Security and the Average Programmer


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” \( \approx \) remote exploit of things
- Fewer and fewer settings where software security doesn’t matter
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
1. Background: Information flow control
2. HiStar
3. IFC for Haskell
4. Experience
- 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
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?
• 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
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 \( L_S \sqsubseteq L_U \).
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 $L_S \sqsubseteq L_U$. 

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 $L_S \sqsubseteq L_U$. 
Transitivity makes it easier to reason about security

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

Example: Label user data so it cannot flow to Internet $\left( L_U \not\sqsubseteq L_{\text{net}} \right)$

- Policy holds regardless of what other software does
  ...so you don’t care what the programmer did

Suppose untrustworthy software labeled $L_{\text{bug}}$ reads user file

- Must have $L_U \sqsubseteq L_{\text{bug}}$
- But since $L_U \not\sqsubseteq L_{\text{net}}$, it follows that $L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$. 
Transitivity makes it easier to reason about security.

Example: Label user data so it cannot flow to Internet \((L_U \not\subseteq L_{\text{net}})\):
- Policy holds regardless of what other software does
  ...so you don’t care what the programmer did

Suppose untrustworthy software labeled \(L_{\text{bug}}\) reads user file:
- Must have \(L_U \subseteq L_{\text{bug}}\)
- But since \(L_U \not\subseteq L_{\text{net}}\), it follows that \(L_{\text{bug}} \not\subseteq L_{\text{net}}\).
\( \sqsubseteq \) is transitive

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

Example: Label user data so it cannot flow to Internet ($L_U \not\subseteq L_{\text{net}}$)
  - 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

\[ \langle \text{top-secret}, \{\text{Nuclear, Crypto}\} \rangle \]
\[ \langle \text{top-secret}, \{\text{Nuclear}\} \rangle \]
\[ \langle \text{top-secret}, \{\text{Crypto}\} \rangle \]
\[ \langle \text{top-secret}, \emptyset \rangle \]
\[ \langle \text{secret}, \{\text{Nuclear}\} \rangle \]
\[ \langle \text{secret}, \{\text{Crypto}\} \rangle \]
\[ \langle \text{secret}, \emptyset \rangle \]
\[ \langle \text{unclassified}, \emptyset \rangle \]

\( L_1 \rightarrow L_2 \) 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

- 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

- 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 \( \star \) lets one bypass restrictions of \( L_{\text{bug}} \) (represented \( \not\subset \))
- Exercising \( \star \) 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]

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

- Privilege $p$ lets one bypass restrictions of $L_{bug}$ (represented $\not\subseteq$)
- Exercising $p$ loosens label requirements to a pre-order, $\subseteq_p$
  - Since $L_{bug} \subseteq_p L_{net}$, Sanitize process can send result to network
- Idea: Set labels so you understand all use of relevant 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 $L_{AB} \sqsubseteq_a L_B$
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 $L_{AB} \sqsubseteq_a L_B$
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
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 thought to be inherently insecure
1 Background: Information flow control
2 HiStar
3 IFC for Haskell
4 Experience
• Haskell is a pure functional language
  - Functions without side effects do not leak data
• Impure computations have type \( \text{IO} \ a \) for some return type \( a \)
  - Haskell’s “Monad” support lets one to introduce other types like \( \text{IO} \)
• Idea: introduce a new labeled \( \text{IO} \) type, \( \text{LIO} \), as substitute for \( \text{IO} \)
  - Internally, \( \text{LIO} \) makes use of \( \text{IO} \) actions, but only after enforcing IFC
  - Type safety and abstraction prevent \( \text{LIO} \) code from executing raw \( \text{IO} \)
• Safe Haskell compiler feature enforces type safety & abstraction
  - Privileged symbols (ending \( \ldots \text{TCB} \)) 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

```haskell
{-# 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
  - VCs contain View and Controller logic
  - MPs contain Model and Policy logic
- Policies enforced using LIO
  - Also isolate spawned programs with Linux namespaces
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
• 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
-- 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
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 your 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.

Browse Posts  Show Tags  Login with Persona  View Users
fork me on: gist|star|github
LearnByHacking

```hs
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))
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/