Pivot

Automatic Blackbox Exploit Generation for Stack-Based Buffer Overflows

Myrto Paraforou
Department of Informatics and Telecommunications, UoA
19 March, 2026

I am Myrto!

1) I graduated from DIT in July, 2025

2) I work on the development of energy market platforms, mainly focusing on backend systems.

3) As a hobby, I enjoy learning about Linux environments, the Linux kernel, and container technologies

4) I consider myself an aspiring systems engineer in the making

Presentation Overview

→ Why do we need exploit automation?

→ A quick demo

→ Pivot’s system internals

Motivation

Binary exploitation is hard for human effort alone:

- Deep system knowledge needed
- Often no source code, must infer behavior via reverse engineering
- Modern mitigations must be bypassed
- Highly case-specific: No universal technique or workflow applies across targets

Motivation

Binary exploitation remains a major security concern:

- Unsafe coding practices and languages still dominate low-level software
- Too many crashes in public software, too little time to manually audit them
- **Automation** is needed to identify which crashes are truly exploitable

A common misconception

vulnerability

exploit

The misconception in the real world :

```
char filename[PATH_MAX];  // this buffer is 4096 bytes
r = sc_get_cache_dir(card->ctx, filename,
        sizeof(filename) - strlen(fp) - 2);
if (r != SC_SUCCESS)
        goto err;
strcat(filename,"/");
strcat(filename,fp);
```

Claude Opus 4.6 identified and classified this as a buffer overflow vulnerability in **OpenSC**, an open-source command-line utility.

The misconception in the real world :

```
        if (strlen(fp) != 64) { /* ASCII-HEX encoded SHA-256 */
                r = SC_ERROR_INVALID_DATA;
                goto err;
        }
        for (i = 0; i < 64; i++) {
                if (isxdigit((unsigned char)fp[i]) == 0) {
                        r = SC_ERROR_INVALID_DATA;
                        goto err;
                }
        }
```

Even though *fp* is not fixed, it is strictly constrained → benign crash → buffer overflow **non-exploitable**

https://red.anthropic.com/2026/zero-days/
https://github.com/OpenSC/OpenSC/blob/master/src/libopensc/card-piv.c#L5168
cybersecurity is about to get weird

The problem:

Many program crashes are **potential exploits** while others are just **benign crashes**.

Is there a way to identify exploitability?

The goal:

Pivot aims to roughly answer the following question:

*“Is this program exploitable?” **

*given a potential buffer overflow and an input that triggers it

Demo time

Exploit eligibility

Reproducing a crash with user input does **not** imply that a program is **exploitable**.

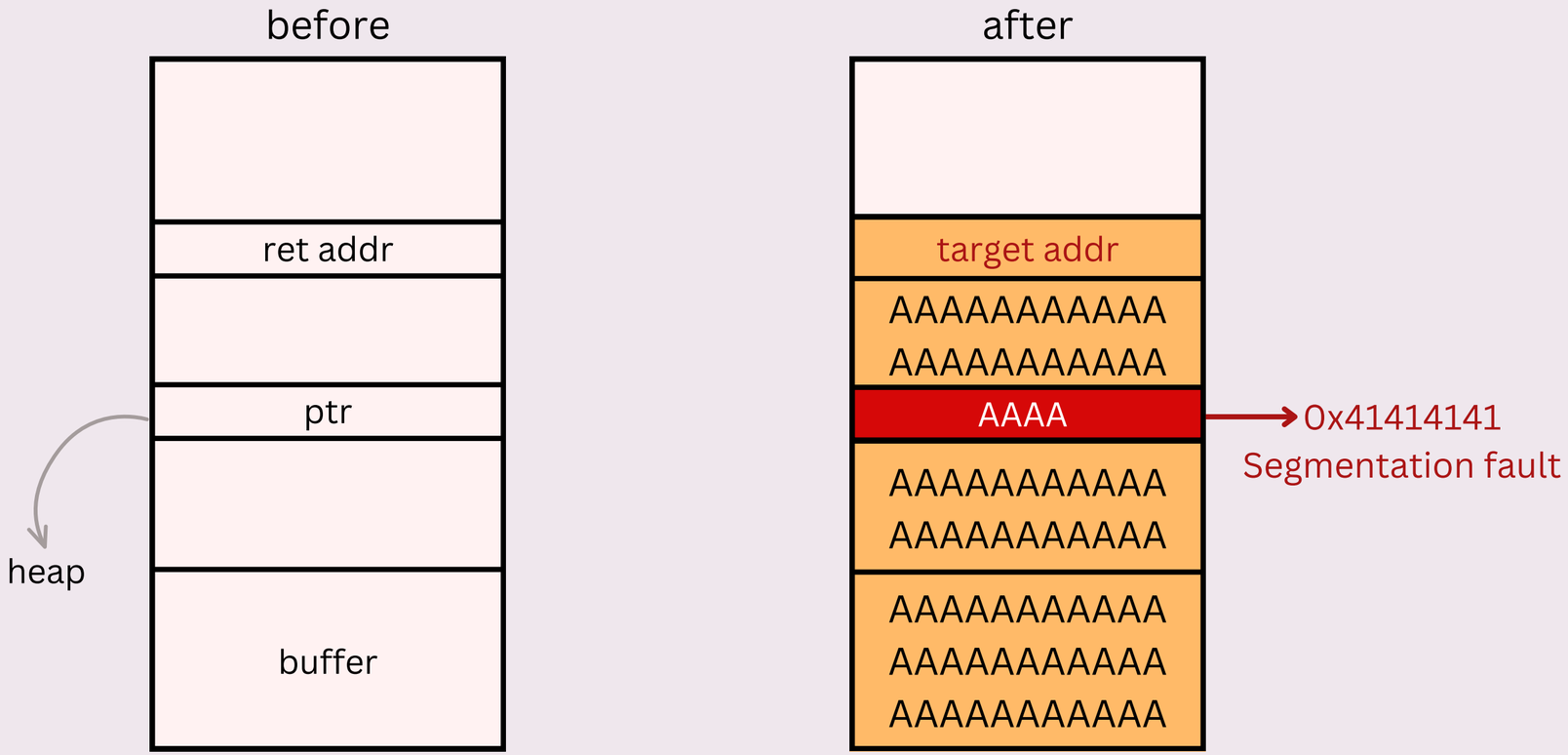
condition for Exploitability: tainting the EIP register

