Building secure systems from buggy code with information flow control

Nickolai Zeldovich

Why is it hard to build secure systems?

- A *single* bug in almost any line of code can lead to a compromise
  - Simple errors in web applications disclose private data of millions of users
  - Even security software has bugs (Symantec anti-virus exposed 200M hosts to attack)

Current strategy

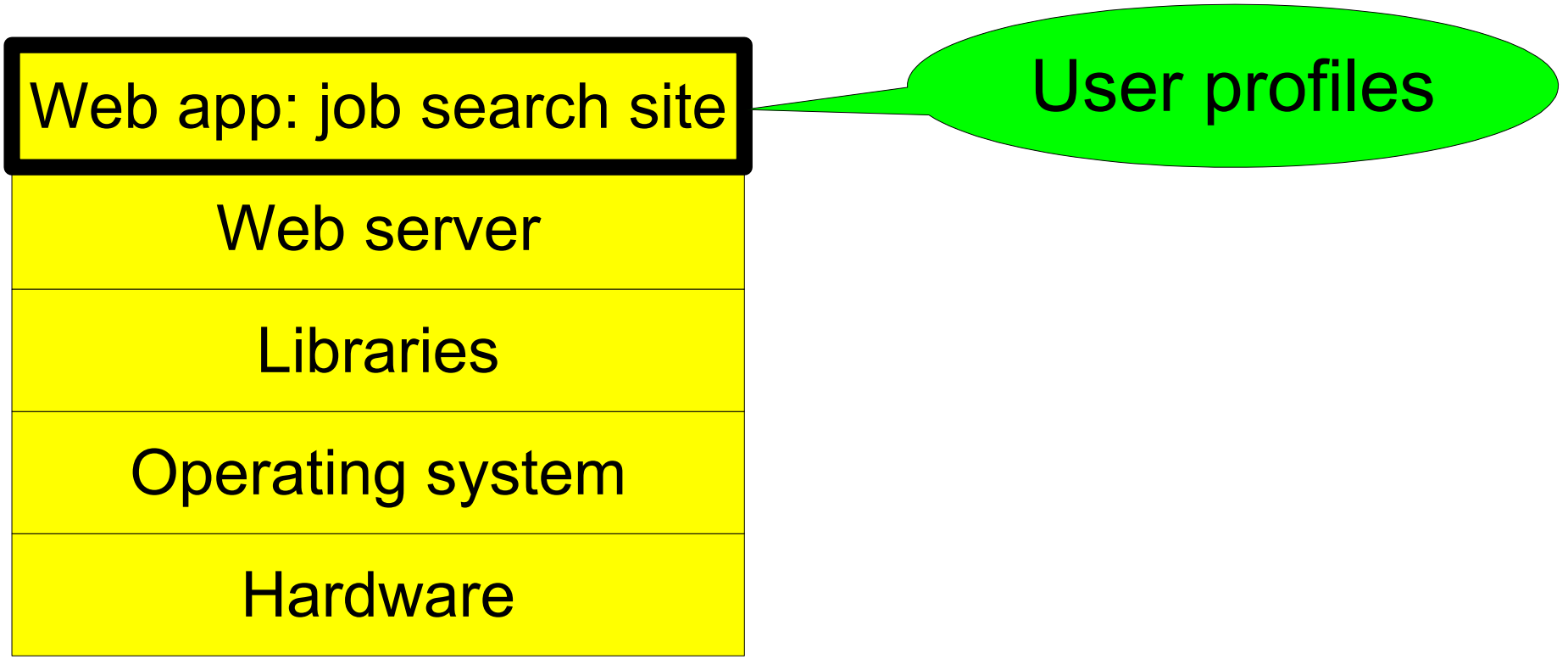
- Lots of techniques for finding and fixing bugs:
  - Buffer overflows
  - Format string bugs
  - SQL injection
  - Integer overflow
  - Temporary file races
  - Missing access checks

Current strategy

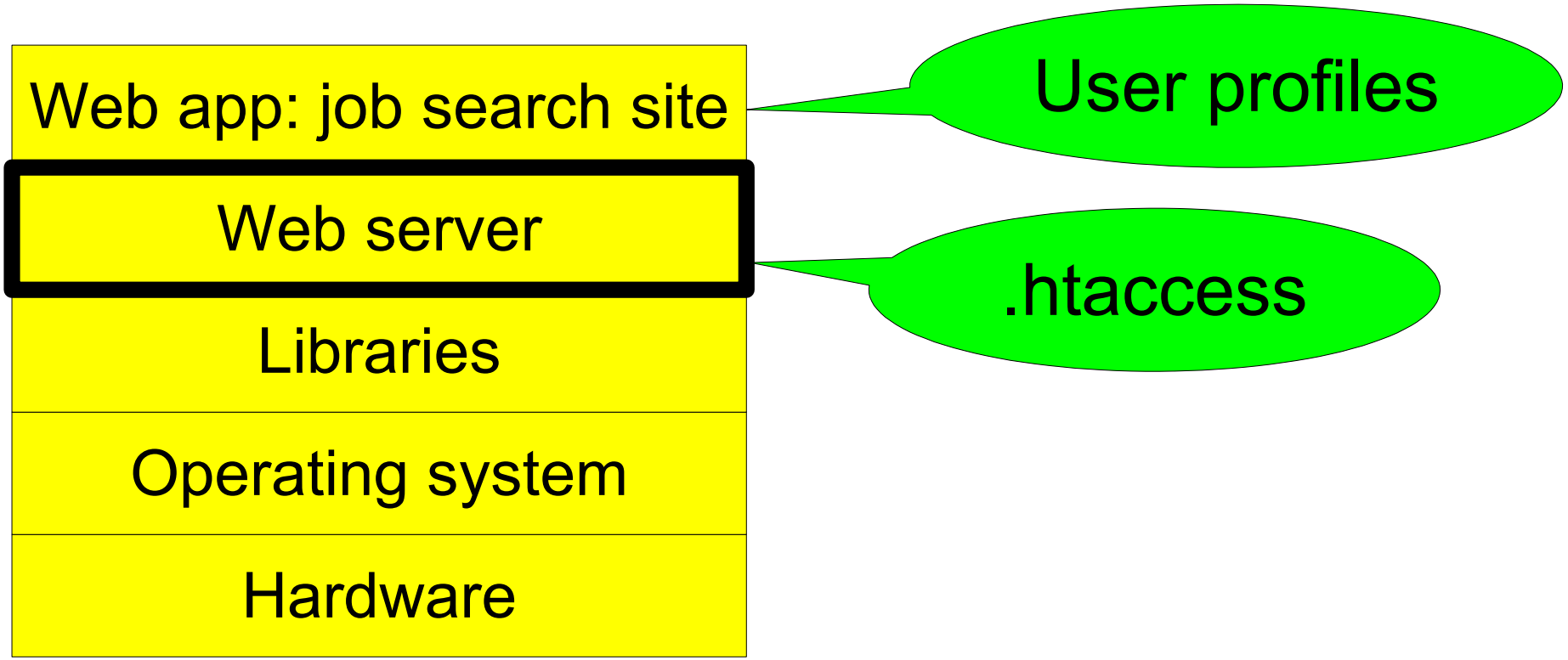
- Lots of techniques for finding and fixing bugs:
  - Buffer overflows
  - Format string bugs
  - SQL injection
  - Integer overflow
  - Temporary file races
  - Missing access checks
- Arms race: who finds the next bug?
  - Experience shows it's impossible to eliminate *all* bugs
- Not sustainable: too risky/costly in the long run!

Example:
Security in a web application

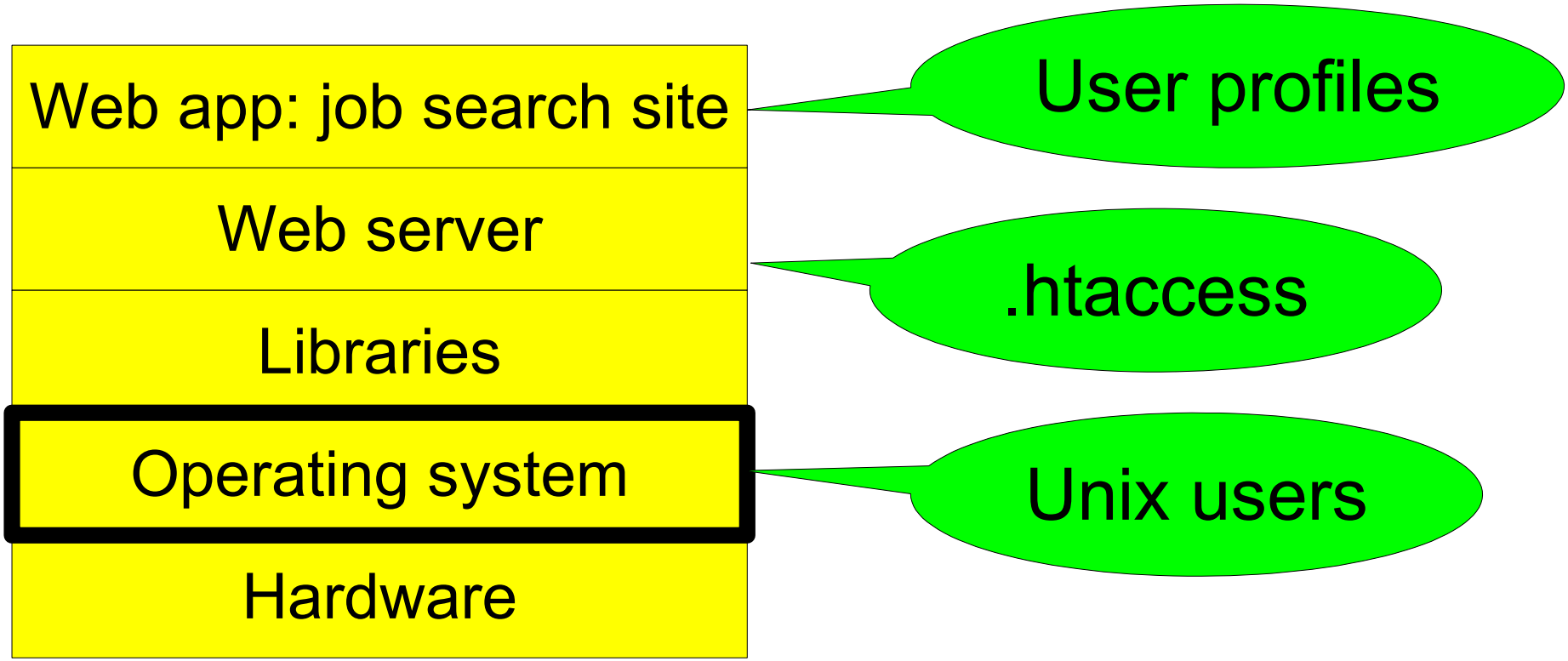
Application enforces security of user profiles



