PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning

Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, Xiangyu Zhang
Background

- **Difficulties in Malware Behavior Analysis**
  - the needed environment or setup may not present
  - recent malware makes use of time-bomb and logic-bomb to hide payload
  - sophisticated malware even use cloaking technique to anti-analysis

- **Forced Execution**
  - penetrate malware self-protection mechanisms and various trigger conditions
  - works by force-setting branch outcomes of some conditional instructions
  - challenge: maintain crash-free execution
X-Force v.s. PMP

• X-Force: heavy-weight
  • track individual instructions
  • reason about pointer alias relations on-the-fly
  • repair invalid pointers by on-demand memory allocation

• PMP: light-weight
  • no tracking individual instructions
  • no on-demand memory allocation and pointer repair
  • pre-allocate a large memory buffer
  • fill the buffer and variables with carefully crafted random values before execution
Example

takes the false branch
is forced to take the true branch

```c
typedef struct{double *f1; long *f2;} T;

void foo() {
    long **a = malloc(...);
    T *b;
    if (cond1()) init(b);
    if (cond2()) {
        long *c = b->f2;
        *(b->f2) = **a; // [0x0008] = [0x0010]
        *(b->f1) = 0.1; // [0xffd0] = 0.1
        long tmp = *c;
    }
}
```

Local Variables

<table>
<thead>
<tr>
<th></th>
<th>c: 0x08</th>
<th>b: 0x20</th>
</tr>
</thead>
</table>

PAMA

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| b |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 01 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 02 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 03 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 04 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 05 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 06 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 07 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 08 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 09 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
```
Architecture of PMP

Path Explorer

Dispatcher

memory scheme 1

memory scheme n

Executor 1

Executor n

40492b:T | 404aec:T |
404e07:T | 401f3f:F |
401ee3:T | 404fdc:F |
404fca:T | 405118:F |
40513a:F | 405144:F |
40517b:F | 40517f:F |
Memory Pre-planning

Program Loading
- crafted file
- mmap
- PAMA
- address space

PAMA Preparation

During Execution
- program entry → Global Variables Init
- call instruction → Local Variables Init
- memory allocation → Heap Variables Init
PAMA Preparation

![Diagram of memory allocation]

- high address (0xffffffff)
- stack
- heap
- .bss
- .text
- Pre-Allocated Memory Area (PAMA)
- low address (0x0)
Variable Initialization

- **Global Variables**
  - read the offset and size information of the .bss segment from the ELF header
  - set .bss segment with random values indicating word-aligned PAMA addresses

- **Heap Variables**
  - intercept all memory allocations
  - set the allocated regions to contain random word-aligned PAMA addresses

- **Local Variables**
  - initialize the entire stack region like a heap region during program loading
  - intercept each function invocation to reinitialize the overwritten stack regions
SCMB and SDMB Properties

• SCMB (Self-Contained Memory Behavior)
  • if the filling values are interpreted as memory address, the corresponding accesses still fall into PAMA
  • violations of SCMB lead to memory access exceptions

• SDMB (Self-Disambiguated Memory Behavior)
  • it is highly unlikely that two semantically unrelated memory operations access the same random address
  • violations of SDMB lead to bogus dependences and corrupted variable values
Example

takes the false branch
is forced to take the true branch

```c
typedef struct{double *f1; long *f2;} T;
void foo() {
    long **a = malloc(...);
    T *b;
    if (cond1()) init(b);
    if (cond2()) {
        long *alias = b->f2;
        *(b->f2) = **a; // [0x0008] = [0x0010]
        *(b->f1) = 0.1; // [0xffd0] = 0.1
        long tmp = *alias;
    }
}
```

Local Variables

<table>
<thead>
<tr>
<th>a: 0x01ed7010</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: 0x20</td>
</tr>
</tbody>
</table>

Heap Region

```
0x1ed7010
```

PAMA

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

0x0000

```
80 fe 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0x0010

```
48 74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0x0020

