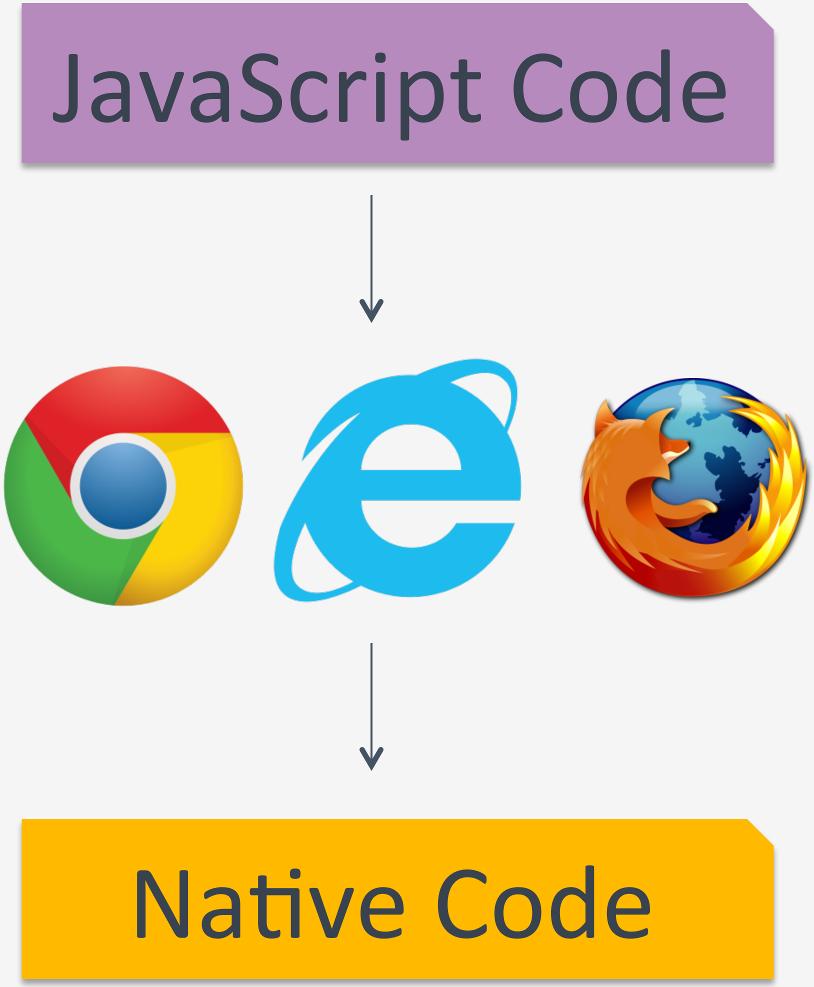
The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines

Michalis Athanasakis	FORTH
Elias Athanasopoulos	FORTH
Michalis Polychronakis	Stony Brook University
Georgios Portokalidis	Stevens Institute of Tech.
Sotiris Ioannidis	FORTH

JIT Engines



JavaScript Code

Native Code

History of JIT exploitation

Data Execution
Inside JIT buffer

Code – Data separation
Finite JIT buffer

Surgical ROP in JIT
(no spray)

