PACStack
an Authenticated Call Stack

Hans Liljestrånd†, Thomas Nyman‡, Lachlan J. Gunn‡, Jan-Erik Ekberg§‡, N. Asokan†‡

†) University of Waterloo, ‡) Aalto University, §) Huawei Technologies
Control-flow attacks

Exploit memory error (e.g. buffer overflow) to:
• Inject shellcode into writable memory (usually stack)
• Corrupts return address to redirect execution flow

Code injection is prevented by modern defenses

Return address corruption still prevalent!
• Code reuse attacks, e.g., ROP

Elias Levy (as Aleph One), Smashing the stack for fun and profit, Phrack 7 (1996)
Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)
ARMv8.3-A Pointer Authentication

General purpose hardware primitive **approximating pointer integrity**

- Ensure **pointers** in memory remain **unchanged**
- Uses message authentication codes embedded in pointers

**Introduced in ARMv8.3-A specification** (2016)

- First compatible processors 2018 (Apple A12 / **iOS12**)
- Userspace support in **Linux 4.21**, in-kernel support in **5.7**
- Instrumentation support introduced in **GCC 7.0** and **Clang 8**
ARMv8.3-A PA – PAC Generation

Adds Pointer Authentication Code (PAC) into unused bits of pointer

- Keyed, tweakable MAC from pointer address and 64-bit modifier
- PA keys protected by hardware, modifier is decided where pointer created and used

---

Reuse attacks on PA

Adversary may reuse PACs

```c
func1 {
    pacia LR, SP
    str LR
    ...
}
```

```
func2 {
    pacia LR, SP
    str LR
    ldr LR
    autia LR, SP
    ret
}
```

```
/* func1() */
brl %func1
```

```
/* func2() */
brl %func2
```

**pacia** – add PAC

**autia** – authenticate
PACStack: high-level idea

Chained MAC of cryptographically bound return addresses
- Modifier bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns

\[
auth_0 = H_k(ret_0, 0) \\
auth_i = H_k(ret_i, auth_{i-1}) \\
auth_n = H_k(ret_n, auth_{n-1})
\]

\[
ret_0, ret_1, ..., ret_n
\]

\[ auth_i, i \in [0, n - 1] \] bound to corresponding return addresses, \[ ret_i, i \in [0, n] \], and \[ auth_n \]
Collisions still happen!

PAC modifiers meant to prevent pointer reuse

But collisions allow reuse
- Registers cannot be directly corrupted
- But values loaded from the stack are vulnerable

Collisions allow reuse of PAC chain
- Harvest and reuse repeatedly for arbitrary paths

\[
\begin{align*}
\text{auth}_C \rightarrow \text{auth}_{CB} &= H_K(\text{ret}_{\text{loader}}, \text{auth}_{CB}) \\
\text{auth}_C \rightarrow \text{auth}_{CA} &= H_K(\text{ret}_{\text{loader}}, \text{auth}_{CA})
\end{align*}
\]
Mitigation of hash-collisions: PAC masking

**Challenge:** PAC collisions occur on average after $1.253 \times 2^{b/2}$ return addresses
- For $b = 16$, $n = 321$ addresses

**Solution:** Prevent recognizing collisions by masking each $auth$
- pseudo-random mask generated using $pacia(0x0, auth_{i-1})$
- exploitation changes from deterministic to probabilistic

<table>
<thead>
<tr>
<th>Attack</th>
<th>w/o Masking</th>
<th>w/ Masking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse previous auth collision</td>
<td>1</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to existing call-site</td>
<td>$2^{-b}$</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to arbitrary address</td>
<td>$2^{-2b}$</td>
<td>$2^{-2b}$</td>
</tr>
</tbody>
</table>

Maximum probability of success for different attacks
Provable security

PA-based schemes can be analyzed as crypto protocols

We prove security reduction for PACStack

• to collision-resistance of PAC primitive

Formal analysis identified vulnerabilities in early design

• Identified collision attack against non-masked PACStack
• Proved masked PACStack secure

\[ G_{\text{ACS}}(V^k, H, C, q) \]

1: \( K \leftarrow \{0, 1\}^k \)
2: // Give \( k \) tokens from call-graph traversals.
3: for \( i \in \{1, \ldots, k\} \) do
4: \( p_i \leftarrow \text{Alias-request}() \)
5: // Is the request for a real path through the call-graph?
6: if \( \exists j: p_j \rightarrow p_{j+1} \notin \text{edges}(C) \) then
7: return 0
8: endif
9: \( \text{auth}_{\text{hub}} \leftarrow \text{Tag}(p_{\text{hub}}, C_{(p_1, \ldots, p_{\text{hub}})} \parallel p_{\text{hub}}) \)
10: \( \text{Authentic response(auth}_{\text{hub}}) \)
11: endfor
12: \( p_{\text{tamper}}, p_{\text{correct}}, \text{auth}_{\text{correct}} \leftarrow \text{Get}() \)
13: \( p_{\text{auth}}, \text{auth}_{\text{auth}}, p_{\text{auth}} \leftarrow \text{ACS-vrfy}() \)
14: // The substituted masked authenticated return address must be different.
15: if \( p_{\text{correct}} = p_{\text{auth}} \land \text{auth}_{\text{correct}} = \text{auth}_{\text{auth}} \) then
16: return 0
17: endif
18: // Does the return pointer authenticate correctly with the adversary’s
19: // new masked authenticated return address as the modifier?
20: if \( H_k(p_{\text{tamper}}, \text{auth}_{\text{auth}}, p_{\text{correct}}) \)
21: \( \neq H_k(p_{\text{tamper}}, \text{auth}_{\text{auth}}, p_{\text{auth}}) \) then
22: return 0
23: endif
24: // Did the adversary provide a valid masked authenticated return address?
25: if \( \text{auth}_{\text{auth}} = H_k(p_{\text{auth}} \cdot \text{auth}) \)
26: return 1
27: endif
28: return 0
SPEC CPU 2017 benchmarks

Performance overhead measured based on 4–cycle PA-analogue

- with masking ~ 3% (geo.mean)
- without masking ~ 1,2 % (geo.mean)
PACStack provides security comparable to shadow stacks
• but, probabilistic vs. deterministic
• Small overhead without additional hardware requirements

PA is a general-purpose mechanism
• Can support novel non-obvious use cases
• New emergent properties when combined with other HW?