For a crash to be transformed into an exploit, it must satisfy the above condition and result in a fault on EIP; typically meaning that control flow reaches the function epilogue.

Exploit eligibility

...but a crash or simply a segfault can happen on any invalid memory access!

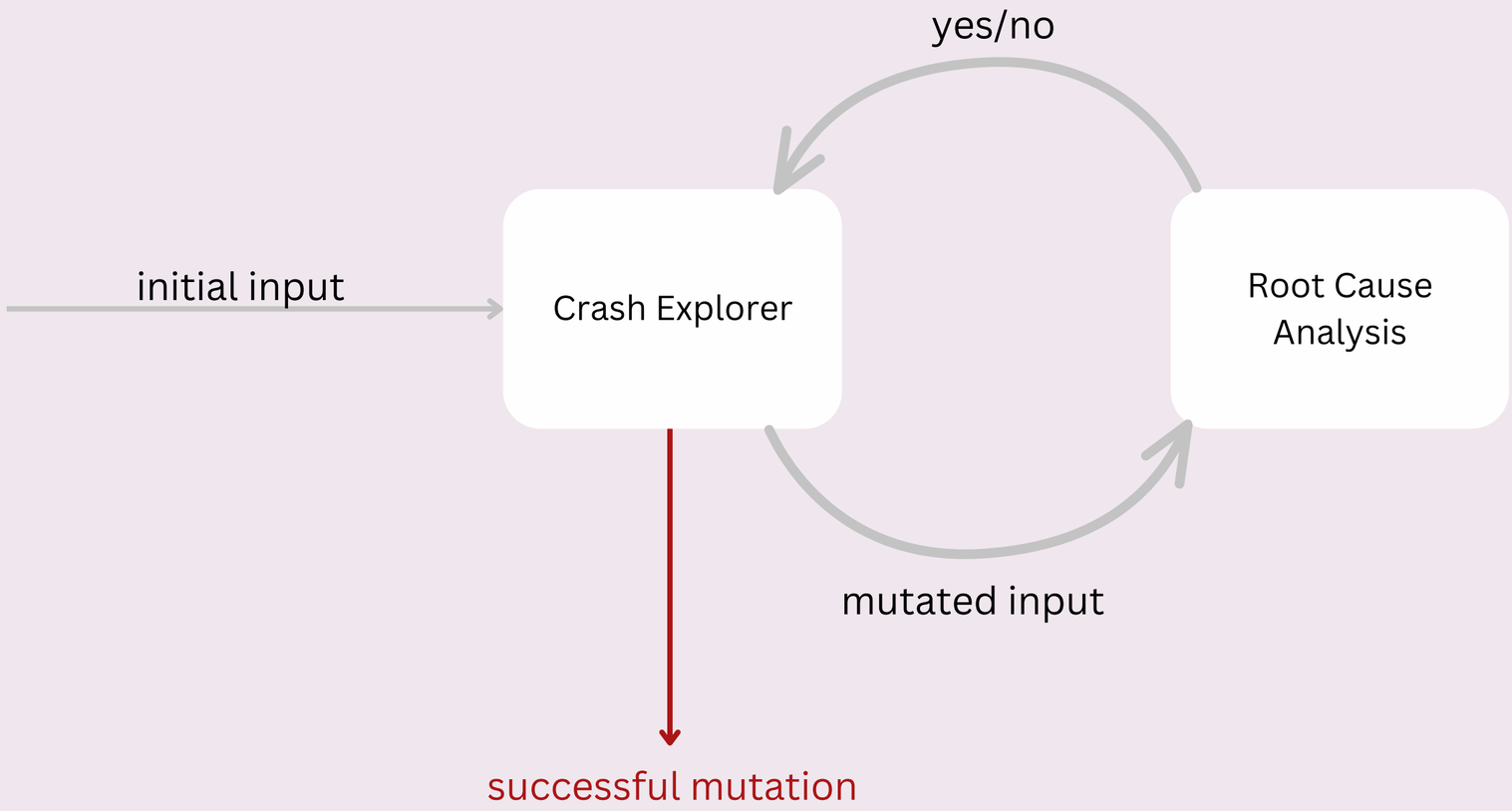
overwriting the stack ⇒ potentially overwriting local pointers



