A Formalisation of an Access Control Framework

joint work with Chunhan Wu and Xingyuan Zhang from the PLA University of Science and Technology in Nanjing

Christian Urban
King's College London

Access Control

- perhaps most known are Unix-style access control systems (root super-user, setuid mechanism)

```
> ls -ld . * */*
drwxr-xr-x 1 alice staff    32768  Apr  2 2010 .
-rw----r-- 1 alice students 31359  Jul 24 2011 manual.txt
-rwsr--r-x 1 bob   students 141359 Jun  1 2013 microedit
dr--r-xr-x 1 bob   staff    32768  Jul 23 2011 src
-rw-r--r-- 1 bob   staff    81359  Feb 28 2012 src/code.c
```

Access Control

more fine-grained access control is provided by

- SELinux
  (security enhanced Linux devloped by the NSA;
  mandatory access control system)

- Role-Compatibility Model
  (developed by Amon Ott;
  main application in the Apache server)

Operations in the OS

using Paulson's inductive method a **state of the system** is a **trace**, a list of events (system calls):

$$[e_1, \ldots, e_2]$$

e ::=	*CreateFile p f*	| *ReadFile p f*	| *Send p i*
|	*WriteFile p f*	| *Execute p f*	| *Recv p i*
|	*DeleteFile p f*	| *Clone p p'*	| *CreateIPC p i*
|	*ChangeOwner p u*	| *ChangeRole p r*	| *DeleteIPC p i*
|	*Kill p p'*		

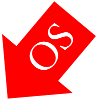
Valid Traces

we need to restrict the traces to **valid traces**:

$$\frac{}{\mathit{valid}\ [\,]} \qquad \frac{\mathit{valid}\ s \quad \mathit{admissible}\ s\ e \quad \mathit{granted}\ s\ e}{\mathit{valid}\ (e{::}s)}$$

Valid Traces

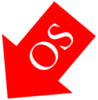
we need to restrict the traces to **valid traces**:



$$\frac{}{\mathit{valid}\ [\,]} \qquad \frac{\mathit{valid}\ s \quad \mathit{admissible}\ s\ e \quad \mathit{granted}\ s\ e}{\mathit{valid}\ (e::s)}$$

Valid Traces

we need to restrict the traces to **valid traces**:



$$\frac{}{\textit{valid}\ [\,]} \qquad \frac{\textit{valid}\ s \qquad \textit{admissible}\ s\ e \qquad \textit{granted}\ s\ e}{\textit{valid}\ (e::s)}$$

$$\frac{p \in \textit{current\_procs}\ s \qquad p' \notin \textit{current\_procs}\ s}{\textit{admissible}\ s\ (\textit{Clone}\ p\ p')}$$

Valid Traces

we need to restrict the traces to **valid traces**:

RC

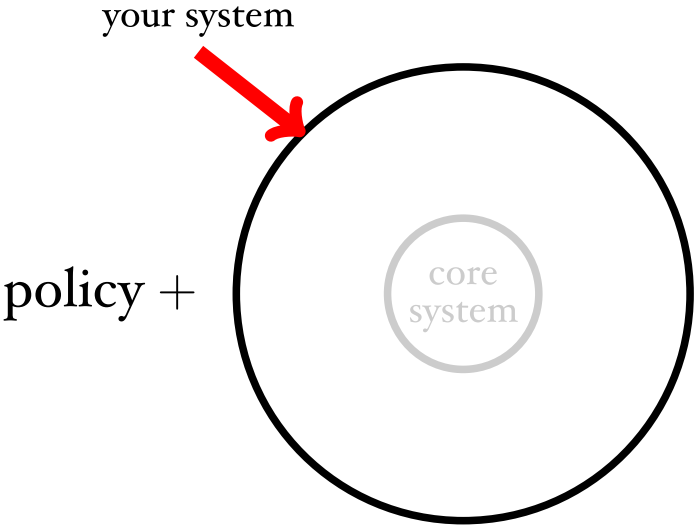
$$\frac{}{\textit{valid}\ []} \qquad \frac{\textit{valid}\ s \quad \textit{admissible}\ s\ e \quad \textit{granted}\ s\ e}{\textit{valid}\ (e::s)}$$

$$\frac{\textit{is\_current\_role}\ s\ p\ r \quad \textit{is\_file\_type}\ s\ f\ t \quad (r,\ t,\ \textit{Execute}) \in \textit{permissions}}{\textit{granted}\ s\ (\textit{Execute}\ p\ f)}$$

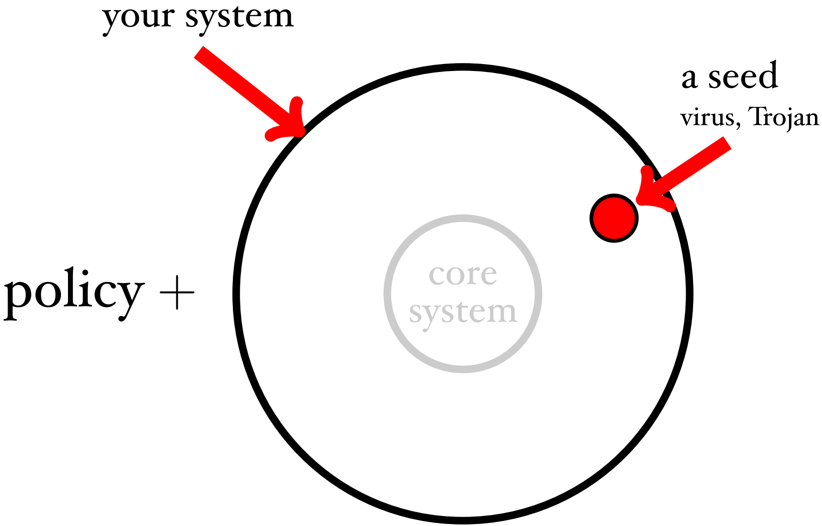
Design of AC-Policies

"what you specify is what you get but not necessarily what you want..."

Testing Policies

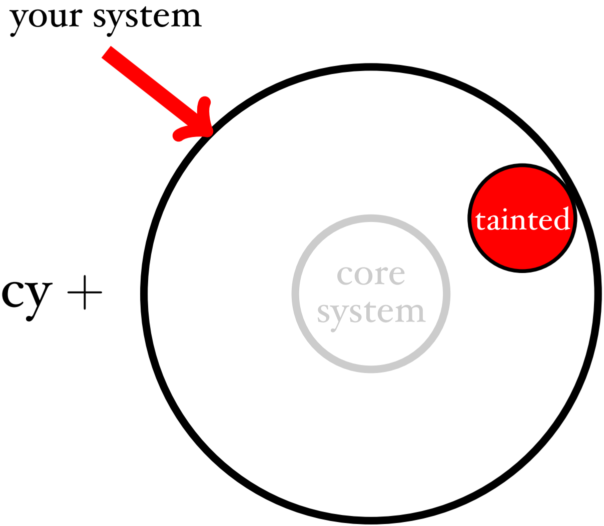


Testing Policies

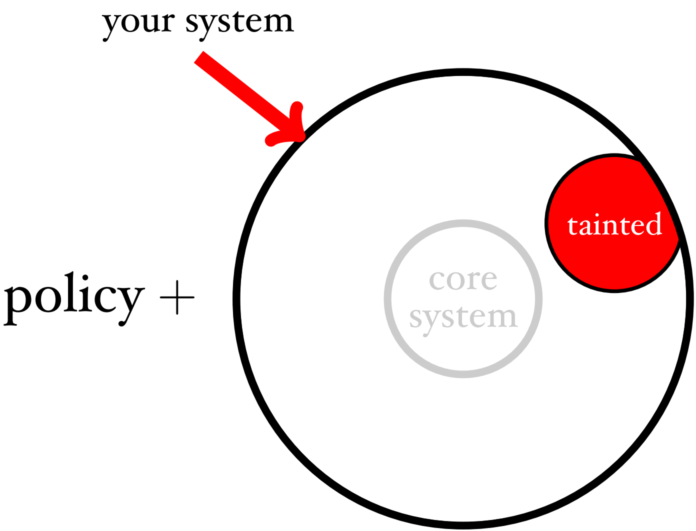


Testing Policies

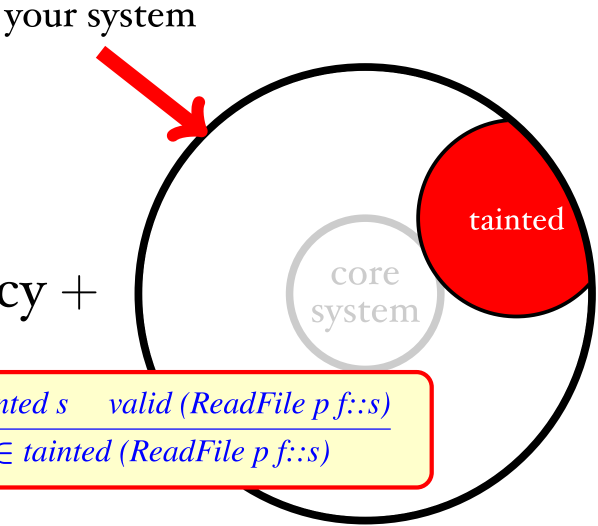
policy +



Testing Policies



Testing Policies

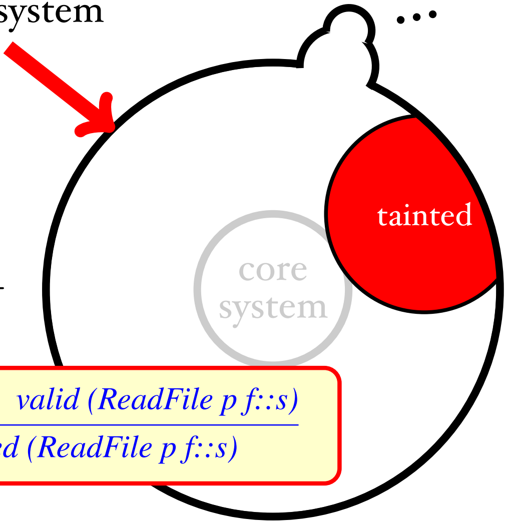


policy +

$$\frac{f \in \mathit{tainted}\ s \qquad \mathit{valid}\ (\mathit{ReadFile}\ p\ f :: s)}{p \in \mathit{tainted}\ (\mathit{ReadFile}\ p\ f :: s)}$$

