



USENIX'23 Artifact Appendix: ICSPatch: Automated Vulnerability Localization and Non-Intrusive Hotpatching in Industrial Control Systems using Data Dependence Graphs

Prashant Hari Narayan Rajput¹, Constantine Doumanidis², Michail Maniatakos²

¹NYU Tandon School of Engineering, Brooklyn, NY, USA

²New York University Abu Dhabi, Abu Dhabi, UAE

A Artifact Appendix

A.1 Abstract

This artifact contains the source code for ICSPatch, a hotpatching tool for control application binaries on Codesys runtime-compatible Programmable Logic Controllers (PLCs). It can detect and patch out-of-bounds write/read, improper input sanitization, and os command injection vulnerabilities in control applications. It can patch these vulnerabilities via an LKM-based (Loadable Kernel Module) patcher or through JTAG.

Evaluating ICSPatch on a live setup requires a Codesys Codesys runtime-compatible PLC with either a Linux OS or JTAG connection. To facilitate a more straightforward evaluation, we also allow hotpatching angr simulation instances loaded with vulnerable memory snapshots of control application binaries in case of missing physical devices. Furthermore, we package ICSPatch in a Docker container to minimize the initial setup steps, supporting multiple platforms. ICSPatch is tested on Wago PFC 100, PFC200 for Linux-5.10.21, and BeagleBone Black for Linux-4.19.82-ti-rt-r31.

A.2 Description & Requirements

A.2.1 How to access

All the documents and source code for ICSPatch is available on GitHub at <https://github.com/momalab/ICSPatch/tree/v1.0>.

[Commit: 40803636849d24ab6a50e1c166d7522c7a1ceb6e]

A.2.2 Hardware dependencies

ICSPatch requires a 32 bit ARM architecture PLC supporting Codesys-runtime. In addition, a readily accessible JTAG port is also required for patching the control applications by using JTAG. However, ICSPatch also supports LKM-based patching, removing the need for an accessible JTAG port.

A.2.3 Software dependencies

ICSPatch is packaged in a Docker container. To run ICSPatch, manually install Docker as explained on this <https://docs.docker.com/engine/install/ubuntu/>.

In case of missing hardware requirements, utilize the captured memory snapshots of control application binaries (included in the repository) and evaluate ICSPatch by hotpatching the angr simulation instance as explained <https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md>.

A.2.4 Benchmarks

We create a synthetic dataset of vulnerable control application binaries with their source code project files present at https://github.com/momalab/ICSPatch/tree/v1.0/experiments/iec_projects and the corresponding memory snapshots for the WAGO PFC 200 included in the repository at the location: <https://github.com/momalab/ICSPatch/tree/v1.0/main/src/bin/internal>. ICSPatch can utilize the control application memory snapshots in the evaluation mode.

A.3 Set-up

A.3.1 Installation

For installing and running ICSPatch on the Docker container, build from the Dockerfile provided in the repository. The steps are explained in the **Installation** section at <https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md>.

Run the following commands to build and run the docker container.

```
cd ICSPatch/main
sudo docker build --pull --rm -f "Dockerfile" -
  ↪ t icspatch:latest ". "
```

```
sudo docker images // List the images
sudo docker run -it icspatch:latest
```

To try ICSPatch on live PLCs, please build the LKM patcher for the target Linux Kernel.

A.3.2 Basic Test

To test the successful installation of ICSPatch, run the command `sudo docker run -it icspatch:latest`. If ICSPatch executes successfully, the following prompt will be displayed on stdout:

```
Select Vulnerability:
-----
0. improper_input
1. oob_write
2. oob_read
3. os_command
4. exit
Choice:
```

A.4 Evaluation Workflow

Artifacts for ICSPatch have a detailed example for evaluating out-of-bound write in a control application binary for desalination plants, located at <https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md> in the section **ICSPatch for Evaluation**.

The overall steps are as follows:

1. Run the following command to execute ICSPatch in the Docker container and select the vulnerability for evaluation by entering the corresponding choice.

```
sudo docker run -it icspatch:latest
```

2. Next, select the appropriate mode of operation for ICSPatch. For evaluating ICSPatch without requiring a physical PLC, select 0, as shown:

```
Select Experiment:
-----
0. Evaluate
1. Live
Choice: 0
```

3. Next, select the test infrastructure when ICSPatch displays the following menu on stdout.

```
Select Infrastructure:
-----
0. aircraft_control
1. anaerobic_reactor
```

```
2. chemical_plant
3. desalination_plant
4. smart_grid
Choice:
```

