

Engineering Heap Overflows with JavaScript

Jake Honoroff
Mark Daniel
Charlie Miller

Independent Security Evaluators

Overview

- Introduction
- Heap Control
- WebKit JavaScript Details and Exploit Description

Heap Overflows

- Can allow for arbitrary bytes written to specific relative offset from pointer on heap
- What if nothing interesting is after buffer?
 - Unmapped memory
 - Unused memory
 - Unpredictable memory

Heap Overflows

- Safe Unlinking means metadata not viable target
- Often, need function pointer(s) as target
 - malloc'd before overflow
 - Used after overflow
 - Reliably positioned
 - No critical data smashed

Motivation

- 2008 CanSecWest Pwn2Own contest: exploit fully patched, default installation of Mac Leopard, Windows Vista, or Ubuntu laptop
- Found buffer overflow in WebKit's JavaScript PCRE (Perl-Compatible Regular Expression) compilation, accessible through Safari on Leopard

Motivation

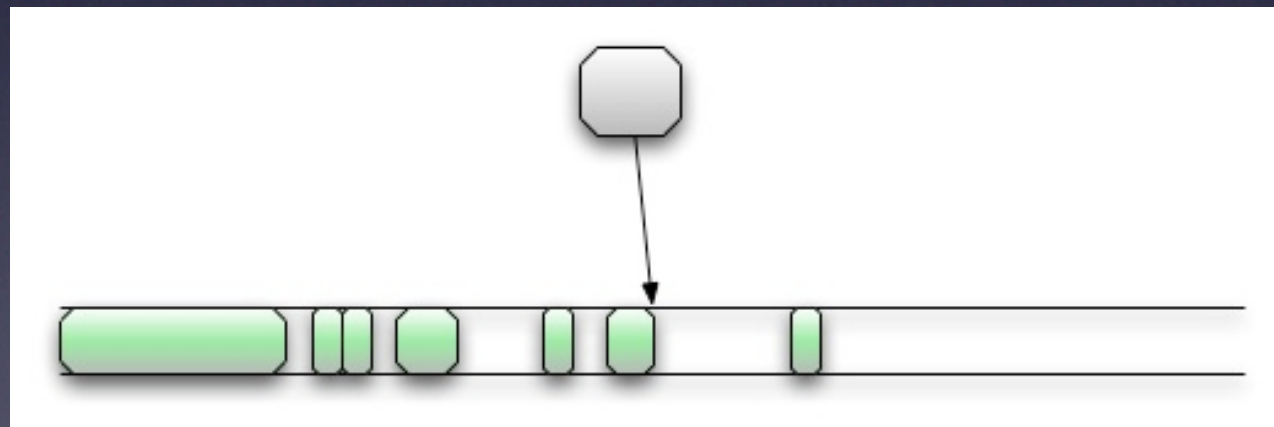
- Overflow itself extremely controllable
 - Size of malloc up to 65535 bytes
 - Exact amount of overflow controllable
 - Bytes in buffer from compiled regular expression: lots of control
- But nothing useful and predictable after buffer...

Heap Control

- Inspired by A. Sotirov's *Heap Feng Shui*
- Assuming non-randomized heap, can control memory after the overflow by:
 - Defragmenting heap
 - Creating holes in heap
 - Landing vulnerable buffer in hole

