MARDU: Efficient and Scalable Code Re-Randomization

SYSTOR '20: Proceedings of the 13th ACM International Systems and Storage Conference

Christopher Jelesnianski (Virginia Tech), Jinwoo Yom (Virginia Tech), Changwoo Min (Virginia Tech), Yeongjin Jang (Oregon State University)
The Fight against Return Oriented Programming (ROP)

What is Return Oriented Programming?
- An attack that reuses program code to achieve *arbitrary code computation*

What are Gadgets?
- Snippets of code that perform specific actions
  - Arithmetic operations
  - Reading/writing to registers
  - Etc.

---

**Attack**
- Code Injection
- Return Oriented Programming (ROP)

**Defense**
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Fine-Grained ASLR & eXecute-only Memory (XoM)
- Continuous Randomization

**Attack Diagram**
- Return Oriented Programming (ROP)
- Just-In-Time ROP (JIT-ROP)
- Blind ROP (BROP) (Code Inference)

**Defense Diagram**
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Fine-Grained ASLR & eXecute-only Memory (XoM)
- Continuous Randomization
Current randomization techniques are good ...

**Code Randomization**

- Address Space Layout Randomization (ASLR)
  - Light-weight
  - Static code layout
  - One leak can compromise entire code base

- Re-Randomization Techniques
  - Continuous churn makes gadgets hard to find
  - High overhead
  - Rely on predictable thresholds such as
    - Time interval
    - Syscall invocation
    - Call history
But they are not practical. Why?

- Users desire **acceptable performance** (<10% avg & worst-case)
- Users desire **strong defenses**
- Users desire **scalability** as more computation is moved to the cloud
  - Have system-wide security coverage including shared libraries
- Achieving all three together is **hard**
Outline

- Introduction
- Challenges
- MARDU Design
- Implementation
- Evaluation
- Conclusion
Challenges for making a practical randomization defense

- **Security** challenges
  - **Code disclosure**: a single leaked pointer allows attacker to obtain code layout of a victim process

- **Performance** challenges
  - **Avoiding useless overwork**: Active randomization wastes CPU cycles in case of “what-if”

- **Scalability** challenges
  - **Code Tracking**: to support runtime re-randomization tracking and updating of pc-relative code is a necessary and expensive evil
  - **Stop-the-world**: Updating shared code on-the-fly is challenging especially in concurrent access
Outline

- Introduction
- Challenges
- MARDU Design
  - Security: Leveraging code trampolines
  - Scalability: Enabling code sharing
  - Performance: Re-randomization without stopping the world
- Implementation
- Evaluation
- Conclusion
Example: Code Control Flow

Source Code

```c
void foo(){
    /* ... */
    bar();
    /* ... */
}

void bar(){
    /* ... */
}
```

Traditional Control Flow

1. ```call foo()```
2. ```call bar()```
3. ```ret```
4. ```ret```
MARDU is secure

- **Code** and **Trampoline** regions protect *forward* edge
  - Trampolines are immutable code targets
  - Protects against code disclosure

- **Shadow stack** protects *backward* edge

- Randomization occurs at:
  - Process startup AND
  - Whenever an attack is detected (*on-demand*)
    - Process crash
    - Execute-only memory violation
Example: Securing MARDU Code

Source Code

```c
void foo()
{ /* … */
  bar(); /* … */
}

void bar()
{ /* … */
}
```

Using Code Trampolines Control Flow

**Code Region**
- `foo_body`:
  ```
  /* ... */
  jmp bar_trampoline()
  /* ... */
  ```
- `foo_ret0`:
  ```
  /* ... */
  jmp ShadowStack_top
  ```
- `bar_body`:
  ```
  /* ... */
  jmp ShadowStack_top
  ```

**Trampoline Region**
- `bar_trampoline`:
  ```
  jmp bar_body
  ```
- `foo_trampoline`:
  ```
  jmp foo_body
  ```
- `foo_ret0_trampoline`:
  ```
  jmp foo_ret0
  ```

Shadow Stack
- `...`
- `foo_ret0_tr`
Outline

- Introduction
- Challenges
- **MARDU Design**
  - Security: Leveraging code trampolines
  - **Scalability:** Enabling code sharing
  - Performance: Re-randomization without stopping the world
- Implementation
- Evaluation
- Conclusion
MARDU is scalable

- MARDU is capable of code sharing (e.g., shared libraries)
  - No previous randomization scheme is capable of runtime re-randomization AND code sharing

- MARDU leverages position independent code (\(-fPIC\)) for easy fixups of PC-relative code.

