ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications

Dimitrios Tychalas
NYU Tandon School of Engineering

Hadjer Benkraouda
New York University Abu Dhabi

Michail Maniatakos
New York University Abu Dhabi
Introduction

ICS Landscape

- Industrial Control Systems (ICS) evolving
  - Industry 4.0, IIoT

- IT/OT convergence

- Most research focused on network/system/operational security
  - What about have software bugs?
Introduction

Control Application Intro

- Control logic in PLCs compiled into a software application
  - Assembly instructions
  - Unique format
- Application written in dedicated languages
  - Third-party libraries
- Application hosted in a runtime process
  - Codesys platform a.k.a. a soft PLC
- Runtime is hosted in a Linux-based OS
  - Single binary, multi-threaded
Research Questions/Threat Model

○ Can control logic applications have software bugs?
  ○ Fuzzing!

○ Can we fuzz a control applications?
  ○ Format is not readily accepted by typical fuzzers

○ Can fuzzing uncover exploitable bugs?
  ○ Reverse shell on a PLC sounds exciting!
What is fuzzing?
- Input control
- Execution control
- Execution feedback
Fuzzing control applications

- Execution control
- Scan cycle task
- Automated cyclic execution!
Fuzzing control applications

Input Control

How to control input?
- Trace input delivery throughout the stack
- Sensor -> PLC I/O -> GPIO -> KBUS -> PLC binary
- Isolate most controllable input delivery stage
- Force new values through system structures

How to synchronize input delivery?
- KBUS_CYCLE_TASK
- Custom system calls (ioctl’s)
Fuzzing control applications

Execution feedback

- Trace the PID family
- Find a malleable predecessor
- Trace control application exit through predecessor
Fuzzing control applications

Instrumentation

- Can we get feedback on the execution of the control application?
  - Instrumentation!

- What to do?
  - No source code
  - Proprietary compiler

- Look for opportunities!
  - NOPs
  - Replace NOPs with controllable code
    - i.e. store the current program counter (PC)
  - Approximate coverage with the PC information
Fuzzing control applications

Experiments

- Fuzz binaries found in the wild
  - Problem: PLC binaries available online are too simple
  - Try something beefier

- Fuzzing an open-source desalination process
  - Bingo!
  - Binary crashed
  - DoS possible
Fuzzing control applications
Experiments (contd.)

- Dig a little bit deeper
  - Can we make more complex PLC binaries
  - *(yes we can!)*
- Introduce interesting stuff
  - Complex structures (arrays, strings, etc.)
  - Notorious functions *(memcpy, memmove etc.)*
  - Fuzz away
- Crashes, lots of crashes
  - Buffer overflows
  - *(Way) Out of bounds read/writes*
- Possibility for complex exploits
  - Reverse shell through TCP
  - Rootkit insertion

<table>
<thead>
<tr>
<th>Control Application</th>
<th>Execution Speed (inputs/sec)</th>
<th>First Crash (time mm:ss)</th>
<th>First crash (inputs)</th>
<th>Crashes (1hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bf_memcpy_1</td>
<td>70.88</td>
<td>3:54</td>
<td>15270</td>
<td>32</td>
</tr>
<tr>
<td>bf_memcpy_6</td>
<td>64.2</td>
<td>3:08</td>
<td>12172</td>
<td>21</td>
</tr>
<tr>
<td>bf_memcpy_8</td>
<td>66.06</td>
<td>4:39</td>
<td>18216</td>
<td>17</td>
</tr>
<tr>
<td>bf_memcpy_12</td>
<td>62.11</td>
<td>7:06</td>
<td>26645</td>
<td>9</td>
</tr>
<tr>
<td>bf_mset_1</td>
<td>64.56</td>
<td>3:28</td>
<td>13441</td>
<td>21</td>
</tr>
<tr>
<td>bf_mset_3</td>
<td>62.68</td>
<td>2:54</td>
<td>10906</td>
<td>24</td>
</tr>
<tr>
<td>bf_mset_5</td>
<td>68.8</td>
<td>4:14</td>
<td>17554</td>
<td>16</td>
</tr>
<tr>
<td>bf_mset_9</td>
<td>69.76</td>
<td>10:23</td>
<td>43530</td>
<td>7</td>
</tr>
<tr>
<td>bf_mmove_1</td>
<td>64.63</td>
<td>2:56</td>
<td>11245</td>
<td>28</td>
</tr>
<tr>
<td>bf_mmove_4</td>
<td>63.1</td>
<td>2:39</td>
<td>10070</td>
<td>24</td>
</tr>
<tr>
<td>bf_mmove_7</td>
<td>66.31</td>
<td>3:49</td>
<td>15317</td>
<td>15</td>
</tr>
<tr>
<td>bf_mmove_12</td>
<td>64.53</td>
<td>13:03</td>
<td>50643</td>
<td>6</td>
</tr>
<tr>
<td>oob_1_arr_1</td>
<td>71.86</td>
<td>0:55</td>
<td>3880</td>
<td>39</td>
</tr>
<tr>
<td>oob_1_arr_6</td>
<td>77.03</td>
<td>1:43</td>
<td>8085</td>
<td>28</td>
</tr>
<tr>
<td>oob_1_arr_9</td>
<td>69.78</td>
<td>1:45</td>
<td>7326</td>
<td>27</td>
</tr>
<tr>
<td>oob_1_arr_13</td>
<td>75.2</td>
<td>3:27</td>
<td>27241</td>
<td>19</td>
</tr>
<tr>
<td>oob_2_arr_1</td>
<td>73.53</td>
<td>1:57</td>
<td>8558</td>
<td>35</td>
</tr>
<tr>
<td>oob_2_arr_5</td>
<td>71.1</td>
<td>2:45</td>
<td>22759</td>
<td>27</td>
</tr>
<tr>
<td>oob_2_arr_8</td>
<td>69.8</td>
<td>3:08</td>
<td>13366</td>
<td>22</td>
</tr>
<tr>
<td>oob_2_arr_13</td>
<td>70.95</td>
<td>3:12</td>
<td>13401</td>
<td>19</td>
</tr>
<tr>
<td>divby0_1</td>
<td>73.68</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>
Fuzzing the runtime