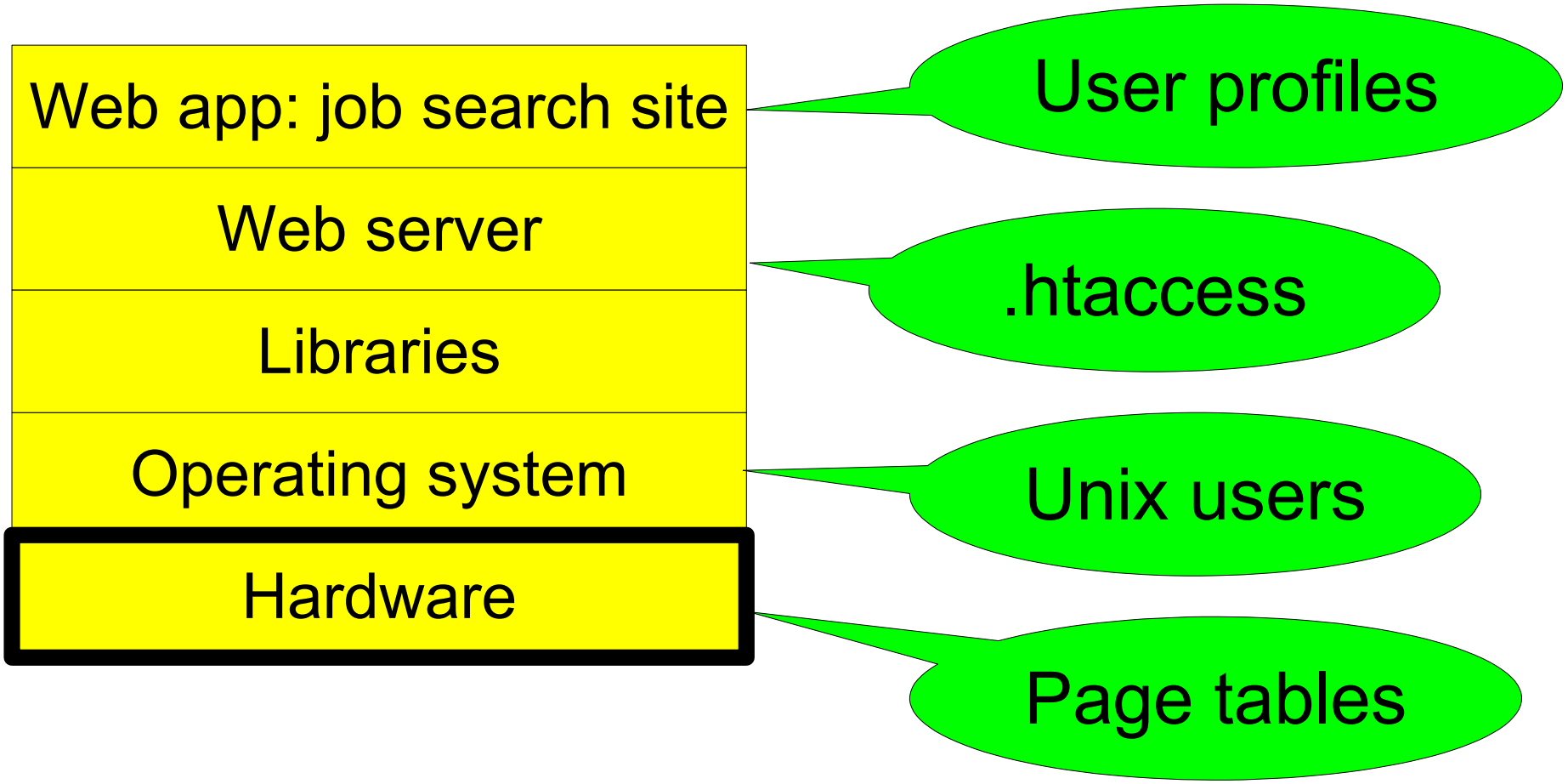
Web server controls who can make what HTTP request



Operating system protects Unix users from each other

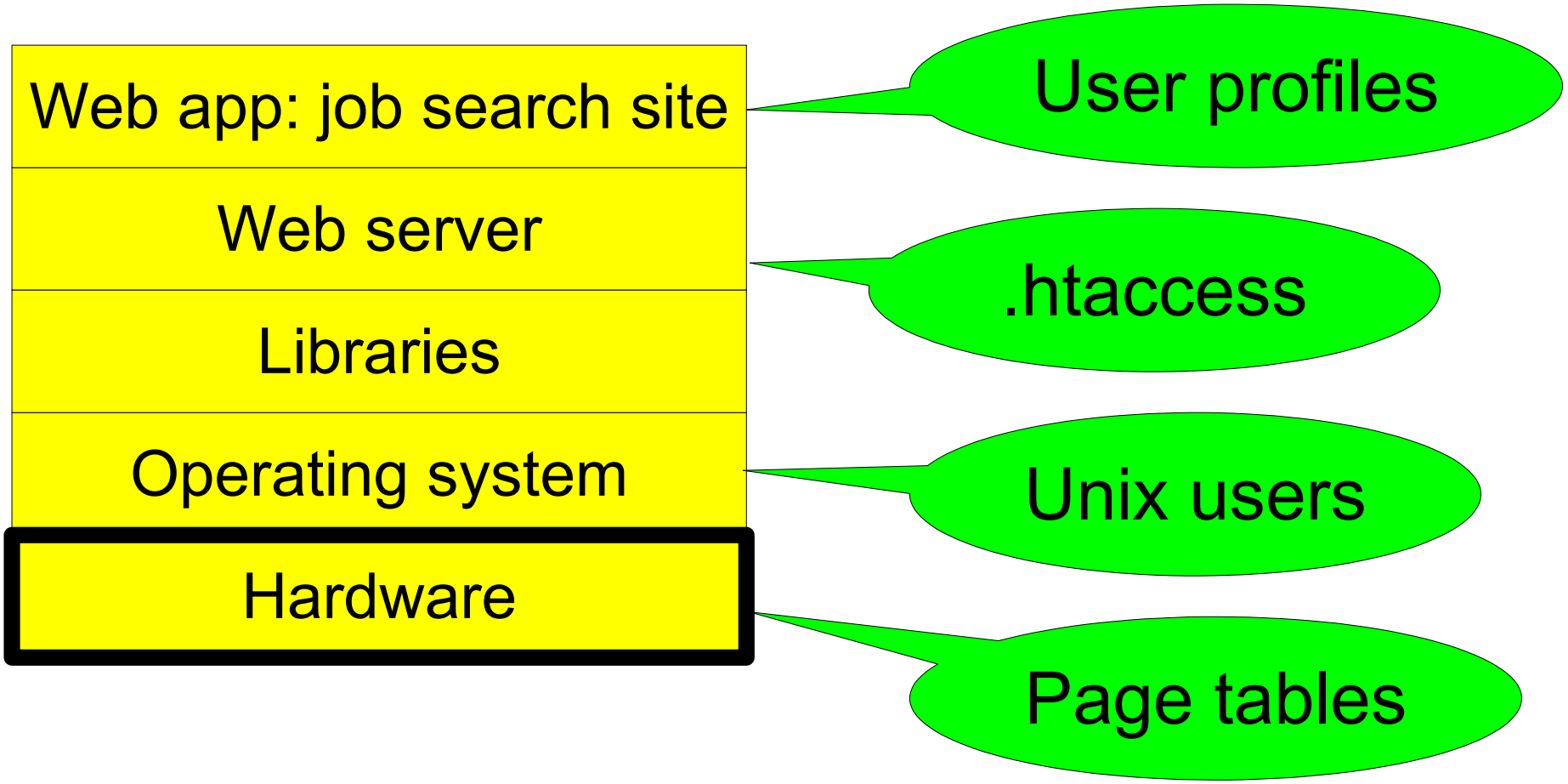


Hardware protects kernel from applications



Security depends on code at every layer being correct

Even if hardware and OS were perfect, application could still be compromised!



Security depends on code at every layer being correct

No way we can get *millions* of bug-free lines of code

Millions of lines, third-party code

Apache: 300,000 lines of code

OpenSSL: 340,000 lines of code

Linux: 570,000+ lines of code

What can we do?

- As long as security is about code being correct, fixing code is the only answer
  - Unfortunately, this approach is doomed
  - Experience shows perfect code is impossible

- This talk: how to build secure systems despite bugs in most code

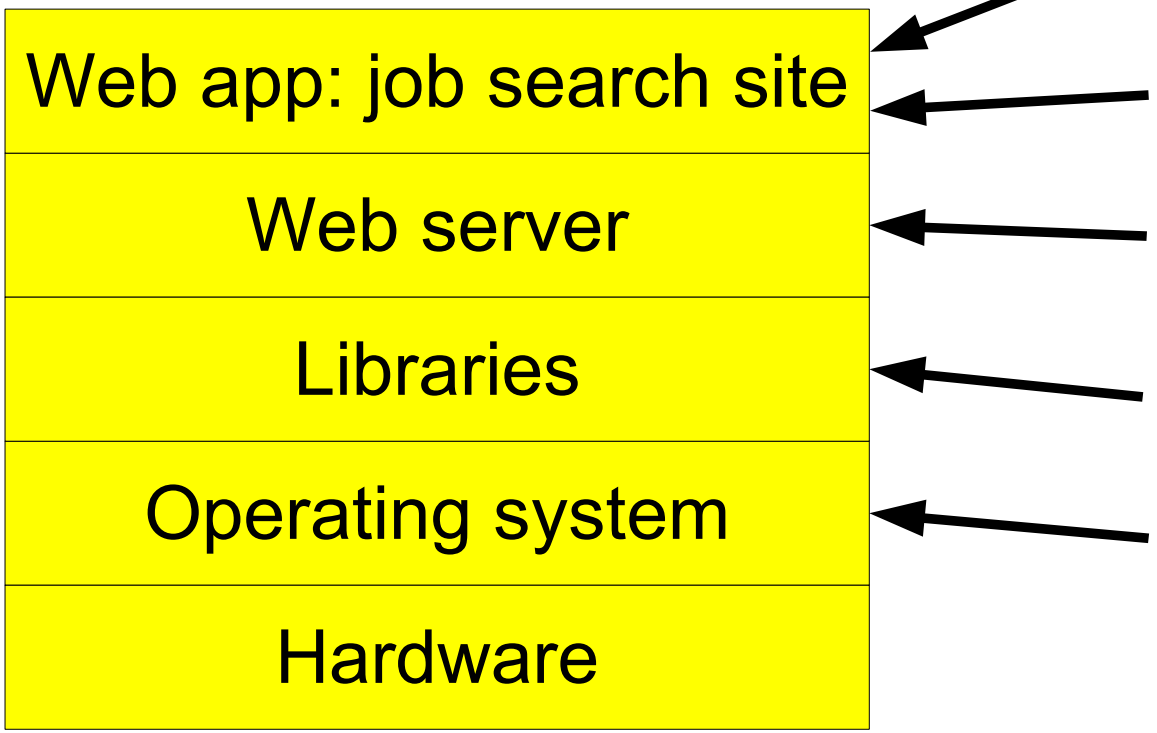
Step back and rethink security

- Most security concerns relate to data movement
  - My financial data cannot be sent over the network
  - My password shouldn't be disclosed to anyone
  - User's profile can only be sent to his/her browser
  - You can sign with this private key, but not reveal it
  - Other users can't read or write my files

- Suppose we could control data movement
  - Doesn't matter what code does, if data is secure
  - Achieves our goal: security despite buggy code

Currently, policies enforced by code all over the place

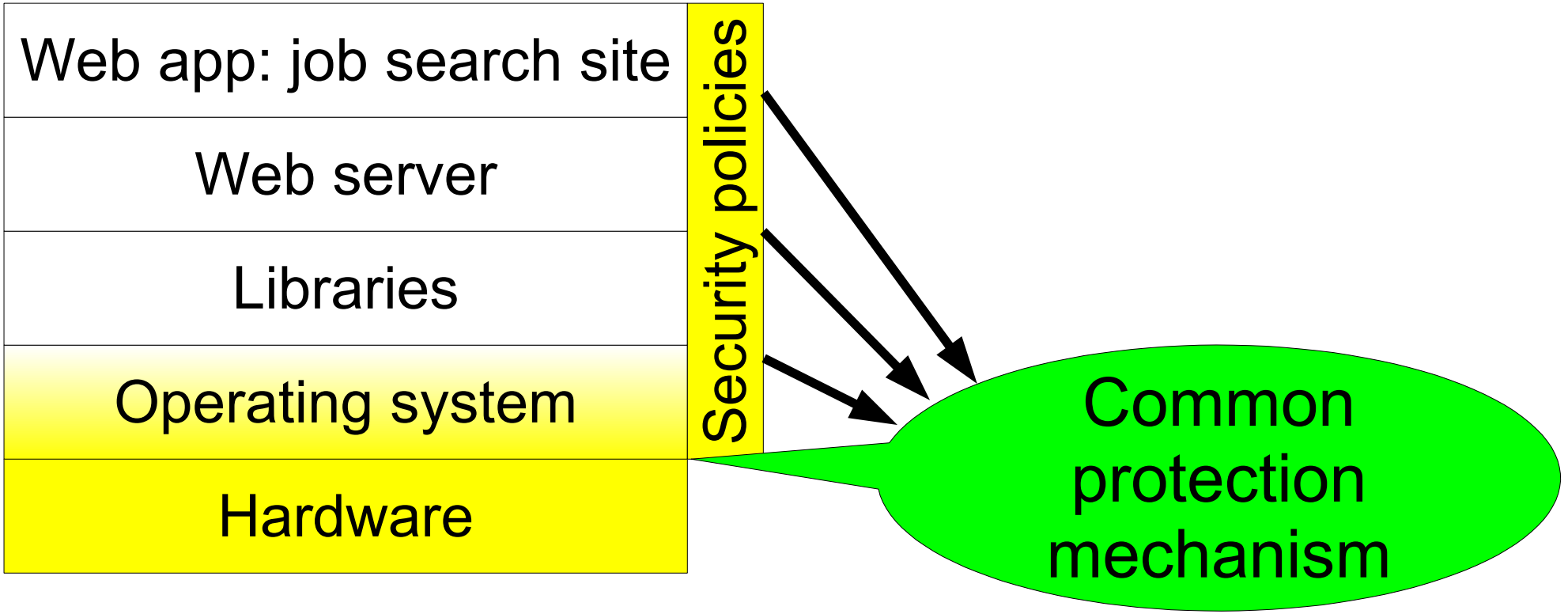
- Code enforcing security must be **trusted = yellow**



