A Survey of Return-Oriented Programming: Attack, Defense and its Benign Use

George W. Wood Jr.
(gwwood@miners.utep.edu)
Javier Soon
(jisoon@miners.utep.edu)

Outline

- Motivation
- Attack Mechanisms
- Defense Mechanisms
- Benign Uses
- Future Work
- Reference

Motivation

- Return Oriented Programming (ROP) - Process by which the $W \oplus X$ security model may be defeated.
- $W \oplus X$ Security Model
  - Memory Space cannot be writable and executable simultaneously
  - 2003 PaX/Exec Shield patches (OpenBSD 3.3/Linux Kernel 2.6.18-8)
  - 2004 Windows Data Execution Protection (DEP)
- Ret to LibC
  - 23 Years Old (1997)
  - Possible without the use of function calls (Shacham, 2007)

Attack Mechanisms - A Comparison

- Traditional Buffer Overflow
  - Goal: Arbitrary Code Execution
  - Method: Overwrite Return Pointer with arbitrary Shell Code
  - Prerequisite Conditions:
    - Overflow Vulnerability
    - Adequate Space for Stable Code Execution
- ROP
  - Goal: Arbitrary Code Execution
  - Method: Overwrite the Return Pointer with a series of pointer to code that already exists in memory
  - Prerequisite Conditions:
    - Overflow Vulnerability
    - Usable Instruction Fragment Addresses
    - Small Code Fragments in Memory

Attack Mechanisms - Visualization

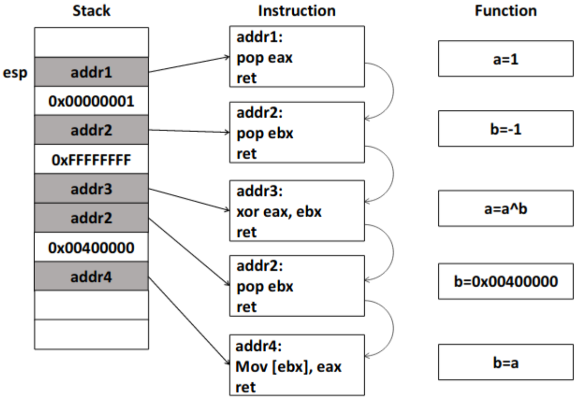


Figure 1: Example of Overflow using ROP

```
IMAGE_BASE_0 = 0x08048000 # ed7ae8f6df9fa6730f4bb3dd6f34601e355fcad9504f252c9e76a68b1df94dd5
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = ''

rop += rebase_0(0x000042b4) # 0x0804c2b4: pop edi; ret;
rop += '//bi'
rop += rebase_0(0x0000401b) # 0x0804c01b: pop esi; ret;
rop += rebase_0(0x000201e0)
rop += rebase_0(0x0000df98) # 0x08055f98: mov dword ptr [esi], edi; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x000042b4) # 0x0804c2b4: pop edi; ret;
rop += 'n/sh'
rop += rebase_0(0x0000401b) # 0x0804c01b: pop esi; ret;
rop += rebase_0(0x000201e4)
rop += rebase_0(0x0000df98) # 0x08055f98: mov dword ptr [esi], edi; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x000042b4) # 0x0804c2b4: pop edi; ret;
rop += p(0x00000000)
rop += rebase_0(0x0000401b) # 0x0804c01b: pop esi; ret;
rop += rebase_0(0x000201e8)
rop += rebase_0(0x0000df98) # 0x08055f98: mov dword ptr [esi], edi; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
# Filled registers: ebx, eax,
rop += rebase_0(0x000016cd) # 0x080496cd: pop ebx; ret;
rop += rebase_0(0x000201e0)
rop += rebase_0(0x0000410a) # 0x0804c10a: pop ebp; ret;
rop += p(0x0000000b)
rop += rebase_0(0x00005c77) # 0x0804dc77: xchg eax, ebp; ret;
# INSERT SYSCALL GADGET HERE
print rop
[INFO] rop chain generated!
(ls/ELF/x86)>
```

Figure 2: Ropper EXECVE Ropchain using /bin/ls

Attack Mechanisms - Extended ROP

- Return-less ROP (Checkoway)
  - Update-Load-Branch Sequence
  - ret becomes pop <reg>; jmp <reg>
- Pure-Call Oriented Programming (PCOP) (Sadeghi)
  - Gadgets end in call opposed to ret
  - Difficult but feasible
  - 2017 First Proof of Concept

