In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication

Sungbae Yoo*, Jinbum Park*, Seolheui Kim1, Yeji Kim1, Taesoo Kim1 2

1 Samsung Research
2 Georgia Institute of Technology
* Primary Co-Authors
Problem: Memory Corruptions are Major Concern in OSes

- 189 memory unsafe CVEs in Linux from 2021 to 2022
- Common attack vector: code-reuse attacks

An attacker manipulates function pointers or return addresses
Promising Defense: Control-flow Integrity (CFI)

- Ensures control-flow transfers remain intact at runtime

CFI checks if the control-flow transfer is allowed

Pointer A

Pointer B

An attacker fails to change control-flow
State-of-the-art of CFI for Commodity OSes

• Type-based CFI
  • Google’s
  
  Still shows large number of allowed target
  → Problem: too course-grained

• Hardware-based CFI
  • iOS Kernel, PARTS, PATTER

Exaining Pointer Authentication on the iPhone XS
Posted by Brandon Azad, Project Zero

Several vulnerabilities to misuse HW
→ Problem: too error-prone
Our Approach: Fine-grained CFI with Hardware Support

- Key idea: Leveraging the common design idioms in OSes
  - Approach 1: Adopting the latest HW-based protection
  - Approach 2: Static validator to avoid mistakes

More fine-grained
Key Enabler: ARM Pointer Authentication (PA)

- ARM PA ensures the integrity of pointers at runtime
- PAC signs a pointer

Diagram:
- Context
- Private Key
- Pointer
- Signing
- PA Code
- Pointer
- Authenticating
- USE
- Signed pointer with the context
Key Enabler: ARM Pointer Authentication (PA)

- ARM PA ensures the integrity of pointers at runtime

- AUT checks the integrity of a pointer and restores the pointer
How to properly Set “context” for Better Precision?

- Naïve solution: using zero
  - # allowed targets: 30K – in Linux

- Strawman solution: using type
  - Max. # allowed targets: 1K - int (*) (struct platform_device *) in Linux
Attack Vectors: Replaying or Substitution

- Re-uses an indirect call with the same context

An attacker finds the signed with the same context

Signed Pointer B w/ ctx C

And then, replaces

USE
Solution: Using more Idiom in Kernel Objects

- An example of actual code in Linux

```c
int __init arch_timer_register(void) {
    ...
    err = request_percpu_irq(ppi,
                             arch_timer_handler_phys,
                             "arch_timer", ...);
    ...
}
```

```c
Int request_percpu_irq(unsigned int irq,
                         irq_handler_t handler,
                         const char *devname, ...) {
    struct irqaction *action = ...
    action->name = devname;
    action->handler = handler;
    ...
}
```

How can we make the context for this as unique as possible?

- Function pointer type
- Constant field using as a identifier
Solution: Using more Idiom in Kernel Objects

- Unique, Invariant, Movable (compatible with memcpy)

```c
struct irqaction {
    irq_handler_t handler;
    const char *name;
}

void func1() {
    struct irqaction *o = ...;
    o->name = "o1";
    o->handler = &target;
}
```
Two Other Attack Vectors: Forging and TOCTOU

- Forging attack
  - Generates a signed pointer using signing gadgets

```
+---------------------+    signing    +---------------------+    authenticating    +---------------------+    USE
| Pointer A           |    pointing   | Signed Pointer B w/ ctx C |    authenticating   | USE
| Pointer B           |                  | An attacker replaces before signing |
```

- Time-of-check to time-of-use (TOCTOU)
  - Manipulates spilled and restored pointers before it uses

```
+---------------------+    pointing    +---------------------+    authenticating    +---------------------+    USE
| Pointer A           |    pointing    | Signed Pointer A w/ ctx C |    authenticating    | USE
| Pointer B           |                  | An attacker replaces before use |
```

Problem: Complex optimization passes in the compiler inadvertently causes the bugs!
Problem: Complex Optimization Passes in Compiler

- Highly sophisticated modern compiler frameworks
  - Unpredictable produced binaries
  - Optimizations could spill out registers to memory

```
x = GET_ADDR(func2)
x = SIGN(x, context)
STORE(x, o->fp)
```

```
adrp x1, func2
... (unpredictable!)
pacia x1, x2
... (unpredictable!)
str x1, [x3]
```
Static Validator: Correctness Check of the Final Binary