4. Some infrastructure might have multiple vulnerable control application binary examples. Please select the target control application binary for evaluation when a menu shows up similar to this:

```
Select Test Sample:
-----
0. bin/internal/chemical_plant/oob_write/
   ↳ code_1
1. bin/internal/chemical_plant/oob_write/
   ↳ code_2
Choice:
```

5. After this, ICSPatch starts loading captured memory snapshots of the selected control application binary with a legitimate input (used to detect crashes that only impact the control application stack). After which the stdout displays the message.

```
- Press Enter to continue to capture
  ↳ exploit input hexdump ...
```

6. Press **Enter** to continue loading control application memory snapshot with exploit input, which results in displaying an output as shown below:

```
*****
RULE: OUT_OF_BOUNDS_WRITE_RULE
MESSAGE: OUT-OF-BOUNDS WRITE VULNERABILITY
  ↳ DETECTED
*****
----- BLOCK DISASSEMBLY -----
Instruction # in block: 8
0xb6bbf8a0: stmhs r3!, {r1, ip}
0xb6bbf8a4: subshs r2, r2, #8
0xb6bbf8a8: stmhs r3!, {r1, ip}
0xb6bbf8ac: subshs r2, r2, #8
0xb6bbf8b0: stmhs r3!, {r1, ip}
0xb6bbf8b4: subshs r2, r2, #8
0xb6bbf8b8: stmhs r3!, {r1, ip}
0xb6bbf8bc: bhs #0xb6bbf89c
----- DEBUG INFO -----
* Instruction Address: 0xb6bbf8a0
* Exploit Memory Address: 0xb617ad5c
* Length: None
* Expression: 0x0
[*] Angr execution time of the control
  ↳ application: 5.889697313308716
* Found start node: 0x83f48d0 ...
```

```

- Localization start address list:
  ↪ [138365136] ...
-----0-----
[*] Starting exploit localization from
  ↪ address 0x83f48d0 ...
[*] Start address: 0xb6193fb4 End Address:
  ↪ 0xb6194018...
[*] Bounded by 0xb6193fb4 - 0xb6194018 ...
[*] Search successful for start node 0
  ↪ x83f48d0 ...
[*] Detected exploit location: 0xb6193ff0:
  ↪ str r6, [sp, #8]
[*] Detected exploit input: 0xb617aca0: ['0
  ↪ x2', '0x0', '0x200']
[*] Memory value at exploit location: 0
  ↪ xb617aca0: 0x00000200
-----0-----

[*] Time for localizing vulnerability:
  ↪ 0.012766838073730469
* Selected vulnerability location is 0
  ↪ xb6193ff0 ...
* Exploit memory location is 0xb617aca0 ...

- Press Enter to continue to patching ...

```

Here, **RULE** displays the name of the vulnerability identification rule triggered for the exploit input and the corresponding message in **MESSAGE**. It also detects the start node for DDG traversal for performing vulnerability localization. The start node in this example is detected as `0x83f48d0`. The traversal successfully detects the exploit instruction location at `0xb6193ff0` and the memory location for the input (to be validated by the patch) at `0xb617aca0`.

7. Press **Enter** to continue patching the vulnerability, which displays patch-related information such as the address table base address and the memory location for an empty location. Press **Y** when the prompt display:

```

[*] Saved patch information detected. Use
  ↪ it? (Y/N):

```

This directs **ICSPatch** to use saved path information rather than connecting to an active local patch server.

8. Finally, **ICSPatch** creates the patch, loads it in the `angr` simulation, and verifies it. Loading the patch in the `angr` simulation instance is similar to writing it into the live PLC with the **LKM** patcher. So, this can successfully test the patch created by **ICSPatch**, and the overall automated process.
9. Since the evaluation of **ICSPatch** does not require a connected PLC, once **Enter** is pressed on the prompt:

```

- Press Enter to continue to patching live
  ↪ PLC ...

```

ICSPatch exits after 10 seconds when failing to connect to a local patch server deployed on a live PLC.

Instructions on [GitHub](#) also elaborate on how to use **ICSPatch** with a live PLC.

A.5 Evaluation and Expected Results

While running the experiments, as explained in Subsection [A.4](#), **ICSPatch** displays the timings (in seconds) corresponding to every operation on the `stdout`. For instance,

```

[*] Time for localizing vulnerability:
  ↪ 0.012766838073730469

```

It should be noted that only the vulnerability localization time is representative of the live PLC scenario. All the other timings will change when tested with a live PLC. Furthermore, the **LKM** patcher captures the timing for the critical operation of redirecting execution flow by overwriting the `ldr` instruction, as explained in the paper.