Fine-grained randomization
Constant blinding

This Work

Bypassing state of the art defenses
Locating randomized gadgets

Threat model

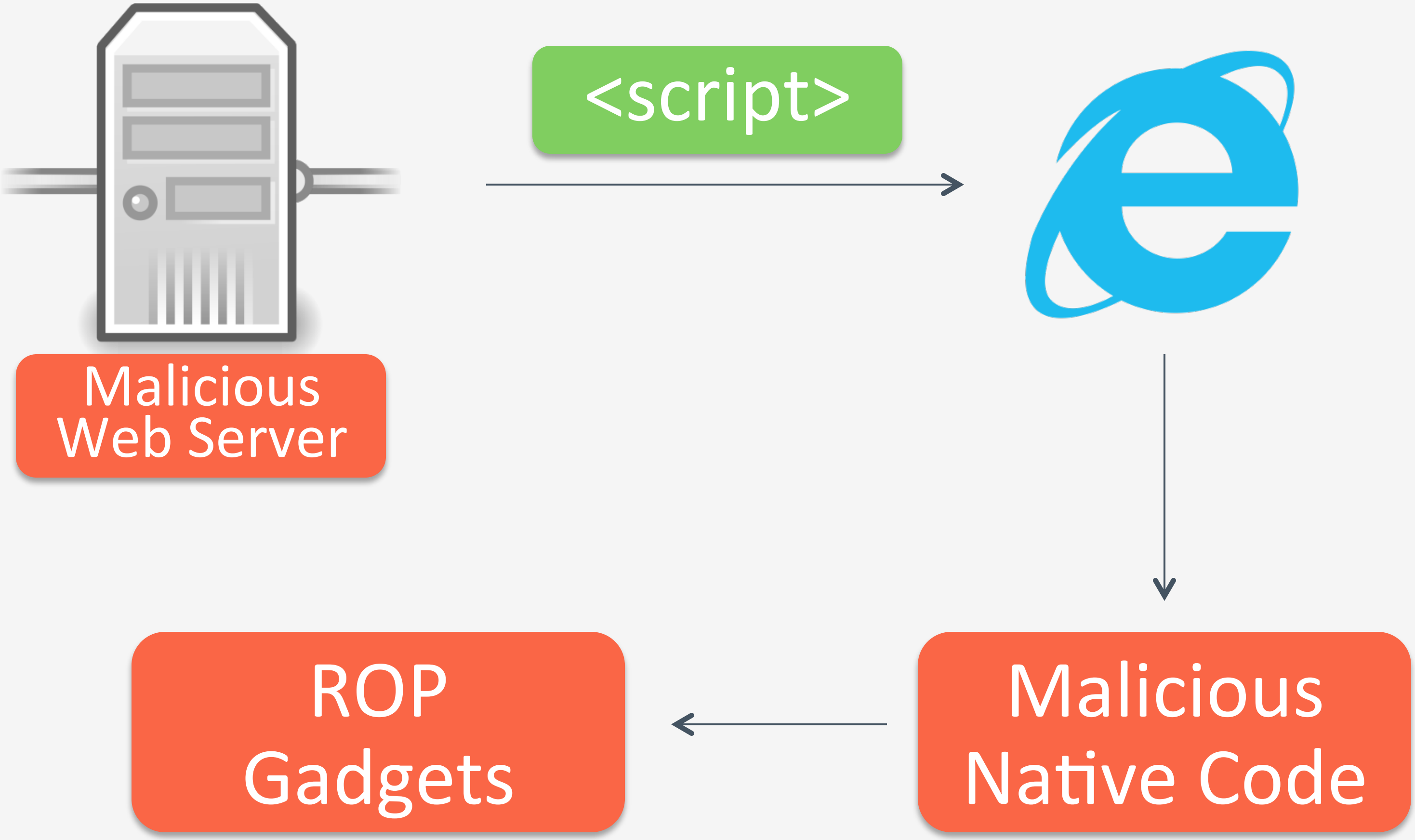
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)

