Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path

Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, Justin Cappos

New York University
Tandon School of Engineering
Motivation

1. Many vulnerabilities exist in the host OS kernel
2. These vulnerabilities can be reached and exploited, even with VMs in place

What do we want when building virtual machines?

1. Sufficient functionality
2. Very few zero-day security bugs
The metrics we have don’t meet our needs

1. Predictive of where bugs will be found
2. Locate areas that have no/very few bugs

- Ozment, et al. [Usenix Security ’06]
- Chou, et al. [SOSP ’01]
Our metric: the **popular paths**

- **Definition:** lines of code in the kernel source files, which are commonly executed in the system’s normal workload.
- **Key insight:** the popular paths contain many fewer bugs!
Our experiments to obtain the popular paths

- Ran top 50 most popular packages according to the Debian popularity contest.
- Two students used their Ubuntu systems for five days.
- We used Gcov 4.8.4 in Ubuntu 14.04 to capture the kernel coverage data.
Bug density comparison among three security metrics

**code age**

- $\leq 1$
- 1-2
- 2-3
- 3-4
- $\geq 4$ (year)

**code in device drivers**

- drivers
- non-drivers

**code in the popular paths**

- popular
- unpopular
- unreachable

100% Linux kernel code

0.2 bugs/KLOC

0 bugs/KLOC

- [1] Ozment, et al. [Usenix Security ’06]
- [3] Li, et al. [USENIX ATC ’17]
popular paths vs. unpopular paths

popular paths
(1 bug)

unpopular paths
(19 bugs)
Our metric: the **popular paths**

- Definition ✓
- How to measure it? ✓
- Is it a good security metric? ✓
- Is it practically useful?
Traditional designs: *check-and-pass-through*
**Lock-in-Pop design**

*lock* applications *into* using only *popular* paths

- safely re-create file directories with basic calls like `open()`, `read()`, `write()`, `close()` to avoid using unpopular paths
- the kernel is used infrequently
- only the popular paths in the kernel is accessed
Our prototype implementation: Lind

- Google’s Native Client (NaCl) [IEEE S&P ’09]: software fault isolation

- Repy Sandbox [CCS ’10]
  - Small sandbox kernel (8K LOC)
  - 33 basic API functions
  - Accessed only a subset of the “popular paths”
  - Real-world deployment in the Seattle project, under security audit for 5+ years
Our prototype implementation: Lind
# Evaluation results: Linux kernel coverage by fuzzing

<table>
<thead>
<tr>
<th>Virtualization system</th>
<th># of bugs</th>
<th>Kernel trace (LOC)</th>
<th>Total coverage</th>
<th>In popular paths</th>
<th>In risky paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LXC</td>
<td>12</td>
<td>127.3K</td>
<td>70.9K</td>
<td>56.4K</td>
<td></td>
</tr>
<tr>
<td>Docker</td>
<td>8</td>
<td>119.0K</td>
<td>69.5K</td>
<td>49.5K</td>
<td></td>
</tr>
<tr>
<td>Graphene</td>
<td>8</td>
<td>95.5K</td>
<td>62.2K</td>
<td>33.3K</td>
<td></td>
</tr>
<tr>
<td>Lind</td>
<td>1</td>
<td>70.3K</td>
<td>70.3K</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Repy</td>
<td>1</td>
<td>74.4K</td>
<td>74.4K</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**Evaluation results: Linux kernel bugs triggered**

<table>
<thead>
<tr>
<th>VM</th>
<th>Bugs Triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Linux</td>
<td>35/35 (100%)</td>
</tr>
<tr>
<td>LXC</td>
<td>12/35 (34.3%)</td>
</tr>
<tr>
<td>Docker</td>
<td>8/35 (22.9%)</td>
</tr>
<tr>
<td>Graphene</td>
<td>8/35 (22.9%)</td>
</tr>
<tr>
<td>Lind</td>
<td>1/35 (2.9%)</td>
</tr>
</tbody>
</table>

Example: CVE-2015-5706, a bug triggered everywhere except Lind
- A rarely-used flag O_TMPFILE reached unpopular lines of code inside fs/namei.c
- Lind is not affected, because it is avoiding unpopular paths by restricting flags
Evaluation results: performance overhead in Lind
Limitations

● Some bugs are difficult to evaluate using our metric.
● Reaching lines of code may not be sufficient to trigger or exploit a bug.
● Lind’s performance could be improved.

Future work

● Removing risky lines from the kernel.
● Build a minimal OS kernel for Docker’s LinuxKit, etc.
Conclusion

- The **popular paths**, contain many fewer bugs.
- *Lock-in-Pop* design
- Our prototype system, Lind, exposes fewer zero-day kernel bugs.
Thank You

Q & A