Protecting the IoT Against Data Leaks through Intra-Process Access Control
Marcela S. Melara, David Liu, Edward W. Felten, Michael J. Freedman
Princeton University

Background
Third-party libraries are extremely prevalent in the IoT:
- 99% of studied IoT applications import at least one third-party library
- Out of top 50 studied Python libraries:
  
<table>
<thead>
<tr>
<th>Library Implementation Language</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written in Python</td>
<td>18.0%</td>
</tr>
<tr>
<td>Run external binaries</td>
<td>40.0%</td>
</tr>
<tr>
<td>Have native dependencies</td>
<td>76.0%</td>
</tr>
<tr>
<td>Use ctypes FFI</td>
<td>40.0%</td>
</tr>
</tbody>
</table>

Table 1: Library implementation language.

Problem: Developers do not/cannot inspect and vet imported third-party code
⇒ security & privacy vulnerabilities go undetected

Motivating Threats:
- Our experiments: successfully modified function pointers, local variables in the Python runtime call stack from (native) libraries.
- Reported vulnerabilities in Python libraries 2012-2017:

<table>
<thead>
<tr>
<th>Vulnerability Class</th>
<th># of Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Leak</td>
<td>15</td>
</tr>
<tr>
<td>Arbitrary Code Execution</td>
<td>12</td>
</tr>
<tr>
<td>Symlink Attack</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3: Top 3 Python library vulnerabilities out of 48 analyzed CVE database reports. We identified 35 distinct vulnerable Python libraries.

Our goal: Prevent malicious third-party libraries from accessing sensors or data other than those intended by the developer.

Prior Work: Why can’t we apply isolation solutions for Android or IoT?
- Isolate libs into separate apps, apply Android permissions (e.g. [1-3])
⇒ Linux-based IoT doesn’t have built-in mandatory access control
- Android native library isolation:
⇒ SFI [4] requires access to library source code
⇒ hardware fault isolation [5] is platform-dependent

Approach
Provide an access control framework at the granularity of libraries (intra-process) that dynamically adjust privilege based on the execution context.

Stack Inspection:
- Reference monitor intercepts relevant syscalls and pauses the corresponding app thread
- Stack Tracer thread in runtime collects paused thread’s call stack info
⇒ pass to reference monitor through a secure comm. channel

Access Control Semantics:
- Ref. monitor makes access decisions on a per-thread basis
- Developer specifies initial permissions of top-level libraries
- Lower-level modules “inherit” the permissions of the closest known caller module at runtime
- System calls are allowed/denied based on the resulting permissions of all libraries in the call stack (i.e. the provenance of the syscall)

Native Code Isolation Mechanism:
- Isolate native third-party libraries into their own memory address space
⇒ Prevent native libraries from manipulating a thread’s call stack info
- Runtime detects calls to third-party native libraries and runs each in a separate process

References: