PoLPer: Process-Aware Restriction of Over-Privileged Setuid Calls in Legacy Applications

Yuseok Jeon    Junghwan Rhee    Chung Hwan Kim
Zhichun Li    Mathias Payer    Byoungyoung Lee    Zhenyu Wu
Outline

- Motivation
- Background
- PoLPer
- Evaluation
- Conclusion
Motivation

- Setuid calls
  - Manage privileges
  - Key function for the principle of least privilege (PoLP)
  - Active target of attack

```
r = setuid(0)
```

Privilege

Shell code, ROP attack, Non-control data attack
Motivation

- Previous solutions still have limitation

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI</td>
<td>Data modification attack detection</td>
</tr>
<tr>
<td>DFI</td>
<td>High overhead</td>
</tr>
<tr>
<td>System call context check</td>
<td>Over approximated rule (only handle call and data contexts)</td>
</tr>
<tr>
<td>Setuid semantic Inconsistency check</td>
<td>Control flow hijacking and data modification attack detection</td>
</tr>
</tbody>
</table>

CFI: control flow integrity
DFI: data flow integrity
Outline

- Motivation
- **Background**
- PoLPer
- Evaluation
- Conclusion
Principle of Least Privilege (PoLP)

- Require minimal privileges
  - Minimized attack surface
  - Limited malware propagation
  - Better stability

- Login programs and daemon launchers
  - Switch their IDs from root to the user ID
  - Setuid calls are used for this change of IDs
Setuid Family System Calls

Use three user IDs as parameters
- Real user ID (real uid, or ruid)
- Effective user ID (effective uid, or euid)
- Saved user ID (saved uid, or suid)

<table>
<thead>
<tr>
<th>User ID (4 types)</th>
<th>setuid()</th>
<th>seteuid()</th>
<th>setreuid()</th>
<th>setresuid()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group ID (4 types)</td>
<td>setgid()</td>
<td>setegid()</td>
<td>setregid()</td>
<td>setresgid()</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Background
- PoLPer
- Evaluation
- Conclusion
PoLPPer

- Focus on process contexts of a setuid call
  - Extracts accurate context information
  - Enforces precise least privileges

- We propose PoLPPer
  - Identifies important process contexts
  - Implements automated context extractor
  - Implements run-time enforcer
PoLPer Overview

**Context Analyzer**

- Setuid(0)
- Binary
- Process hierarchy
- Call Stack

**Context Enforcer**

- Setuid(0)
- Binary

- Process hierarchy
- Call Stack

**Process Contexts**

- Static Analysis
- Dynamic Analysis

**Success**

- Allowed

**Failure**

- Denied
- Logged
Process Hierarchy Context

- Leverage different units of execution to decompose functionalities.
• Only child process can access setuid calls
Process Data Context

- Need to handle various parameter setting patterns

<table>
<thead>
<tr>
<th>Case</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Constant</td>
<td>Variable</td>
<td>Variable</td>
<td>Variable</td>
</tr>
<tr>
<td>Value</td>
<td>100</td>
<td>1000</td>
<td>100, 200</td>
<td>200</td>
</tr>
</tbody>
</table>

- Use backward data-flow analysis
- Record together with the process hierarchy context
Process Call Context

- Identify code location to identify setuid call
- Dynamic analysis for high accuracy call context
- Record together with the process hierarchy context
Run-time Enforcement

- Use Kprobes, a kernel-based probing mechanism
  - Hooks on the entry points of setuid calls
  - Extracts process hierarchy, data, and call contexts
  - Compares with the profile that was previously extracted
Outline

• Motivation
• Background
• PoLPer
• Evaluation
• Conclusion
## More Accurate Rule Generation

<table>
<thead>
<tr>
<th>Programs</th>
<th>Process hierarchy context disabled</th>
<th>Process hierarchy context enabled</th>
<th>Rule cut (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sudo</td>
<td>352</td>
<td>196</td>
<td>44</td>
</tr>
<tr>
<td>Xterm</td>
<td>576</td>
<td>296</td>
<td>49</td>
</tr>
<tr>
<td>Cron</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Telnet</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Telnet-Login</td>
<td>6</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>Login</td>
<td>4</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>SSH &amp; SCP</td>
<td>182</td>
<td>88</td>
<td>52</td>
</tr>
<tr>
<td>WireShark</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Nginx</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Micro-benchmark

![Graph showing verification time (µs) vs call context number for different data context counts.]

