Security and the Average Programmer

Silas Boyd-Wickizer, Pablo Buiras*, Daniel Giffin, Stefan Heule, Eddie Kohler, Amit Levy, David Mazières, John Mitchell, Alejandro Russo*, Amy Shen, Deian Stefan, David Terei, Edward Yang, and Nickolai Zeldovich

Stanford and *Chalmers

Tuesday, April 15, 2014

Software vulnerabilities are everywhere

- High-profile software (nginx, Symantec)
- But also web applications (Paymaxx)
  - One-off designs receive little outside scrutiny
  - See a wide range of programmer abilities (unlike core components such as kernels)
- Also embedded systems (fridge, TV)
- "Internet of things" $\overset{?}{\approx}$ remote exploit of things
- Fewer and fewer settings where software security doesn't matter

The only solution

The median programmer must build secure systems.

- Sadly, I won't tell you how to make this happen today, but
- Information flow control (IFC) has made progress towards the goal

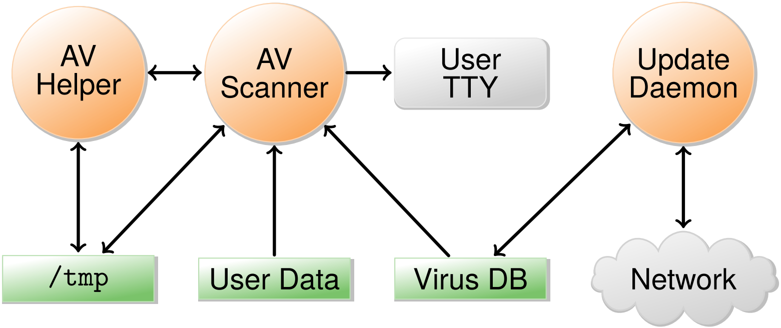
Steps towards the goal

- Allow experts to incorporate third-party code into secure systems
  - Achievable if you are willing to use a new operating system (HiStar)
  - Compatibility issues make it hard to deploy a new OS
- Allow experts to manage non-experts building secure systems
  - Possible if you teach people a new language (Haskell)
  - Ideas may be transferable to mainstream languages (e.g., JavaScript)
- Allow *anyone* to hire non-experts to build secure systems
  - This is *the* big open problem
  - IFC is a plausible approach, and we have some experience pointing to the remaining difficulties

Outline

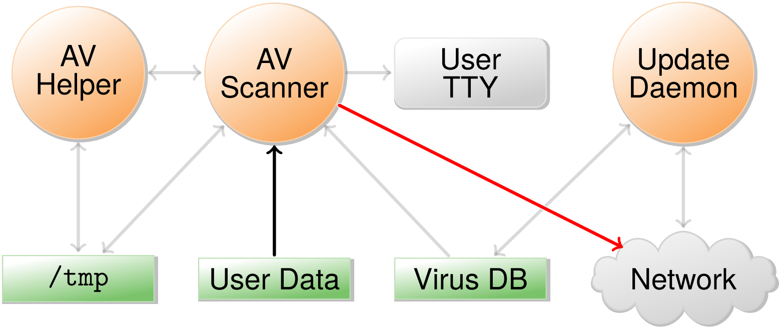
1. Background: Information flow control
2. HiStar
3. IFC for Haskell
4. Experience

Example: Anti-virus software



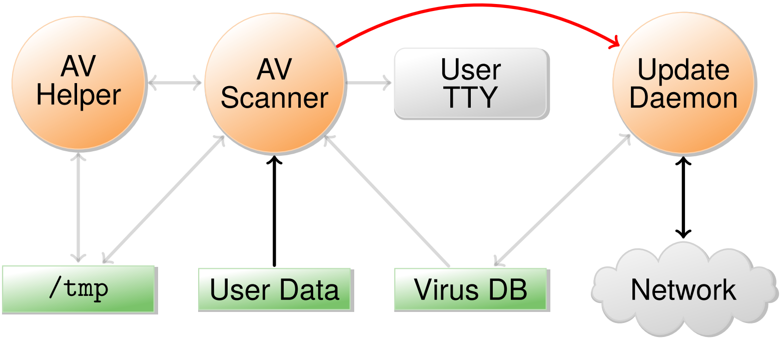
- Symantec AV (deployed on 200M machines) had remote exploit
- Can the OS provide security despite Symantec's programmers?
  - Prevent leaking contents of private files to network
  - Prevent tampering with contents of files

Example: Anti-virus software



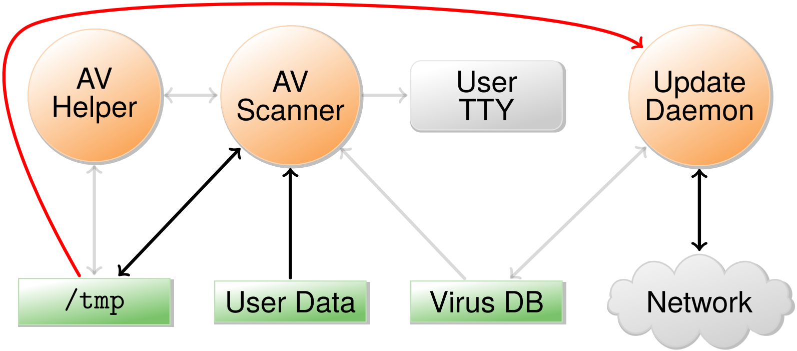
- Scanner can write your private data to network
- Prevent scanner from invoking any system call that might send a network message?

