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
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
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 email stolen from T-mobile
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 email stolen from T-mobile
- 10,000 students data compromised at Stanford
- 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 largely untrustworthy
- Legitimate software is often vulnerable
- Users willingly run malicious software
  - Malware, spyware, ...
- No sign that this problem is going away
- Reduce the amount of trusted software
  - Focus on **security of data**, not security of code
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

- Virus Scanner
- /tmp
- Virus Database
- Update Process
- Network

- Private User Files
- fcntl locking

- 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 design

• Narrow kernel interface
  – Simple objects with simple, well-defined operations
  – Strictly control information flow for every operation
  – Overall approach: *make everything explicit*

• Unix support implemented as user-level library
  – Composes safe kernel operations to implement Unix
  – Composing is safe, since information flow is transitive
  – 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 ➔ 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
Unix File Descriptors

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

Thread A

Address Space A

File Descriptor Segment (O_RDONLY)
Seek pointer: 0xa32f

Thread B

Address Space B
HiStar File Descriptors

- All shared state is now explicitly labeled
- Reduce problem to object read/write checks
Taint Tracking Strawman

write(File)

Tainted Thread A

File

Thread B
Taint Tracking Strawman

- Propagate taint when writing to file

write(File)

Tainted Thread A → File → Thread B
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

File

Thread B

read(File)

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

write(File 1)

Secret = 1

File 0

File 1

Thread B

Network
Strawman has Covert Channel

Tainted Thread A

File 0

File 1

Thread B

read(File 0)
read(File 1)

Network

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)
```

Network

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
Reading a tainted file

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

Thread C

Tainted Thread A

Directory

Untainted File

Thread B

Taint self

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

```
CREATE ...
SELECT ...
```

![Diagram showing IPC with tainted client]

- IPC with tainted client
  - Taint server thread during request

```
CREATE ...
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
  - Secrecy preserved?

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Client Thread</th>
<th>IPC Port</th>
<th>Results</th>
<th>Create</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Server Threads

DB Server
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?
“Owner” privilege

- Star can get around information flow restrictions
- Small, trusted shell can isolate a large, frequently-changing virus scanner

Diagram:
- Alice's Files
- Virus Scanner
- Alice's shell
- Network
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
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
  – 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
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
% 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
How to really reboot?

- Separate command called “ureboot”

- Kills all processes except itself (`ureboot`)
  - Delete all containers, except for the file system
  - FS containers have special bit that excludes threads

- Starts a new `init` process
  - It will start everything else (TCP/IP stack, sshd, ...)

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: highly centralized

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

User: Bob
Pass: 1bob

Login Process (runs as root)

/etc/shadow:
Alice: H(alic3)
Bob: H(1bob)
Login on HiStar: less trusted code

- Login process requires no privileges
- Each user can provide their own auth. service

User: Bob
Pass: 1bob

Alice's **Auth. Service**
PW: H(atic3)

Bob's **Auth. Service**
PW: H(1bob)
Login on HiStar: less trusted code

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

Login Process

Pass: 1bob

Alice's Auth. Service

PW: H(alic3)

Bob's Auth. Service

PW: H(1bob)

OK
Login on HiStar: less trusted code

- No code runs with every user's privilege
  - Each user trusts their 300-line authentication agent

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

- OS ensures password is not disclosed
  - Even if user mistypes username and gives password to a malicious authentication agent (see paper)
HiStar SSL Web Server

- Only small fraction of code (green) is trusted

310 lines

User's browser

inetd

RSA key

User authentication

User data
HiStar SSL Web Server

- Only small fraction of code (green) is trusted
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

User's browser

inetd

SSL

RSAd

RSA key

httpd trusted with user's privilege, credentials

User authentication

User data
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

- How to track information flow across machines?
Problem: Who can we trust?

- No single fully-trusted kernel to make decisions
Globally-trusted authority?

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

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

![Diagram of decentralized design](image)
Solution: Users define trust with self-certifying categories

- Category (taint color) is a public key $C$

- To trust host $H$ with your secret data, sign delegation certificate ($H$ trusted with $C$) using $C^{-1}$

- Anyone can verify delegation certificates based on the category name (public key)
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

File Server

Attacker Process

Exporter

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

Attacker Process

2nd attacker Process

Delegations: Host X: “☆”

Exporter
Strawman has covert channel

Private User Files

File Server

1st bit

Attacker Process

2nd attacker Process

Exporter

Delegations: Host X: "★"

1st Process

2nd attacker Process
Strawman has covert channel

Private User Files

1st bit

File Server

Attacker Process

Delegate to Y

Exporter

Delegations:
Host X: “★”
Host Y: “★”

2nd attacker Process
Strawman has covert channel

Private User Files

File Server

Attacker Process

2nd attacker Process

Delegations:
Host X: “★”
Host Y: “☆”

Exporter

Send to Y
Solution: Stateless exporter

- Delegations are self-authenticating
Sender supplies delegations

- Result only depends on caller-supplied data
  - Verify certificate signatures using category pub. key
Recall: HiStar SSL Web Server

- Only small fraction of code (green) is trusted
Scalable, Distributed Web Server

- Same security properties (except trust exporters)
  - No fully-trusted machines: limits effect of compromise

![Diagram showing components and security properties]

- User's browser
- inetd
- SSL
- RSAd
- RSA key
- httpd
- Application code
- User authentication
- User data
Conclusion

• HiStar reduces amount of trusted code
  – Safely run untrusted code on confidential data
• Kernel interface eliminates covert channels
  – Make everything explicit: labels, resources
• Unix library makes Unix information flow explicit
  – Superuser by convention, not by design
• No fully-trusted machines in distributed system

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

Linux
HiStar
OpenBSD
Benchmarks, relative to Linux

7.5x slower

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

- Linux
- Apache
- Unified
- Separated
- Distributed

Scalability of application servers (Fixed number of other servers)

1 2 3
Language-based security

- Much more fine-grained control
- Resource allocation covert channels hard to fix
- 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 that foo halts to remove taint on thread
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