1. Complete protection
   : All indirect branches have to be authenticated before use

2. No time-of-check-time-of-use (TOCTOU)
   : Raw pointers after PA instructions are never stored back in memory

3. No signing oracle
   : There must be no gadget that signs an attacker-chosen pointer

4. No unchecked control-flow change
   : All direct modifications of program counter register must be validated
Problem: Preemption Hijacking Attack

- Attackers can occur preemption when they want in kernel
- Preemption context save/restore can be used as a signing oracle

1. An attacker occurs preemption at an arbitrary moment
2. The CPU context is stored to memory
3. An attacker replaces the saved registers to a malicious pointer
4. The CPU context is loaded from memory
5. Then, The malicious Pointer will be signed
Solution: Preemption Context Protection

- Whole preemption context signing via key-chaining technique
  - Prevents substitution attack to part of preemption context

Context's address

Current timestamp

time_pac

Two variables are added into preemption context

Reg_0

Reg_1

Reg_2

Reg_max

pac

An attacker fails to replace part of preemption context

Pointer A

Pointer B

Preemption context
Another Attack Vector: Brute-forcing in Kernel

- Enumerates all possible PA code bits (generally $2^{15}$)

Solution: If an attack is detected, just panicking giving delays with increasing exponentially
System Overview: PAL

Context Analyzer (Semi automated tool)

1. Kernel code → precision level
2. Annotated kernel code
3. PA-protected kernel binary (w/o validation)

Precision analysis report

Compiler instrumentation

Kernel Infrastructure

Static validator

Validated PA-protected kernel binary

If not validated, Feedback!

If validated

Manual annotation (optional)

Manual annotation (optional)

Manual annotation (optional)
Implementation

• Applied to Linux(Tizen, Apple M1 mini), FreeBSD

• PAL
  • GCC plugin (forward-edge) : 3,632 LoC (C++)
  • GCC (backward-edge) : 127 LoC changes
  • Static validator : 848 LoC (Python)
  • Context analyzer : 1943 Loc (C++)

• Infrastructure
  • Linux: 491 LoC changes
  • FreeBSD: 258 LoC changes
Evaluation – Comparing with other approaches

- Google’s - Allowed targets for indirect calls

![Graph showing indirect calls for Google's and PAL with different target counts.

<table>
<thead>
<tr>
<th>#targets</th>
<th>Google’s PAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤5</td>
<td>55.0% 84.9% 88.6% 90.8%</td>
</tr>
<tr>
<td>&gt;100</td>
<td>7.0% 2.8% 1.6% 0.08%</td>
</tr>
<tr>
<td>Max</td>
<td>1,153    35,264 30,622 207</td>
</tr>
</tbody>
</table>

- iOS kernel – Indirect calls sharing the same context

<table>
<thead>
<tr>
<th>#contexts</th>
<th>iOS Kernel</th>
<th>PAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤5</td>
<td>62.2 %</td>
<td>94.9 %</td>
</tr>
<tr>
<td>&gt;100</td>
<td>21.2 %</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Max</td>
<td>6,513</td>
<td>70</td>
</tr>
</tbody>
</table>
Evaluation - Performance

- Micro-benchmark : LMBench
  - Latency: 0-3µs (median. 7%)

- Macro-benchmark : Apache
  - RPi3: 1.06%, Mac mini: 0.75%

Binary increase

<table>
<thead>
<tr>
<th></th>
<th>5.12.0-rc-1/Mac mini</th>
<th>4.19.49/RPi3</th>
<th>FreeBSD/Qemu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock</td>
<td>123.5 MB</td>
<td>19.9 MB</td>
<td>5.9 MB</td>
</tr>
<tr>
<td>w/ PAL</td>
<td>130.7 MB</td>
<td>23.0 MB</td>
<td>6.4 MB</td>
</tr>
<tr>
<td>Overhead</td>
<td>7.2 / 5.8%</td>
<td>3.1 / 15.6%</td>
<td>0.5 / 8.5%</td>
</tr>
</tbody>
</table>
Conclusion

• PAL is a new in-kernel CFI based on ARM PA
  • Leverage the common design idioms in OSes
  • Check the correctness of the final binary

• PAL considers kernel’s characteristics such as preemption

• PAL is fully evaluated on real HW supporting ARM PA
  • Negligible overhead in most workloads
Thank you

Contacts:
sungbae.yoo@samsung.com / ysbnim@gmail.com
jinb.park@samsung.com / jinb.park7@gmail.com

Source code (To be released)
https://github.com/SamsungLabs/PALinux