Example: Anti-virus software



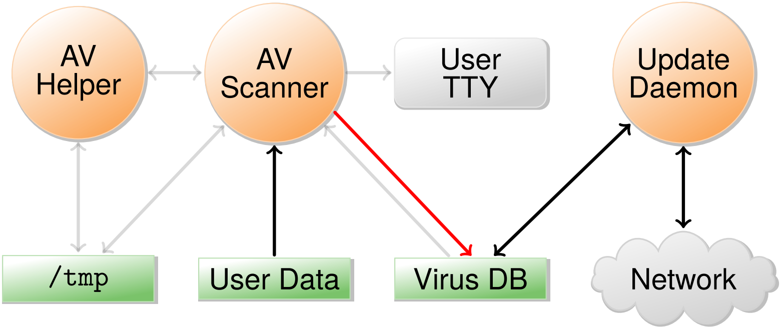
- Scanner can send private data to update daemon
- Update daemon sends data over network
  - Can cleverly disguise secrets in order/timing of update requests
- Block IPC & shared memory system calls in scanner?

Example: Anti-virus software



- Scanner can write data to world-readable file in `/tmp`
- Update daemon later reads and discloses file
- Prevent update daemon from using `/tmp`?

Example: Anti-virus software

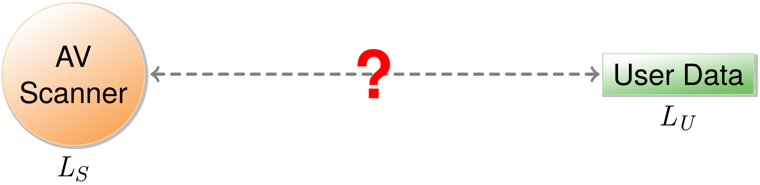


- Scanner can acquire read locks on virus database
  - Encode secret user data by locking various ranges of file
- Update daemon decodes data by detecting locks
  - Discloses private data over the network
- Have trusted software copy virus DB for scanner?

The list goes on

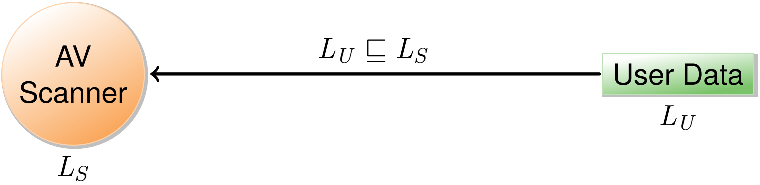
- Scanner can call setproctitle with user data
  - Update daemon extracts data by running ps
- Scanner can bind particular TCP or UDP port numbers
  - Sends no network traffic, but detectable by update daemon
- Scanner can relay data through another process
  - Call ptrace to take over process, then write to network
  - Use sendmail, httpd, or portmap to reveal data
- Disclose data by modulating free disk space
- Can we ever convince ourselves we've covered all possible communication channels?
  - Not without a more systematic approach to the problem

Background: Information flow control



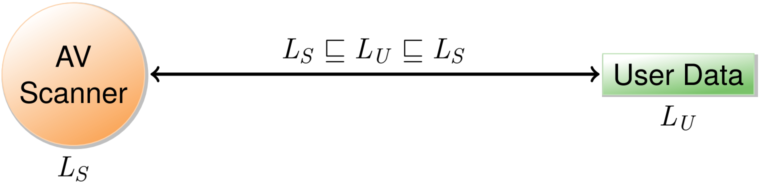
- Every piece of data in the system has a label
- Every process/thread/subject has a label
- Labels are partially ordered by $\sqsubseteq$ ("can flow to")
- Example: Scanner (labeled L_S) accesses user file (labeled L_U)
    - Check permission by comparing L_S and L_U
    - File read? Information flows from file to scanner. Require: $L_U \sqsubseteq L_S$.
    - File write? Information flows in both directions. Require: $L_U \sqsubseteq L_S$ and $L_S \sqsubseteq L_U$.

Background: Information flow control



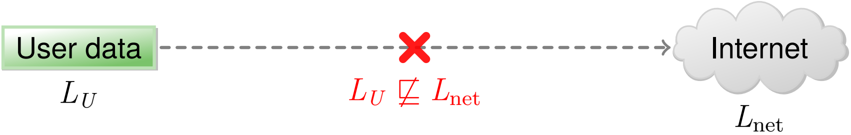
- Every piece of data in the system has a label
- Every process/thread/subject has a label
- Labels are partially ordered by $\sqsubseteq$ ("can flow to")
- Example: Scanner (labeled L_S) accesses user file (labeled L_U)
  - Check permission by comparing L_S and L_U
  - File read? Information flows from file to scanner. Require: $L_U \sqsubseteq L_S$.
  - File write? Information flows in both directions. Require: $L_U \sqsubseteq L_S$ and $L_S \sqsubseteq L_U$.