- Gadget free environment

  Software compiled with G-free framework

- Browser-specific defenses
  - Fine-grained randomization
  - Constant blinding

High level approach



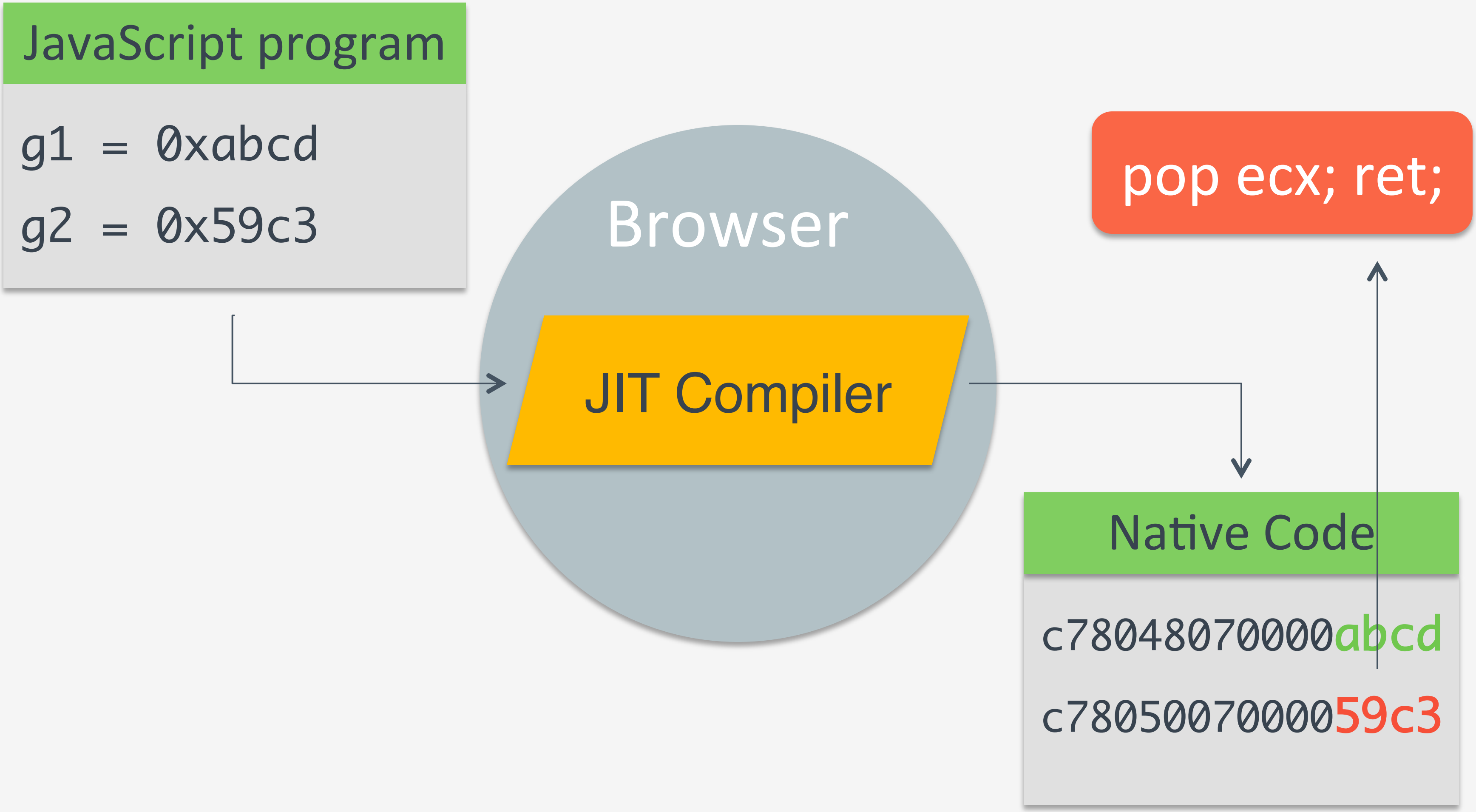
Malicious Web Server

<script>

Malicious Native Code

ROP Gadgets

How the attack works



JavaScript program

```
g1 = 0xabcd
g2 = 0x59c3
```

Native Code

```
c78048070000abcd
c7805007000059c3
```

Attacked browsers

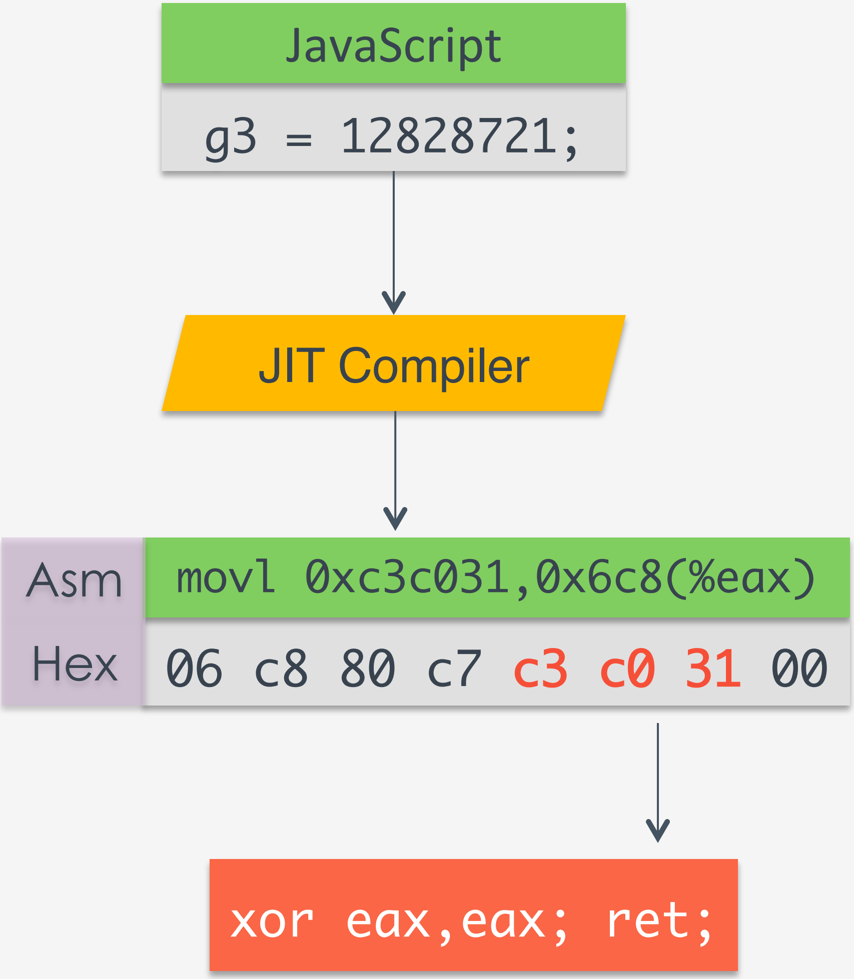
- Mozilla Firefox
  - OS: Linux 32-bit
  - JS Engine: SpiderMonkey

- Internet Explorer
  - OS: Windows 8.1 64-bit
  - JS Engine: Chakra

Exploiting Mozilla

- Target: Call mprotect()
- Required gadgets

```
pop %ebx; ret;
pop %ecx; ret;
xor %eax , %eax; ret;
mov 0x7d , %al; ret;
xor %edx , %edx; ret;
mov 0x7 , %dl; ret;
int 0x80; ret;
```