- User's profile can only be sent to his/her browser
- My financial data cannot be sent over the network
- My password shouldn't be disclosed to anyone
- You can sign with this private key, but not reveal it
- Other users can't read or write my files

Goal:
Building block for security

- Code enforcing security must be **trusted = yellow**
- Provide common mechanism apps can use



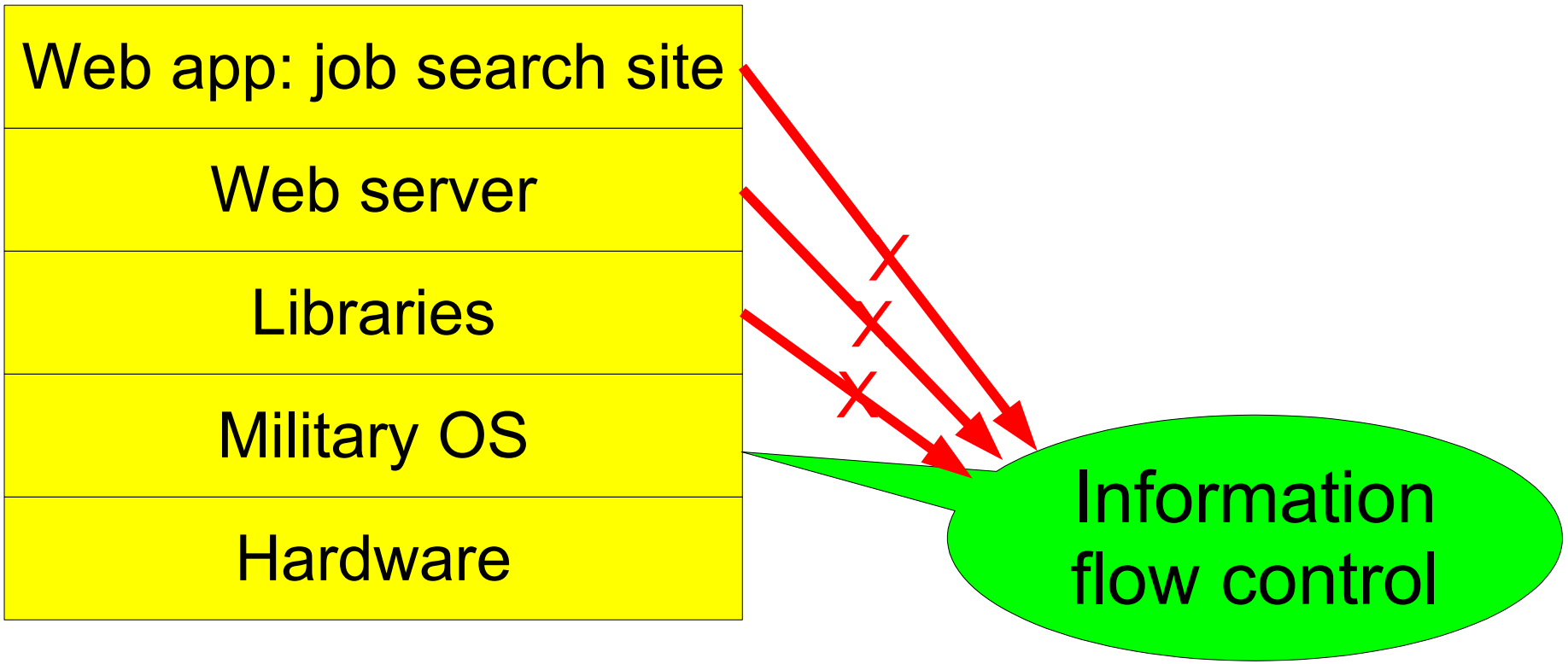
This talk: common mechanism should control data movement

- Data movement works across layers
  - Same data in memory pages, files, user profiles
  - High-level data movement policy translates to low-level OS/HW mechanism that can enforce it
- Allow building secure systems despite buggy code!
  - 100 lines of code will enforce security for complex apps

Information flow control (IFC)

[Bell-LaPadula '73, Biba '77, Denning '75]

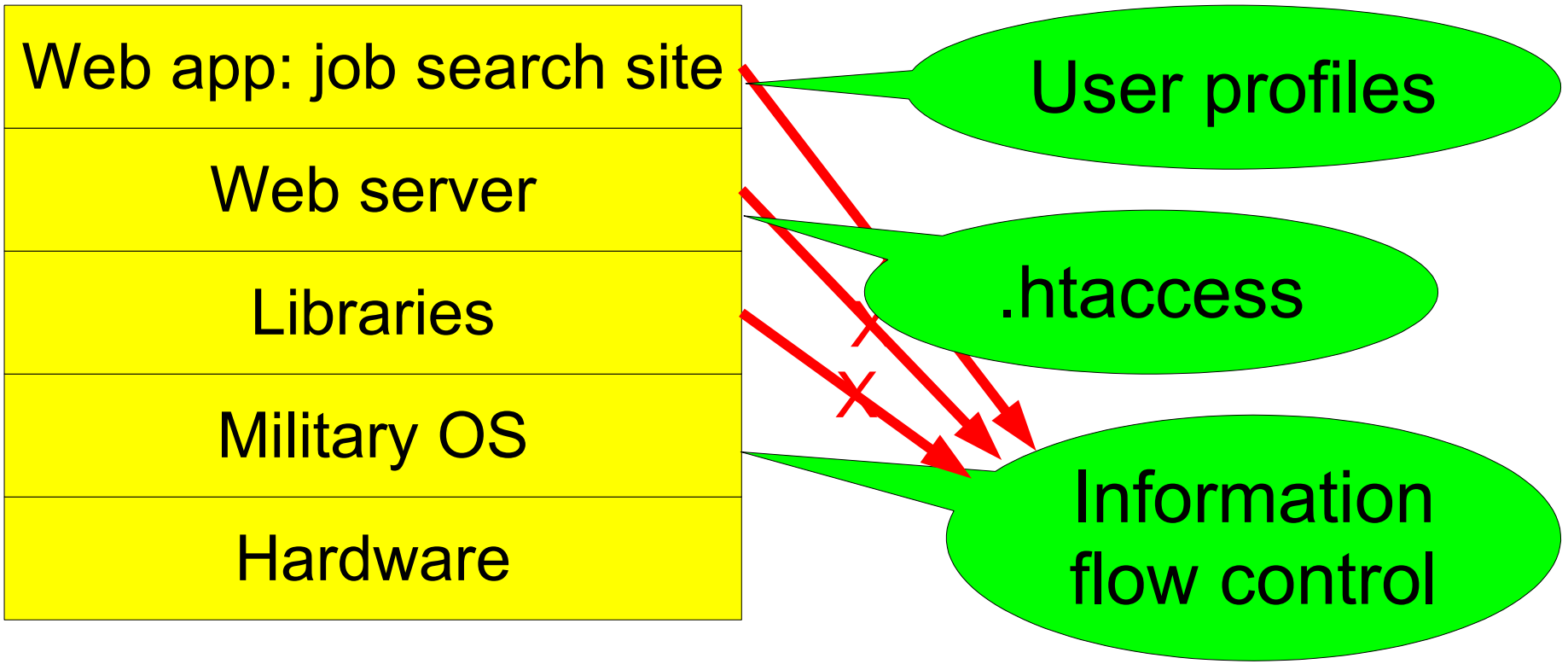
- Military systems [IX, Adept-50, KeySafe, VMM SecKern]
  - Top-secret process can't write unclassified files



Information flow control (IFC)

[Bell-LaPadula '73, Biba '77, Denning '75]

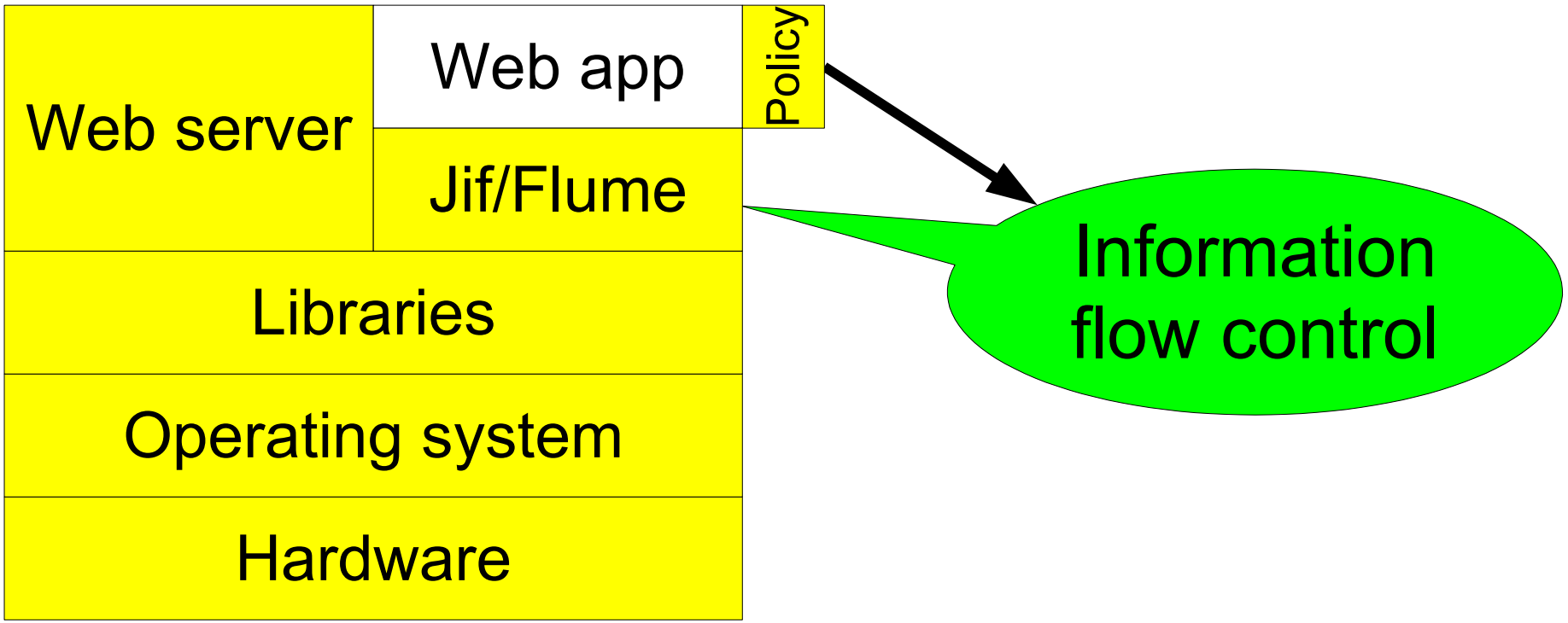
- Military systems [IX, Adept-50, KeySafe, VMM SecKern]
  - Top-secret process can't write unclassified files



Information flow control (IFC)