Background: Information flow control



- Every piece of data in the system has a label
- Every process/thread/subject has a label
- Labels are partially ordered by $\sqsubseteq$ ("can flow to")
- Example: Scanner (labeled L_S) accesses user file (labeled L_U)
  - Check permission by comparing L_S and L_U
  - File read? Information flows from file to scanner. Require: $L_U \sqsubseteq L_S$.
  - File write? Information flows in both directions. Require: $L_U \sqsubseteq L_S$ and $L_S \sqsubseteq L_U$.

$\sqsubseteq$ is transitive



User data

L_U

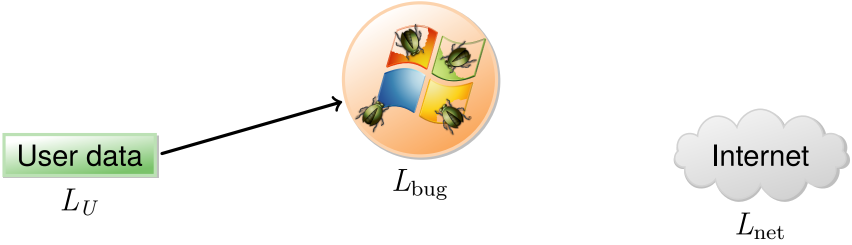
$L_U \not\sqsubseteq L_{\text{net}}$

Internet

L_{net}

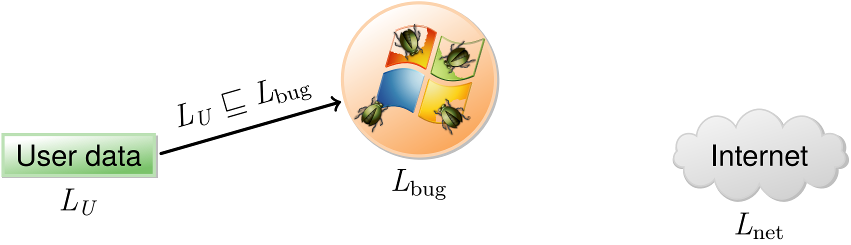
- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet ($L_U \not\sqsubseteq L_{\text{net}}$)
  - Policy holds regardless of what other software does
    …so you don't care what the programmer did

$\sqsubseteq$ is transitive



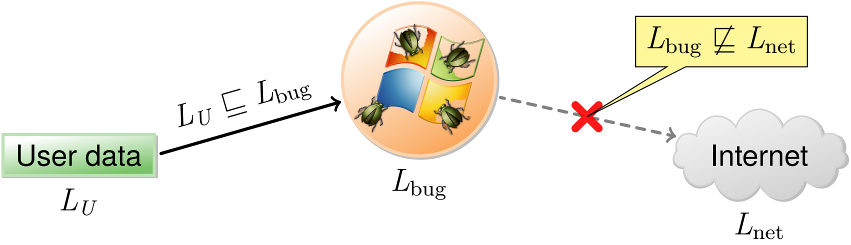
- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet ($L_U \not\sqsubseteq L_{\text{net}}$)
  - Policy holds regardless of what other software does
    …so you don't care what the programmer did
- Suppose untrustworthy software labeled L_{bug} reads user file
  - Must have $L_U \sqsubseteq L_{\text{bug}}$
  - But since $L_U \not\sqsubseteq L_{\text{net}}$, it follows that $L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$.

$\sqsubseteq$ is transitive



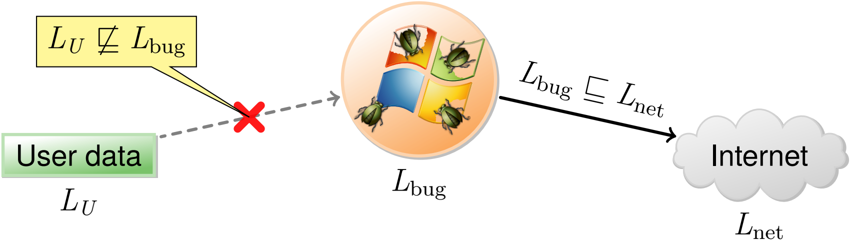
- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet ($L_U \not\sqsubseteq L_{\mathrm{net}}$)
  - Policy holds regardless of what other software does
    …so you don't care what the programmer did
- Suppose untrustworthy software labeled L_{bug} reads user file
  - Must have $L_U \sqsubseteq L_{\mathrm{bug}}$
  - But since $L_U \not\sqsubseteq L_{\mathrm{net}}$, it follows that $L_{\mathrm{bug}} \not\sqsubseteq L_{\mathrm{net}}$.

