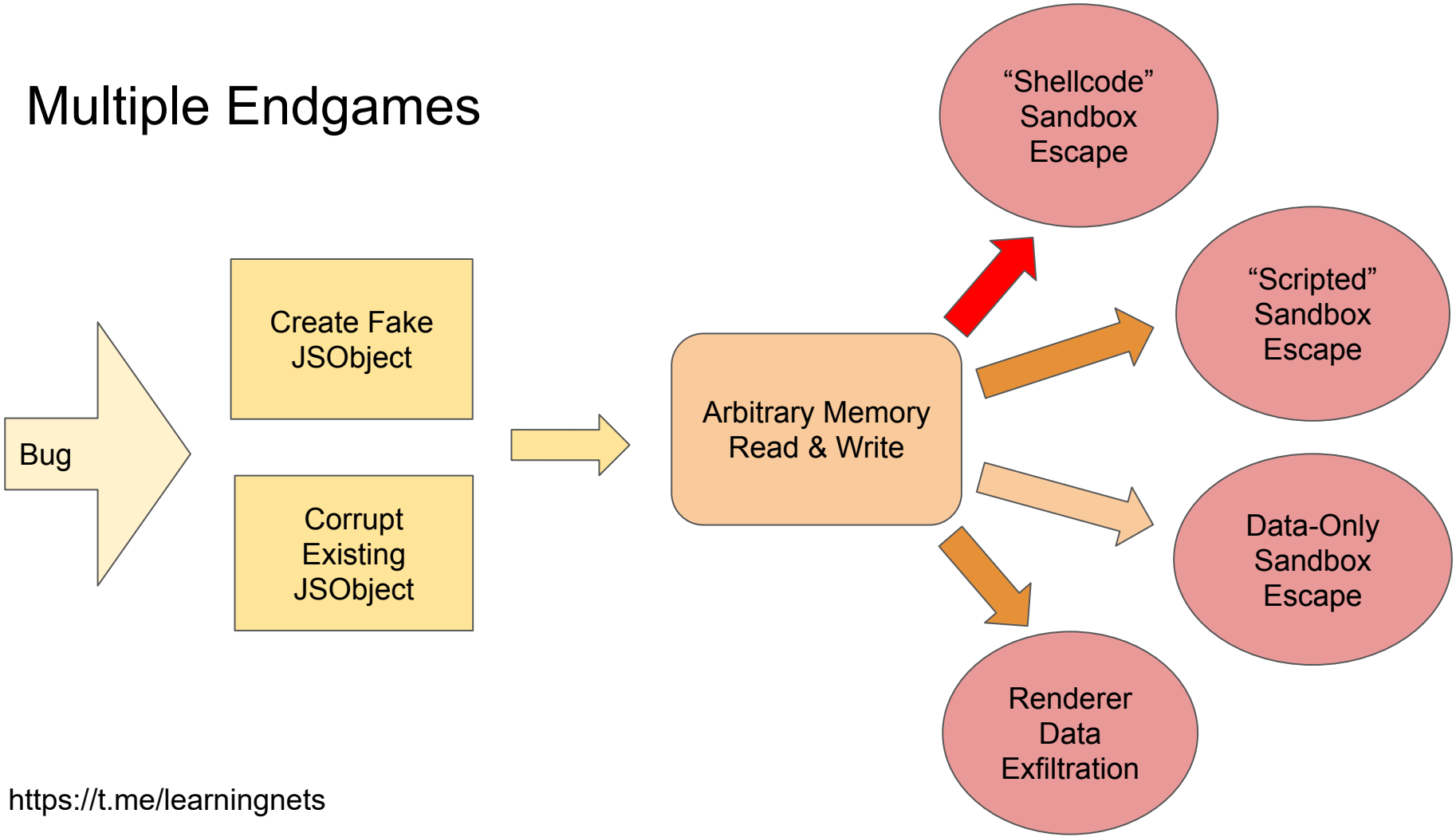
Mitigating Cross Origin Attacks

Chrome and Firefox have enabled "Site Isolation"