JavaScript

```
g3 = 12828721;
```

JIT Compiler

Asm	movl 0xc3c031,0x6c8(%eax)
Hex	06 c8 80 c7 c3 c0 31 00

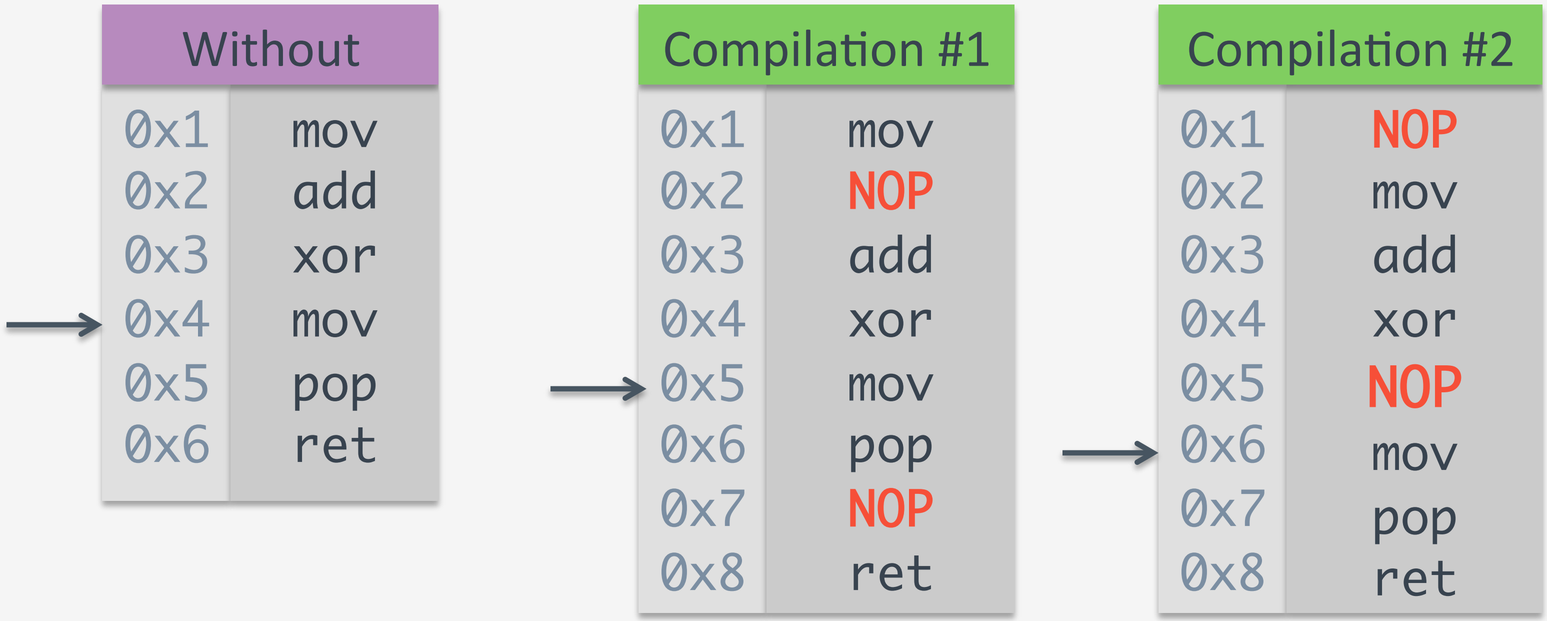
```
xor eax,eax; ret;
```

Internet Explorer defenses

- Fine-grained randomization
- Constant blinding

Fine-grained randomizations

Randomize JIT Code buffer by inserting NOP instructions each time code is compiled.