$\sqsubseteq$ is transitive


User data
L_U
$L_U \sqsubseteq L_{\text{bug}}$
L_{bug}
$L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$
Internet
L_{net}

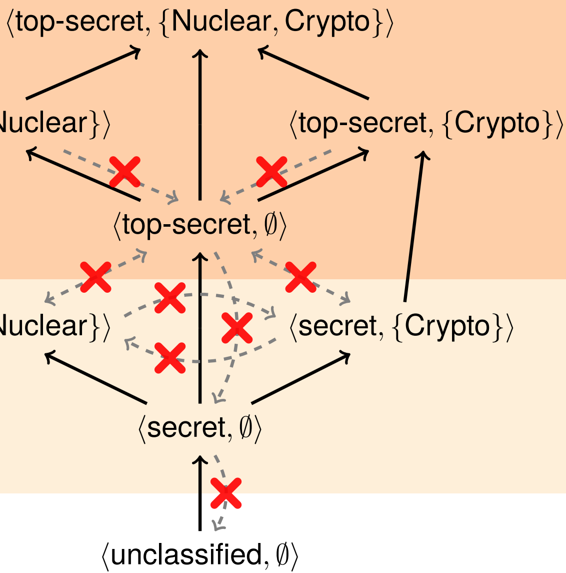
- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet ($L_U \not\sqsubseteq L_{\text{net}}$)
  - Policy holds regardless of what other software does
    …so you don't care what the programmer did
- Suppose untrustworthy software labeled L_{bug} reads user file
  - Must have $L_U \sqsubseteq L_{\text{bug}}$
  - But since $L_U \not\sqsubseteq L_{\text{net}}$, it follows that $L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$.

$\sqsubseteq$ is transitive


$L_U \not\sqsubseteq L_{\text{bug}}$
User data
L_U
L_{bug}
$L_{\text{bug}} \sqsubseteq L_{\text{net}}$
Internet
L_{net}

- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet ($L_U \not\sqsubseteq L_{\text{net}}$)
  - Policy holds regardless of what other software does
    …so you don't care what the programmer did
- Conversely, a process that *can* write to network cannot read the file

Traditionally labels form static lattice



$\langle$top-secret, $\{$Nuclear, Crypto$\}\rangle$

$\langle$top-secret, $\{$Nuclear$\}\rangle$

$\langle$top-secret, $\{$Crypto$\}\rangle$

$\langle$top-secret, $\emptyset\rangle$

$\langle$secret, $\{$Nuclear$\}\rangle$

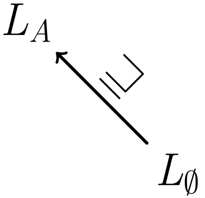
$\langle$secret, $\{$Crypto$\}\rangle$

$\langle$secret, $\emptyset\rangle$

$\langle$unclassified, $\emptyset\rangle$

$L_1 \longrightarrow L_2$
means $L_1 \sqsubseteq L_2$

Dynamic labels can express per-user policy



- E.g., use $L_\emptyset$ for public data, L_A for user A's private data
- If new user B joins web site, introduce new label L_B for his data
    - A and B cannot read each other's private data
- Mix A's and B's private data? Need label $L_{AB} = L_A \sqcup L_B$
- But what if A wants to make her data public?

Dynamic labels can express per-user policy

$$\begin{tikzcd}
L_A \arrow[rr, dashed, leftrightarrow, "\times"] & & L_B \\
& L_\emptyset \arrow[ul, "\sqsubseteq"] \arrow[ur, "\sqsubseteq"'] &
\end{tikzcd}$$

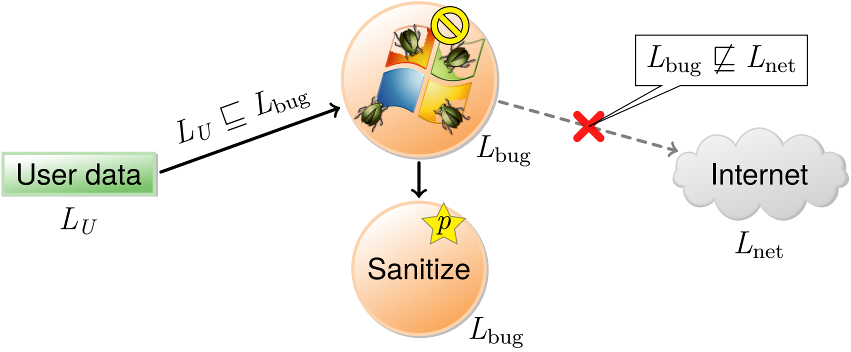
- E.g., use $L_\emptyset$ for public data, L_A for user A's private data
- If new user B joins web site, introduce new label L_B for his data
    - A and B cannot read each other's private data
- Mix A's and B's private data? Need label $L_{AB} = L_A \sqcup L_B$
- But what if A wants to make her data public?

Dynamic labels can express per-user policy

$$\begin{tikzcd}
& L_{AB} & \\
L_A \arrow[ur, "\sqsubseteq"] & & L_B \arrow[ul, "\sqsubseteq"'] \\
& L_\emptyset \arrow[ul, "\sqsubseteq"] \arrow[ur, "\sqsubseteq"'] &
\end{tikzcd}$$

