Securing Untrustworthy Software Using Information Flow Control

Nickolai Zeldovich

Joint work with: Silas Boyd-Wickizer, Eddie Kohler, David Mazières
Problem: Bad Code

- PayMaxx divulges social security numbers
  - Sequential account number stored in the URL
  - First account had SSN 000-00-0000, no password
- CardSystems loses 40,000,000 CC numbers
- Secret service mail stolen from T-mobile
- 10,000 users compromised at Stanford (CDC)
- Don't these people know what they're doing?
Problem: Bad Code

- Even security experts can't get it right
- May 2006: Symantec AV 10.x remote exploit
  - Software deployed on 200,000,000 machines
  - Without this software, machines also vulnerable
  - You just can't win
- If Symantec can't get it right, what hope is there?
Solution: Give up

- Accept that software is untrustworthy
- Legitimate software is often vulnerable
- Users willingly run malicious software
  - Malware, spyware, ...
- No sign that this problem is going away
- Make software less trusted
Example: Virus Scanner

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

Goal: private files cannot go onto the network

Virus Scanner

Private User Files

/tmp

Virus Database

Update Process

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, ...

```
setproctitle: 0x6e371bc2
```

Update Process

- Private User Files
- /tmp
- Virus Database
- Network
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 control information flow

- 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 control information flow

- 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
  - Overall theme: make everything explicit

- 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
Unix File Descriptors

Process A -> Process B

File Descriptor (O_RDONLY)

Kernel State
Unix File Descriptors

- Tainted process only talks to other tainted procs

![Diagram showing Unix File Descriptors]

- Process A
- Process B
- File Descriptor (O_RDONLY)
- Kernel State

Note: Tainted process only talks to other tainted procs.
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
- Reduce problem to object read/write checks
Taint Tracking Strawman

- Tainted Thread A
- File
- Thread B

write(File)
Taint Tracking Strawman

- Propagate taint when writing to file
Taint Tracking Strawman

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

Tainted Thread A → File → Thread B

read(File)
Taint Tracking Strawman

Thread A

Tainted

File

ACCESS DENIED

Thread B

read(File)
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

Secret = 1

write(File 1)
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Thread B

read(File 0)
read(File 1)

Secret = 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?

```
read(File 0)
read(File 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

Network

send email: “secret=1”

send email: “secret=0”

Secret = 1
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

Wrapper
Thread C

Tainted
Thread A

Directory

Untainted
File

Tainted
File

Thread B

readdir(): T. File's label
Reading a tainted file

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

Wrapper Thread C

Thread A

Tainted File

Directory

Untainted File

Taint self

Thread B

Tainted File

Tainted Thread A
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

```sql
SELECT ...
```
Problems with IPC

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

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

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

Results

Time
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

Alice's Files

Virus Scanner

Network
“Owner” privilege

- Star can get around information flow restrictions
- 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
HiStar root privileges are explicit

- Kernel gives no special treatment to root
HiStar root privileges are explicit

- Users can keep secret data inaccessible to root
What to do with inaccessible files?

- No one has privilege to access Bob's Secret Files.
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: root container
- Remove recalcitrant users
Persistent Storage

- Unix: file system implemented in the kernel
  - Potential covert channels: mtime, atime, link count, ...

- HiStar: Single-level store (like Multics / EROS)
  - All kernel objects stored on disk
  - Memory is just a cache of disk objects
Single-level store

% ssh root@histar
HiStar#
Single-level store

% ssh root@histar
HiStar# reboot
Single-level store

% ssh root@histar
HiStar# reboot
rebooting...

Kernel checkpoints to disk:
- Threads
- Address spaces
- Segments (memory)
- ...
and then reboots machine
Single-level store

% ssh root@histar
HiStar# reboot
rebooting...
done
HiStar#

Kernel checkpoints to disk:
• Threads
• Address spaces
• Segments (memory)
• ...
and then reboots machine

Kernel boots up, reads in:
• Threads
• Address spaces
• Segments (memory)
• ...
and continues as before!
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
  – login with user-supplied authentication code (next)
  – Privilege-separated web server
Login on Unix: highly centralized

- Difficult and error-prone to extend login process
  - Any bugs can lead to complete system compromise!

```
/etc/shadow:
Alice: H(alic3)
Bob: H(1bob)
```

User: Bob
Pass: 1bob

Login Process (runs as root)

/etc/shadow:
Login on HiStar: less trusted code

- Login process requires no privileges
- Each user can provide their own auth. service
Login process requires no privileges
Each user can provide their own auth. service
Login on HiStar: less trusted code

Pass: 1bob

OK

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

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

Login Process
Login on HiStar: less trusted code

- No code runs with every user's privilege

- Users supply their own authentication code
  - Password checker, one-time passwords, ...

- OS ensures password is not disclosed
  - Even if user mistypes username, gives password to attacker's authentication code (not described)
HiStar SSL Web Server

- Only small fraction of code (green) is trusted