- MARDU supports mixed instrumented and non-instrumented libraries
Example: Sharing MARDU code

- **MARDU-compiled Binary/Library**
  - **Code Region (C)**
  - **Trampoline Region (T)**
  - **Fixups**

- **.text Section**

- **MARDU Patch Info Section**

- **In-Kernel Randomized code cache**
  - **Random Offset**
  - **Place Code Region**
  - **Place Trampoline Region**

- **Perform patching**

- **Map Kernel Memory**

- **Map Kernel Memory**
  - **0xffffffff81171000**
Example: Sharing MARDU code

In-Kernel Randomized code cache

<table>
<thead>
<tr>
<th>Process 1 Userspace</th>
<th>Process 2 Userspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>C T</td>
<td>C T</td>
</tr>
<tr>
<td>0xffffffff81171000</td>
<td>0xffffffff81171000</td>
</tr>
<tr>
<td>libc.so</td>
<td>libc.so</td>
</tr>
<tr>
<td>0x7fa67811a000</td>
<td>0x7fb67811b000</td>
</tr>
<tr>
<td>webserver</td>
<td>dbserver</td>
</tr>
<tr>
<td>5 0x7fa67811a000</td>
<td>6 0x7fb67811b000</td>
</tr>
</tbody>
</table>

libc.so

0xffffffff81171000
Outline

- Introduction
- Challenges
- MARDU Design
  - Security: Leveraging code trampolines
  - Scalability: Enabling code sharing
  - Performance: Re-randomization without stopping the world
- Implementation
- Evaluation
- Conclusion
Re-Randomization without stopping the world

In-Kernel Randomized code cache

MARDU Process 1
Userspace

webserver

libc.so

MARDU Process 2
Userspace

dbserver

libc.so

EXPOSED

0xffffffff81171000

EXPOSED

EXPOSED
Re-Randomization without stopping the world

- Gadgets previously deduced are now stale
- Randomization is repeated whenever another attack event is detected
- Randomization is replicated for ALL ASSOCIATED shared code of a victim process

In-Kernel Randomized code cache

1. Map new region
2. Map Code v2 to userspace
3. Map Trampoline v2 to userspace
4. Unmap old region

Process 1
Userspace

MARDU

webserver

libc.so

C v1
C v2
T v2

0xffffffff81171000

0xffffffff2245d000

0xffffffffffffff2245d000

0xffffffffffffff81171000
MARDU is performant

- Trampolines
  - No Runtime Instrumentation Tracking

- Trampolines leverage immutable code
  - No stop-the-world mechanisms

- Re-active re-randomization
  - Only when attack detected (on-demand)
  - Responsibility of exiting (crashed) process/thread
Outline

- Introduction
- Challenges
- MARDU Design
- Implementation
- Evaluation
- Conclusion
MARDU Implementation

- Working Prototype

- Compiler
  - LLVM/Clang 6.0.0
  - 3.5K LOC

- Kernel
  - X86-64 Linux 4.17.0
  - 4K LOC

- musl LibC
  - General C library
Outline

- Introduction
- Challenges
- MARDU Design
- Implementation
- Evaluation
  - How to evaluate MARDU?
  - Security: MARDU against popular ROP attacks
  - Performance: Compute Bound -> minimal runtime overhead
  - Scalability: Concurrent Web server -> negligible runtime overhead and scalability
- Conclusion
How to evaluate MARDU?

1) How secure is MARDU, against current known and popular attacks on randomization?

2) How much performance overhead does MARDU impose?

3) How scalable is MARDU in terms of load time, memory savings, and re-randomization, particularly for concurrent processes (such as a real-world web server)?
Outline

- Introduction
- Challenges
- MARDU Design
- Implementation
- **Evaluation**
  - How to evaluate MARDU?
  - **Security**: MARDU against popular ROP attacks
  - **Performance**: Compute Bound -> minimal runtime overhead
  - **Scalability**: Concurrent Web server -> negligible runtime overhead and scalability
- Conclusion
How MARDU defends against popular ROP

- **Blind ROP (BROP) & Code Inference Attacks**
  - MARDU: XoM protected code triggers a permission violation and re-randomization of code
  - MARDU: Re-randomization makes all previous collected layout information stale
  - MARDU: Usage of trampolines & function granularity randomization makes correlation prediction challenging for attackers

- JIT-ROP Attacks
- Low Profile Attacks
- Code Pointer Offsetting Attacks
Outline

- Introduction
- Challenges
- MARDU Design
- Implementation
- Evaluation
  - How to evaluate MARDU?
  - Security: MARDU against popular ROP attacks
  - Performance: Compute Bound -> minimal runtime overhead
  - Scalability: Concurrent Web server -> negligible runtime overhead and scalability
- Conclusion
Experimental Setup and Applications

- **Experimental Setup**
  - All programs compiled with MARDU LLVM compiler and `-O2 -fPIC` optimization flags
  - Platform:
    - 24-core (48-Hardware thread) machine with two Intel Xeon Silver 4116 CPUs (2.10 GHz)
    - 128 GB DRAM

- **Applications**
  - SPEC CPU 2006 (All C applications)
  - NGINX web server
How MARDU performs

CPU Intensive Benchmark (SPEC CPU 2006)

Web server (NGINX)

NGINX AVG Degradation: 4.4%
MARDU randomization with scalability

- Re-randomization latency scales approximately linearly with number of fixups required.
- Cold start randomization latency for any number of workers for NGINX is 61ms.
- Re-randomization latency plateau's even when under attack.

![gobmk: Re-randomization latency (ms) vs. Attack interval](image-url)
Conclusion

We propose MARDU, an re-randomization approach to thwart return oriented programming (ROP) attacks

- MARDU randomizes **re-actively, on-demand** to minimize performance overhead
  - Active randomization is relic of the past

- MARDU is the first randomization scheme capable of runtime re-randomization **with code sharing**
  - Scalable to apply across entire system
  - Randomization of all shared code associated with compromised process/thread

Thank You!