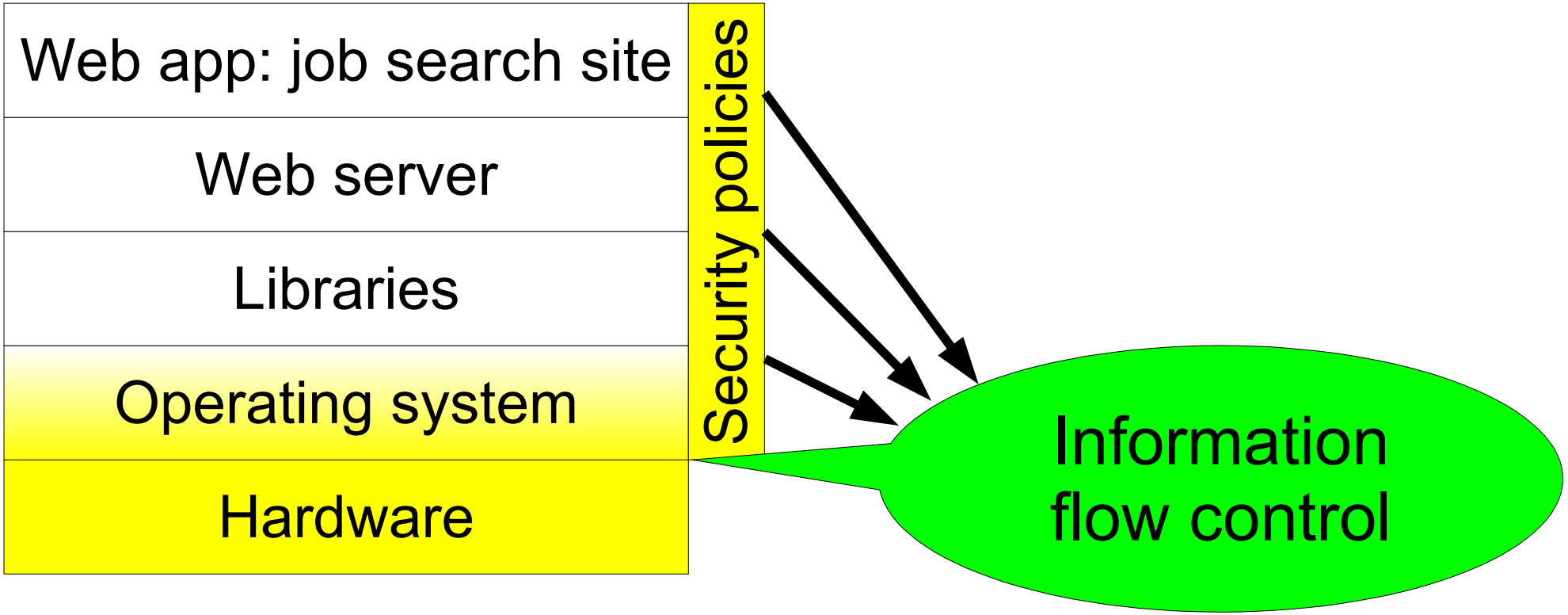
[Bell-LaPadula '73, Biba '77, Denning '75]

- More recent systems Jif [Myers '01], Flume [Krohn '07]
  - Still lots of yellow code to get right – millions LoC



This talk: IFC should be fundamental mechanism

- All other protection can then be built on top of IFC
- IFC enforced in 20,000 line kernel or in hardware



One mechanism will be used for security by everyone

- Information flow control mechanism
  - Associate a *label* with data
    - Applies to data at all levels of abstraction
  - Labels follow data when it moves around
  - Labels specify what can happen to the data, regardless of how many times it moves

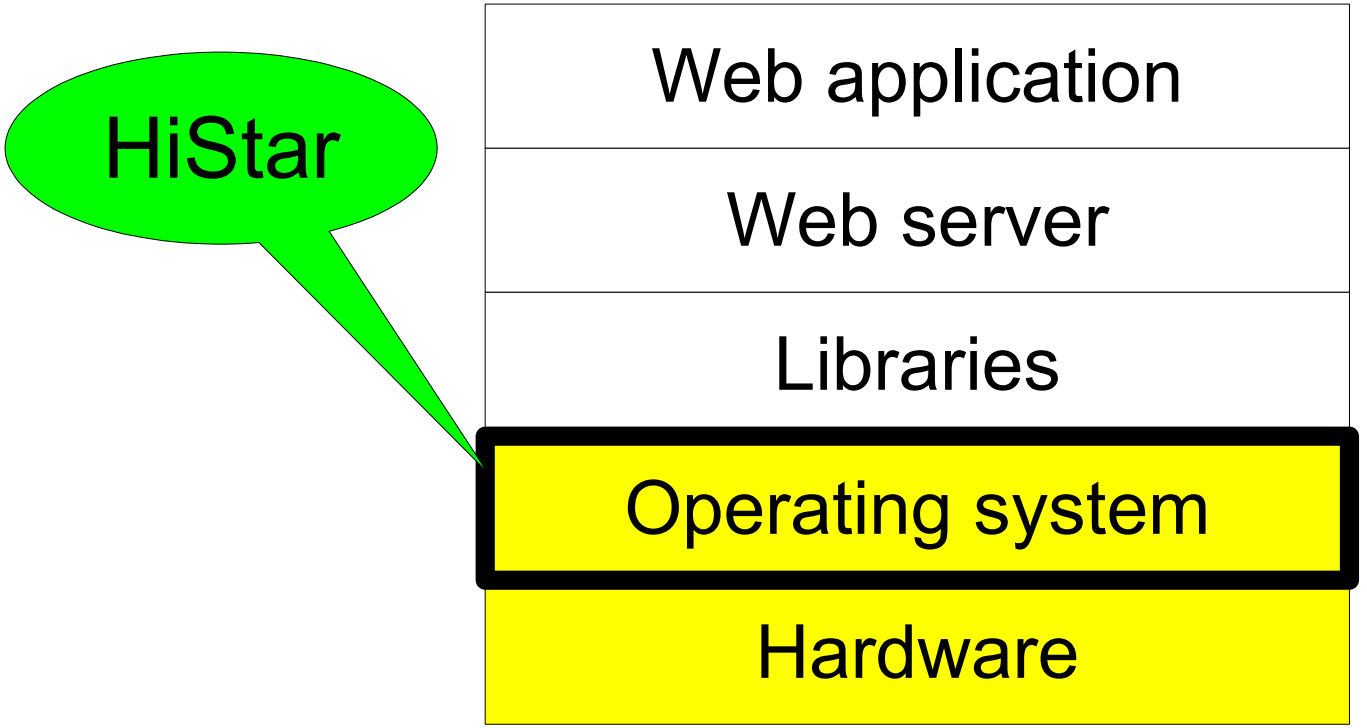
- If we get this simple mechanism right, then most other code won't have to worry about security

Not obvious how to build all protection on top of IFC

- How to implement user accounts with IFC?
  - Military systems had separate user protection mech.
- How to give users access to same mechanism?
  - Same mechanism should enforce app. policies
  - What if users create processes they can't kill?
- How do we manage such a system?
  - Without any separate “superuser” mechanism

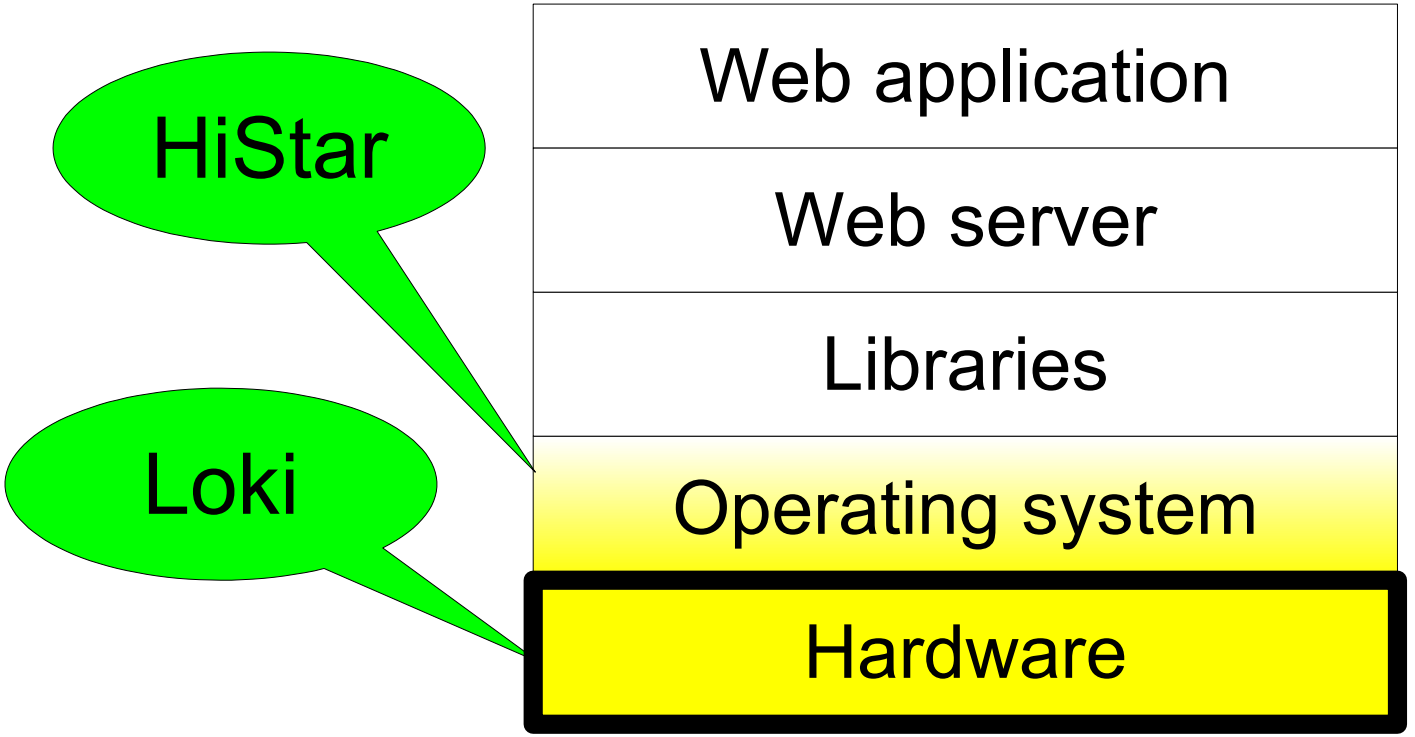
Outline: Three systems based on information flow control

HiStar: collaboration with Silas Boyd-Wickizer, David Mazières



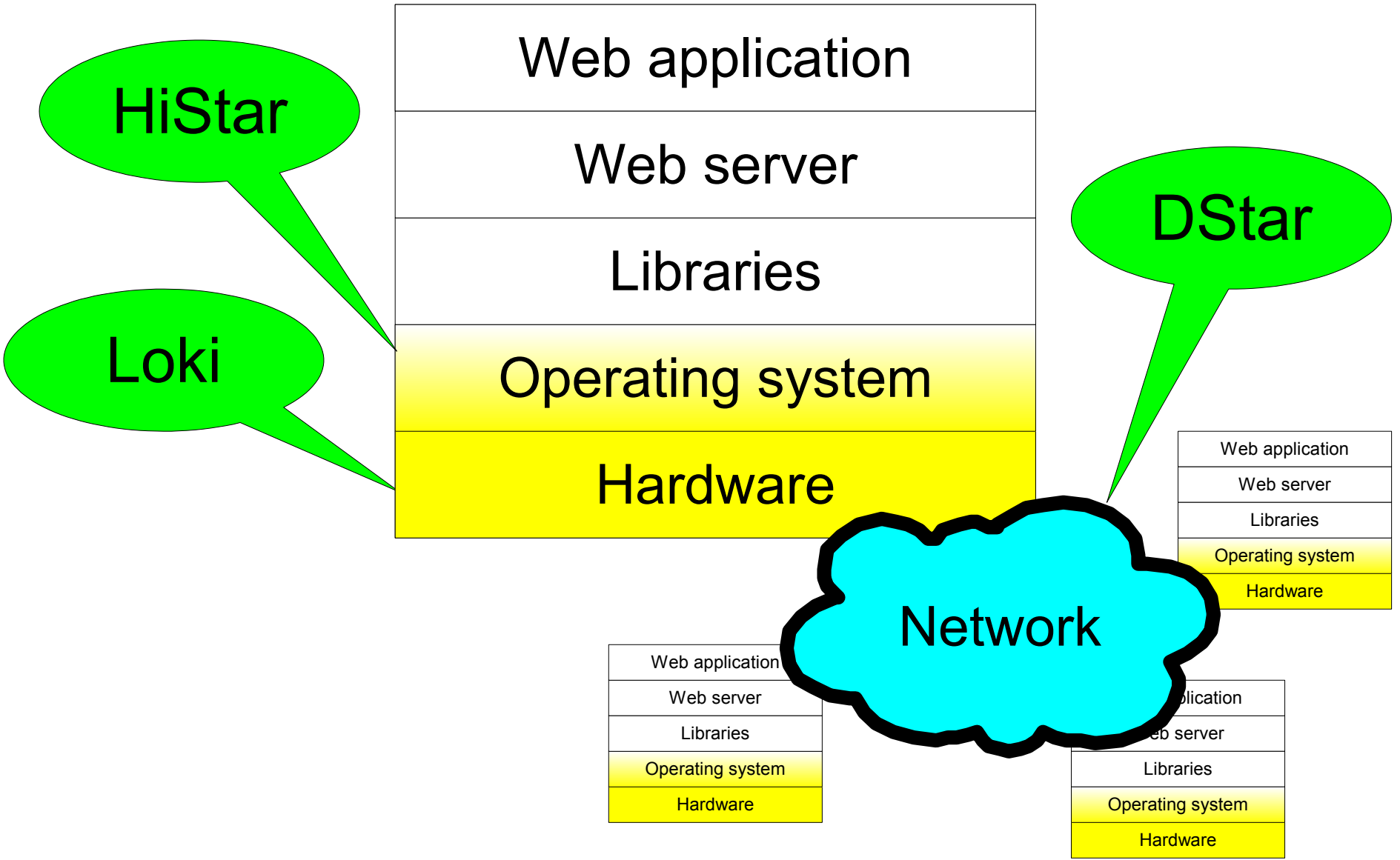
Outline: Three systems based on information flow control