- Goal: Protect a program at binary level
- Method: Detecting the frequency and length of code fragments ending with ret
- How:
  - Count length of code fragment
  - Count length of contiguous code fragments ending with ret
  - Depends on the length of the gadget instruction and the number of consecutive times of gadget instruction and the number of consecutive times of gadgets defined
  - Works best when combined with control flow integrity method

- Goal: Increase difficulty to obtain addresses
- Method: Randomizes base addresses
- Why:
  - ROP depends on the use of existing instructions in memory (attacker needs to know where is what)
  - ASLR (Address space layout randomization) randomizes of stacks, heaps, external libraries
  - As address are now randomized, gadgets cannot be easily found for an attack

- Goal: Detect ROP attack
- Method: Modifies code layout during compile time, or by using a dynamic approach (KBouncer)
- Why modify code layout:
  - Code is added that can check behavior of free branch instructions
  - If the targeted free branch instruction fails a defense system is triggered
  - Can also re-write binary to eliminate unintended gadgets
- Why use KBouncer:
  - Can be used in the binary level as it is a dynamic approach
  - Number of ret is the same as number of call
  - When system call is launched, check whether every ret was located after the corresponding call site of the call function
  - counting the length of contiguous code fragment ending with ret

Benign Uses - Program Stenography

- Goal: Hide certain instructions and functions
- Why:
  - Existing techniques of stenography may violate $W \oplus X$, or mandatory code signing security mechanisms
  - Due to being a dynamic attack, a disassemble can not pick up the unintended gadgets in a binary using static analysis.

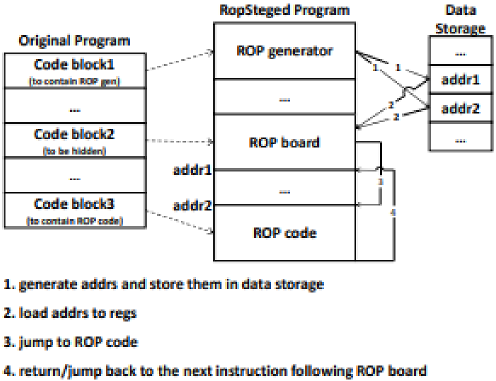
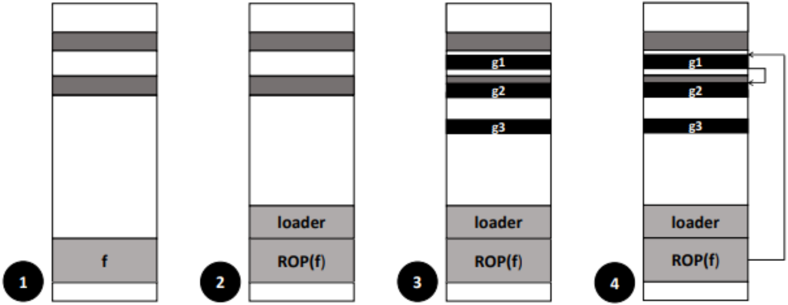


Figure 3: Example RopSteg

- Goal: Verify no manipulation
- Why:
  - If there is manipulation the gadget verification will fail
  - Gadgets overlap that are written and or found in the binary or constructed by code rewrite to generate a verification function
  - Gadgets will be created from rewrite-able gadgets, new gadgets will be marked as overlapping
  - Gadgets will be Turing complete compliant



Select the verification function (f), and the code regions which require protection (dark gray rectangles).

Translate the verification function into a ROP chain (the addresses of the gadgets are not yet known).

Find already existing gadgets, and insert new ones that overlap with the regions to protect (g1 – g3).

Recompile the ROP chain to use the actual gadgets. Prioritize the gadgets overlapping protected regions.

Figure 4: Example Parallax

Benign Use - Software Watermarking

- Goal: Watermark a program/software
- How:
  - Find useful gadgets located in libraries
  - Create "carriers" which are functions that are split up into other functions
  - The author embeds small pieces of code in the "carriers" reducing suspicion
  - Chaining the "carriers" with special gadgets, use stack-shifting gadgets at the end of them so that each of the segments is responsible to relocate the stack frame correctly to the exact memory address of the next one
  - Triggering ROP with function pointer overwriting

Future Work

Further exploration and investigation of non-destructive uses of ROP in computing.

Reference

- J. Wang, P. Xie, Y. Wang and Z. Rong, "A Survey of Return-Oriented Programming Attack, Defense and Its Benign Use," 2018 13th Asia Joint Conference on Information Security (AsiaJCIS), Guilin, 2018, pp. 83-88, doi: 10.1109/AsiaJCIS.2018.00022.