Make JIT-Spray Great Again

Robert Gawlik and Thorsten Holz

Ruhr-Universität Bochum

W00T '18

August 13/14 2018 – Baltimore
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

- Performance gain compared to interpreted execution
Just-In-Time Compilation (JIT)

- Several compilers and optimization layers

**JS:**
- Baseline, IonMonkey
- ChakraCore (2 Tier JIT)
- TurboFan

**Java:**
- HotSpot JIT

**Java:**
- Baseline, DFG, FTL

**PHP:**
- HHVM JIT

**ActionScript:**
- NanoJIT

**Linux Kernel:**
- eBPF

**.Net:**
- RyuJIT
Just-In-Time Compilation (JIT)

- Several compilers and optimization layers
  - Baseline, Baseline

JS:
  - TurboFan
  - ChakraCore (2 Tier JIT)

ActionScript:
  - NanoJIT

Linux Kernel:
  - eBPF

.Net:
  - RyuJIT

More than 13 compilation engines. What could possibly go wrong?
JIT-Spray (x86)
JIT-Spray

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

\[
c = 0xa8909090 \\
c += 0xa8909090
\]

x86 Disassembly @ offset 1

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>90</td>
<td>05</td>
<td>00</td>
<td>a8</td>
<td>05</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>05</td>
<td>08</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Emitted JIT Code

\[
00: b8909090a8 \text{ mov eax, 0xa8909090} \\
05: 05909090a8 \text{ add eax, 0xa8909090}
\]
JIT-Spray

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

2. Force allocations to predictable address regions

```cpp
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){
    JIT()
  }

  0x20202020: 90
  0x20202021: 90
  0x20202022: 90
  0x20202023: 90
  0x20202024: a805
  0x20202025: 90
  0x20202026: 90
  0x20202027: 90

  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)

• Long XOR sequence gets compiled to XOR instructions

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

```assembly
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
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:
  - 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)
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)

<table>
<thead>
<tr>
<th></th>
<th>V8</th>
<th>IE9</th>
<th>Jaeger Monkey</th>
<th>Trace Monkey</th>
<th>LLVM</th>
<th>JVM</th>
<th>Flash / Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Page Permissions</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Guard Pages</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JIT Page Randomization</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Constant Folding</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Constant Blinding</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Allocation Restrictions</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Random NOP Insertion</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Random Code Base Offset</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- In depth analysis of LLVM and Firefox JIT engines
- JIT-Spray techniques (i.e. with floating point values)
- JIT gadget techniques
- Comparison of JIT hardening measurements
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
Exploit Your Java Native Vulnerabilities on Win7/JRE7 in One Minute (Yuki Chen, 2013)

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

```plaintext
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
```
ASM.JS JIT-Spray on OdinMonkey (x86)
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

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

Simple ASM.JS module

```javascript
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

```javascript
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

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

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

Simple ASM.JS module

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

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

```javascript
Successfully compiled asm.js code (total compilation time 3ms)
```
ASM.JS

Inject Code to Predictable Addresses

• Request ASM.JS module several times

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

Inject Code to Predictable Addresses

```
"use asm"

function asm_is_function(){
    var val = 0x1c2c3c4;
    return val | 0;
}
```

Value appears in machine code

```
01000023 b3c4c3c2c1  mov    eax,0C1C2C3C4h
01000028 6690    xchg    ax,ax
0100002a 83c404  add     esp,4
0100002d c3  ret
```
ASM.JS

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
...```
ASM.JS

Inject Code to Predictable Addresses

- Unblinded constant 0xc1c2c3c4 appears many times

- many module requests yield many copies
ASM.JS

Inject Code to Predictable Addresses

- Unblinded constant `0xc1c2c3c4` appears many times

- many module requests yield many copies
- aligned to predictable addresses
ASM.JS

Inject Code to Predictable Addresses

• Unblinded constant `0xc1c2c3c4` appears many times

- many module requests yield many copies
- aligned to predictable addresses

→ ASM.JS JIT-Spray unlocked
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 Payloads

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
05: 90  nop
06: 90  nop
07: 90  nop
08: 90  nop
```
ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

• Setting array elements

```javascript
'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

```javascript
'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
```
ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- Setting array elements

```javascript
stdlib.Uint32Array
function() {
  var arr = new Uint32Array(10);
  arr[0] = 0x00ce9090;
  arr[1] = 0x00ce9090;
  arr[2] = 0x00ce9090;
  arr[3] = 0x00ce9090;
  arr[4] = 0x00ce9090;
  arr[5] = 0x00ce9090;
  arr[6] = 0x00ce9090;
  arr[7] = 0x00ce9090;
  arr[8] = 0x00ce9090;
  arr[9] = 0x00ce9090;
}
```
ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- Setting array elements

2 payload bytes connect with jumps
ASM.JS Payloads

ASM.JS Statements Suitable to Embed Code

- Foreign function call with double values

```javascript
"use asm"

var ffi_func = ffi.func

function asm_js_function()
{
    var val = 0.0
    val = +ffi_func(2261634.5098039214, 156842099844.51764, 1.0843961455707782e+16, 7.477080264543605e+20)
```

// 0x4141414141414141
// 0x4242424242424242
// 0x4343434343434343
// 0x4444444444444444
ASM.JS Payloads

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
- same executable region
- continuous in address space

→ gapless, arbitrary shellcode possible
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

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)
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 on architecture with **fixed instruction length** and **instruction alignment**
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
More Flaws beyond JIT-Spray
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)
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)
Summary
Summary

- JIT-Spray simplified client-side exploitation in the past
- ASM.JS in OdinMonkey (x86) was the perfect JIT-Spray target
- JIT-Spray was possible on x86 and ARM
- JIT-Spray seems 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?