Loki: collaboration with Michael Dalton, Hari Kannan, Christos Kozyrakis

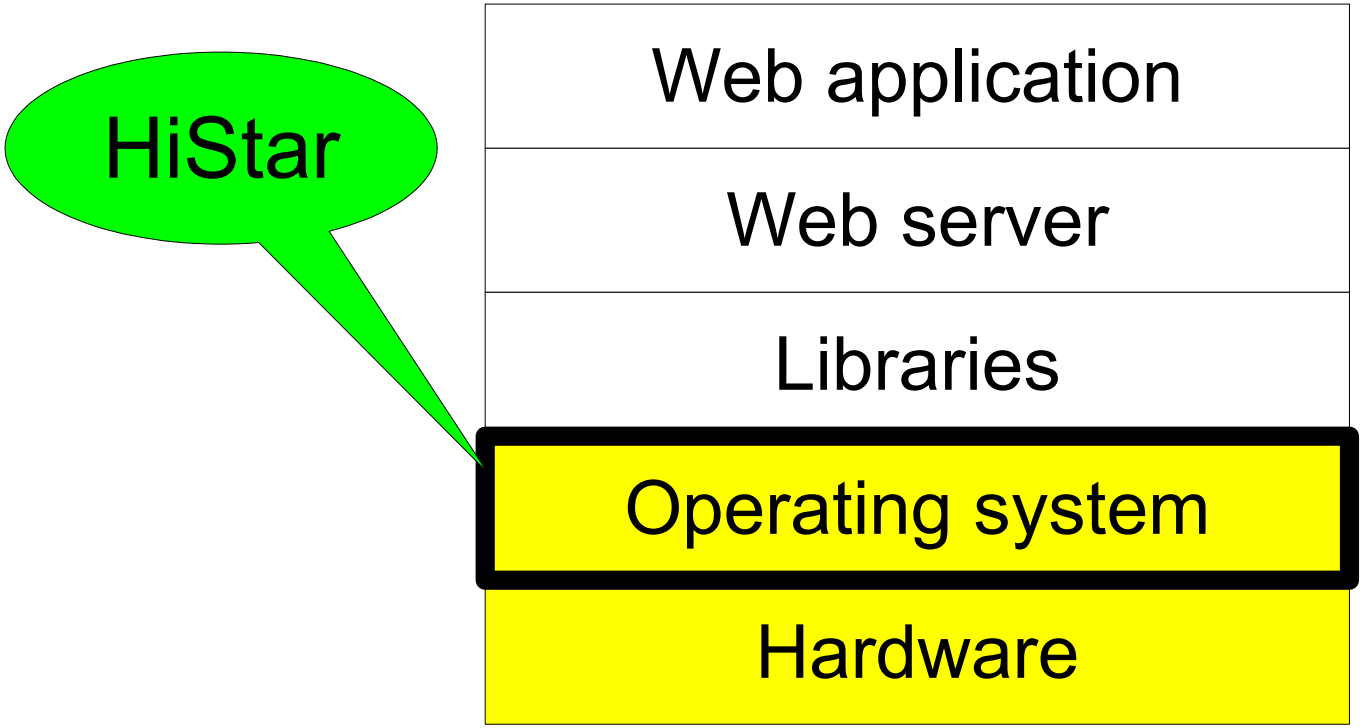


Outline: Three systems based on information flow control

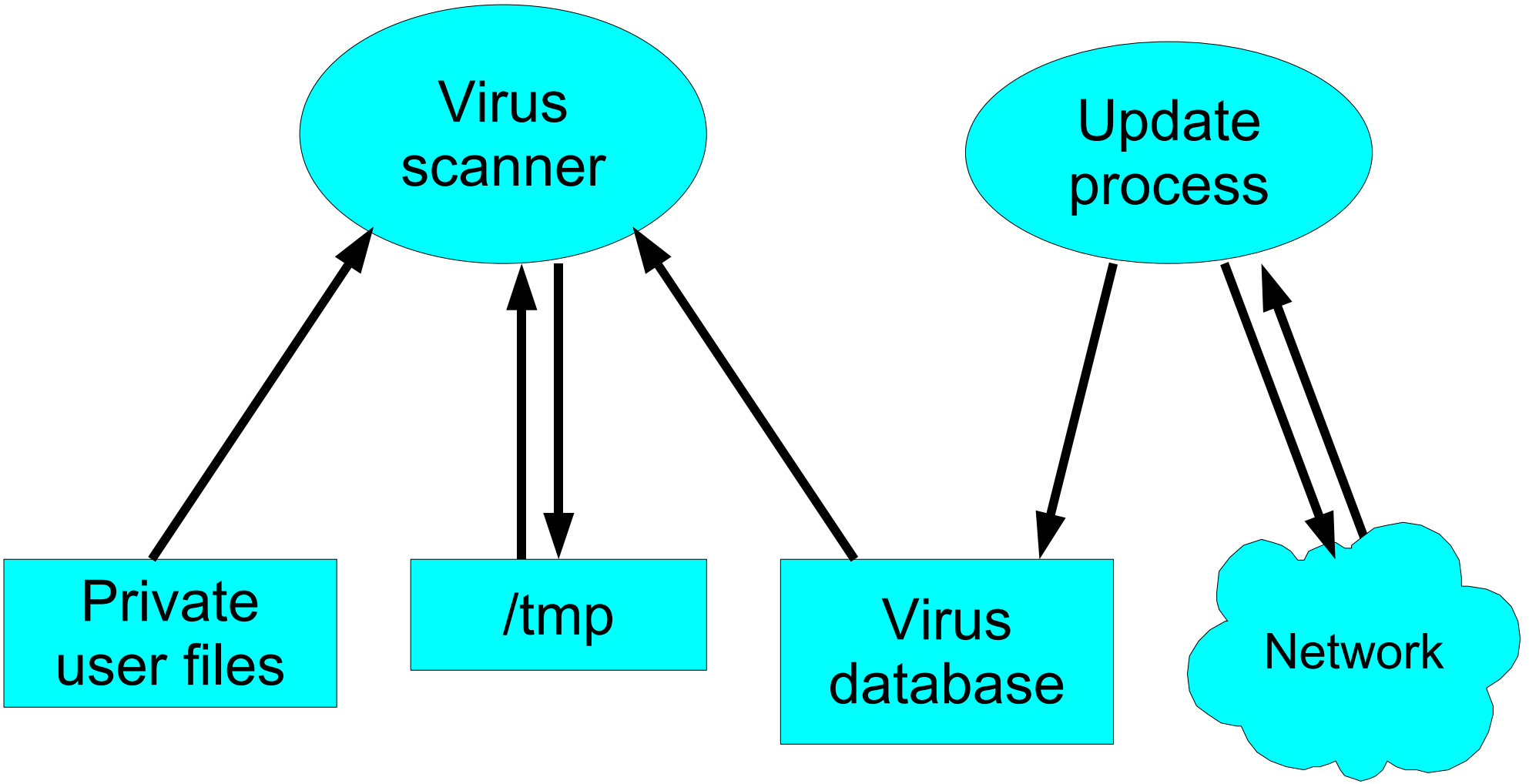
DStar: collaboration with Silas Boyd-Wickizer, David Mazières