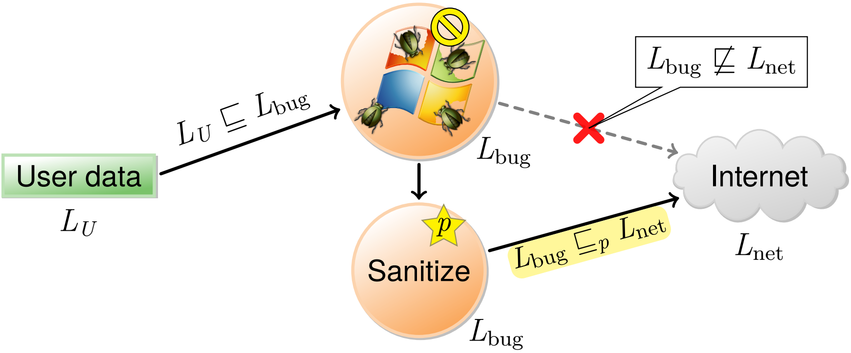
- E.g., use $L_\emptyset$ for public data, L_A for user A's private data
- If new user B joins web site, introduce new label L_B for his data
  - A and B cannot read each other's private data
- Mix A's and B's private data? Need label $L_{AB} = L_A \sqcup L_B$
- But what if A wants to make her data public?

Decentralized information flow control [Myers]



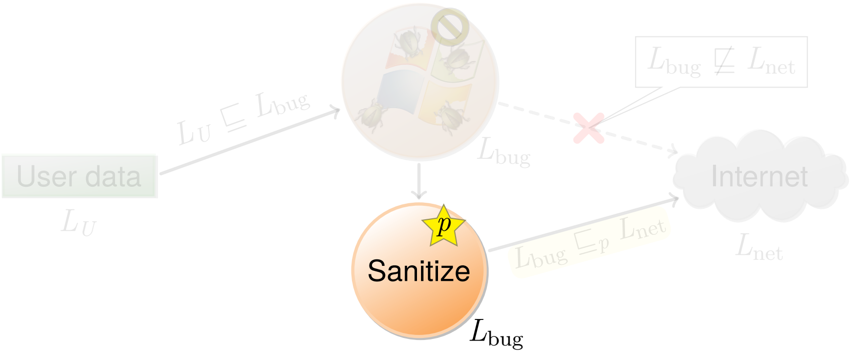
- Privilege p lets one bypass restrictions of L_{bug} (represented 🛇)
- Exercising p loosens label requirements to a pre-order, $\sqsubseteq_p$
  - Since $L_{\text{bug}} \sqsubseteq_p L_{\text{net}}$, Sanitize process can send result to network
- Idea: Set labels so you understand all use of relevant privileges

Decentralized information flow control [Myers]


$L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$
$L_U \sqsubseteq L_{\text{bug}}$
User data
L_U
L_{bug}
Internet
p
Sanitize
$L_{\text{bug}} \sqsubseteq_p L_{\text{net}}$
L_{net}
L_{bug}

- Privilege p lets one bypass restrictions of L_{bug} (represented 🚫)
- Exercising p loosens label requirements to a pre-order, $\sqsubseteq_p$
  - Since $L_{\text{bug}} \sqsubseteq_p L_{\text{net}}$, Sanitize process can send result to network
- Idea: Set labels so you understand all use of relevant privileges

Decentralized information flow control [Myers]


$L_U \sqsubseteq L_{\mathrm{bug}}$
$L_{\mathrm{bug}} \not\sqsubseteq L_{\mathrm{net}}$
User data
L_U
L_{bug}
Internet
p
Sanitize
$L_{\mathrm{bug}} \sqsubseteq_p L_{\mathrm{net}}$
L_{net}
L_{bug}

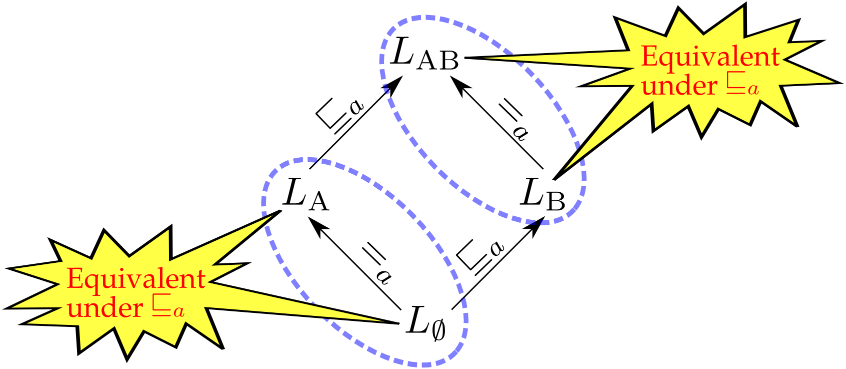
- Privilege p lets one bypass restrictions of L_{bug} (represented 🛇)
- Exercising p loosens label requirements to a pre-order, $\sqsubseteq_p$
  - Since $L_{\mathrm{bug}} \sqsubseteq_p L_{\mathrm{net}}$, Sanitize process can send result to network
- Idea: Set labels so you understand all use of relevant privileges

Example privileges

$$\begin{tikzcd}
& L_{\mathrm{AB}} & \\
L_{\mathrm{A}} \arrow[ur, "\sqsubseteq"] & & L_{\mathrm{B}} \arrow[ul, "\sqsubseteq"'] \\
& L_{\emptyset} \arrow[ul, "\sqsubseteq"'] \arrow[ur, "\sqsubseteq"] &
\end{tikzcd}$$