```
d0 ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0xffff0

```
20 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

b->f2

b->f1

Implementation

• Based on QEMU User-Mode Emulator
  • instrument conditional jumps and indirect jumps to enforce path scheme
  • currently supports ELF binary on x86_64 platform

• Practical Challenges
  • handling file and network I/O, infinite loop and recursion
  • allocation of large PAMA
  • misaligned memory access
Probability Analysis

- **Definition**
  - \( \text{PA} \): set of all possible addresses within PAMA
  - \( \text{WA} \): word-aligned subset of \( \text{PA} \), \( \text{FV} \): random subset of \( \text{WA} \)
  - \( S = |\text{PA}| = |\text{WA}| \times 8 \): size of PAMA, \( d = |\text{FV}|/|\text{WA}| \): diversity of filling values

- **Probabilistic Guarantee of SCMB**
  \[
P_{\text{err1}} = P((x+\alpha) \notin \text{PA} \mid x \in \text{FV}) = \frac{\alpha}{S-8} \cdot \left(1 - \frac{8}{d \cdot S}\right) 
  \]
  (1) error1: out-of-bound access

- **Probabilistic Guarantee of SDMB**
  \[
P_{\text{err2}} = P(x = y \mid x \in \text{FV}, y \in \text{FV}) = \frac{8}{d \cdot S} 
  \]
  (2) error2: coincidental address collision

  \[
P_{\text{err3}} = P(l(x, \beta) \cap l(y, \gamma) \neq \emptyset \mid x \in \text{FV}, y \in \text{FV}) \\
  \leq \frac{64}{d^2 \cdot S^2} + (1 - \frac{8}{d \cdot S})^2 \cdot \frac{\beta + \gamma - 8}{S-8} 
  \]
  (3) error3: coincidental address overlap
Probabilities of Errors in a Typical Setting

• Typical Setting
  • 4-MB pre-allocated memory area ($S = 0x400000$)
  • 2 executors ($n = 2$)
  • diversity of filling values $d$ is set to be $1$
  • $\alpha = 8$, $\beta = 0x1000$, $\gamma = 0x1000$

• Probabilities of Errors
  • $P_{err1} = 1.9073e-06$
  • $P_{err2} = 1.9073e-06$
  • $P_{err3} = 0.00195$
Evaluation Settings

• Subjects:

• Computing Resources
  • 8-core CPU (Intel® Core™ i7-8700@ 3.20GHz)
  • 16G main memory

• Time budget
  • no time limit for Spec2000
  • 5 minutes for each malware sample
Evaluation on SPEC2000

- SPEC2000: a well-known benchmark set
  - 12 real-world programs
  - some of them are large (e.g., 176.gcc)

- Comparison
  - execution outcomes
  - code coverage
  - memory dependence
Evaluation on **SPEC2000**

- **Execution Outcomes**
  - PMP is 84 times faster than X-Force
  - the failure rate is similar

- **Code Coverage**
  - PMP has comparable code coverage with X-Force (83.8% v.s. 82.7%)
  - PMP achieves 100% code coverage for some programs while X-Force does not

- **Memory Dependence**
  - X-Force has 6.5 times more false positives than PMP
  - X-Force has 10% more false negatives than PMP
Evaluation on Malware

• 400 Malware Samples
  • half of them are from VirusTotal
  • half of them are from Padawan

PADAWAN VS X-force Habo Analysis System

cuckoo system time fast-forwarding anti-virtualization-detection cuckoo ++
Evaluation on Malware

- PMP reports more than twice syscall sequences of that of other tools
- PMP is 9.8 times faster than X-Force
- PMP yields 1.5 times longer path schemes than X-Force

(a) number of exposed syscall sequences  (b) executions per second  (c) length of path scheme
Case Study: C&C Bot Malware Sample

• Simplified Code Snippet

```c
01 char *data = read_file("/sys/class/dmi/id/product_name");
02 if (contains(data, "VirtualBox", "VMware")) {
03   remove_self_and_exit();
04   while (1) {
05     char *ip = select_intranet_ip(ip_list);
06     char *vuln = select_known_vuln(vuln_list);
07     if (connect_and_check(ip, vuln)) {
08       send_info_to_server(ip, vuln);
09       send_payload(ip, vuln);
10     }
11   }
```

VM detector
communication to the selected IP address
sending host info and payload

• Comparison of Different Tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Cuckoo</th>
<th>Habo</th>
<th>Padawan</th>
<th>Cuckoo++</th>
<th>X-Force</th>
<th>PMP</th>
</tr>
</thead>
<tbody>
<tr>
<td># syscall sequences</td>
<td>153</td>
<td>169</td>
<td>292</td>
<td>221</td>
<td>274</td>
<td>705</td>
</tr>
</tbody>
</table>
Availability

Experimental version of PMP:
https://github.com/pmp-tool/PMP
Thank you!

Q & A