Torches on Pitchfork: Multi-feature Evaluation of a Security-oriented Programming Toolchain

Nik Sultana

*Illinois Institute of Technology*

Learning from Authoritative Security Experiment Results (LASER) 2022
System release

- http://pitchfork.cs.iit.edu

- Everything is released **except for exploit code**:
  - libcompart
  - Pitchfork
  - examples of applying libcompart & Pitchfork
  - FreeBSD ports analysis
- Apache 2.0 license
Motivation: Software Security

Increased trend in # of CVEs:
Good: we know about problems.
Bad: there are more problems.

Ack: Graph generated using dataset from https://www.cve-search.org/dataset/
What is Privilege Separation? (privsep)

- Compartmentalize code + data. Early application: SSH server.
- Monolithic application: Concurrent set of cooperating programs.
- Monolithic application: often common privileges throughout.
- Distributed system: granularity of privilege allocation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privileges</td>
<td></td>
</tr>
</tbody>
</table>
What is Privilege Separation? (privsep)

- **Compartmentalize code + data.** Early application: servers: SMTP, SSH.

- Monolithic application → Concurrent set of cooperating programs.
  
  - Monolithic application: often common privileges throughout.

- **Distributed system:** granularity of privilege allocation.
• Main benefit: **vulnerability containment.** Best case: if a vulnerability is exploitable, then fewer privileges can be abused.
Privsep

• **Implementing** privsep: usually a lot of work. Changing software without introducing bugs.
• **Implementing** privsep: usually a lot of work. Changing software without introducing bugs.

• There are many **decision** to take (and retake later) wrt what+how to separate (see yellow bubbles above).
Privsep

Heuristics:
- Components needing specific access.
- Dependencies incl. libraries.
- Cross-domain interfaces (e.g., parts of network, filesystem)
Privsep

- **Drawbacks** include:
  Inertia wrt splitting software, introduction of new failure modes (hello distributed systems), performance overhead, inertia wrt maintainability and portability (e.g., if use hardware enforcement).

Some parts are buggy?  
*Fewer privileges = fewer problems.*

Equally trusted?  
*Need further splits?*

Too high? 
*Can lower further? Need further splits?*
Research Goal

(Widely-applicable) tool support for privsep
(Longstanding) Research Goal

Widely-applicable tool support for privsep

Foundations:
- compartment model
- tool infrastructure
- software-level

(This paper)
Research Goal

Widely-applicable tool support for privsep

Foundations:
- compartment model
- tool infrastructure
- software-level

Artefacts:
+ tooling
+ several examples
+ supporting scripts & documentation

(This paper)
Compartment Model

- **Organization:**
  - **Domain:** Shared memory/handles/resources across compartments
  - **Compartments:** Sharing across segments.
  - **Segments:** code + data.

- **Special compartments:** Main, Monitor — always in domain0.

- Implementation: pluggable API for communication, configuration and enforcement.

- Generalization and Tooling
  vs Flexibility:
  General but restrictive
Example of what’s enabled

- Organization:
  - **Domain**: one on each machine
  - **Compartments**: one in each domain.
  - **Segments**: 2 in Classified, 1 in Main.

- Communication channel over TCP.

- Machine and network-level policy+enforcement.
The **system** has two components based on a **model**: 