- Consider again the simple two user lattice
- Let a be user A's privileges
- User A should be allowed to make her own data public
- She can because $L_A \sqsubseteq_a \ L_\emptyset$ and $L_{AB} \sqsubseteq_a \ L_B$

Example privileges



- Consider again the simple two user lattice
- Let a be user A's privileges
- User A should be allowed to make her own data public
- She can because $L_A \sqsubseteq_a L_{\emptyset}$ and $L_{AB} \sqsubseteq_a L_B$

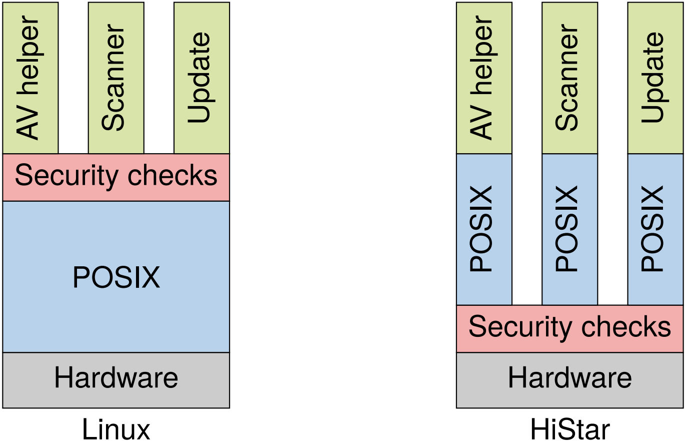
Outline

1. Background: Information flow control
2. HiStar
3. IFC for Haskell
4. Experience

HiStar OS

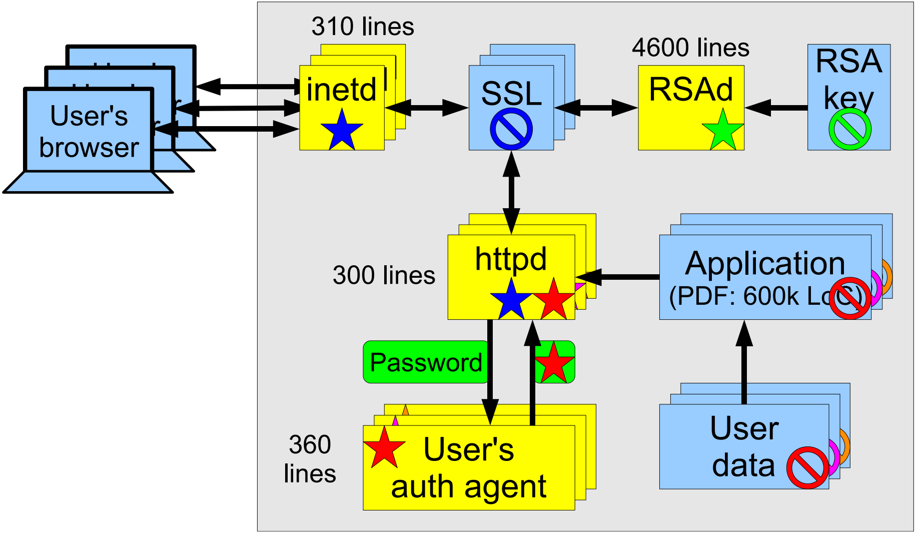
- Clean-slate OS that makes all information flow explicit
- Key feature: partial declassification privileges
  - All other security features built on partial declassification
- Example: user IDs
  - Each uid implemented as two privileges, one for reading and one for writing user's files
  - User's login shell receives privileges after authentication
- Example: web security
  - Each web user is associated with unique privileges
  - Ensures Paymaxx-style dump-the-database attacks not possible

HiStar architecture



- Kernel provides six simple object types
    - Simple enough that information flow is unambiguous
- Layer POSIX API as untrusted library on top of kernel

Web server


User's browser
310 lines
inetd
SSL
4600 lines
RSAd
RSA key
300 lines
httpd
Application
(PDF: 600k LoC)
Password
360 lines
User's auth agent
User data

What we learned from HiStar

- Nickolai Zeldovich can secure 1,000,000+ lines of third-party code
  - But he is *not* the median programmer to say the least
- System-wide egalitarian access control is practical
- Dynamic IFC enforcement can avoid implicit flows
  - Dynamic IFC was previously through to be inherently insecure

Outline

1. Background: Information flow control
2. HiStar
3. IFC for Haskell
4. Experience

Why Haskell?

