Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems

Florent Moriconi, Axel Neergaard, Lucas Georget, Samuel Aubertin, Aurélien Francillon

florent.moriconi@eurecom.fr
Compilers are widely used

Most software is not written directly in machine code (assembler)
Usual software development process

Write code in C, C++, Python, Java, Go, you-name-it

Ask a compiler to generate machine code

Run machine code

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

```
L0: MOV R1, #a ; Address of a
    MOV R2, #b ; in R1, of b in R2
L1: LD R3, (R1) ; Import bits in R3
    CMP R3, #0 ; IF-condition
    BNE L3 ;
L2: MOV R4, #1 ; IF-branch
    JMP L4 ;
L3: MOV R4, #0 ; ELSE-branch
L4: ST (R2), R4 ;
    JMP L1 ;
```
Usual software development process

**Write** code in C, C++, Python, Java, Go, you-name-it

**Ask a** **compiler** to generate machine code

**Run** **machine code**

```cpp
#include <iostream>
int main() {
    std::cout << "Hello World!";
    return 0;
}
```

Clean source code

Malicious **compiler**

Malicious **machine code**
Self-hosted architecture

Compiler **source code**

Ask a **compiler** to generate machine code

Run **machine code**

Clean **source code**

**Malicious compiler**

**Malicious machine code**

*initial infection + self-hosted architecture = persistent malware*
Self-hosted architecture

Compiler **source code**

- Ask a **compiler** to generate machine code

- Run **machine code**

  - Clean **source code**
  - **Malicious** compiler
  - **Malicious** machine code

  \[ \text{initial infection} + \text{self-hosted architecture} = \text{persistent malware} \]

  “The moral is obvious. **You can’t trust code that you did not totally create yourself.** [...] No amount of source-level verification or scrutiny will protect you from using untrusted code.”, Ken Thompson [1]

---

Revisiting Thompson idea

Is the Thompson idea applicable to **Continuous Integration** systems?

*Is a vulnerability identified in 1984 still applicable in 2023?*
What is continuous integration?

- Github Actions
- Gitlab CI
- Travis CI
- Jenkins
- CircleCI
The use of custom images for CI

Modern CI are based on containers

Custom CI images are widely used

Why?
Avoid reinstalling your tools at each CI run
Consistent CI images between runs
Faster startup time
Self-hosted CI architecture is common

How do you build your custom CI images?

Using your CI!

This is a self-hosted architecture
Not all software in CI image are self-hosted

Not self-hosted
(i.e., X will not be used to build X)
🚫 Code linter
🚫 Code scanner
🚫 Docker daemon (DIND, host daemon)

Self-hosted
✅ Docker client
✅ Shell
Initial infection

- Malicious commit\(^1\)
  - history rewrite

- Compromise CI container\(^2\)
  - Dependency confusion\(^3\)

- Image registry compromise

Initial infection is required **only once**!

---

\(^1\) Q. Wu and K. Lu, “On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits”, 2021


\(^3\) Alex Birsan, “Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies”, 2021, Medium
Infecting a Continuous Integration system

Source code

Cl Build image source code
+ self-injection
+ attack payload

Docker container

Cl container

Docker image

Cl image

Initial infection

Self-reproduction & payload

Cl Build image source code

Cl container

Cl image

Target software source code

Target software
Malicious docker client can manipulate Dockerfile!

But also **exfiltrating secrets** in environment variables, **scanning internal network**, etc.
Proof of Concept

- Based on Gitlab CI
- Self-injecting on CI update
- Add authentication bypass backdoor to a Python API

```python
ALLOWED_TOKENS = ["ltlz9b0vC19hIB103HWwHktK9"]

@app.get("/")
async def root(token: str = Query(..., description="Token to access the resource")):  
    # We know this code is vulnerable to timing attacks, but this is out of scope here :)
    if token in ALLOWED_TOKENS:   
        return Response(status_code=200, content="Access allowed :\n")  
    else:  
        return Response(status_code=403, content="Access denied!\n")
```
Proof of Concept

```python
ALLOWED_TOKENS = ["ltlz9b0vC19hIB103HwWkT9"]

@app.get("/")
async def root(token: str = Query(..., description="Token to access the resource")):
    # We know this code is vulnerable to timing attacks, but this is out of scope here
    if token in ALLOWED_TOKENS:
        return Response(status_code=200, content="Access allowed :)\n")
    else:
        return Response(status_code=403, content="Access denied!\n")
```

SCM

CI

Production

$ curl http://127.0.0.1:8080?token=backdoortoken
Access allowed :)
Conclusion

Thompson’s idea can be applied to CI systems

Your CI system can be **malicious** even if the **source code is clean** of malicious code

**Self-reproduction** allow long-term compromise

**Initial infection** is feasible in practice: malicious commit, dependency confusion, registry compromise, etc.
Conclusion

Thompson’s idea can be applied to CI systems

Your CI system can be malicious even if the source code is clean of malicious code

Self-reproduction allow long-term compromise

Initial infection is feasible in practice: malicious commit, dependency confusion, registry compromise, etc.

Any question?