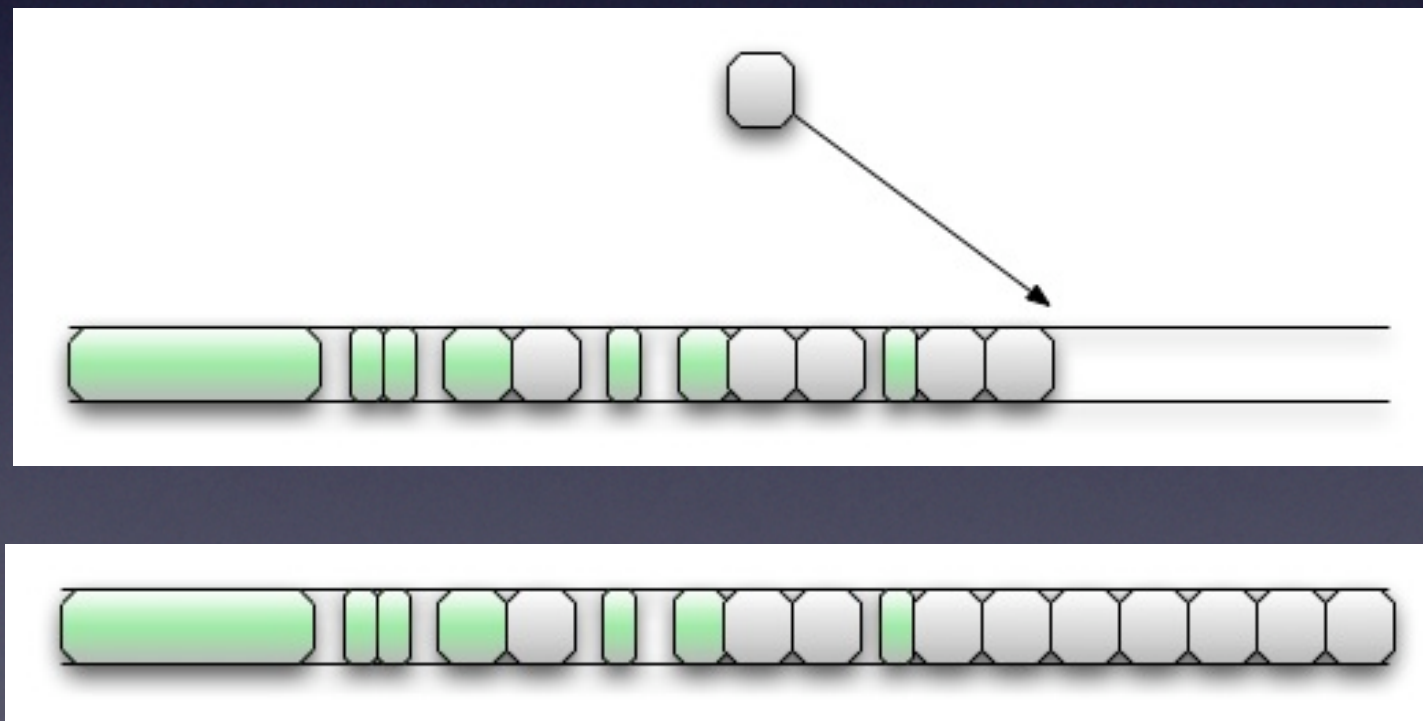
Fragmented Heap

- Non-randomized heaps in unpredictable state with holes from freed buffers. Future allocations land in unpredictable location



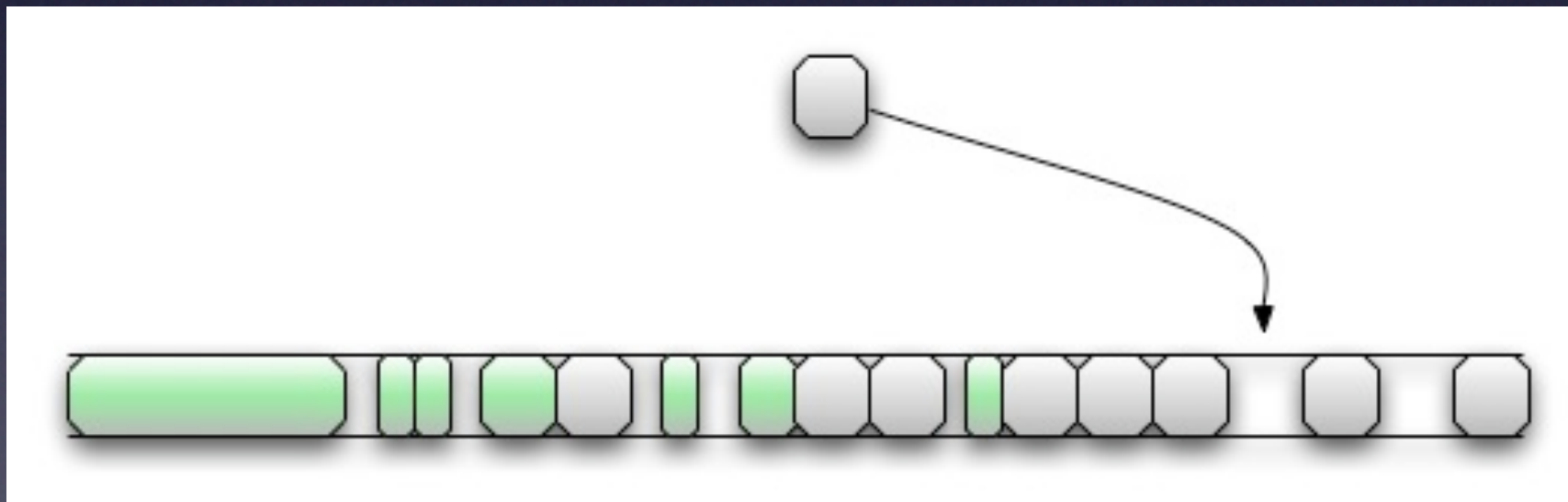
Defragmenting heap

- Can fill these holes with allocations of many same-sized buffers, eventually leading to a long sequence of same-sized buffers



