Record-Replay Architecture as a General Security Framework

Yasser Shalabi\(^1\), Mengjia Yan\(^1\), Nima Honarmand\(^2\), Ruby B. Lee\(^3\), and Josep Torrellas\(^1\)

\(^1\)University of Illinois at Urbana-Champaign

\(^2\)Stony-Brook University

\(^3\)Princeton University

HPCA 2018
Motivation

- Security continues to grow in importance
  - More impact
- Cost-effective complete defenses
- Attackers continuously evolving to bypass defenses
  - Flexibility in defenses would be useful
Example

- Stack-Overflow and Stack Smashing
  - Stack-Overflow allows rewriting stack
  - Stack Smashing was the initial technique, where malware injected onto stack executed there
Example

```
void vulnerable(char * str) {
    char buffer[128];
    //Buffer overflow!
    strcpy(array, str);
    return;
}
```
Example

void vulnerable(char * str)
char buffer[128];
//Buffer overflow!
strcpy(array, str);
return;

Benign Execution

Application Binary

Application Stack

6157FF800FF0ABDDEC39
38EBF0FAB0AC81778015

50C657FEE7C38EF0C61
57FF800F038EBAC817780
15F0C6157FF800C3DEE7
938AC36157FF800FF0AB
DDEE7938EC30FAB0AC81
5F0C657FEE7C38EF0C6
157FF800F03C36157FF80
330FA
B0AC815F0C657FEE7

Example Call
Example

```c
void vulnerable(char * str)
char buffer[128];
//Buffer overflow!
strcpy(array, str);
return;  //return;
```
Example

```c
void vulnerable(char * str)
char buffer[128];
//Buffer overflow!
strcpy(array, str);
return;
```

Malicious Execution

Application Binary

Application Stack

6157FF800FF0ABDDEC39
38EBF0FAB0AC81778015

Call

buffer[0:127]
Example

void vulnerable(char * str)
    char buffer[128];
    //Buffer overflow!
    strcpy(array, str);
    return;

Example ret PC

Application Binary

Malicious Execution

6157FF800FF0ABDDEE7938EF0C61
38EBF0FAB0AC81778015

vulnerable

Application Stack

buffer[0:127]

call

PC

PC
Example

- Stack-Overflow and Stack Smashing
  - Stack-Overflow allows rewriting stack
  - Stack Smashing was the initial technique, where malware injected onto stack executed there
- Defenses Upper Hand: W^X -- "No eXecute"
Example

```c
void vulnerable(char * str)
char buffer[128];
//Buffer overflow!
strcpy(array, str);
return;
```

Malicious Execution

Application Binary

Application Stack

Return Address

buffer[0:127]
Example

- Stack-Smashing
  - Stack-Overflow allows rewriting stack
  - Initial attacks were to inject malware onto stack and redirect execution there
- Defenses Upper Hand: W^X -- “No eXecute”
- Code Reuse Attacks emerged to bypass
  - Return Oriented Programming used ret instructions
  - Allows indirect execution of the corrupted stack
Return Oriented Programming

- Gadget is a useful piece of code terminated by a ret instruction

```c
void vulnerable(char * str)
{
    char buffer[128];
    //Buffer overflow!
    strcpy(array, str);
    return;
}
```

Application Binary

Application Stack
Return Oriented Programming

```c
void vulnerable(char * str)
char buffer[128];
//Buffer overflow!
strcpy(array, str);
return;
```

Application Binary

Application Stack
Return Oriented Programming

Application Binary

Application Stack

Buffer overflow!

void vulnerable(char * str)
{
    char buffer[128];
    strcpy(array, str);
    return;
}

pop R1
ret
pop R1
ret
pop R1
ret

i-acoma group
Enter RnR
Record and Deterministic Replay (RnR)

- Technique for reproducing past executions faithfully
- As the program executes, it is recorded
  - Non-Deterministic (ND) events saved in a log
- In the replayed execution:
  - ND events trapped and reproduced using values from the logs
  - Deterministic events can be reproduced by re-execution
Enter RnR-Safe
Our Proposal: RnR-Safe

• Use RnR for threat detection and attack verification
• **Recording Machine:** detect all threats and raise alarm
  • IF intrusion possible → raise alarm
  • Ok to have false positives!
• **Replay Machine:** investigate alarm and verify attack
  • Expensive software analysis to determine if an alarm is:
    • True intrusion
    • False alarm
    • Possible to further characterize intrusions
Contribution: RnR as a Security Framework (RnR-Safe)
Contribution: RnR as a Security Framework (RnR-Safe)
Summary

• Recorder
  • Records inputs, captures alarms and passes them to replayers

• Checkpointing Replayer
  • Consume log, produce checkpoints, invoke alarm replayers

• Alarm Replayers
  • Replay from recent checkpoint and determine if there was a valid reason for the alarm
Using RnR-Safe
# How to Use RnR-Safe

