Controlling the Flow of Information Despite Untrustworthy Software

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, David Mazières
Problem: Bad Code

- PayMaxx divulges social security numbers
  - Test account had SSN 000-00-0000, no password
  - Then access records by consecutive serial number
- CardSystems loses 40,000,000 CC numbers
- Secret service mail stolen from T-mobile
- 10,000 rec. letters stolen here at Stanford
- Don't these people know what they're doing?
Problem: Bad Code

- Actually, yes.
- Even security experts can't get it right.
- May 2006: Symantec AV 10.x remote exploit
  - Software deployed on 200,000,000 machines
  - Software without which machines also vulnerable (damned if you run it, damned if you don't)
- If Symantec can't even always get it right, what hope is there?
Solution: Give up

- Accept that software is untrustworthy
- Users willingly run malicious software
  - Malware, spyware, ...
- Even legitimate software is often vulnerable
- No sign that this problem is going away
  - Can't fix bugs as fast as we create them
- So make software less trusted
  - This is the goal of the HiStar OS
Example: Virus Scanner

- Can we eliminate trust in ClamAV?

- Goal: private files cannot go onto the network
Information Flow Control

- Goal: private files cannot go onto the network
Buggy scanner leaks private data

- Must restrict sockets to protect private data
Buggy scanner leaks private data

- Must restrict scanner's ability to use IPC
Buggy scanner leaks private data

- Must run scanner in chroot jail
Buggy scanner leaks private data

- Must run scanner with different UID
Buggy scanner leaks private data

- Must restrict access to /proc, ...
Buggy scanner leaks private data

- Must restrict FS'es that virus scanner can write
Buggy scanner leaks private data

- List goes on – is there any hope?
What's going on?

- Kernel not designed to enforce these policies
- Retrofitting difficult
  - Need to track potentially any memory observed or modified by a system call!
  - Hard to even enumerate
What's going on?

- Kernel not designed to enforce these policies

- Retrofitting difficult
  - Need to track potentially any memory observed or modified by a system call!
  - Hard to even enumerate
HiStar Solution

- Make all state explicit, track all communication
HiStar: Contributions

- Narrow kernel interface, few comm. channels
  - Minimal mechanism: enough for a Unix library
  - Strong control over information flow

- Unix support implemented as user-level library
  - Unix communication channels are made explicit, in terms of HiStar's mechanisms
  - Provides control over the gamut of Unix channels
HiStar kernel objects

Container (Directory)

- Segment (Data)
- Address Space
- Thread
- Gate (IPC)
- Device (Network)
HiStar kernel objects

Think of labels as a “tainted” bit
HiStar: Unix process

Process Container

Thread
Address Space
Code Segment
Data Segment
Unix File Descriptors

Process A

File Descriptor (O_RDONLY)

Process B

Kernel State
Unix File Descriptors

- Tainted process only talks to other tainted procs

![Diagram showing file descriptor (O_RDONLY) and kernel state](image)
Unix File Descriptors

- Lots of shared state in kernel, easy to miss
HiStar File Descriptors

Thread A

Address Space A

Thread B

Address Space B

File Descriptor Segment (O_RDONLY)
Seek pointer: 0xa32f
HiStar File Descriptors

- All shared state is now explicitly labeled
- Just need segment read/write checks
Taint Tracking Strawman

write(File)

Tainted Thread A

File

Thread B
Taint Tracking Strawman

- Propagate taint when writing to file
Taint Tracking Strawman

- Propagate taint when writing to file
- What happens when reading?

Read (File)
Taint Tracking Strawman

Thread A

File

Thread B

read(File)

Tainted

ACCESS
DENIED
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Thread B

Network

Secret = 1
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Network

write(File 1)

Secret = 1
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Thread B

Network

Secret = 1

read(File 0)
read(File 1)
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Thread B

Network

send email: “secret=1”

Secret = 1
Strawman has Covert Channel

- What if we taint B when it reads File 1?
Strawman has Covert Channel

- What if we taint B when it reads File 1?