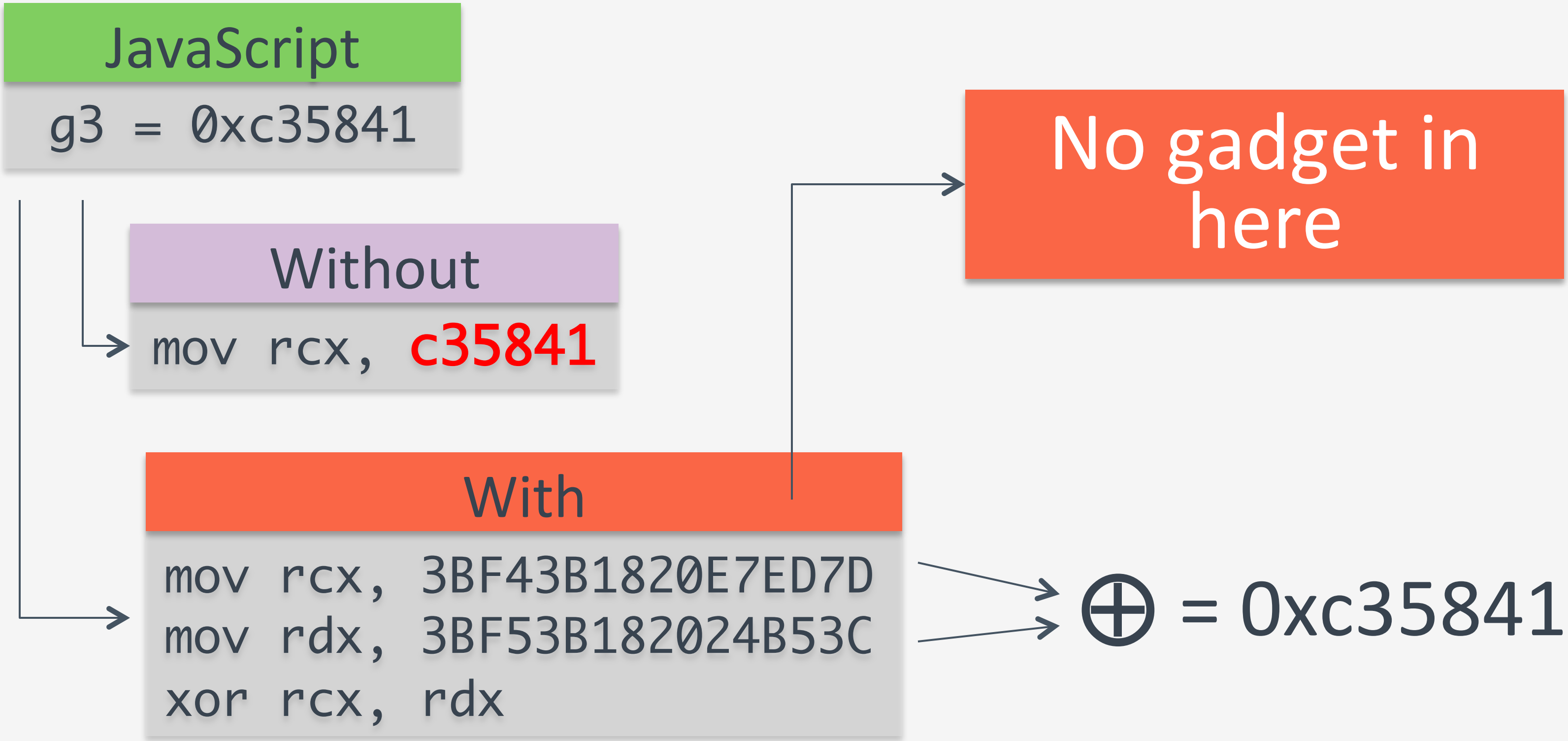
	Without	
	0x1	mov
	0x2	add
	0x3	xor
→	0x4	mov
	0x5	pop
	0x6	ret

	Compilation #1	
	0x1	mov
	0x2	NOP
	0x3	add
	0x4	xor
→	0x5	mov
	0x6	pop
	0x7	NOP
	0x8	ret

	Compilation #2	
	0x1	NOP
	0x2	mov
	0x3	add
	0x4	xor
	0x5	NOP
→	0x6	mov
	0x7	pop
	0x8	ret

Constant Blinding

- XOR all immediate values with a secret cookie
- Emit code that XORs the value at runtime



JavaScript

```
g3 = 0xc35841
```

Without

```
mov rcx, c35841
```

With

```
mov rcx, 3BF43B1820E7ED7D
mov rdx, 3BF53B182024B53C
xor rcx, rdx
```

$\oplus$ = 0xc35841

No gadget in here

Bypassing IE's JIT Defenses 1/3

- Target: Call VirtualProtect()
- Required gadgets

```
pop %r8; ret;
pop %r9; ret;
pop %rcx; ret;
pop %rdx; ret;
pop %rax; ret;
```

- IE only blinds immediate values larger than 2 bytes

- We can still use 2-byte immediate values to generate gadgets

Bypassing IE's JIT Defenses 2/3

- Creating r8, r9 gadgets

Example JS source

```
function r8(x) { return 0x5841 }
function r9(x) { return 0x5941 }
```

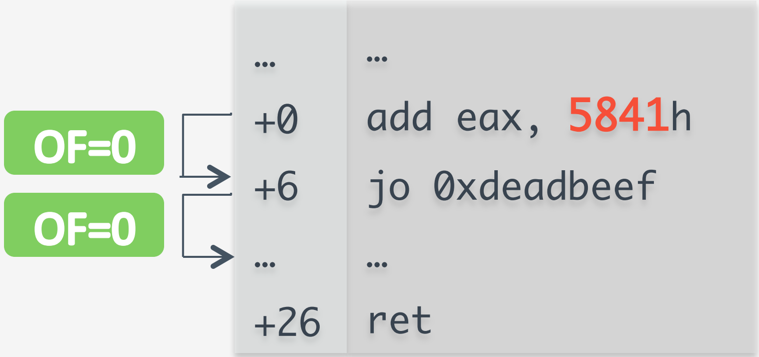
11 instructions – 26 bytes long

Gadget r8

```
pop r8
add rax,al
jo 0xdeadbeef
mov rcx, 1000
or rax,rcx
add rsp,30
pop rbx
pop rsi
mov rsp,rbp
pop rbp
ret
```