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



Multiple Endgames



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

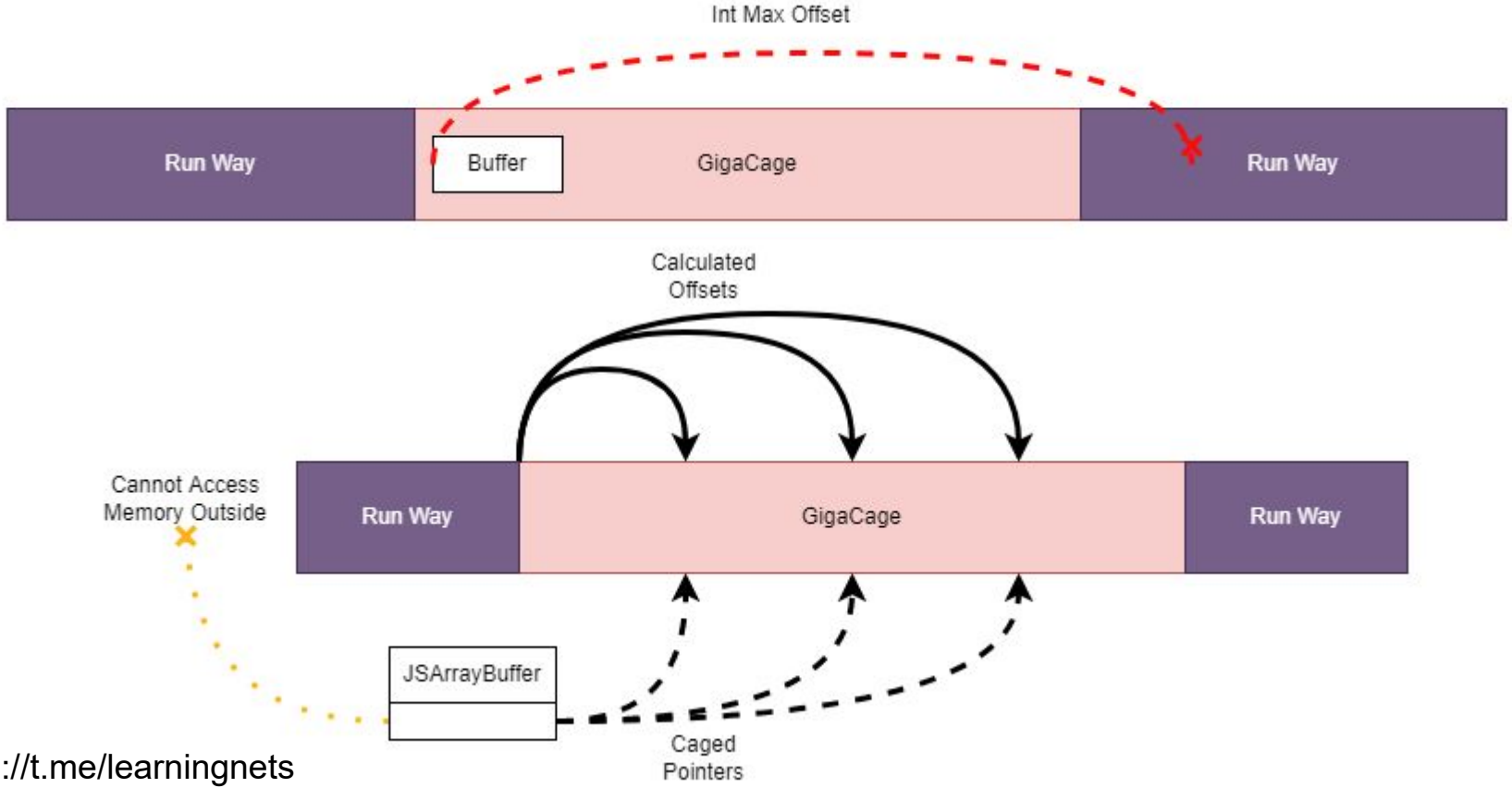
JavaScriptCore's GigaCage

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:

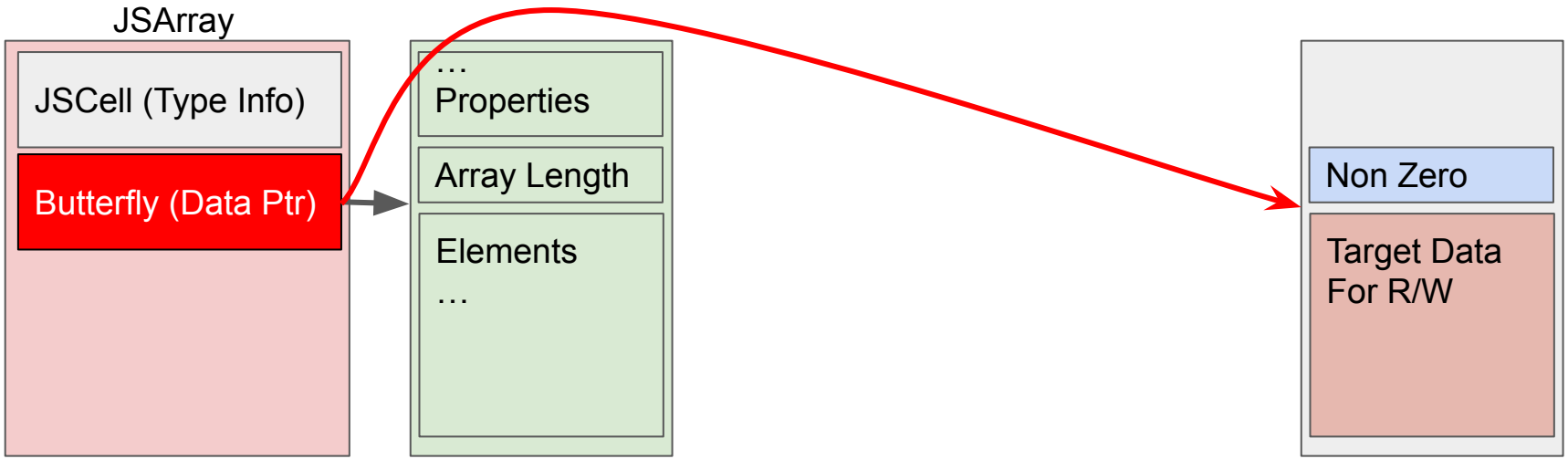
- Explicit caged typing of the pointer
- Correct uncaging implementation when accessing (such as in the JIT)

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…[ref]



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

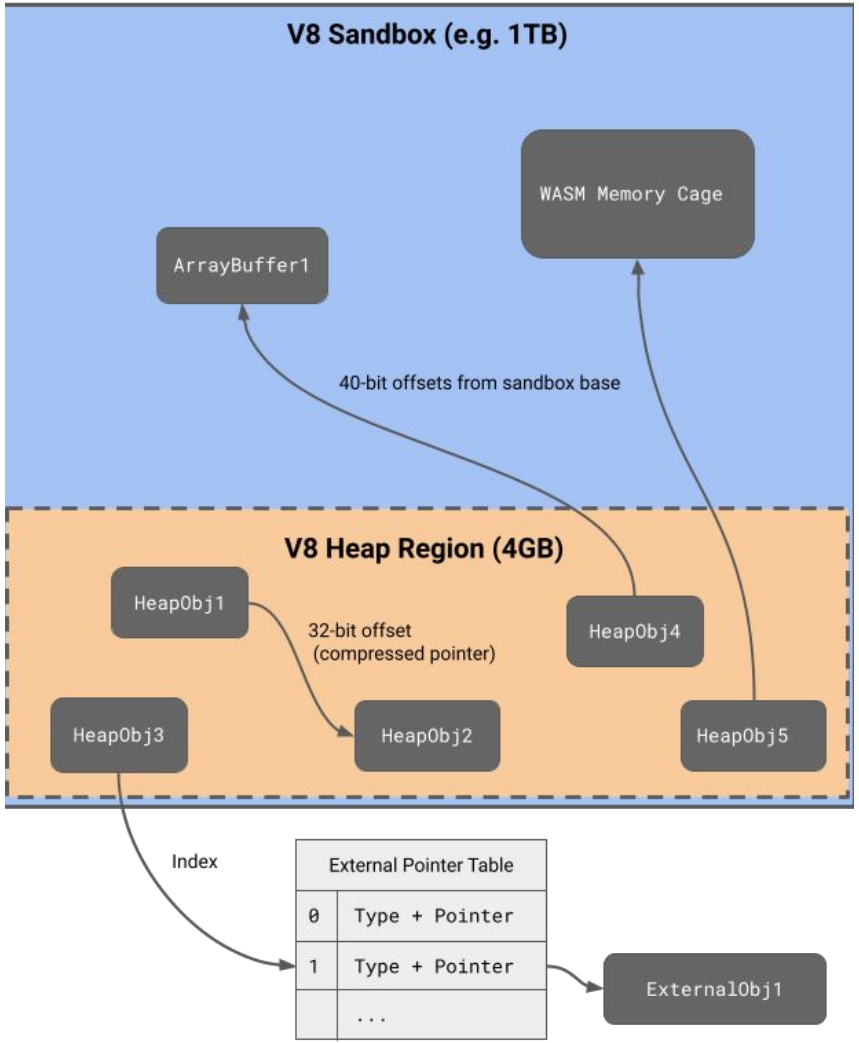
All JS objects confined to sandbox memory

