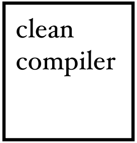
Automata and Formal Languages (9)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)
Slides:	KEATS (also home work is there)

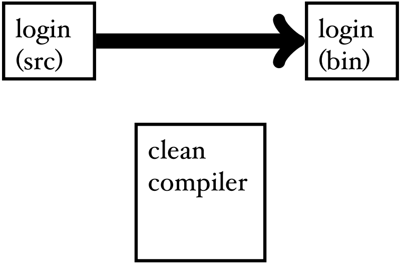
**Using a compiler,
how can you mount the
perfect attack against a system?**

What is a perfect attack?

1. you can potentially completely take over a target system
2. your attack is (nearly) undetectable
3. the victim has (almost) no chance to recover

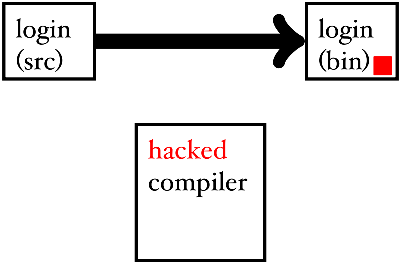


clean compiler


login
(src)

login
(bin)

clean
compiler

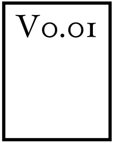


login (src)

login (bin)

hacked compiler

my compiler (src)

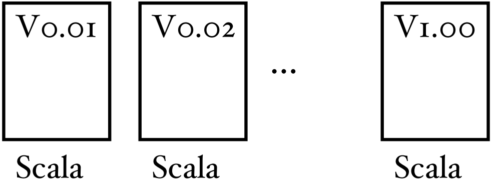


V0.01

Scala

host language

my compiler (src)

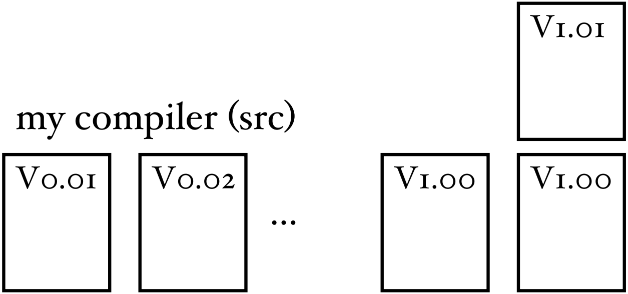


V0.01 V0.02 ... V1.00

Scala Scala Scala

host language

my compiler (src)



V0.01

V0.02

...

V1.00

V1.00

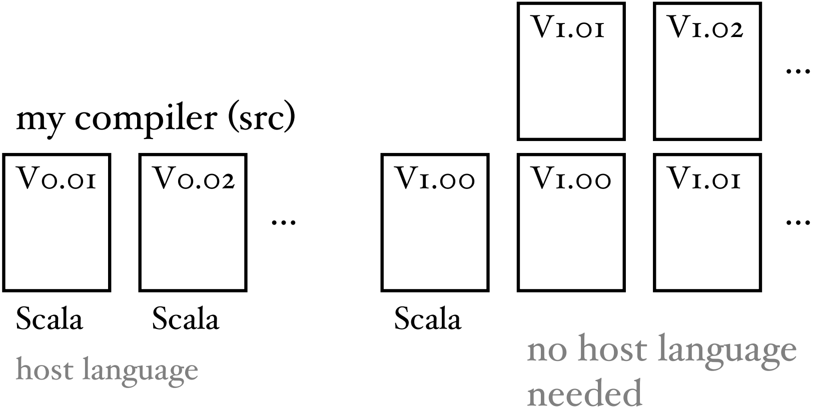
V1.01

Scala

Scala

Scala

host language



my compiler (src)

V0.01

V0.02

...

Scala

Scala

host language

V1.01

V1.02

...

V1.00

V1.00

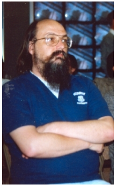
V1.01

...

Scala

no host language needed

Hacking Compilers



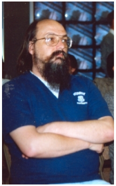
Ken Thompson
Turing Award, 1983

Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.

No amount of source level verification will protect you from such Thompson-hacks.

Therefore in safety-critical systems it is important to rely on only a very small TCB.

