Anti Return-Oriented Programming (ROP)

A Moving Target Defense

Omer Boehm
Ayman Jarrous
Duv Murik

IBM Research Lab - Haifa
Return oriented programming (ROP) attack

- Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as $W \oplus X$ memory and code signing.

- In this technique, an attacker gains control of the call stack or heap etc. to hijack program control flow and then executes carefully chosen machine instruction sequences typically ending with a ‘ret’ instruction (‘gadgets’) allowing an attacker to perform arbitrary operations.

* $W \oplus X$ - Each memory page is either Writable OR Executable
Return oriented programming (ROP) attack

“good” codes may do “bad” behavior

Application Code → “Good” Behavior

Attacker Code → “Bad” Behavior
Return oriented programming (ROP) attack

Ordinary programming

| instr | instruction | instruction | instruction |

Instruction Pointer

The value of Instruction Pointer defines the program control flow
Return oriented programming (ROP) attack

ROP programming

The content of the Stack defines the program control flow
Return oriented programming (ROP) attack

Pulls an address into ebx and a value into eax.

The attacker control the stack's contents and so can make sure these instructions obtain the desired.

Chaining these together allows an attacker to edit the contents of any memory address the current running thread has permission to alter – this is called arbitrary write.
Return oriented programming (ROP) attack

Areas of strength

- Turing-complete (there are ROP compilers)
- Circumvent Data Execution Prevention (NX)
- Circumvent code signing and trusted computing
Existing solutions

- Address Space Layout Randomization (ASLR)
  - ASLR is the state-of-the-art protection against ROP attacks.
  - Randomly choose base address of stack, heap and code segment
  - Randomly pad stack frames and dynamic memory allocation requests (malloc)
  - Randomize location of Global Offset Table
  - Not all files support ASLR
  - Using memory disclosure, the base address of the text section, stack, etc. can be discovered.
  - In 32 bit OS, the randomization range of the address space is small and attackers try to guess or infer the base.
Existing solutions

- **Hardware**
  - Shadow stack – size, legacy code
  - Branch recording units (LBR, BHRB) – size, pattern based, trigger

- **Software tools**
  - kBouncer, ROP guard, ROP pecker
  - Triggers - (e.g. calls to winAPI’s, access violation when execution reaches a new page, ret miss-predictions)
Existing solutions - limitations

- Kernel Level Approaches (32 bit processes on linux)
  - Low entropy: heap 13 bit, stack 24 bit
    - De-Randomization attack can defeat ASLR in minutes
  - Kernel support required
  - Pad wastes memory space.
  - Performance overhead

- User Level Approaches
  - Usually easily circumvented if approach is known
  - Limited, hard, ware resources
  - Runtime overhead, pattern based
Solution - Source

- Create many versions of the executable
  - At compilation time, we generate many versions of the executable.
  - These are distinct in binary form, but perform identical functionality
  - Libraries can be shuffled too (assuming we have source code)
  - Attackers have no knowledge regarding memory layout and content of their target
    - Attackers can’t gather addresses of ROP widgets to create attack payload
Solution - Source

- How to generate distinct executables/libraries?
  - Shuffle functions order
  - Shuffle objects order
  - Insert empty sections between functions
  - NOPs? Illegal instructions? Dead code
  - Change code alignment
Solution - Source

$ SHAKEDOWN_SEED=123 shakedown-clang
mytest.c -o mytest-123
Shakedown: seed=123: Module mytest.c:
shuffling 5 functions... done
$ nm -n mytest-123
...
0000000000400530 t myfunc_a
0000000000400550 T main
0000000000400580 T myfunc_c
00000000004005b0 t myfunc_b
...

$ SHAKEDOWN_SEED=456 shakedown-clang
mytest.c -o mytest-456
Shakedown: seed=456: Module mytest.c:
shuffling 5 functions... done
$ nm -n mytest-456
...
0000000000400530 t myfunc_b
0000000000400550 T myfunc_c
0000000000400580 T main
00000000004005b0 t myfunc_a
...

© 2017 IBM Corporation
Solution - binary

- Critical Observation
  - Attackers use absolute addresses during the attacks

- Nullify Attackers’ Assumption
  - Makes the memory locations of program objects unpredictable
  - Forces attackers to guess memory location with low probability of success

- Low overhead
- Works on legacy code – backwards compatible – non intrusive
- Instruction level Permutation
- Can be done at load-time (requires OS loader support)
- Code and data segments are permuted with fine-grained randomization.
Solution – binary challenge

- What parts of an binary file needs rewriting?
- How do we find the correct locations of those parts and rewrite them?
- How those parts affect each other during run time?
- How to find cross-references between program objects
Solution - binary

Read and ‘analyze’ the binary

Divide it to distinct chunks

Generate random permutation

- The shuffling has no overhead.
- No need for source code
- Support legacy code

- A different permutation is generated per-invocation
- Supports executable binaries and dlls

Update references to/from the new code/data layout
Thanks