All other sensitive memory is outside:

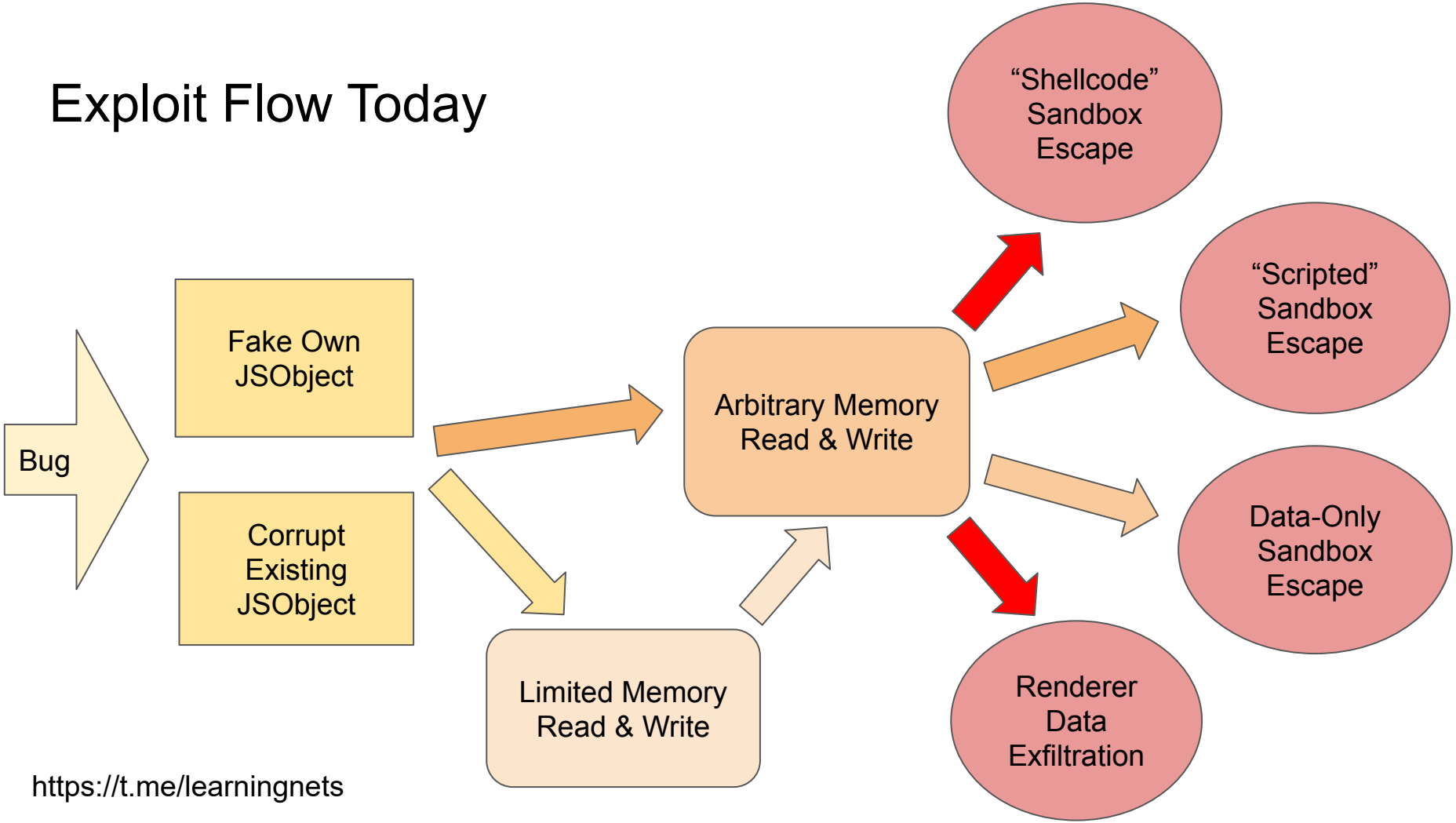
- External pointers (and type) in table
- JIT compiler structures and code
- Any reference to other memory

Exploit now relies on unsound behavior of external objects and code it has handles to

(similar to a sandbox escape…)



Exploit Flow Today



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)