Making Holes

- Free every other buffer, such that subsequent allocations fall in hole with controllable memory after



Defragmenting Heap

- Easy in Javascript:

```
var bigdummy = new Array(1000);  
for(i=0; i<1000; i++){  
    bigdummy[i] = new Array(size);  
}
```

- Each call to `new Array(size)` results in allocation of $4 * size + 8$ bytes on heap from `ArrayStorage` object

Making Holes

- First, delete the JavaScript objects:

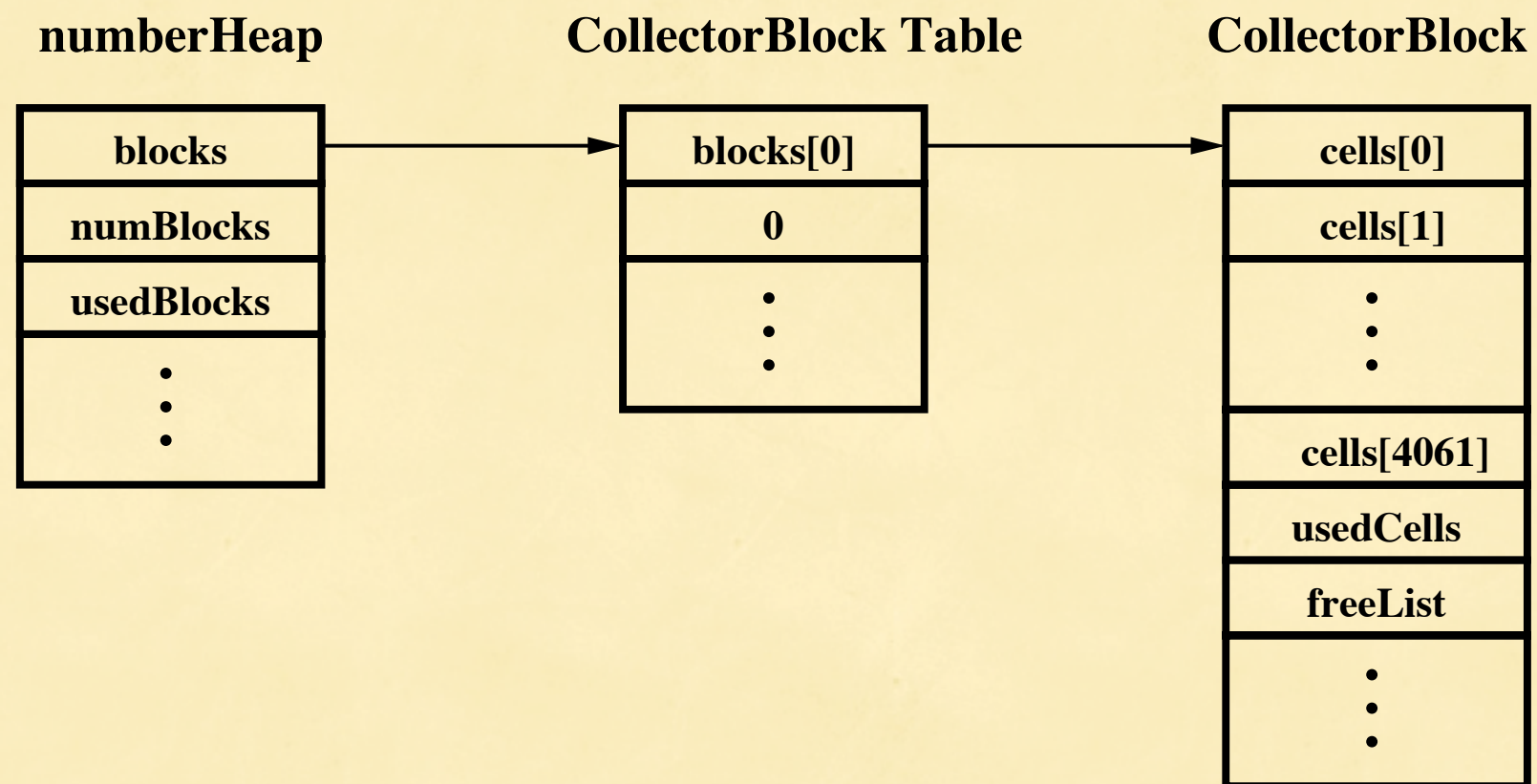
```
for(i=900; i<1000; i+=2){  
    delete(bigdummy[i]);  
}
```