- Haskell is a pure functional langauge
  - Functions without side effects do not leak data
- Impure computations have type `IO a` for some return type `a`
  - Haskell's "Monad" support lets one to introduce other types like `IO`
- Idea: introduce a new *labeled IO* type, `LIO`, as substitute for `IO`
  - Internally, `LIO` makes use of `IO` actions, but only after enforcing IFC
  - Type safety and abstraction prevent `LIO` code from executing raw `IO`
- Safe Haskell compiler feature enforces type safety & abstraction
  - Privileged symbols (ending `...TCB`) are inaccessible from safe code

Example: Wrapping IO abstractions

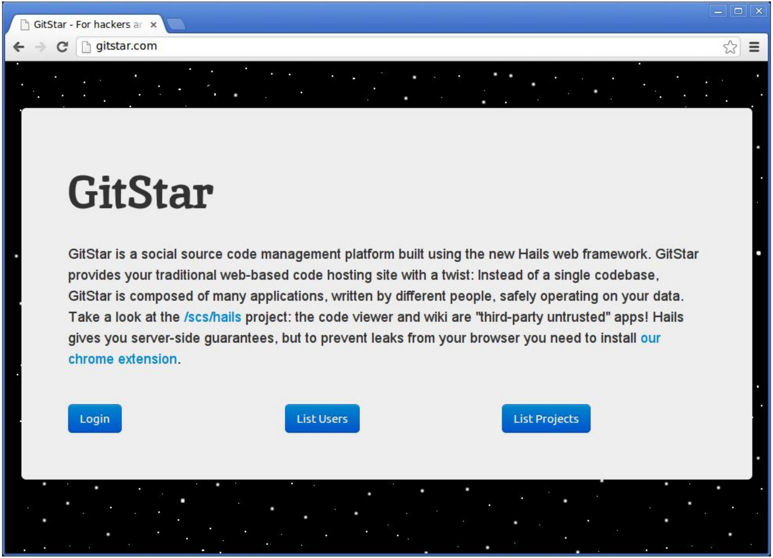
- Wrap `IO` abstractions into generic labeled objects
  - `blessTCB` transforms an `IO` function into an `LIO` action on a labeled version of the same type
  - `LIO` version checks labels before performing action
- E.g., Haskell `MVar` abstraction provides mutable variables
  - `LIO` version called `LMVar` merely a wrapped `MVar`

```
{-# LANGUAGE Trustworthy #-}
    ⋮
type LMVar l a = LObj l (MVar a)

takeLMVar :: Label l => LMVar l a -> LIO l a
takeLMVar = blessTCB "takeLMVar" takeMVar

putLMVar :: Label l => LMVar l a -> a -> LIO l ()
putLMVar = blessTCB "putLMVar" putMVar
    ⋮
```

Hails: An LIO web framework

- Introduces Model-Policy-View-Controller paradigm
- A Hails server comprises two types of software package
  - *VC*s contain View and Controller logic
  - *MPs* contain Model and Policy logic
- Policies enforced using LIO
  - Also isolate spawned programs with Linux namespaces

GitStar

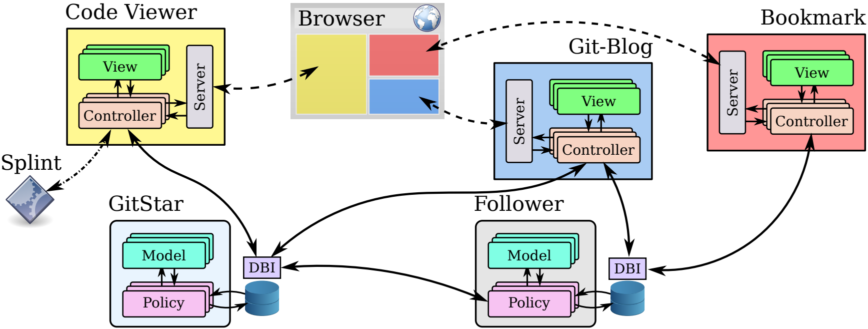


GitStar

GitStar is a social source code management platform built using the new Hails web framework. GitStar provides your traditional web-based code hosting site with a twist: Instead of a single codebase, GitStar is composed of many applications, written by different people, safely operating on your data. Take a look at the /scs/hails project: the code viewer and wiki are "third-party untrusted" apps! Hails gives you server-side guarantees, but to prevent leaks from your browser you need to install our chrome extension.

- Public GitHub-like service supporting private projects

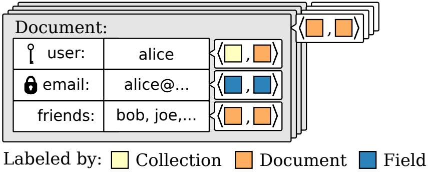
Simplified GitStar architecture



- Two MPs: *GitStar* hosts git repos, *Follower* stores a relationship between users
- Three different VC apps make use of these MPs
  - VCs can be written after the fact w/o permission of MP author
  - `LIO` ensures they cannot misuse data

What policy looks like

```
-- Set policy for "users" collection:
collection "users" $ do
  -- Set collection label:
  access $ do
    readers ==> anybody
    writers ==> anybody
  -- Declare user field as a key:
  field "user" key
  -- Set document label, given document doc:
  document $ \doc -> do
    readers ==> anybody
    writers ==> ("user" ‘from‘ doc) \/ _Follower
  -- Set email field label, given document doc:
  field "email" $ labeled $ \doc -> do
    readers ==> ("user" ‘from‘ doc)
                \/ fromList ("friends" ‘from‘ doc)
                \/ _Follower
    writers ==> anybody
```