Information flow control in an OS

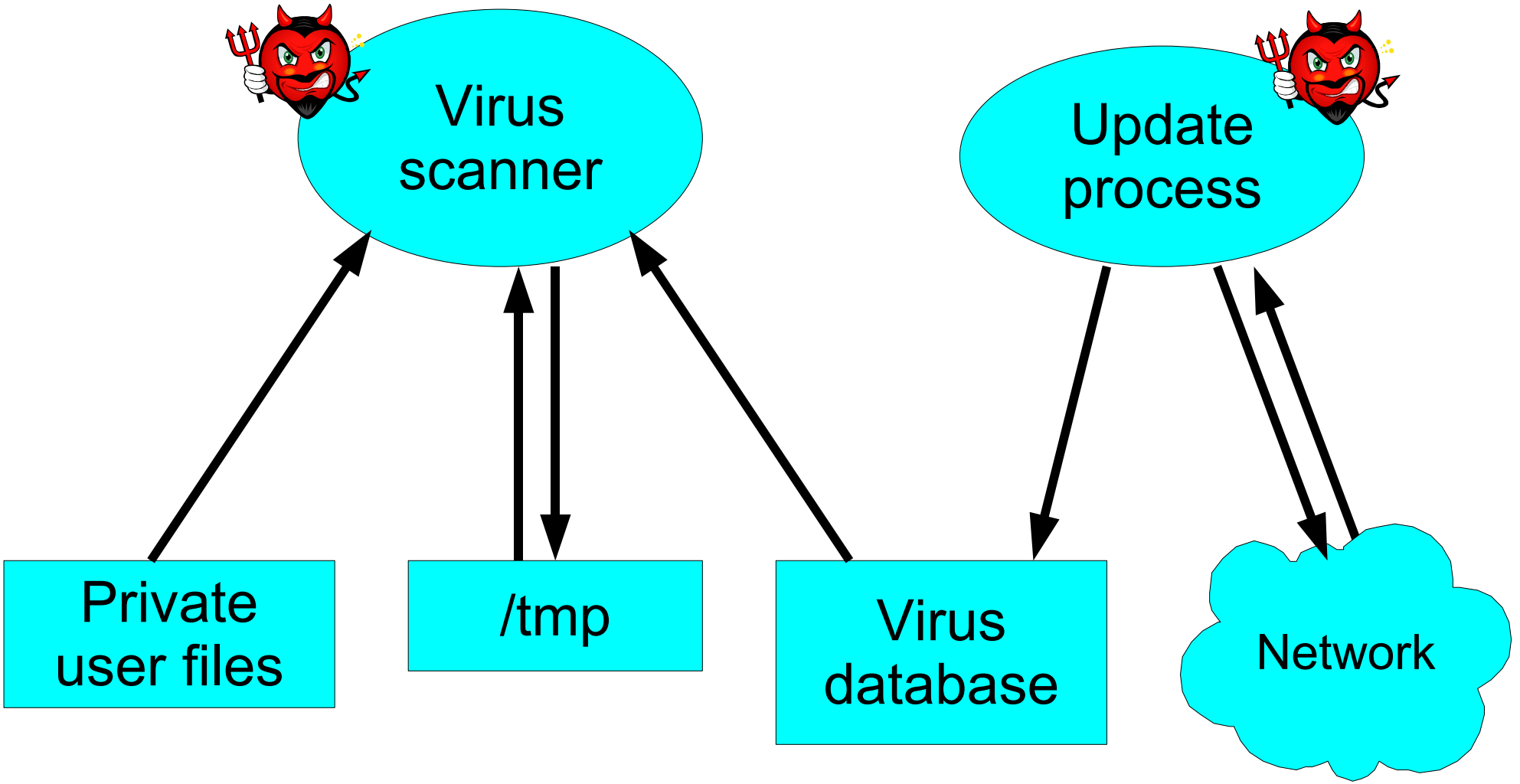


Example: Virus scanner



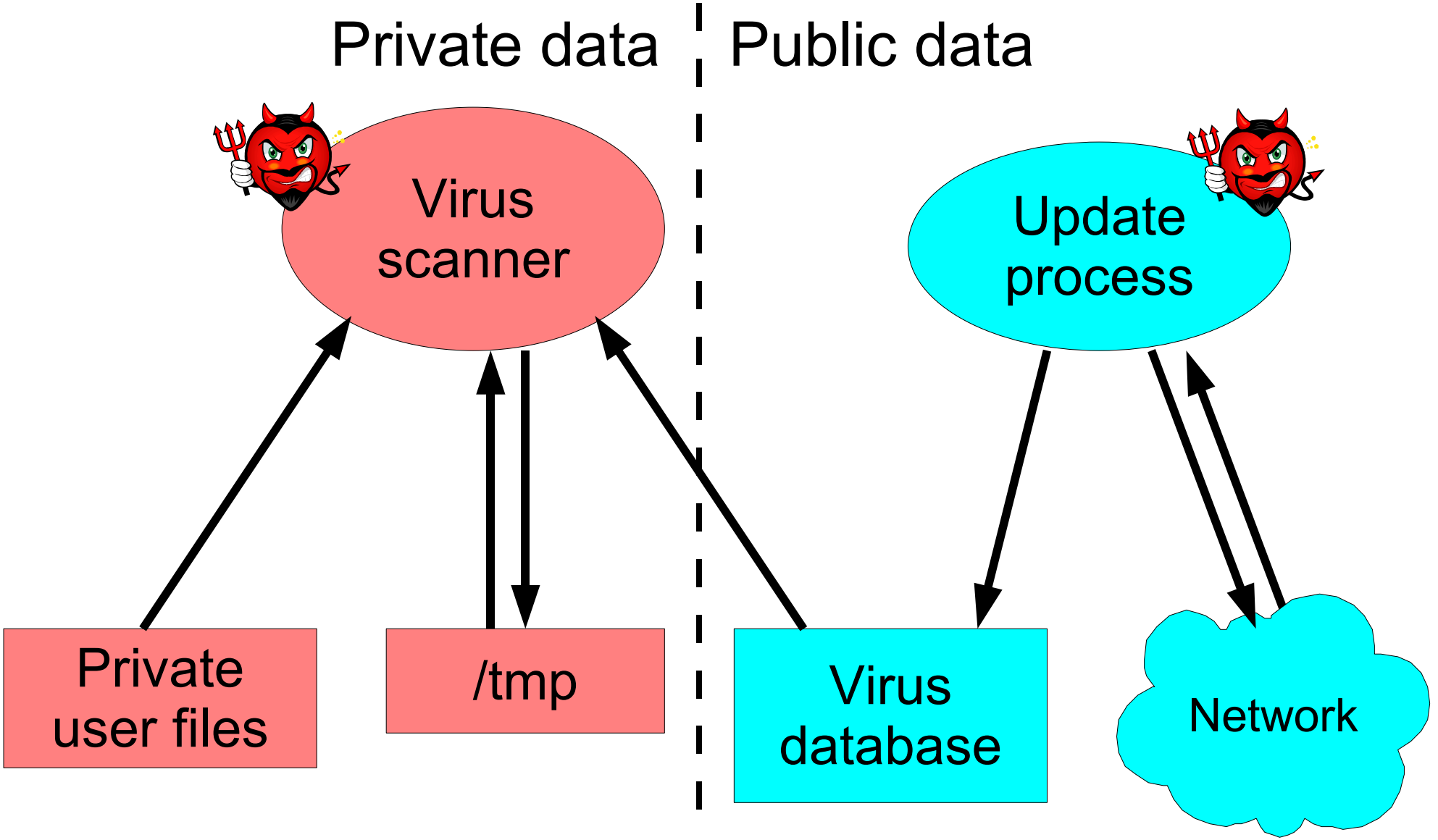
Example: Virus scanner

Can we confine a compromised scanner on Unix?



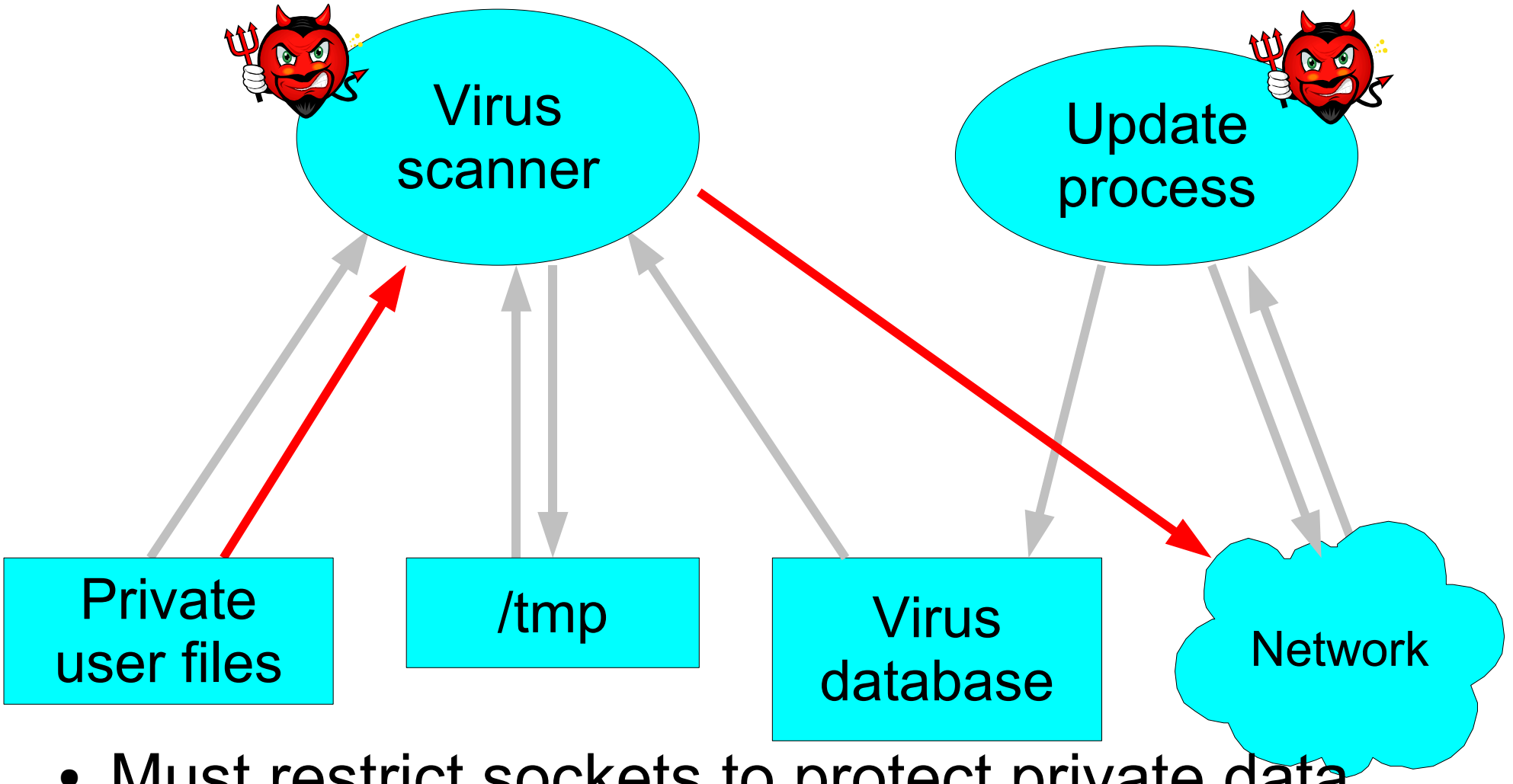
Goal: private files not corrupted or sent over network

Seems like a job for IFC!



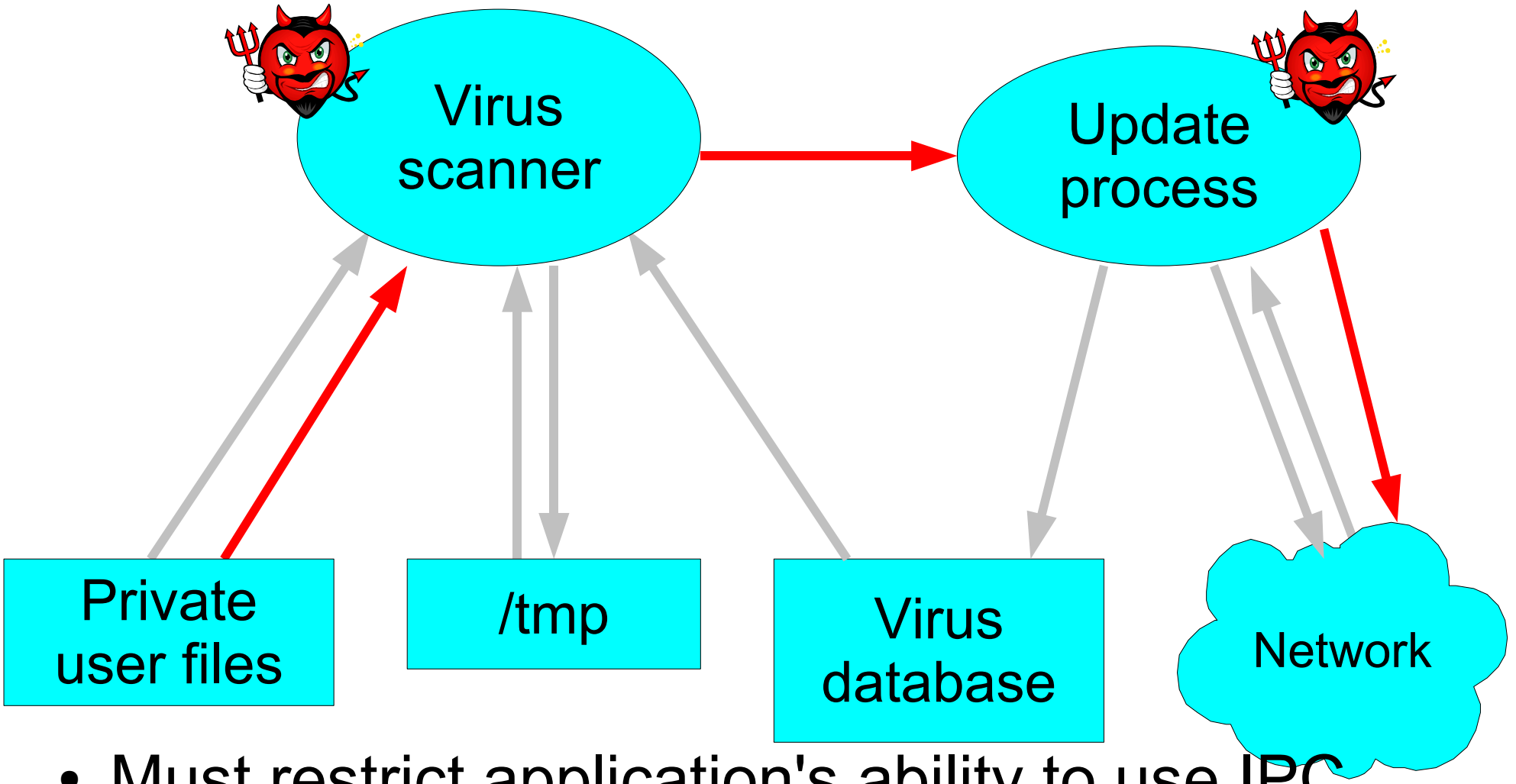
Goal: private files not corrupted or sent over network

Cannot prevent app from leaking data



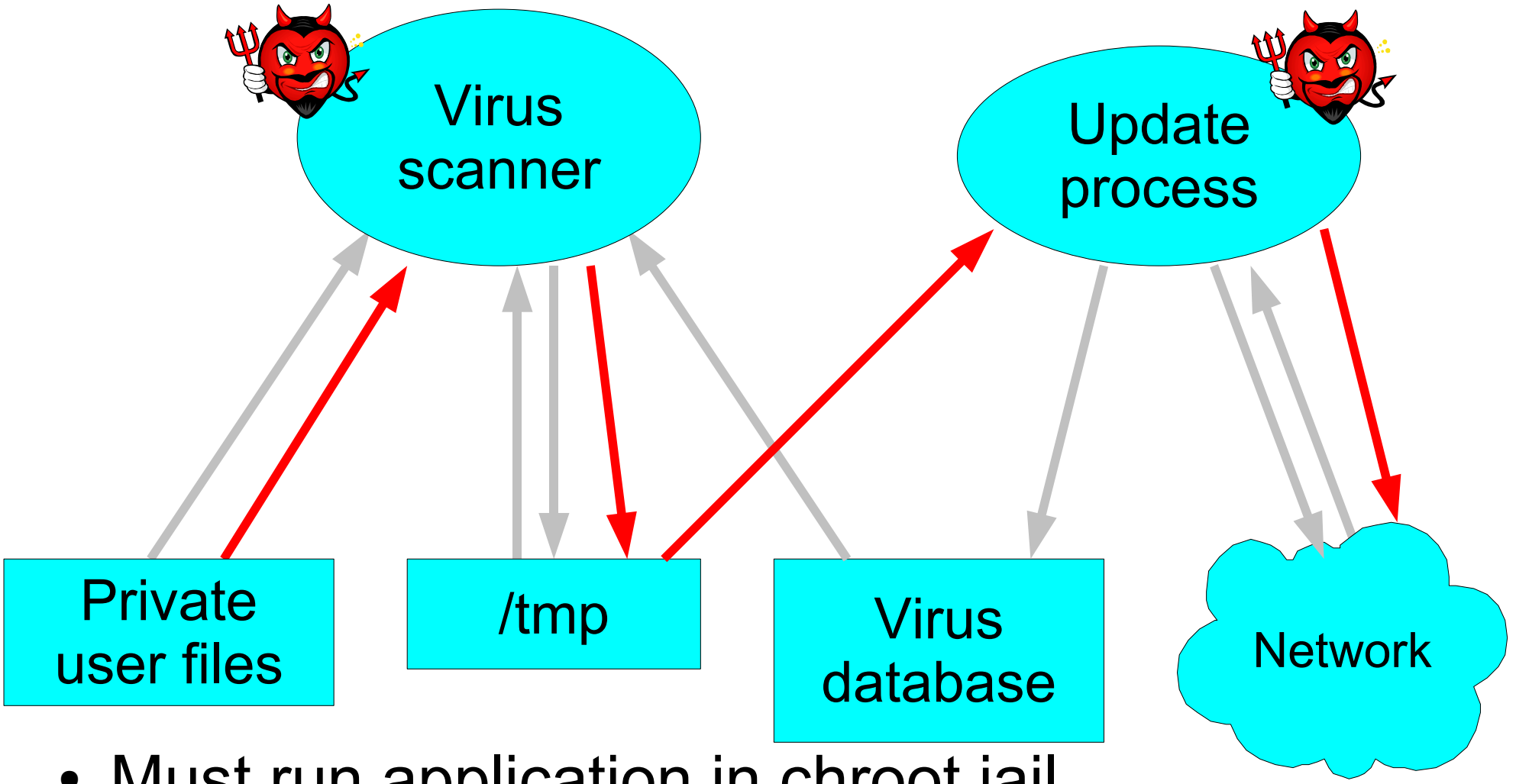
- Must restrict sockets to protect private data