- But associated storage not freed until Garbage Collection
- In IE, can call `CollectGarbage()`
- In WebKit, need to trigger GC

WebKit GC Overview

- Maintains two structures, `primaryHeap` and `numberHeap`
- Arrays of `CollectorBlock` objects
- Used to store array of `Cells`
- `Cells` contain JavaScript objects

numberHeap Diagram



WebKit GC Overview

- `primaryHeap` used to store objects deriving from `JLObject`, grows with significant browsing
- `numberHeap` used to store `NumberImp` objects, short-lived number objects corresponding to arithmetic computations

Triggering GC

- Expiration of GC timer
- Allocation request when `CollectorBlocks` are full
- Allocation of object(s) with sufficiently large associated storage

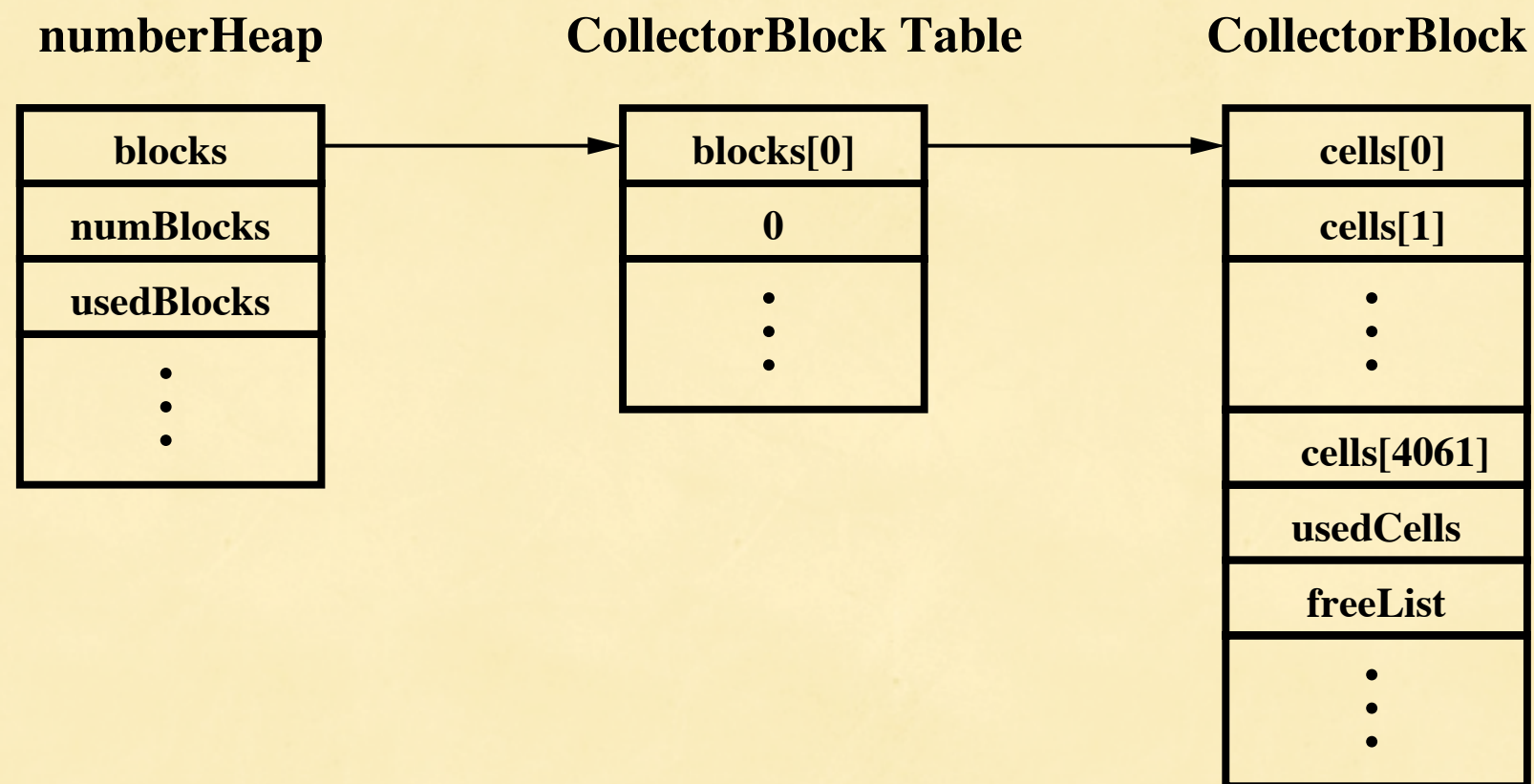
Triggering GC

- ~~Expiration of GC timer~~
- Allocation request when `CollectorBlocks` are full
- Allocation of object(s) with sufficiently large associated storage