- What about the runtime itself?
  - Control applications can write way outside its memory space
  - Fuzzing crashed the whole runtime

- Must assess the runtime itself
  - Complex
  - Super large
  - Built-in anti-debugging
Fuzzing the runtime

- Divide and conquer
- Target dynamically loaded libraries
- Reverse-engineer context
- Instantiate in C harness
- Fuzz the produced ELF!
- Fuzzers galore

```c
#include <LIBDBUSCOMMON_H>
int main(int argc, char **argv){
    kbus_ksock_t ksock = fopen("/dev/kbus0", "r");
    kbus_ksock_write_data(ksock, &argv, 32);
}
```
Fuzzing the runtime

Experiments

- Limited function eligibility
  - ~250 out of more than 5000
- Focus on “external facing” functions
- Network protocols
- Parsers
- Found crashes
  - Recognized a couple of known CVEs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Crashes (1hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KbusRegisterRequestWrite</td>
<td>Kbus write function</td>
<td>2</td>
</tr>
<tr>
<td>KbusRegisterRequestRead</td>
<td>Kbus read function</td>
<td>1</td>
</tr>
<tr>
<td>kbus_ksock_write_data</td>
<td>Kbus write function</td>
<td>4</td>
</tr>
<tr>
<td>kbus_ksock_read_data</td>
<td>read function</td>
<td>7</td>
</tr>
<tr>
<td>XMLParse</td>
<td>XML Parsing Function</td>
<td>8</td>
</tr>
<tr>
<td>SysSockRecv</td>
<td>TCP receive data</td>
<td>6</td>
</tr>
<tr>
<td>CMAddCoponentKbus</td>
<td>Kbus instantiation</td>
<td>4</td>
</tr>
<tr>
<td>pthread_create</td>
<td>Creates runtime thread</td>
<td>8</td>
</tr>
<tr>
<td>pthread_rwlock_unlock</td>
<td>Updates thread privileges</td>
<td>2</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Joins PLC task threads</td>
<td>1</td>
</tr>
<tr>
<td>pthread_setschedparam</td>
<td>Sets scheduler thread policy</td>
<td>1</td>
</tr>
<tr>
<td>GetLoginName</td>
<td>Receives input login name</td>
<td>7</td>
</tr>
<tr>
<td>SysLibStrepy</td>
<td>String copy custom function</td>
<td>2</td>
</tr>
<tr>
<td>SysLibStrcmp</td>
<td>String compare custom function</td>
<td>5</td>
</tr>
<tr>
<td>SysComWrite</td>
<td>System communication output</td>
<td>7</td>
</tr>
<tr>
<td>SysComRead</td>
<td>System communication input</td>
<td>2</td>
</tr>
<tr>
<td>GetHookName</td>
<td>Get name of hooked function</td>
<td>6</td>
</tr>
<tr>
<td>CopyRtsMetrics</td>
<td>Copies PLC data</td>
<td>8</td>
</tr>
<tr>
<td>getsnam</td>
<td>Returns info from shadow file</td>
<td>5</td>
</tr>
</tbody>
</table>
Conclusion

- PLC binaries are inherently robust
  - High-level nature of the PLC programming languages
  - Very well-defined problems they are addressing

- However, the control binaries get more complex
  - Mimicking typical software evolution
  - Exploitable vulnerabilities can be introduced

- PLC runtimes suffer like high-level software
  - E.g. typical C/C++ developed software
  - This can compromise the whole industrial control system computation stack

- Fuzzing is great!
  - Even for Industrial Control Systems
  - Despite the presence of heavy I/O and scan cycles
Thank you

Dimitris Tychalas
dimitris.tychalas@nyu.edu

Hadjer Benkraouda
hb992@nyu.edu

Mihalis Maniatakos
mihalis.maniatakos@nyu.edu