Tainted Thread A

File 0 → Thread 0

File 1 → Thread 1

read(File 0)

read(File 1)

Network

Secret = 1
Strawman has Covert Channel

- What if we taint B when it reads File 1?

Tainted Thread A

File 0

Thread 0

File 1

Thread 1

Network

Secret = 1

send email: "secret=1"

send email: "secret=0"
HiStar: Immutable File Labels

- Label (taint level) is state that must be tracked
- Immutable labels solve this problem!
Who creates tainted files?

- Tainted thread can't modify untainted directory to place the new file there...
HiStar: Untainted thread pre-creates tainted file

- Existence and label of tainted file provide no information about A
Reading a tainted file

- Existence and label of tainted file provide no information about A
Reading a tainted file

- Existence and label of tainted file provide no information about A

Thread C

Thread A

Tainted File

Directory

readdir(): T. File's label

Untainted File

Thread B
Reading a tainted file

- Existence and label of tainted file provide no information about A
- Neither does B's decision to taint

Thread C

Thread A

Directory

Untainted File

Taint self

Thread B

Tainted File

Tainted File
HiStar avoids file covert channels

- Immutable labels prevent covert channels that communicate through label state
- Untainted threads pre-allocate tainted files
  - File existence or label provides no secret information
- Threads taint themselves to read tainted files
  - Tainted file's label accessible via parent directory
Problems with IPC

- IPC with tainted client
  - Taint server thread during request
Problems with IPC

- IPC with tainted client
  - Taint server thread during request

![Diagram of IPC with labeled components: Client Thread, IPC Port, IPC Return, Server Threads, Create, SELECT ...]
Problems with IPC

- **IPC with tainted client**
  - Taint server thread during request

![Diagram showing IPC issues with tainted client](image)
Problems with IPC

- IPC with tainted client
  - Taint server thread during request
  - Secrecy preserved?

Results
Problems with IPC

- IPC with tainted client
  - Taint server thread during request
  - Secrecy preserved?
- Lots of client calls
  - Limit server threads? Leaks information...
  - Otherwise, no control over resources!
Gates make resources explicit

- Client donates initial resources (thread)
Gates make resources explicit

- Client donates initial resources (thread)
- Client thread runs in server address space, executing server code
Gates make resources explicit

- Client donates initial resources (thread)
- Client thread runs in server address space, executing server code
Gates make resources explicit

- Client donates initial resources (thread)
- Client thread runs in server address space, executing server code
- No implicit resource allocation – no leaks
How do we get anything out?

- Network
- Virus Scanner
- Alice's Files
“Owner” privilege

- Yellow objects can only interact with other yellow objects, or objects with yellow star
- Small, trusted shell can isolate a large, frequently-changing virus scanner
Multiple categories of taint

- Owner privilege and information flow control are the only access control mechanism
- Anyone can allocate a new category, gets star
What about “root”? 

- Huge security hole for information flow control
  - Observe/modify anything – violate any security policy

- Make it explicit
  - Can be controlled as necessary
HiStar root privileges are explicit

- Kernel gives no special treatment to root

```
★ Alice's shell

Alice's Files

★ root's shell

★ Bob's shell

Bob's Files
```
HiStar root privileges are explicit

- Users can keep secret data inaccessible to root

Alice's shell

Bob's shell

Bob's Secret Files

Bob's Files

Alice's Files
What about inaccessible files?

- No one has privilege to access Bob's Secret Files.
HiStar resource allocation
HiStar resource allocation

- Create a new sub-container for secret files
HiStar resource allocation

- Create a new sub-container for secret files
HiStar resource allocation

- Create a new sub-container for secret files
- Bob can delete sub-container even if he cannot otherwise access it!
HiStar resource allocation

- Create a new sub-container for secret files
- Bob can delete sub-container even if he cannot otherwise access it!
HiStar resource allocation

- Create a new sub-container for secret files
- Bob can delete sub-container even if he cannot otherwise access it!
HiStar resource allocation

- Root has control over all resources, via the root container

- Bob's Files
- Bob's Container
- Bob's shell
- Root Container
- root's shell
Persistent Storage