Triggering GC

- ~~Expiration of GC timer~~
- Allocation request when `CollectorBlocks` are full
- ~~Allocation of object(s) with sufficiently large associated storage~~

numberHeap Diagram



Triggering GC

- Since numberHeap reliably has only one allocated CollectorBlock, filling its 4062 Cells should trigger GC
- Use manipulation of doubles in JavaScript:

```
for(i=0; i<4100; i++){  
    a = .5;  
}
```

Prepare Blocks

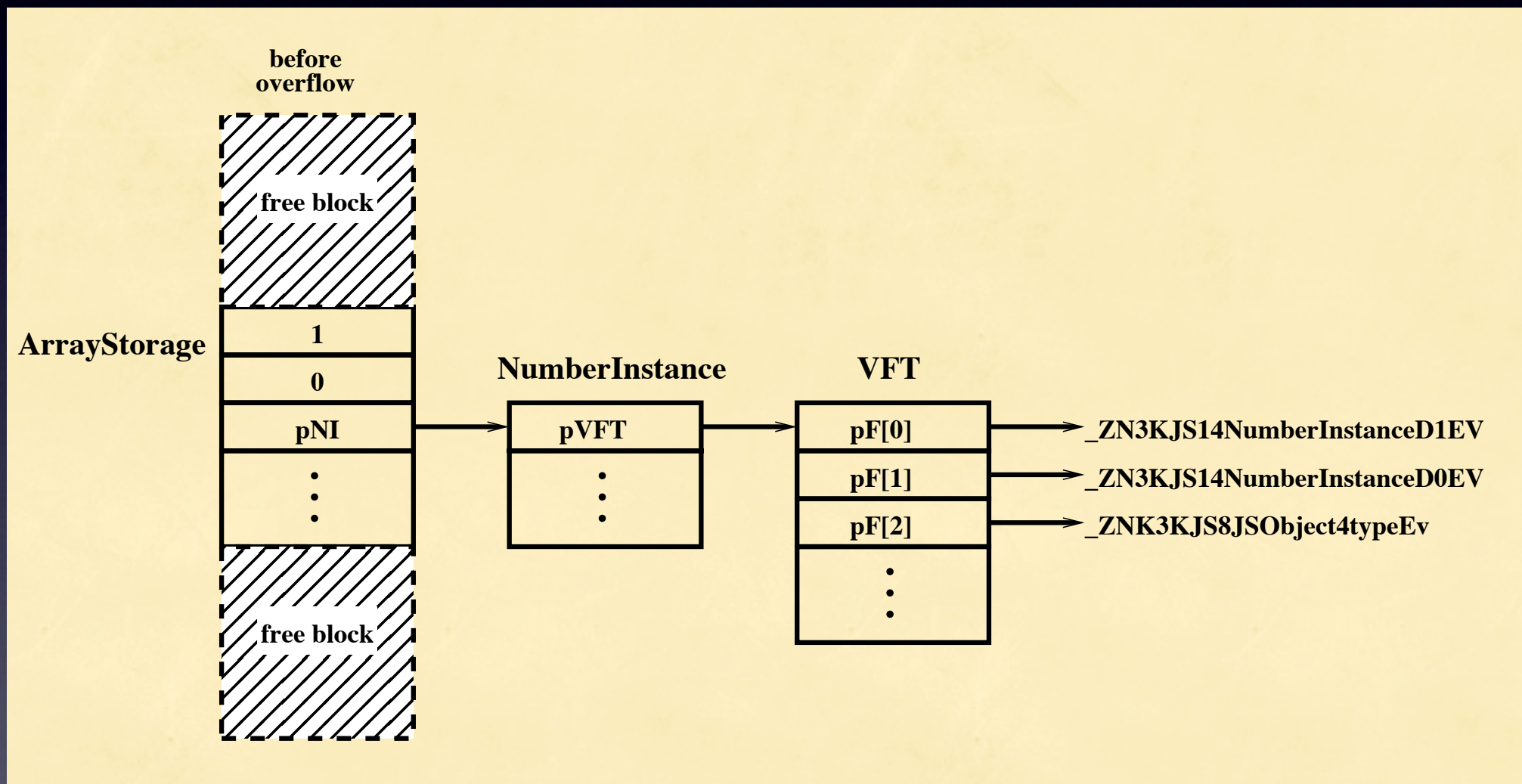
- Recall, the holes were made by freeing alternating `ArrayStorage` objects:

```
for(i=900; i<1000; i+=2){  
    delete(bigdummy[i]);  
}
```