Bypassing IE's JIT Defenses 3/3

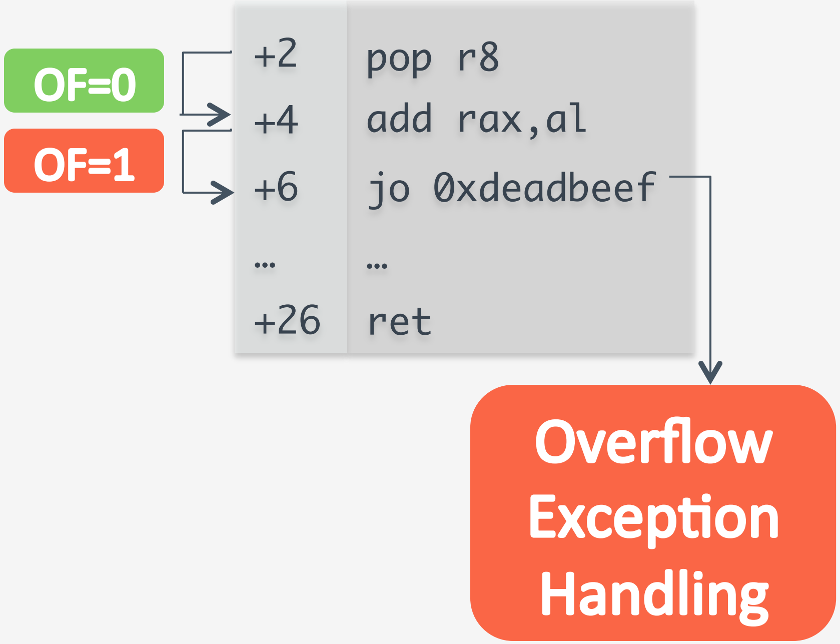
- Usable r8, r9 gadgets by altering Overflow Flag (OF)
  - Normal Execution
  - Malicious Execution



```
…     …
+0    add eax, 5841h
+6    jo 0xdeadbeef
…     …
+26   ret
```

OF=0

OF=0



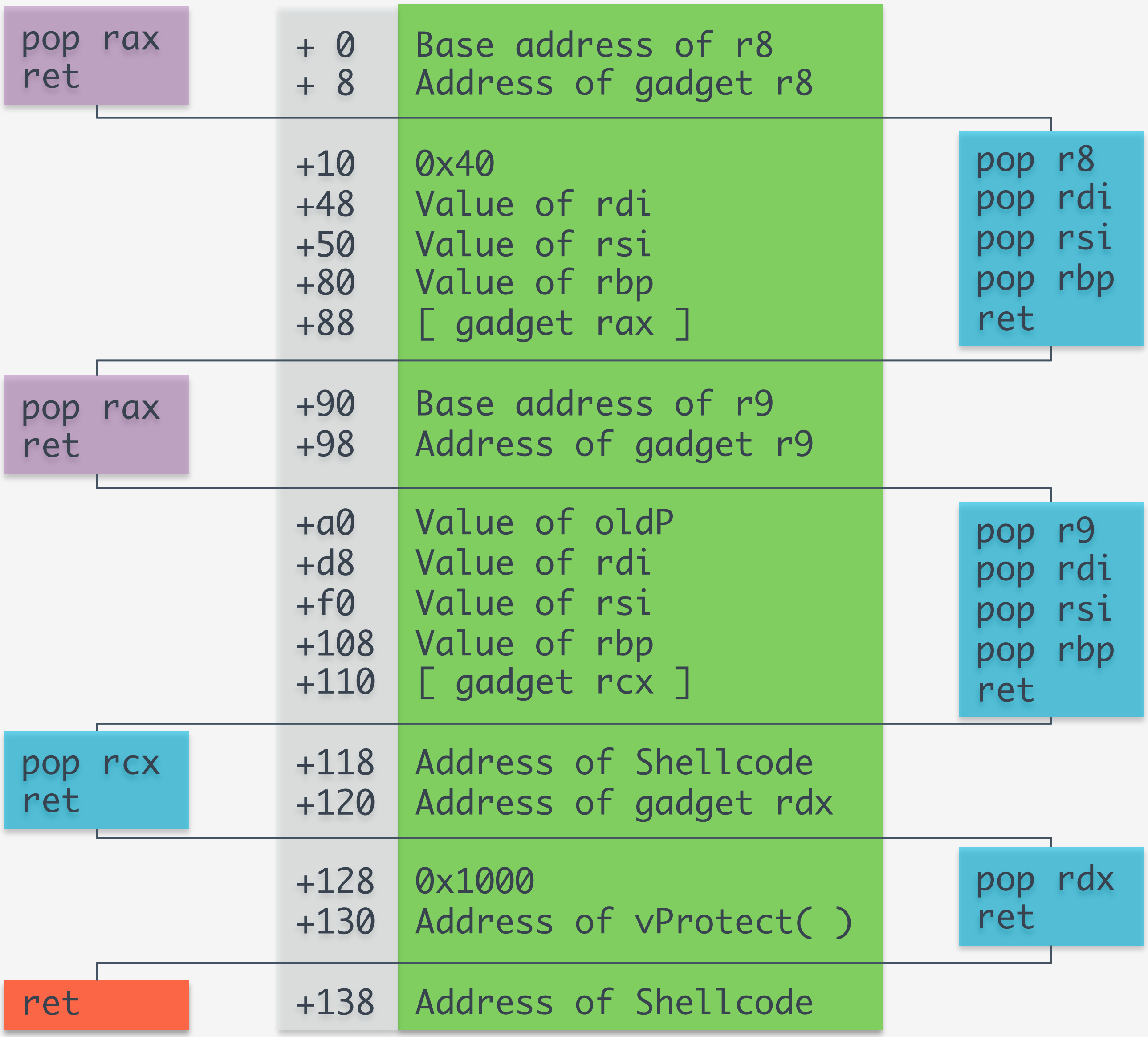
```
+2    pop r8
+4    add rax,al
+6    jo 0xdeadbeef
…     …
+26   ret
```

