Software security in the Internet of things


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)
- Now embedded systems (fridge, TV, car)
  - “Internet of things” ? remote exploitability of things
  - New mindset for embedded programmers
The median programmer must build secure systems.

- Information flow control (IFC) has made progress towards the goal
- Can IFC help in the Internet of things?
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 are transferable to mainstream languages (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 & Hails
3. Experience
4. IFC in the Internet of things
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?
• 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 \((L_U \nsubseteq L_{net})\)

- 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 \((L_U \nsubseteq L_{\text{net}})\)
- 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 \(L_U \nsubseteq L_{\text{net}}\), it follows that \(L_{\text{bug}} \nsubseteq L_{\text{net}}\).
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 $L_{\text{bug}}$ 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 \(L_{\text{bug}} \not\subseteq L_{\text{net}}\).
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 \[ L_1 \subseteq L_2 \]
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 $p$ lets one bypass restrictions of $L_{bug}$ (represented $\circlearrowleft$)
- 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 $L_{\text{bug}}$ (represented $\not\subseteq$)

- Exercising $p$ loosens label requirements to a pre-order, $\subseteq_p$
  - Since $L_{\text{bug}} \subseteq_p L_{\text{net}}$, 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 $L_{\text{bug}}$ (represented $\otimes$)
- Exercising $p$ loosens label requirements to a pre-order, $\subseteq_p$
  - Since $L_{\text{bug}} \subseteq_p L_{\text{net}}$, Sanitize process can send result to network
- Idea: Set labels so you understand all use of relevant privileges
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 $L_{AB} \sqsubseteq_a L_B$
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 $L_A \subseteq_a L_\emptyset$ and $L_{AB} \subseteq_a L_B$
1. Background: Information flow control
2. HiStar & Hails
3. Experience
4. IFC in the Internet of things
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
- 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
• 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 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 ..\text{TCTB}) are inaccessible from safe code
• 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
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
1. Background: Information flow control

2. HiStar & Hails

3. Experience

4. IFC in the Internet of things
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

+ APIs and languages can change programmer behavior
  - Much more effective than trying to “teach security”
+ Teaching people Haskell much easier than deploying a new OS
  - 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
1. Background: Information flow control
2. HiStar & Hails
3. Experience
4. IFC in the Internet of things
Putting computation on the computer

- There are some good reasons to compute in the cloud
  - Derive value for users by computing over multiple users’ data
- Then there are some very non-fundamental ones
  - NAT makes it a pain to connect to home devices [IPv6]
  - Users don’t trust themselves to manage storage [Ori]
  - Can’t tap unused cycles on users’ desktop machines/routers
- IFC works well for hosting untrusted code (e.g., Hails)
- Define API for devices to offload computation to desktop or router
  - Enforce privacy locally rather than depending on cloud
Information flow pictures concisely express security properties

Can we generate pictures and policy from same source?

Can visualizing information flow help users devise policy?
  - Hard for OS people to answer alone... good area for collaboration
• IFC directly captures many high-level security goals
• Security can depend on much smaller pieces of code
• IFC may be much easier to verify than full functional correctness
  - Particularly with some help from the programming language
Secure Computer Systems

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