- Using below code we put a new `NumberInstance` object in the surrounding `ArrayStorage` objects

```
for(i=901; i<1000; i+=2){  
    bigdummy[i][0] = new Number(i);  
}
```

ArrayStorage Diagram Before Overflow



Trigger Overflow

- Finally, we are ready for the overflow.
- The buffer will land in a hole
- Beginning of overflow preserves the 8-byte `ArrayStorage` metadata
- At third dword we place pointer to self-referential NOP sled followed by shellcode

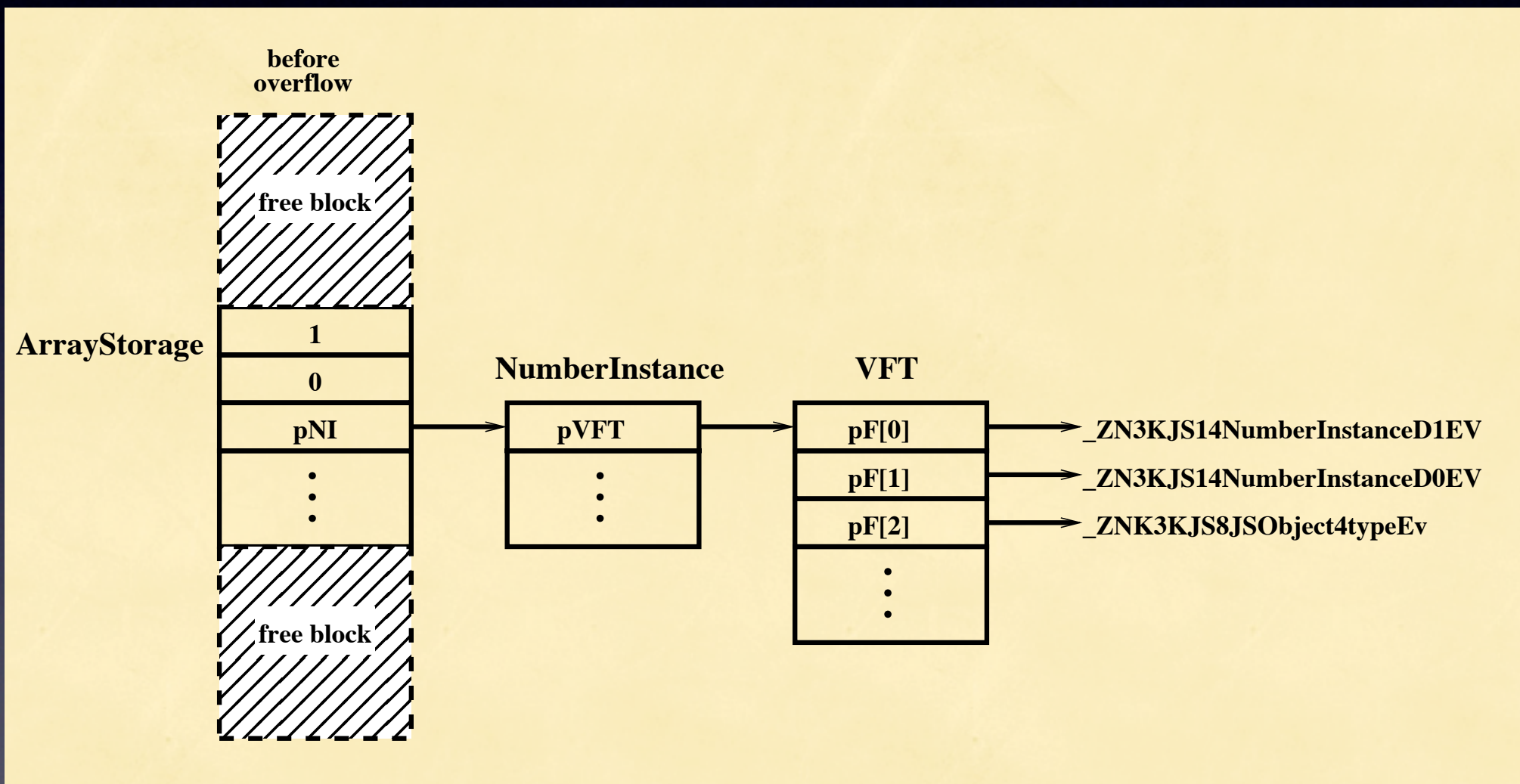
Gaining Control

- Just need to get function pointer called:

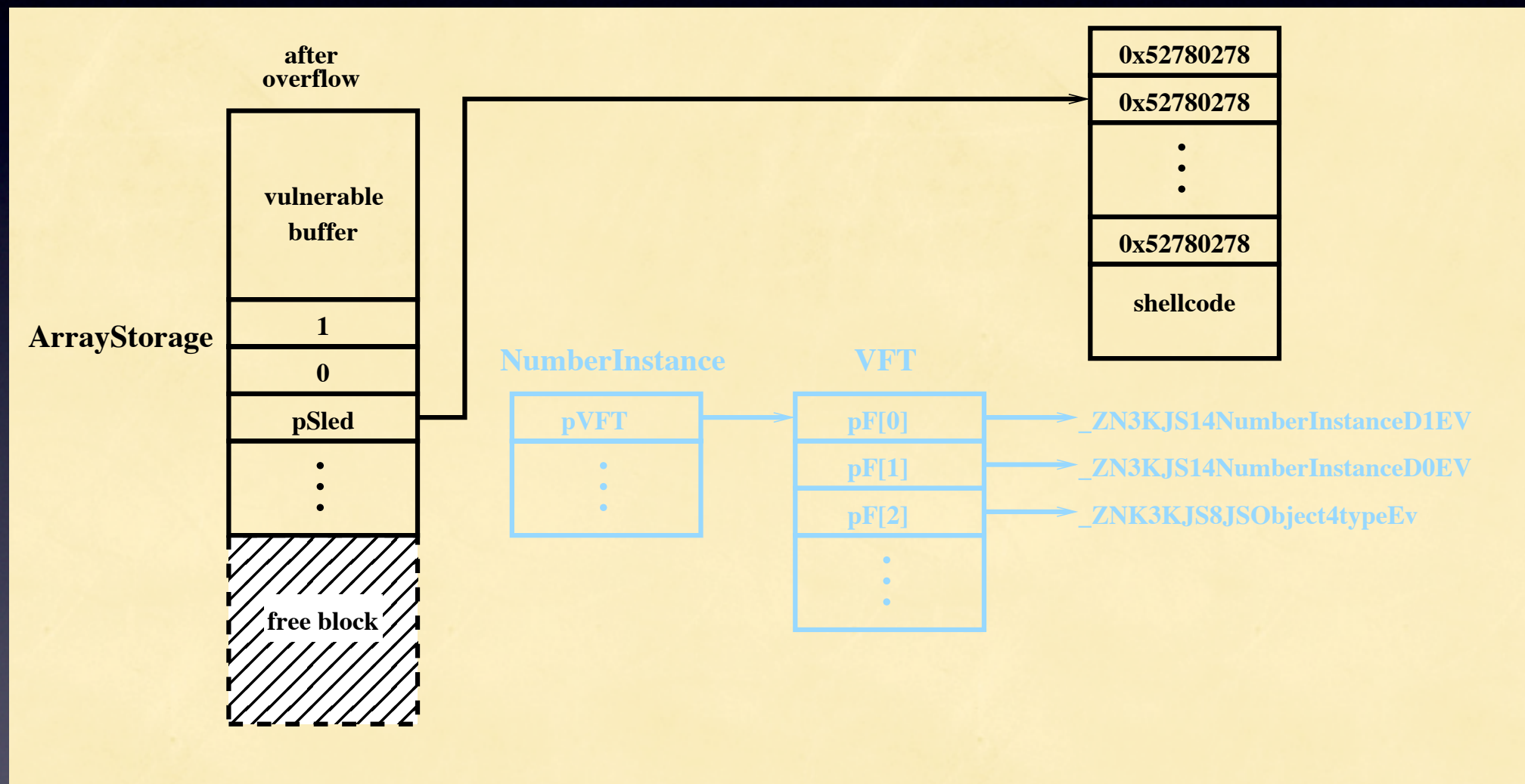
```
for(i=901; i<1000; i+=2){  
    document.write(bigdummy[i][0] + "<br />");  
}
```