OF=0

OF=1

Overflow Exception Handling

Internet Explorer ROP stack



```
pop rax
ret
```

+ 0	Base address of r8
+ 8	Address of gadget r8
+10	0x40
+48	Value of rdi
+50	Value of rsi
+80	Value of rbp
+88	[gadget rax]
+90	Base address of r9
+98	Address of gadget r9
+a0	Value of oldP
+d8	Value of rdi
+f0	Value of rsi
+108	Value of rbp
+110	[gadget rcx]
+118	Address of Shellcode
+120	Address of gadget rdx
+128	0x1000
+130	Address of vProtect( )
+138	Address of Shellcode

```
pop r8
pop rdi
pop rsi
pop rbp
ret
```

```
pop rax
ret
```

```
pop r9
pop rdi
pop rsi
pop rbp
ret
```

```
pop rcx
ret
```

```
pop rdx
ret
```

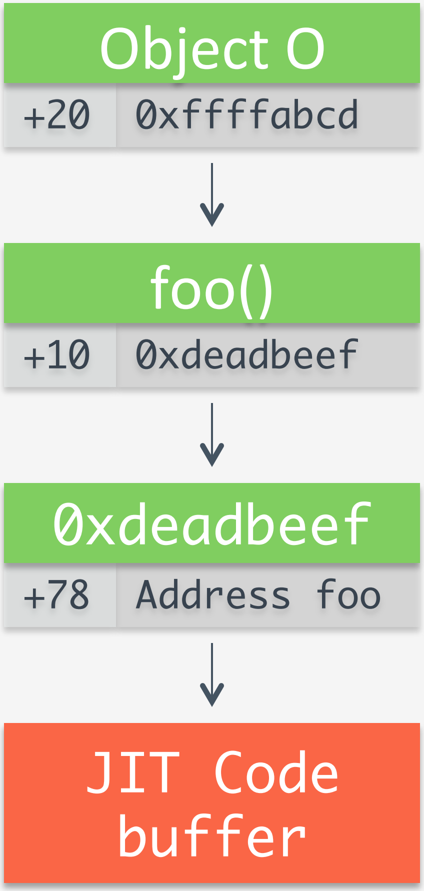
```
ret
```

Locating Gadgets

Just-in-time code reuse: On the effectiveness of fine grained address space layout randomization [Security & Privacy 2013]

Example JS source

```
O = new Object()
O.g1 = 0xc358
O.g2 = 0xc359
function foo(x) { return 0x5841 }
O.func = foo
```



Evaluation of Constant Blinding

Why not blind all immediate values?