- Pitchfork 1 2
- libcompart 3
Pitchfork

1. Source code
2. Annotated source code
3. Annotation analysis
4. Transformed source code
5. Runtime API
6. Compilation
7. Debugging
if(console_type == BEEP_TYPE_CONSOLE) {
  pitchfork_start("Privileged");
  if(ioctl(console_fd, KIOCSOUND, period) < 0) {
    putchar('\a'); /* Output the only beep we can, in an 
                effort to fall back on usefulness */
    perror("ioctl");
  }
  pitchfork_end("Privileged");
} else {
  /* BEEP_TYPE_EVDEV */
  struct input_event e;
  e.type = EV_SND;
  e.code = SND_TONE;
  e.value = freq;
  pitchfork_start("Privileged");
  if(write(console_fd, &e, sizeof(struct input_event)) < 
   0) {
    putchar('\a'); /* See above */
    perror("write");
  }
  pitchfork_end("Privileged");
}
```c
#include "netpbm_interface.h"

int
main(int argc, const char * argv[]) {
  +compart_init(NO_COMPARTS, comparts, default_config);
  +convertTIFF_ext = compart_register_fn("libtiff", &
       ext_convertTIFF);
  +parseCommandLine_ext = compart_register_fn("cmdparse"
       , &ext_parseCommandLine);
  +compart_start("netpbm");

  struct CmdlineInfo cmdline;
  TIFF * tiffP;
  FILE * alphaFile;
  FILE * imageoutFile;

  pm_proginit(&argc, argv);
  -parseCommandLine(argc, argv, &cmdline);
  +struct extension_data arg;
  +args_to_data_CommandLine(&arg, argc, argv);
  +arg = compart_call_fn(parseCommandLine_ext, arg);
  +args_from_data(&arg, &cmdline);
  -tiffP = newTiffImageObject(cmdline.inputFilename);
  -if (cmdline.alphaStdout)
    ...
  -TIFFClose(tiffP);
  +args_to_data(&arg, &cmdline);
  +arg = compart_call_fn(convertTIFF_ext, arg);
  pm_strfree(cmdline.inputFilename);
```
Torches on Pitchfork: Multi-feature Evaluation of a Security-oriented Programming Toolchain

Nik Sultana
Illinois Institute of Technology

Security Experiment Results
Food for thought

- How to identify+scope the security problem?
- How to show the problem begin solved? Can this scale with size, complexity and variety of problem instances? (programs)
- How to understand newly-introduced problems?
Food for thought

- Evaluation goals
- Evaluation process

Challenges:
- Skills + Time needed to reproduce exploit. Scaling the eval.
- Generalizability of analysis + transformation.
- User study.
- Reasoning about incomplete info — likelihood of introducing bugs.

Plans for post-workshop: above + more software analysis
Evaluation

(Many more details in the paper)

• Applicability
  • Examples
  • Maintainability
  • Convenience

• Security
  • Known CVEs
  • Heuristics

• Overhead: running time, memory, binary size.
Evaluation

- Applicability
- Examples
- Maintainability
- Convenience
- Security
  - Known CVEs
- Heuristics
- Overhead: running time, memory, binary size.

<table>
<thead>
<tr>
<th>Software</th>
<th>CVE-***</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>beep</td>
<td>2018-0492</td>
<td>Race condition</td>
</tr>
<tr>
<td>PuTTY</td>
<td>2016-2563</td>
<td>Stack buffer overflow</td>
</tr>
<tr>
<td>wget</td>
<td>2016-4971</td>
<td>Arbitrary file writing</td>
</tr>
<tr>
<td>wget</td>
<td>2017-13089</td>
<td>Stack buffer overflow</td>
</tr>
</tbody>
</table>
Evaluation

- Applicability
  - Examples
- Maintainability
- Convenience
- Security
  - Known CVEs
  - Heuristics
- Overhead: running time, memory, binary size.

### Software Plat. Separation Goal

<table>
<thead>
<tr>
<th>Tool</th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cURL</td>
<td>L</td>
<td>Command invocation, parsing, file transfer.</td>
</tr>
<tr>
<td>Evince</td>
<td>L</td>
<td>libspectre dependency—see §2.</td>
</tr>
<tr>
<td>git</td>
<td>L</td>
<td>Historical vulnerability [13].</td>
</tr>
<tr>
<td>ioquake3</td>
<td>m</td>
<td>Applying server updates.</td>
</tr>
<tr>
<td>tifftopnm</td>
<td>L</td>
<td>Separating parsers—see §C.</td>
</tr>
<tr>
<td>nginx</td>
<td>L</td>
<td>HTTP request parsing</td>
</tr>
<tr>
<td>redis</td>
<td>L</td>
<td>Isolating low-use commands.</td>
</tr>
<tr>
<td>tcpdump</td>
<td>F</td>
<td>Leveraging Capsicum [68].</td>
</tr>
<tr>
<td>uniq</td>
<td>L</td>
<td>Network-facing code—see §2.</td>
</tr>
<tr>
<td>Vitetris</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