- Data context count 1
- Data context count 2
- Data context count 5
- Data context count 10

Verification time (µs)

Call context number
## End-to-end Benchmarks

- Show near zero overhead

<table>
<thead>
<tr>
<th>Programs</th>
<th>Base (s)</th>
<th>PoLPer (s)</th>
<th>Setuid call (#)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping</td>
<td>9.0019</td>
<td>9.0039</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>Nginx</td>
<td>11.522</td>
<td>11.539</td>
<td>0</td>
<td>0.14</td>
</tr>
<tr>
<td>Apache</td>
<td>18.250</td>
<td>18.286</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>Telnet</td>
<td>1.001</td>
<td>1.004</td>
<td>6</td>
<td>0.29</td>
</tr>
<tr>
<td>SCP</td>
<td>0.1656</td>
<td>0.1665</td>
<td>28</td>
<td>0.54</td>
</tr>
</tbody>
</table>
## Real-world Exploits

<table>
<thead>
<tr>
<th>Exploit Pattern</th>
<th>Vul. Program</th>
<th>Exploit Name (EDB)</th>
<th>Setuid Syscall Exploited</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PoLPerr CFI NCI</td>
</tr>
<tr>
<td>Modify Setuid Parameters</td>
<td>Sudo</td>
<td>(N/A)</td>
<td>setuid</td>
<td>√ X √</td>
</tr>
<tr>
<td></td>
<td>Wu_ftpd</td>
<td>(N/A)</td>
<td>setuid</td>
<td>√ X √</td>
</tr>
<tr>
<td>Run setuid call to create a root shell</td>
<td>Overlayfs</td>
<td>37292-2015</td>
<td>setresuid, setresgid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>39230-2016</td>
<td>setresuid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>Glibc</td>
<td>209-2000</td>
<td>setuid, setgid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>Mkdir</td>
<td>20554-2001</td>
<td>setuid, setgid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>KApplication</td>
<td>19981-2000</td>
<td>setuid, setregid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>Suid_dumpable</td>
<td>2006-2006</td>
<td>setuid, setgid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>Execve/ptrace</td>
<td>20720-2001</td>
<td>setuid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>Splitvt</td>
<td>20013-2000</td>
<td>setuid</td>
<td>√ √ X</td>
</tr>
<tr>
<td></td>
<td>OpernMovieeditor</td>
<td>2338-2006</td>
<td>setuid, setgid</td>
<td>√ √ X</td>
</tr>
</tbody>
</table>

CFI: control flow integrity
NCI: non-control data integrity
Case Study: Sudo

```c
struct user {
    uid_t uid;
    ...
}

struct user ud;
ud.uid = getuid();

//in sudo_debug()
vfprintf (...);

//in sudo_askpass()
setuid (ud.uid);
```

Table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>setuid</td>
<td>(Profile) uid = getuid()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(exploit) 0</td>
</tr>
</tbody>
</table>

Call Stack:

<table>
<thead>
<tr>
<th>#</th>
<th>Offset</th>
<th>File</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>0x32 + 0xb75f7b4</td>
<td>../libc.so.6</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>0x8053080</td>
<td>../bin/sudo</td>
<td>sudo_askpass</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x8049dd1</td>
<td>../bin/sudo</td>
<td></td>
</tr>
</tbody>
</table>

Sudo code example
Outline

- Motivation
- Background
- PoLPerc
- Evaluation
- Conclusion
Conclusion

- Extracts only the required contexts of setuid calls
- Prevents setuid exploits with negligible overhead
- Enforces PoLP using a combination of different process contexts
THANK YOU!

Q&A