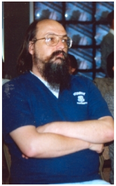
Hacking Compilers



Ken Thompson
Turing Award, 198

1) *Assume you ship the compiler as binary and also with sources.*
2) *Make the compiler aware when it compiles itself.*
3) *Add the Trojan horse.*
4) *Compile.*
5) *Delete Trojan horse from the sources of the compiler.*
6) *Go on holiday for the rest of your life. ;o)*

Hacking Compilers



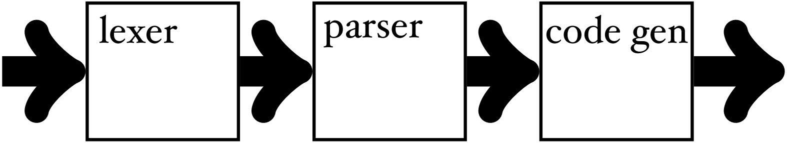
Ken Thompson
Turing Award, 1983

Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.

No amount of source level verification will protect you from such Thompson-hacks.

Therefore in safety-critical systems it is important to rely on only a very small TCB.

Our Compiler



lexer input: string

lexer output: sequence of tokens
(white space and comments filtered out)

parser output: abstract syntax tree

code gen output: assembler byte code /
assembler machine code

For-Loops

`for` Id `:=` AExp `upto` AExp `do` Block

```
for i := 2 upto 4 do {
  write i
}
```

While-Language

$$
\begin{array}{lcl}
\mathit{Stmt} & \rightarrow & \mathrm{skip}\\
 & | & \mathit{Id} := \mathit{AExp}\\
 & | & \mathrm{if}\ \mathit{BExp}\ \mathrm{then}\ \mathit{Block}\ \mathrm{else}\ \mathit{Block}\\
 & | & \mathrm{while}\ \mathit{BExp}\ \mathrm{do}\ \mathit{Block}\\
 & | & \mathrm{write}\ \mathit{Id}\\
 & | & \mathrm{read}\ \mathit{Id}\\
\mathit{Stmts} & \rightarrow & \mathit{Stmt}\ ;\ \mathit{Stmts}\\
 & | & \mathit{Stmt}\\
\mathit{Block} & \rightarrow & \{\mathit{Stmts}\}\\
 & | & \mathit{Stmt}\\
\mathit{AExp} & \rightarrow & \ldots\\
\mathit{BExp} & \rightarrow & \ldots
\end{array}
$$

Interpreter

$$
\begin{array}{lcl}
\text{eval}(n, E) & \overset{\text{def}}{=} & n \\
\text{eval}(x, E) & \overset{\text{def}}{=} & E(x) \quad \text{lookup } x \text{ in } E \\
\text{eval}(a_1 + a_2, E) & \overset{\text{def}}{=} & \text{eval}(a_1, E) + \text{eval}(a_2, E) \\
\text{eval}(a_1 - a_2, E) & \overset{\text{def}}{=} & \text{eval}(a_1, E) - \text{eval}(a_2, E) \\
\text{eval}(a_1 * a_2, E) & \overset{\text{def}}{=} & \text{eval}(a_1, E) * \text{eval}(a_2, E) \\
\\
\text{eval}(a_1 = a_2, E) & \overset{\text{def}}{=} & \text{eval}(a_1, E) = \text{eval}(a_2, E) \\
\text{eval}(a_1\, != a_2, E) & \overset{\text{def}}{=} & \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E)) \\
\text{eval}(a_1 < a_2, E) & \overset{\text{def}}{=} & \text{eval}(a_1, E) < \text{eval}(a_2, E)
\end{array}
$$

Interpreter (2)

$$
\begin{array}{lcl}
\text{eval}(\text{skip}, E) & \overset{\text{def}}{=} & E \\
\text{eval}(x := a, E) & \overset{\text{def}}{=} & E(x \mapsto \text{eval}(a, E)) \\
\multicolumn{3}{l}{\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) \overset{\text{def}}{=}} \\
& & \text{if eval}(b, E) \text{ then eval}(cs_1, E) \\
& & \qquad\qquad\quad\;\; \text{else eval}(cs_2, E) \\
\multicolumn{3}{l}{\text{eval}(\text{while } b \text{ do } cs, E) \overset{\text{def}}{=}} \\
& & \text{if eval}(b, E) \\
& & \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\
& & \text{else } E \\
\text{eval}(\text{write } x, E) & \overset{\text{def}}{=} & \{\ \text{println}(E(x))\ ;\ E\ \}
\end{array}
$$

Compiling Writes

write x

```
.method public static write(I)V          (library function)
   .limit locals 5
   .limit stack 5
   iload 0
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(I)V
   return
.end method
```

iload $E(x)$
invokestatic write(I)V

```
.class public XXX.XXX
.super java/lang/Object

.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit locals 200
   .limit stack 200
```

(here comes the compiled code)

```
   return
.end method
```