Cannot prevent app from leaking data



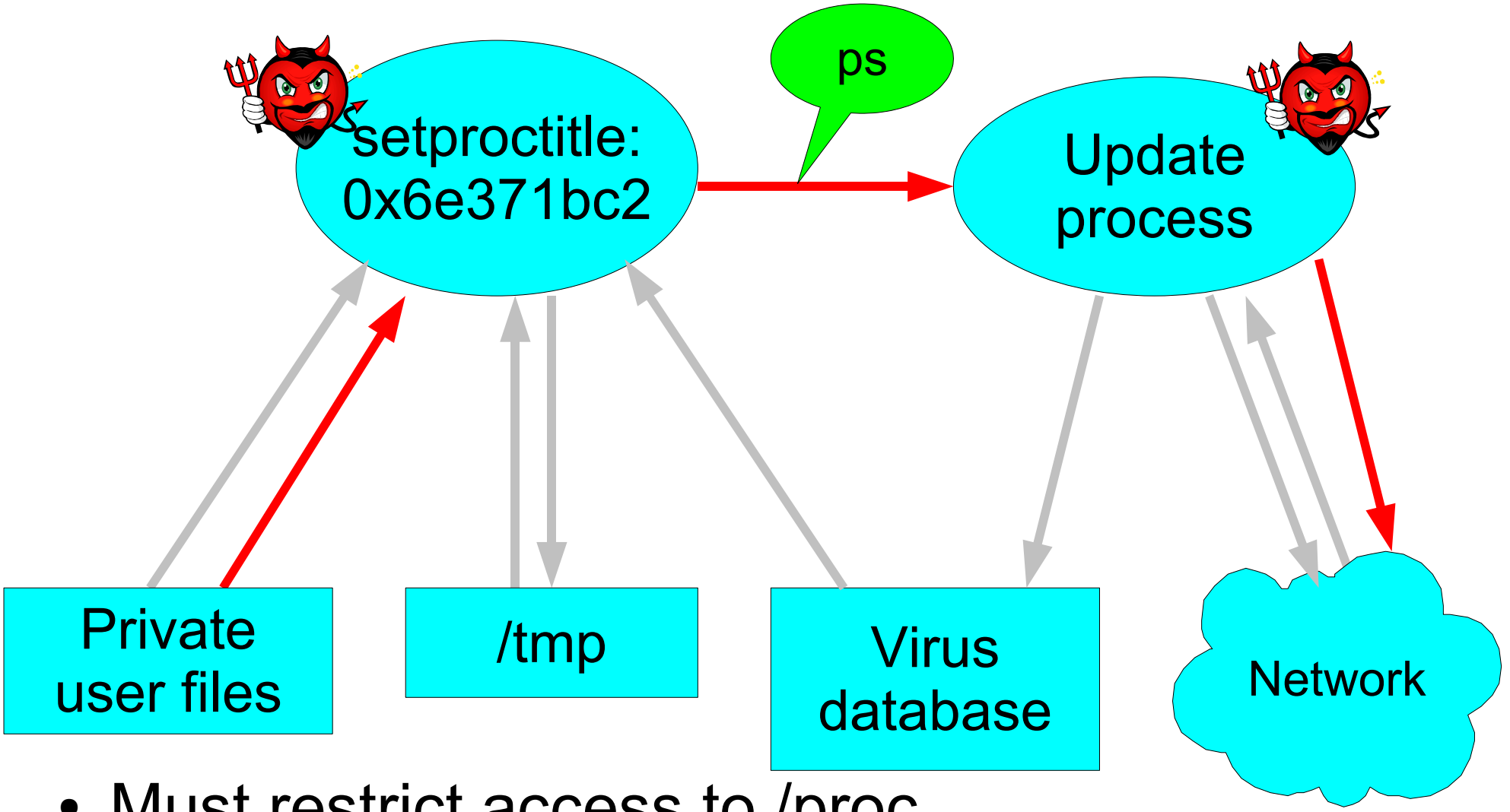
- Must restrict application's ability to use IPC

Cannot prevent app from leaking data



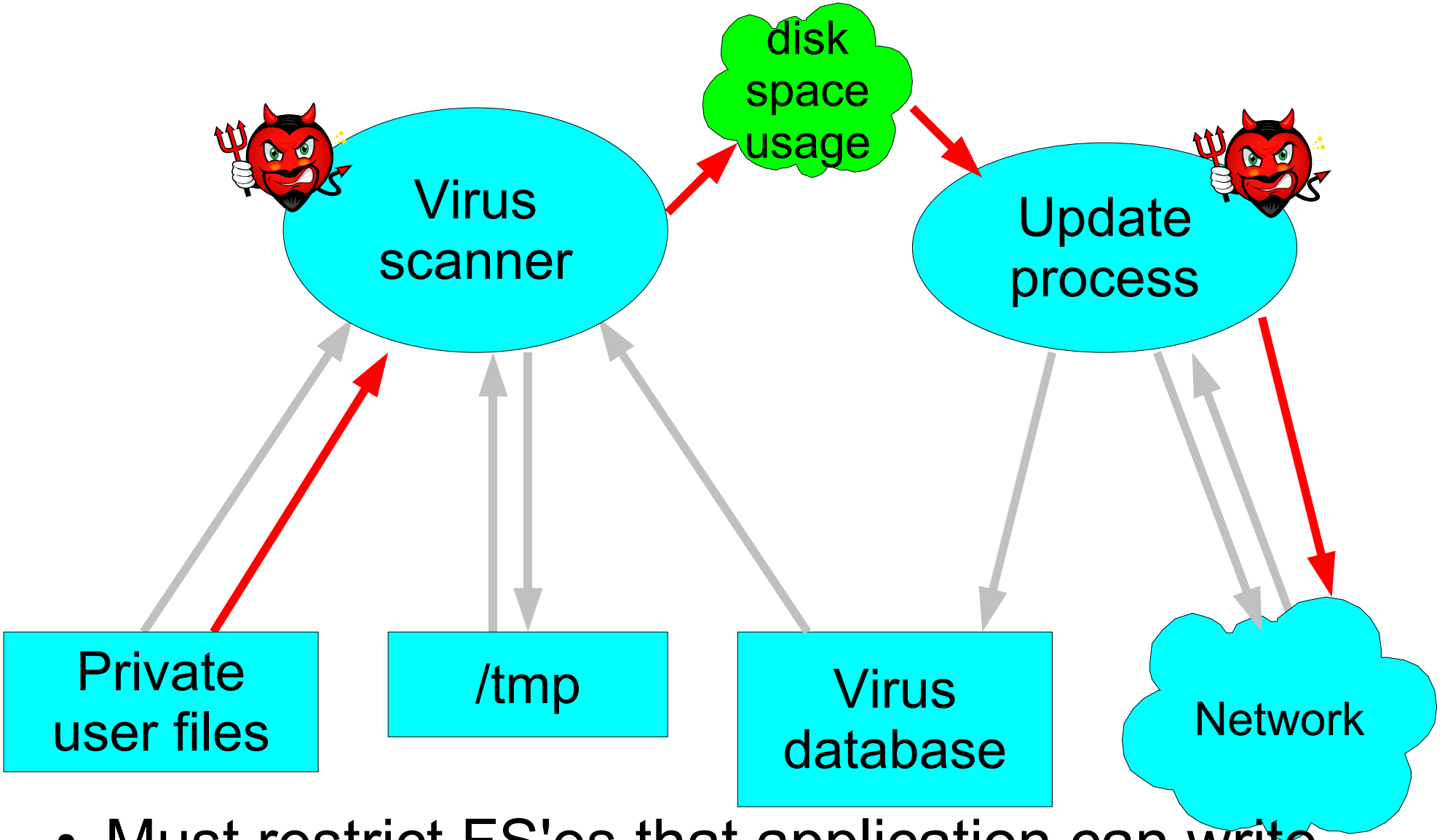
- Must run application in chroot jail

Cannot prevent app from leaking data



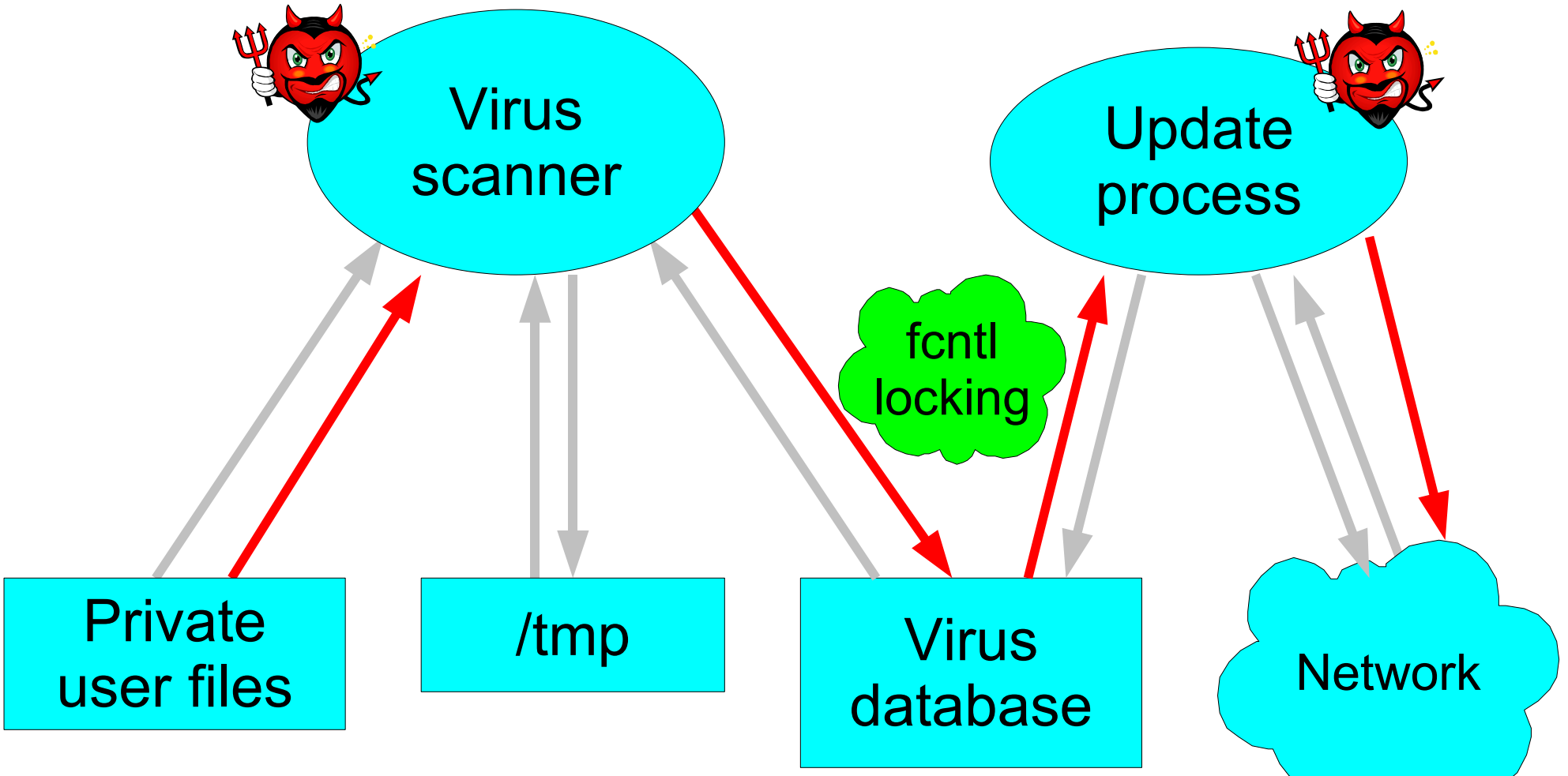
- Must restrict access to /proc, ...