Testing Policies

your system



policy +

$$\frac{f \in \mathit{tainted}\ s \qquad \mathit{valid}\ (\mathit{ReadFile}\ p\ f::s)}{p \in \mathit{tainted}\ (\mathit{ReadFile}\ p\ f::s)}$$

A Sound and Complete Test

- working purely in the *dynamic world* does not work – infinite state space
- working purely on *static* policies also does not work – because of over approximation
  - suppose a tainted file has type *bin* and
  - there is a role r which can both read and write *bin*-files

A Sound and Complete Test

- working purely in the *dynamic world* does not work – infinite state space
- working purely on *static* policies also does not work – because of over approximation
  - suppose a tainted file has type *bin* and
  - there is a role r which can both read and write *bin*-files
  - then we would conclude that this tainted file can spread

A Sound and Complete Test

- working purely in the *dynamic world* does not work – infinite state space
- working purely on *static* policies also does not work – because of over approximation
  - suppose a tainted file has type *bin* and
  - there is a role r which can both read and write *bin*-files
  - then we would conclude that this tainted file can spread
  - but if there is no process with role r and it will never been created, then the file actually does not spread

A Sound and Complete Test

- working purely in the *dynamic world* does not work – infinite state space

- working purely on *static* policies also does not work – because of over approximation
  - suppose a tainted file has type *bin* and
  - there is a role r which can both read and write *bin*-files
  - then we would conclude that this tainted file can spread
  - but if there is no process with role r and it will never been created, then the file actually does not spread

- our solution: take a middle ground and record precisely the information of the initial state, but be less precise about every newly created object.

Results about our Test

- we can show that the objects (files, processes, ...) we need to consider are only finite (at some point it does not matter if we create another *bin*-file)

Thm (Soundness)
If our test says an object is taintable, then it is taintable in the OS, and we produce a sequence of events that show how it can be tainted.

Results about our Test

- we can show that the objects (files, processes, ...) we need to consider are only finite (at some point it does not matter if we create another *bin*-file)

Thm (Soundness)
If our test says an object is taintable, then it is taintable in the OS, and we produce a sequence of events that show how it can be tainted.

Thm (Completeness)
If an object is taintable and *undeletable**, then our test will find out that it is taintable.

* an object is *undeleteable* if it exists in the initial state, but there exists no valid state in which it could have been deleted

Why the Restriction?

- assume a process with *ID* is tainted but gets killed by another process
- after that a proces with the same *ID* gets *re-created* by cloning an untainted process
- clearly the new process should be considered *un*tainted

Why the Restriction?

- assume a process with *ID* is tainted but gets killed by another process
- after that a proces with the same *ID* gets *re-created* by cloning an untainted process
- clearly the new process should be considered *un*tainted

unfortunately our test will not be able to detect the difference (we are less precise about newly created processes)

Why the Restriction?

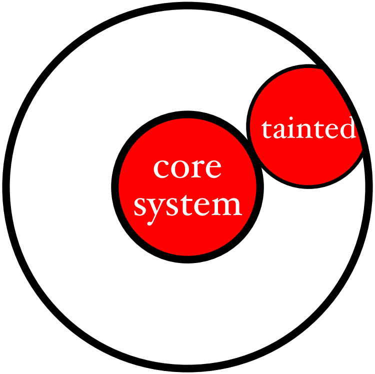
- assume a process with *ID* is tainted but gets killed by another process
- after that a proces with the same *ID* gets *re-created* by cloning an untainted process
- clearly the new process should be considered *un*tainted

unfortunately our test will not be able to detect the difference (we are less precise about newly created processes)

Is this a serious restriction? We think not ...

Core System

Admins usually ask whether their policy is strong enough to protect their core system?



core system files are typically undeletable

Conclusion

- we considered the Role-Compatibility Model used for securing the Apache Server

  13 events, 13 rules for OS admisibility, 14 rules for RC-granting, 10 rules for tainted

- we can scale this to SELinux

  more fine-grainded OS events (inodes, hard-links, shared memory, ...)

22 events, 22 admisibility, 22 granting, 15 taintable

Conclusion

- we considered the Role-Compatibility Model used for securing the Apache Server

  13 events, 13 rules for OS admisibility, 14 rules for RC-granting, 10 rules for tainted

- we can scale this to SELinux

  more fine-grainded OS events (inodes, hard-links, shared memory, ...)

  22 events, 22 admisibility, 22 granting, 15 taintable

- hard sell to Ott (who designed the RC-model)
- hard sell to the community working on access control (beyond *good science*)