- Conversion to string results in VFT call

ArrayStorage Diagram Before Overflow



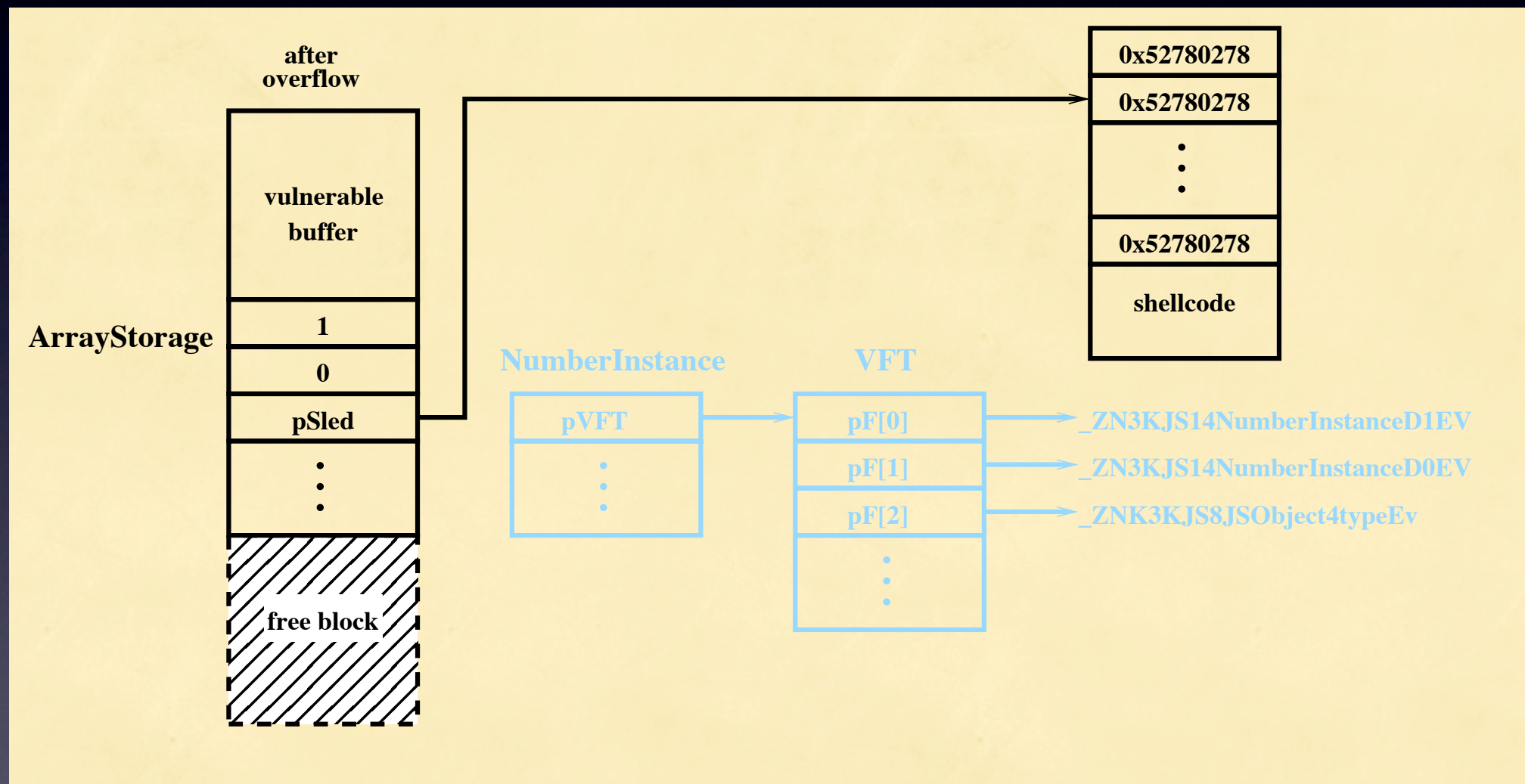
After Overflow



Self-Referential NOP sled

- Using heap spray, we have sled mapped at address 0x52.....
- Use spray of 0x52780278
 - 78 02: js +0x2
 - 78 52: js +0x52
- If 4-byte alignment not necessary, can use 0x...02eb

After Overflow



Conclusions

- Given non-randomized allocator, JavaScript can be used to reliably exploit heap overflows via:
 - Heap defragmentation
 - Creation of holes in heap
 - Positioning of function pointers
 - Getting overwritten VFT called

Questions?