- Applicability
  - Examples
- Maintainability
- Convenience
- Security
  - Known CVEs
- Heuristics
- Overhead: running time, memory, binary size.

\[
\text{SAR} = \frac{\text{#LOC Synthesized}}{\text{#Lines of Annotation}}
\]

<table>
<thead>
<tr>
<th>Soft.</th>
<th>#LOC</th>
<th>#Annot.</th>
<th>#LOC Synthesized Compart.</th>
<th>#LOC Synthesized De/mash.</th>
<th>SAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>beep</td>
<td>372</td>
<td>9</td>
<td>133</td>
<td>245</td>
<td>42</td>
</tr>
<tr>
<td>PuTTY</td>
<td>123K</td>
<td>6</td>
<td>52</td>
<td>29</td>
<td>13.5</td>
</tr>
<tr>
<td>wget$^6$</td>
<td>62.6K</td>
<td>3</td>
<td>65</td>
<td>168</td>
<td>77.7</td>
</tr>
<tr>
<td>wget$^7$</td>
<td>62.8K</td>
<td>8</td>
<td>57</td>
<td>38</td>
<td>11.9</td>
</tr>
</tbody>
</table>
Evaluation

- Applicability
  - Examples
- Maintainability
- Convenience
- Security
  - Known CVEs
- Heuristics
- Overhead: running time, memory, binary size.
Evaluation

- Applicability
- Examples
- Maintainability
- Convenience
- Security
  - Known CVEs
  - Heuristics
- Overhead: running time, memory, binary size.

Memory stability wrt quantity & size of data exchange.
Food for thought

• How to identify+scope the security problem?
  Existing literature on privsep.
  Non-specialized, commodity hardware & kernel. “Realism”.
  CVEs in third-party, widely-used programs. (CVEs that allow code injection or exfiltration).
  Written in C, “warts and all”. Unmodified compiler toolchains.

Security Experiment Results
Food for thought

- How to show the problem being solved?
  - Reproduce CVEs — not all attempts were productive for this research (discussed in an appendix).
  - Classify CVEs?

- How to show the problem being solved?
  - Reproduce CVEs — not all attempts were productive for this research (discussed in an appendix).
  - Classify CVEs?

  Trial and error. Starting with simple/short programs. Recreated problem from literature.

  Work up to more types of software. Generality analysis.

Thanks to community

Different experiment methodologies: security, performance, applicability.
Food for thought

- How to understand newly-introduced problems?
  Very hard to prove a negative.
  
  Does this ultimately require verification?
  
  Practical under approximation: tests still run, usage still works (so no newly-introduced problems wrt those instances), but no airtight evidence that no problems have been introduced.
  
  Other practical issues: build scripts, portability and complexity of the resulting system.
Things that didn’t work

• Some partitionings: e.g.,
  
  • **CVE-2015-6565** (openssh) involved a bad permissioning decision. In general, can partitioning mitigate against bad configuration decisions? Doesn’t partitioning add another layer of configuration?
  
  • **CVE-2018-10933** (libssh) involved flawed state machine.
  
  • Eval environment diversity: leads to complexity in the paper. Better to have a single environment for all use cases?
  
  • Test setup inertia wrt some use-cases (library versioning) — this would have been easy to overcome, but at the cost of a little more engineering and fiddling.
  
  • Conceptual/algebraic approach to describe partitions, too simplistic.
Food for thought

- Evaluation goals
- Evaluation process
- Challenges:
  - Skills + Time needed to reproduce exploit. Scaling the eval.
  - Generalizability of analysis + transformation.
  - User study. **How to quantify benefit of using a specific defense?**
  - Reasoning about incomplete info — likelihood of introducing bugs.

Plans for post-workshop: above + more software analysis
Torches on Pitchfork:
Multi-feature Evaluation of a Security-oriented Programming Toolchain

Nik Sultana
*Illinois Institute of Technology*

Learning from Authoritative Security Experiment Results (LASER) 2022