• Unix: file system implemented in the kernel
  – Many potential pitfalls leading to covert channels: mtime, atime, link counts, ...
  – Would be great to implement it in user-space as well

• HiStar: Single-level store (like Multics / EROS)
  – All kernel objects stored on disk – memory is a cache
  – No difference between disk & memory objects
  – Eliminates need for trusted boot scripts
File System

- Implemented at user-level, using same objects
- Security checks separate from FS implementation
HiStar kernel design

• Kernel operations make information flow explicit
  – Explicit operation for thread to taint itself
    • Kernel never implicitly changes labels
  – Explicit resource allocation: gates, pre-created files
    • Kernel never implicitly allocates resources

• Kernel has no concept of superuser
  – Users can explicitly grant their privileges to root
  – Root owns the top-level container
Applications

- Many Unix applications
  - gcc, gdb, openssh, ...

- High-security applications alongside with Unix
  - Untrusted virus scanners (already described)
  - VPN/Internet data separation (see paper)
  - login with user-supplied authentication code (next)
  - Privilege-separated web server
Login on Unix

- Login must run as root
  - Only root can setuid() to grant user privileges

- Why is this bad?
  - Login is complicated (Kerberos, PAM, ...)
  - Bugs lead to complete system compromise
Login on HiStar

User: Bob
Pass: 1bob

Login Process

Alice's Auth. Service
PW: $H(\text{alic3})$

Bob's Auth. Service
PW: $H(1bob)$

- Each user can provide their own auth. service
Login on HiStar

- Each user can provide their own auth. service
Login on HiStar

Login Process

Pass: 1bob

Alice's Auth. Service
PW: H(alic3)

Bob's Auth. Service
PW: H(1bob)

OK
Password disclosure

- What if Bob mistypes his username as “alice”?
Password disclosure

- What if Bob mistypes his username as "alice"?
Avoiding password disclosure

- It's all about information flow
  - HiStar enforces:
    - “Password cannot go out onto the network”

- Details in the paper
Privilege-separated web server

- **launcher**
- **netd**
- **network**
- **SSLd**
- **RSAd**
- **private key**
- **httpd**
- **login**
- **user files**
- **application code**
Reducing trusted code

- HiStar lets developers to reduce trusted code
  - No code with every user's privilege during login
  - No trusted code needed to initiate authentication
  - 110-line trusted wrapper for complex virus scanner
  - Web server isolates different users' app. code

- Small kernel: <20,000 lines of code
Benchmarks, relative to Linux

Comparable performance to Linux and OpenBSD

Application-level benchmarks and disk benchmarks

- gcc
- wget
- Clam AV
- pipe
- disk read
- disk write
- create 10k files
- fork exec

Linux
HiStar
OpenBSD
HiStar allows use of group sync. Application either runs to completion, or appears to never start (single-level store)
Benchmarks, relative to Linux

7.5x slower

Linux: 9 syscalls per iteration
HiStar: 317 syscalls per iteration

Linux: fork exec 0.5
HiStar: fork exec 7.5
HiStar Conclusion

- HiStar reduces amount of trusted code
  - Enforce security properties on untrusted code using strict information flow control
- Kernel interface eliminates covert channels
  - Make everything explicit: labels, resources
- Unix library makes Unix information flow explicit
  - Superuser by convention, not by design
What about Asbestos?

- Different goal: Unix vs. specialized web server
  - HiStar closes covert channels inherent in the Asbestos design (mutable labels, IPC, ...)
  - Lower-level kernel interface
    - Process vs Container+Thread+AS+Segments+Gates
    - 2 times less kernel code than Asbestos
    - Generality shown by the user-space Unix library
  - System-wide support for persistent storage
    - Asbestos uses trusted user-space file server
  - Resources are manageable
    - In Asbestos, reboot to kill runaway process
How is this different from EROS?

- To isolate in EROS, must strictly partition the capabilities between isolated applications.
- Labels enforce policy without affecting structure.
  - Can impose policies on existing code (see paper).