ROPecker
A Generic and Practical Approach
For Defending Against ROP Attacks

Yueqiang Cheng$, Zongwei Zhou*, Miao Yu*,
Xuhua Ding$ and Robert H. Deng$

$ School of Information Systems, Singapore Management University
* ECE Department and CyLab, Carnegie Mellon University
Return-Oriented Programming

- ROP attack is a code-reuse attack
  - No injected malicious code
- To launch an ROP attack
  - Identify intended gadgets
    - End with indirect branches, e.g., ret, jmp, call
    - Small size
    - Sparsely distributed -> (imply) needing large code base
  - Chain identified gadgets
Existing Approaches

1. Generic
2. Transparent
3. Efficient

Performance Overhead

Low

High

No Code Instr.

Code Instr.

Program Binary

Source Code

Requirements

ROPecker Is HERE!

Return-less, EuroSys’10
G-Free, ACSAC’10
CFLocking, ACSAC’11

ILR and IPR, S&P’12
Binary Stirring,
CCS’12
CCFIR, S&P’13
KBouncer,
USENIX’13

None

ROPdefender, AsiaCCS’11

ROPecker Is HERE!

3. Efficient

1. Generic

2. Transparent

CFLocking,
ACSAC’11

G-Free,
ACSAC’10

ROPecker

None

ROPdefender,
AsiaCCS’11

Return-less,
EuroSys’10

G-Free,
ACSAC’10

CFLocking,
ACSAC’11

ILR and IPR,
S&P’12

Binary Stirring,
CCS’12

CCFIR,
S&P’13

KBouncer,
USENIX’13

None

ROPdefender,
AsiaCCS’11

Return-less,
EuroSys’10

G-Free,
ACSAC’10

CFLocking,
ACSAC’11

ILR and IPR,
S&P’12

Binary Stirring,
CCS’12

CCFIR,
S&P’13

KBouncer,
USENIX’13
Assumptions

• DEP mechanism is enabled
  – NO attempt to protect self-modified applications
• ROP gadgets are sparsely distributed
  – Need large code base for collecting intended gadgets

• NOT rely on ASLR mechanism
• NOT rely on side information
Design Rationale

• Evidence of ROP Attack
  – Reliable - adversary can not modify them to evade detection
  – Sound – solid evidence

• Timing of detection
  – Event driven (NOT busy monitoring)
  – Non-bypassable
Design Overview

• Reliable and Solid evidence - ROP gadget chain
  – Last Branch Record (LBR)
  – Runtime execution flow

• Timing of checking
  – When the execution flow jumps out of the sliding window
ROPecker Architecture

Offline Phase
- App X Binary
- Pre-processor
- Instruction & Gadget Database

Run-time Phase
- CPU Execution Trace
- Monitor Window
- Stack
- Runtime Info

Kernel
- ROPecker Kernel Module

Apps

ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks
Offline Pre-processing

• Instruction Disassembling
  – Reliable
    • Only disassemble 6 instructions in a time (not the whole application)
  – Efficient
    • Only do once for each application/library

• Extracted instruction information is saved in database
  – Save runtime cost
Sliding Window

• Refers to a small portion of code pages
  – Only code pages within the window are executable

• Non-bypassable
  – No enough gadgets within the window for ROP attackers

• Efficient
  – Temporal and spatial locality feature - sliding window could be non-contiguous code pages
Sliding Window Update

Sliding window has 2 code pages

```c
1: int helper(int cmd, char* in){
2:   log(cmd, in);
3:   switch(cmd){
4:     case CMD_START:
5:       start(inputs);
6:       break;
7:     ....
8:   }
9: }
```
Detection Algorithm

Checking Point Triggered

1. Condition Filtering
   - Yes
   - No

2. Past Chain Detection
   - Yes
   - No

3. Future Chain Detection
   - Yes
   - No
   - No

ROP Attack

Continue The Original Execution Flow

No ROP
Condition Filtering

- ROPecker is able to distinguish the exceptions triggered by the sliding window from others
  - PID
  - Error code

<table>
<thead>
<tr>
<th>Present bit</th>
<th>Read/write bit</th>
<th>User/supervisor bit</th>
<th>Reserved bit</th>
<th>NX bit</th>
</tr>
</thead>
</table>

- Error code triggered by sliding window is 0x15
- All other error codes are not 0x15 in the normal executions
Past and Future Chain Detection

• Past execution traces in LBR
  – LBR number is limited, e.g., 16 records
  – Filter out noise records

• Future gadget chain
  – Directly calculate using (offline) generated database
    • Reduce emulation times
  – Seldom triggered Runtime emulation
    • Low performance overhead
Discussions

• Gadget chain threshold
  – The threshold is small, e.g., 5 for Apache
  – Performance degradation is limited

• Stack pivoting
  – ROPecker (kernel) is able to know the position of stack
Implementation and Evaluation

• Implemented on Linux (Ubuntu 12.04 with kernel 3.2.0-29-general-pae)
• New tools for offline processor
  – diStorm + Perl + objdump + readelf
• Kernel module consists of 7K SLOC
  – Runtime emulator (from Xen) is 4.4K SLOC
  – Use NX mechanism for sliding window creation
  – Modify IDT for page fault interception
Space Evaluation

• The databases for all 2393 shared libraries under /lib and /usr/lib of the Ubuntu Linux 12.04 distribution is about 210MB
  – On average, the database of each lib is 90KB
  – Compressed to about 19MB using bzip2

• Each database only has one copy in memory
  – e.g., all protected processes share one libc database
Performance Evaluation

• Micro-benchmark
  – Past gadget chain detection – 0.07µs
  – Future gadget chain detection – 0.91µs (w/o emulation), 2.61µs (with emulation)

• Macro-benchmark
  – SPEC INT2006 - 2.60% overhead on average
  – Bonnie++ - 1.56% overhead
  – Apache - 0.08% overhead on typical (4KB) HTTP communications
Limitations

• Short gadget chain
  – e.g., one-gadget ROP attack

• Long gadgets
  – e.g., a gadget with 20 instructions

• Gadget Gluing Attack
  – Constructing a special gadget which consists of two short code sequences glued together by a direct branch instruction
Conclusions

• ROPecker
  – Efficiently, transparently and effectively defend against ROP attacks
  – Without relying on any other side information or binary instrumentation.
  – Small space and performance cost

• Code is available, please contact with strongerwill@gmail.com
Thanks Questions?

Yueqiang Cheng

strongerwill@gmail.com