```
User's browser
inetd
310 lines
User authentication
User data
RSA key
```
HiStar SSL Web Server

- Only small fraction of code (green) is trusted

![Diagram showing HiStar SSL Web Server components and trusted code]
HiStar SSL Web Server

- OpenSSL only trusted to encrypt/decrypt
HiStar SSL Web Server

- OpenSSL cannot disclose certificate private key
HiStar SSL Web Server

- httpd trusted with user's privilege, credentials
HiStar SSL Web Server

- Application code cannot disclose user data
HiStar allows developers to reduce trusted code

- No code with every user's privilege during login
- No trusted code to initiate authentication
- 110-line trusted wrapper for large virus scanner
- Web server isolates different users' app code

- Small kernel: under 20,000 lines of code
HiStar controls one machine

- Can enforce security for small web server

```
Web Server

httpd

Application code

User data
```
Large services are distributed

- Must use multiple machines for scalability
  - Tainted processes cannot use network in HiStar

Front-end Server

httpd

Application Server

Application code

Data Server

User data

?
Problem: Who can we trust?

- No single fully-trusted kernel to make decisions

```
Front-end Server
  httpd

Application Server
  Application code

Data Server
  User data

Attacker's Server
```
Globally-trusted authority?

- Made sense for local kernel (HiStar), but not here
  - Problems with scalability, security, trust

Front-end Server

httpd

Application code

User data

Data Server

Attacker's Server

Global Network Authority?
Decentralized design

• When it is safe to contact another machine?
  – Any query may leak information to attacker!

httpd

Local Authority

Application code

Local Authority

User data

Local Authority

Attacker's Server
Solution: Self-authenticating categories

- Category (taint color) is a public key $C$
- If you know private key $C^{-1}$, you own (“star”) $C$
- To trust host $H$ with your secret data, sign delegation ($H$ is trusted to handle $C$) using $C^{-1}$
- Category can “speak for itself”
Naming machines: Strawman

Category C

“Trusts”
Signed by C^{-1}

Hostname H.com
Naming machines: Strawman

Category C

Hostname H.com

Signed by C

Trusts

Host key K

Has key

Verisign

Has IP

DNS

IP address 1.2.3.4
Naming machines: Strawman

- Can we reduce trust of Verisign, DNS?
Name hosts by public key

- Trust the public key instead of the hostname!

Category \( C \) \( \rightarrow \) "Trusts" \( \rightarrow \) Host key \( K \)

Signed by \( C^{-1} \)

Hostname \( H.com \)

"Has key" Verisign

"Has IP" DNS

IP address 1.2.3.4
Hosts sign their IP address

- Design separates trust from distribution, policy

Category C

"Trusts"

Signed by C⁻¹

Hostname H.com

Signed by K⁻¹

"Has key"

Verisign

"Has IP"

DNS

My IP is

IP address 1.2.3.4
Exporter daemons

- HiStar enforces information flow locally
- Exporters send UDP-like messages with labels
  - Not part of kernel – only in TCB for distributed apps
  - Need delegations to determine if recipient is trusted
Strawman: Exporter stores delegations

- Delegation: User trusts host X with his data
Strawman:
Exporter stores delegations

- Delegation: User trusts host X with his data
Strawman has covert channel

Private User Files

Attacker Process

Exporter

Delegations: Host X: "★"
Strawman has covert channel

Private User Files

File Server

Attacker Process

Exporter

Delegations: Host X: “★“
Strawman has covert channel

Private User Files

File Server

Attacker Process

2\textsuperscript{nd} attacker Process

Exporter

Delegations: Host X: “\textstar"
Strawman has covert channel

Private User Files

File Server

Attacker Process

2nd attacker Process

Exporter

Delegations: Host X: “★”
Strawman has covert channel

Private User Files

File Server

1st bit

Attacker Process

Exporter

2nd attacker Process

Delegations: Host X: ““
Strawman has covert channel

Private User Files

1st bit

File Server

Attacker Process

2nd attacker Process

Exporter

Delegations:
Host X: “★”
Host Y: “★★”
Strawman has covert channel

Private User Files

Attacker Process

File Server

Exporter

Delegations:
Host X: “★”
Host Y: “★★”

Send to Y

2nd attacker Process
Solution: Stateless exporter

- Delegations are self-authenticating
Sender supplies delegations

- Result only depends on sender-supplied data
Exporter's interface

- void send(ip_address, tcp_port, wire_message, delegation_set)

- struct wire_message {
    pubkey recipient_exporter;
    slot recipient_slot;
    category_set label;
    category_set grant_ownership;
    delegation_set dset;
    opaque data;
};
Exporter's interface

- void send(ip_address, tcp_port, 
  wire_message, delegation_set)

- struct wire_message {
  pubkey recipient_exporter;
  slot recipient_slot;
  category_set label;
  category_set grant_ownership;
  delegation_set dset;
  opaque data;
};

Convince sending exporter it's safe to send message:

Category delegations + Address delegation (secrecy)
Exporter's interface

- void send(ip_address, tcp_port, wire_message, delegation_set)

- struct wire_message {
  pubkey recipient_exporter;
  slot recipient_slot;
  category_set label;
  category_set grant_ownership;
  delegation_set dset;
  opaque data;
};

Convince recipient exporter it's safe to accept message:

Category delegations