visual representation of a **non-exploit-eligible crash**

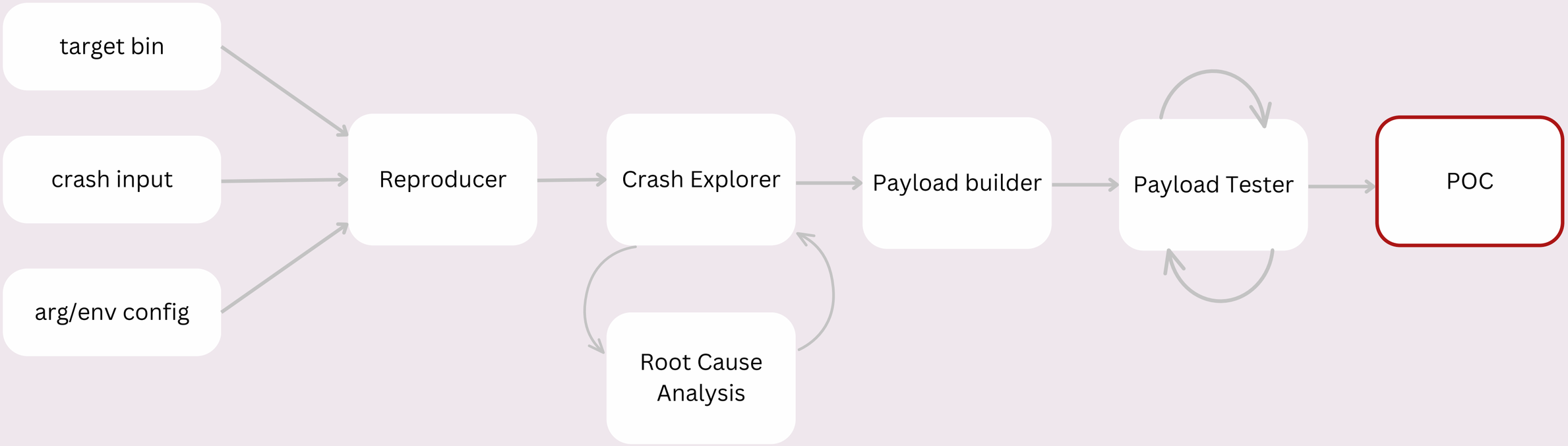
Pivot’s approach on Crash Exploration

abstractly...



*Root Cause Analysis determines whether the provided mutation produces an exploit-eligible crash

System Diagram



Evaluation

Pivot got tested under 2 axes:

a) exploration phase

- Number of mutations explored
- State tree depth
- Time taken

result:

→ up to 100 mutations were explored per target

→ average time under 3 sec

b) exploit phase

- Average number of attempts under ASLR
- Time to successful exploit

result:

→ up to 75 attempts per target

→ average time under 2 sec

Assumptions & Limitations

- Target has a **known** crashing input
- Vulnerability triggered via **stdin**, **argv**, or **env var**
- **NX** and **canaries disabled** for full functionality
- Exploitation relies on **shellcode injection** into the **stack**
- **No fuzzing** or bug discovery, input is user-provided
- Strictly **controlled** and defined **environment**

Thank you for your time!