<table>
<thead>
<tr>
<th>Attack</th>
<th>Alarm Trigger</th>
<th>Possible First Detection Technique</th>
<th>Role of Replay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP (this paper)</td>
<td>RAS mis-prediction</td>
<td>Manage a multi-threaded RAS, use a whitelist</td>
<td>Execute a kernel-compatible shadow stack algorithm</td>
</tr>
<tr>
<td>Jump Oriented Programming (JOP)</td>
<td>Stray indirect branch or call</td>
<td>Table of begin and end addresses of the most common functions</td>
<td>Verify if the target is one of the less common functions</td>
</tr>
<tr>
<td>Denial of Service (DOS)</td>
<td>Kernel scheduler inactivity</td>
<td>Counter of number of context switches</td>
<td>Identify reason for low switching frequency</td>
</tr>
</tbody>
</table>

**Table 1: Examples of potential RnR-Safe uses.**
Summary: How to Use RnR-Safe

1. Design a reliable alarm trigger (HW or SW)
   • Guarantee that an attack will trip the alarm
2. Define an accurate replay SW analysis
   • Replay must perfectly discriminate between false alarms and true intrusions
3. Minimize false alarm rate by enhancing trigger
Using RnR-Safe to Prevent ROP Attacks

RAS, Our Alarm Trigger
Return Address Stack (RAS)

- Used to predict the target of return instructions

One more RET?
Using the RAS to Detect ROPs

• The RAS mis-prediction is valid detector ROPs
• If a ROP attack occurs, at least one of the returns will mis-predict
Using the RAS to Detect ROPs

- The RAS mis-prediction is valid detector ROPs
  - If a ROP attack occurs, at least one of the returns will mis-predict
- **Challenge:** Not every misprediction is a ROP attack
  - Underflow
  - Multi-Threading
  - Non-Procedural Returns
  - Imperfect Nesting
Reducing False Alarms

Again, in context of RAS based ROP detection
Problem: Supporting a Multi-Threaded Environment

• As threads are de-scheduled the RAS entries are left behind
  • Will lead to mis-predictions (false alarms)
• **Solution:** Spill the RAS to hyper-visor memory
  • Hyper-visor can introspect OS to determine thread-id of upcoming thread
  • RAS can be swapped during VM exits and re-entries
Structures Used to Support Multiple Threads

Memory Structure (Software)

| Cnt | Cnt | Cnt | Cnt |

Processor Structures (Hardware)

BackRASptr

RAS

Thread ID

i

j

BackRAS array
Problem: Supporting Non-Procedural Returns

• Legal for SW to use push+ret instructions to branch
• We only found one source of this
  • Within the kernel scheduling code
  • Only three possible destinations
• Solution: White-list all valid (src, dst)
  • Challenge with white-list is potential for bypass
  • So we specifically white-list the 3 possible (src,dst) pairs
Problem: RAS Underflow and Imperfect Nesting

- For underflow, imperfect nesting:
  - **Solution:** Leave to the alarm replayer
- It is possible for checkpoint-replay to suppress alarms
  - Log the underflows into Eviction Records
  - Requires VMExit on underflow
  - The checkpoint replayer can suppress any alarms from underflows with valid eviction records
Performance Discussion
Experimental Evaluation

- Goal of the evaluation:
  - Assess the overhead of recording
  - Assess the overhead of replay on a different machine
  - Determine if false-alarm frequency can be reasonably contained
Recording Overhead

- Recording adds on average only 27%
Checkpoint replayer has comparable execution speed to the recorded
Most Kernel False Alarms Suppressed

- Practically no false alarms for the replay to handle
Alarm Replayer Execution Time

- Alarm replayer is expensive, but rarely invoked
Conclusions

• RnR-Safe is a framework where RnR is used to complement HW security features.

• Alarm Replay gives security features breathing room:
  • Can suffer false-positives and still be useful.
  • Can trade complexity for imprecision.

• Checkpointing replayer has comparable execution speed to the recorder (thus, can replay all the time).

• Alarm replayer has to handle few false positives.
Thank you! Questions?
Recording Overhead

![Chart showing recording overhead distribution for various applications]

- Rdtsc
- Interrupt
- Pio/Mmio
- Network

Applications:
- apache
- fileio
- make
- mysql
- radiosity
Replay Overhead

![Bar chart showing overhead distribution for different applications and categories: Rdtsc, Interrupt, RAS, Pio/Mmio, Network, Chk. The applications shown are apache, fileio, make, mysql, and radiosity. The overhead distribution percentages are indicated for each application and category.]
Alarm Replayer

• Alarm replays are invoked by the checkpoint replayer as alarms are encountered
• The alarm replayer needs to perform analysis such that
  • RAS underflows detected
  • Imperfect nesting detected
• Starts from most recent checkpoint