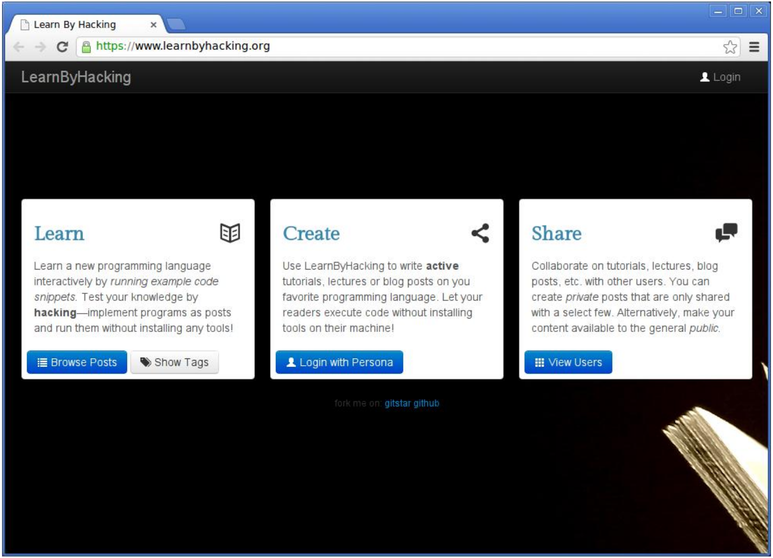
Document:

user:	alice	⟨🟨,🟧⟩
email:	alice@...	⟨🟦,🟦⟩
friends:	bob, joe,...	⟨🟧,🟧⟩

⟨🟧,🟧⟩

Labeled by: 🟨 Collection 🟧 Document 🟦 Field

LearnByHacking



Learn

Learn a new programming language interactively by *running example code snippets*. Test your knowledge by **hacking**—implement programs as posts and run them without installing any tools!

Create

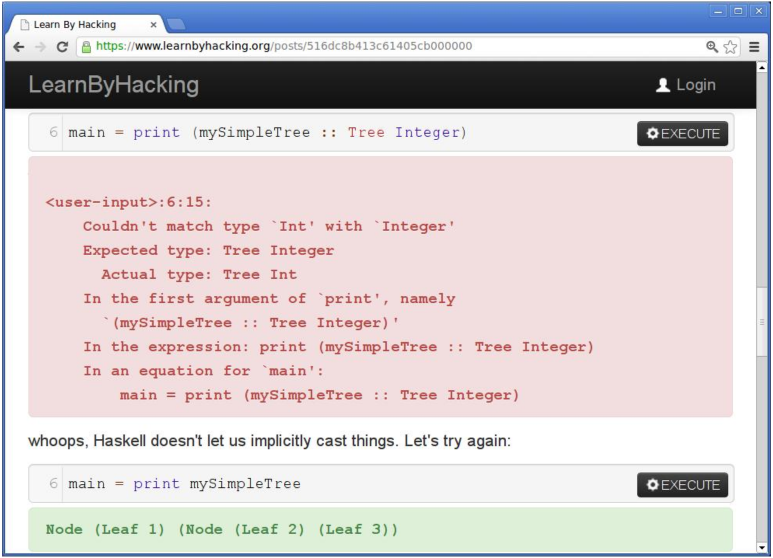
Use LearnByHacking to write **active** tutorials, lectures or blog posts on you favorite programming language. Let your readers execute code without installing tools on their machine!

Share

Collaborate on tutorials, lectures, blog posts, etc. with other users. You can create *private* posts that are only shared with a select few. Alternatively, make your content available to the general *public*.

fork me on: gitstar github

LearnByHacking


Learn By Hacking
https://www.learnbyhacking.org/posts/516dc8b413c61405cb000000
LearnByHacking
Login
EXECUTE

```
main = print (mySimpleTree :: Tree Integer)
```

```
<user-input>:6:15:
    Couldn't match type `Int' with `Integer'
    Expected type: Tree Integer
      Actual type: Tree Int
    In the first argument of `print', namely
      `(mySimpleTree :: Tree Integer)'
    In the expression: print (mySimpleTree :: Tree Integer)
    In an equation for `main':
        main = print (mySimpleTree :: Tree Integer)
```

whoops, Haskell doesn't let us implicitly cast things. Let's try again:

```
main = print mySimpleTree
```

```
Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
```

Outline

1. Background: Information flow control
2. HiStar
3. IFC for Haskell
4. Experience

Three usability data points

1. One high-school student hired at Stanford
2. Four (screened) Brandeis students in Lincoln labs evaluation study
3. Four Stanford students (hired blind, no experience necessary)

[Disclaimer: all programmers compensated in dollars.]

A few highly subjective conclusions

- + Teaching people Haskell much easier than deploying a new OS
  - Libraries, stack overflow, IRC…community has critical mass
  - People's willingness to learn new languages may be increasing
- + People generally had an easy time writing VCs
  - Which is good because VCs are larger and more numerous than MPs
- - Students struggled with policy
  - The policy DSL was introduced later, and helped some
- - It doesn't work to prototype an app, then add policy

- We've come a long way since HiStar's labels, which could mystify even senior systems researchers
  - E.g., Stanford team built task management system with rich policies
  - #1 challenge is enabling more people to understand, express policy

Questions

Secure
Computer
Systems

`http://www.scs.stanford.edu/`