- Platform: SunSpider Benchmark Suite

- Log all JIT instructions *actually* executed
- Count all immediate-related ones
- Calculate their CPU cycles
- Evaluate the overhead

- Additional CPU cycles required is an average of 45% with a maximum of 80%

Possible Defenses

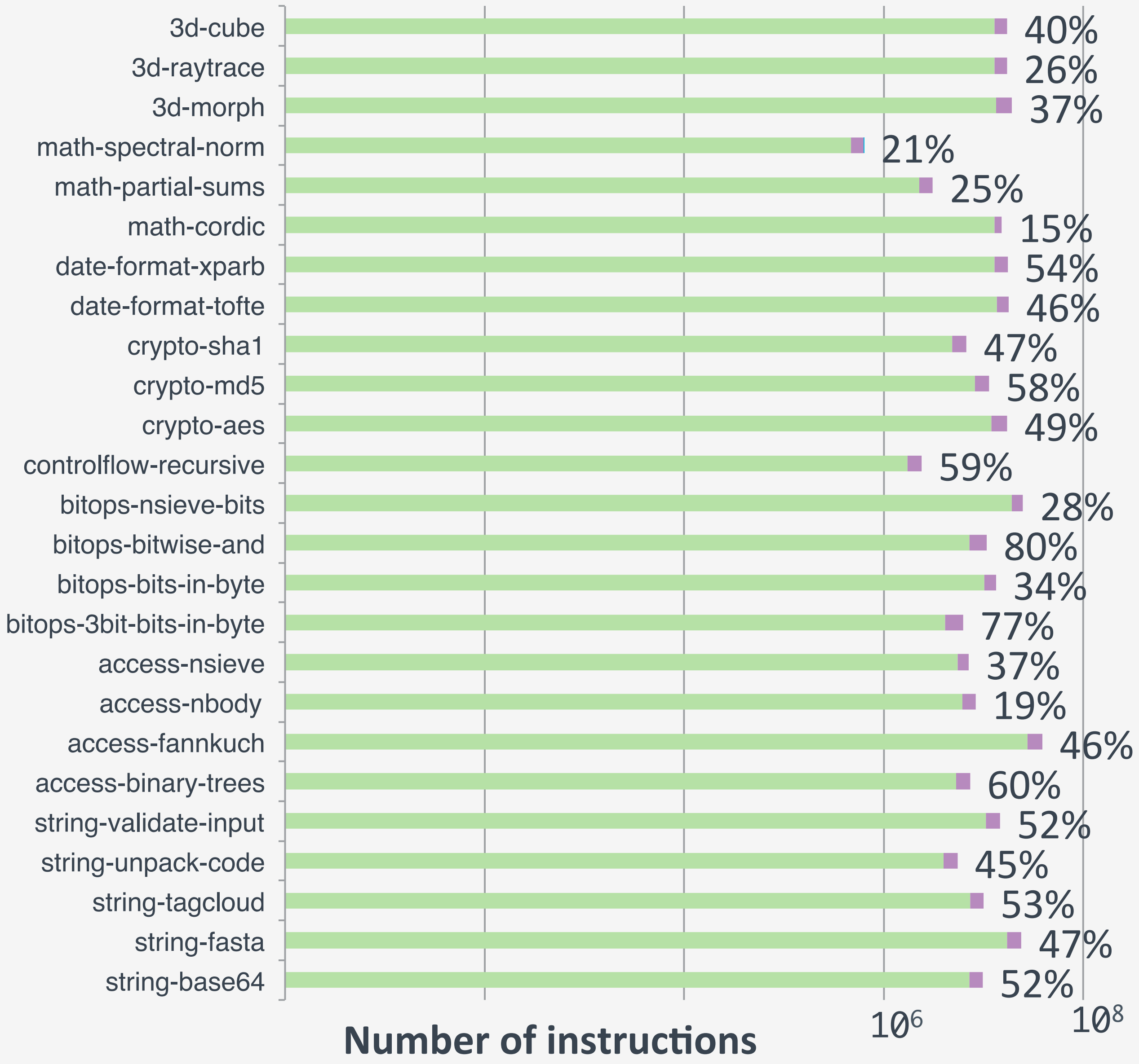
- Internet Explorer
- Librando [CCS 2013]
- JIT Code analysis
- JavaScript analysis

Conclusions

- State of the art defenses can be bypassed
- Gadgets can be generated and located despite fine-grained randomization and constant blinding
- Browsers are still vulnerable!
- Possible defenses are not as easy as they seem or have not been adopted yet

Questions?

More data about constant blinding


3d-cube 40%
3d-raytrace 26%
3d-morph 37%
math-spectral-norm 21%
math-partial-sums 25%
math-cordic 15%
date-format-xparb 54%
date-format-tofte 46%
crypto-sha1 47%
crypto-md5 58%
crypto-aes 49%
controlflow-recursive 59%
bitops-nsieve-bits 28%
bitops-bitwise-and 80%
bitops-bits-in-byte 34%
bitops-3bit-bits-in-byte 77%
access-nsieve 37%
access-nbody 19%
access-fannkuch 46%
access-binary-trees 60%
string-validate-input 52%
string-unpack-code 45%
string-tagcloud 53%
string-fasta 47%
string-base64 52%
10^6
10^8
Number of instructions