(integrity)
RPC using exporter messages

• Much like RPC over UDP
  – Allocate resources to receive the reply
  – Send the request
  – Wait for reply message to arrive
  – Periodically retransmit or time out

• RPC library manages delegations
  – Untrusted by OS, exporters
Security details

• All messages encrypted+MAC on the network
  – Session keys between each pair of exporters

• Ownership and address delegations expire
  – Compromised machine only affects recent users
  – Exporters periodically broadcast address delegations

• Trusted exporter: 3,700 lines of C++ (plus libs)
  – Enforces policy on arbitrary untrusted code
Incremental deployment

- Run untrusted perl code on HiStar, from Linux
  - Well-defined security properties specified by label
Recall: HiStar SSL Web Server

- Only small fraction of code (green) is trusted

![Diagram showing HiStar SSL Web Server components]

- User's browser
- inetd: 310 lines
- SSL: 340K lines
- RSAAd: 4600 lines
- RSA key: 680K lines: PDF maker
- httpd: 300 lines
- Application code
- User authentication
- User data
Scalable, Distributed Web Server

- Same security properties (but trust exporters)

![Diagram showing network components and security levels with line counts: 310 lines for inetd, 340K lines for SSL, 4600 lines for RSAd, 300 lines for httpd, and RSA key with trust levels indicated. User's browser connects to inetd, which then connects to SSL, RSAd, and httpd. Application code and user data are also shown with trust levels.]
Conclusion

- Shown how to reduce amount of trusted code
  - Trusted: 20,000 line kernel + 3,700 line exporter
  - Enforce security of arbitrary distributed application
- Explicit information flow removes covert channels
  - Even root privileges can be made explicit
- No need for globally-trusted authority
  - Self-authenticating categories make trust explicit

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

• Hard to enforce correctness, progress
  – Malicious code cannot leak your data
  – But if you give it write access, it can corrupt it!

• Applicable to servers, not obvious for desktops
  – May need to provide trusted path to and from user

• Fine-grained isolation requires code changes
  – Code not always structured along information flow

• Covert channels are inevitable
Potential ways to reduce covert channels

• One idea: “secure” scheduler for sensitive data
  – Preempt based on instruction counts instead of time
  – Prohibit process from yielding CPU to others

• Only incur overhead for, e.g. checking password
  – Spend a deterministic 0.1 sec CPU time for login
Verifying security

• Verifying the design
  – Can objectively determine if something is safe
  – Model-checking subset of syscalls (Taral Joglekar)
    • Seems to provide non-interference

• Verifying the implementation
  – Symbolic execution (Peter Pawlowski, Daniel Dunbar)
    • Found two bugs in HiStar (and a few more in EXE)
  – Static taint analysis (Suhabe Bugrara, Peter Hawkins)
    • No user pointer derefs (where alias analysis terminates)
How to really reboot?

- Separate command called “ureboot”
- Kills all processes except itself (*ureboot*)
  - Delete containers, except for the file system
  - FS containers have special bit that excludes threads
- Start a new *init* process
  - It will start everything else (TCP/IP stack, sshd, …)
Benchmarks, relative to Linux

Comparable performance to Linux and OpenBSD
Application-level benchmarks and disk benchmarks

Linux
HiStar
OpenBSD
Benchmarks, relative to Linux

217x faster!
Synchronous creation of 10,000 files

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

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

Linux
HiStar
OpenBSD
Web server: “PDF maker” app

Throughput on one server, req / second

- Linux
- Apache
- Unified
- Separated
- Distributed
Web server: “PDF maker” app

Throughput on one server, req / second

Scalability of application servers
(Fixed number of other servers)

- Linux
- Apache
- Unified
- Separated
- Distributed
Related Work

- Asbestos inspired this work
- Capability-based systems: KeyKOS, EROS
- Distributed capability systems: Amoeba
- Language-based security: Jif, Joe-E
Asbestos: Built for a 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
Labels vs capabilities

- Both provide strong isolation

- Capabilities: determine privilege before starting
  - Restricts program structure

- Labels: can change privilege levels at runtime
  - Thread can raise label to read a secret file
  - Label change prevents writing to non-secret files
  - Easier to apply to existing code
Labels in a capability OS

Capability Wrapper → Process A → Capability Wrapper

A's label
Distributed Capabilities (Amoeba)

- Servers require properly-signed capabilities

- Attacker cannot make up arbitrary capabilities
  - Must authenticate to access user's file server

- Attacker can create capabilities for his server
  - Cannot prevent code from “calling home”
Language-based security

- Much more fine-grained control
- Resource allocation covert channels hard to fix
- Many similar problems in structuring code

  - if (secret == 1)
    foo();
    printf("Hello world.\n");
  - If secret is tainted, foo runs tainted
  - printf only runs if foo terminates
  - Must prove halting to remove taint on thread