Cannot prevent app from leaking data



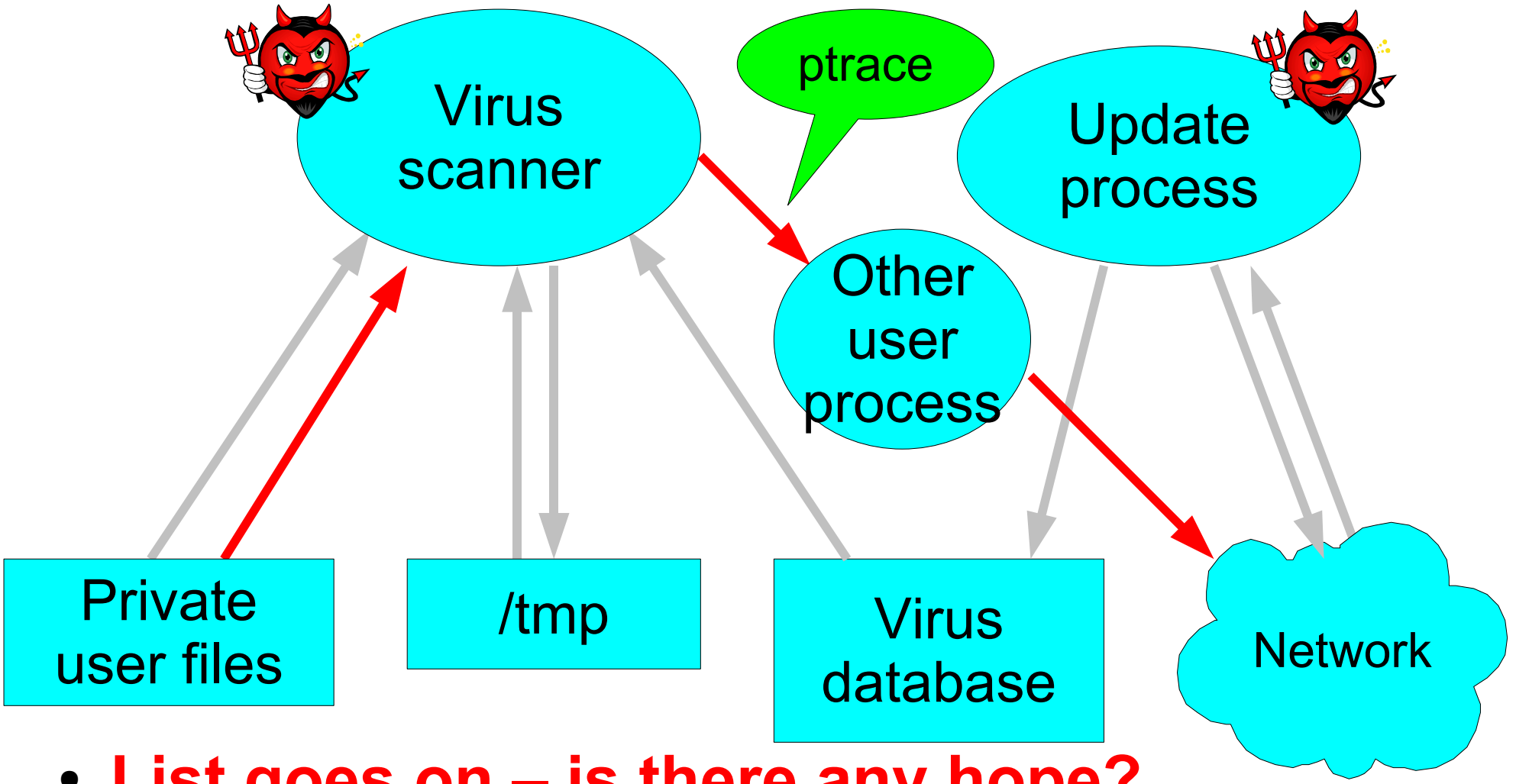
- Must restrict FS'es that application can write

Cannot prevent app from leaking data



- Cannot allow file locking or synchronization

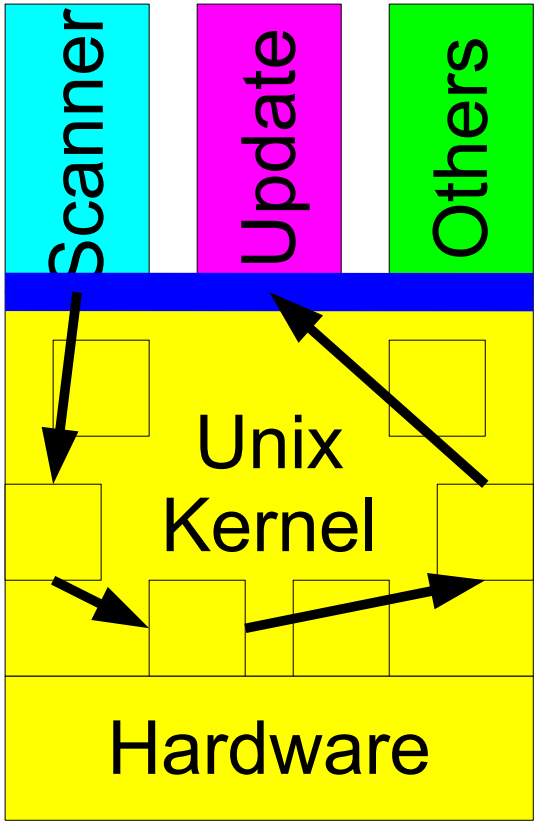
Impossible to prevent simple app from leaking data?



- **List goes on – is there any hope?**

Unix interface is too high-level to control information flow

= Unix API



Unix

(1) Too many ways for data to move around

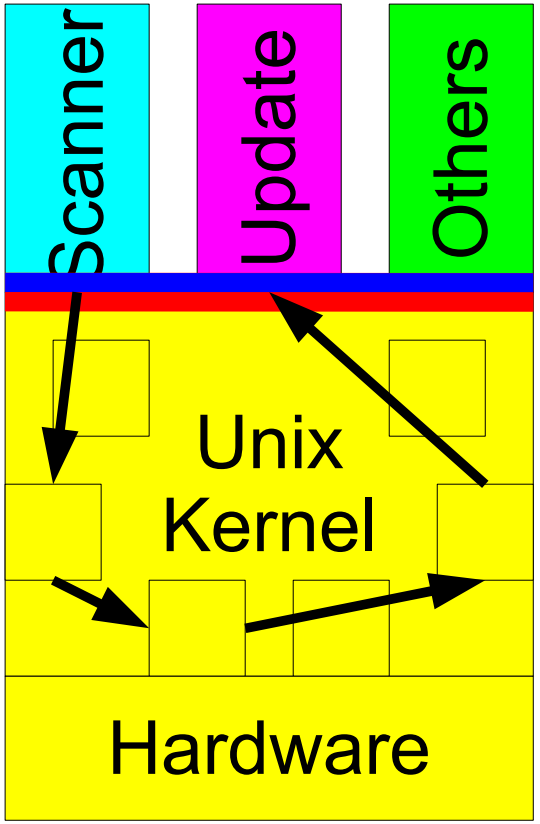
(2) Protection for processes and files – not data!

- Process can read one file, write data to another file with different protection

Unix interface is too high-level to control information flow

= Unix API

= Security checks



Unix

(1) Too many ways for data to move around

(2) Protection for processes and files – not data!

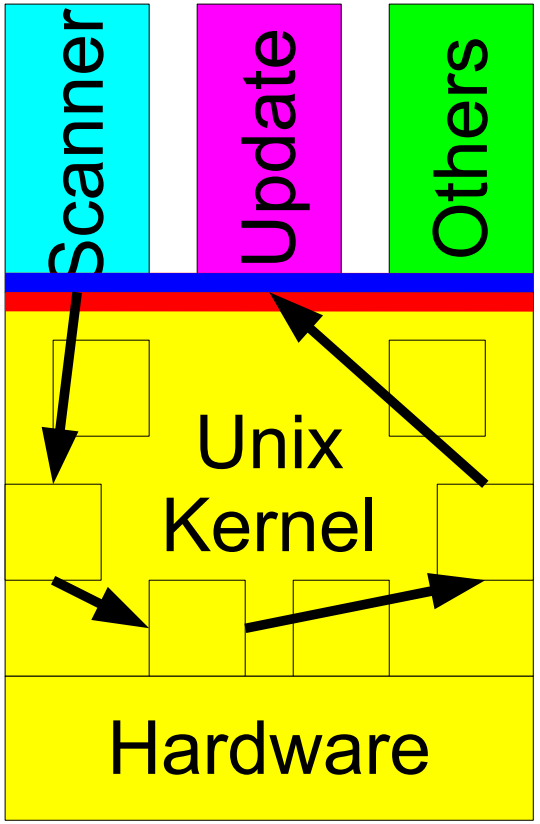
- Process can read one file, write data to another file with different protection

→ Unix API poor choice for information flow control

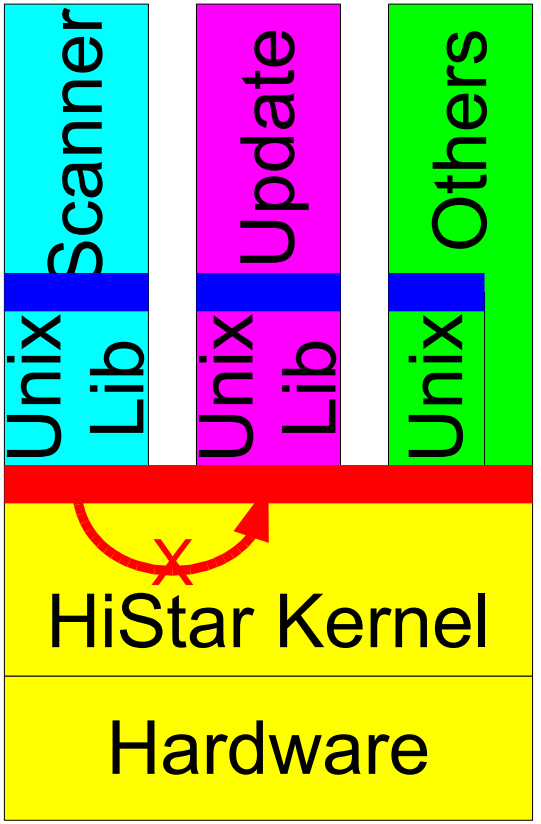
HiStar solution: Lower-level interface, Protect data

= Unix API

= Security checks



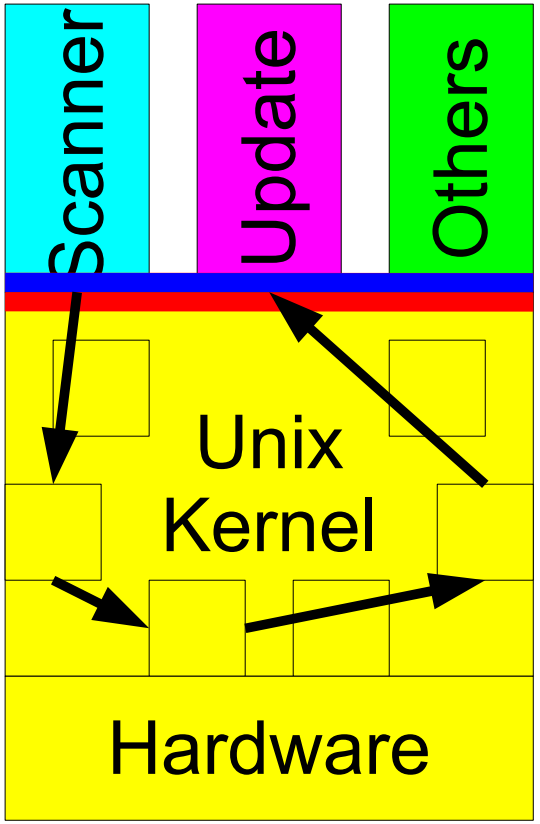
Unix



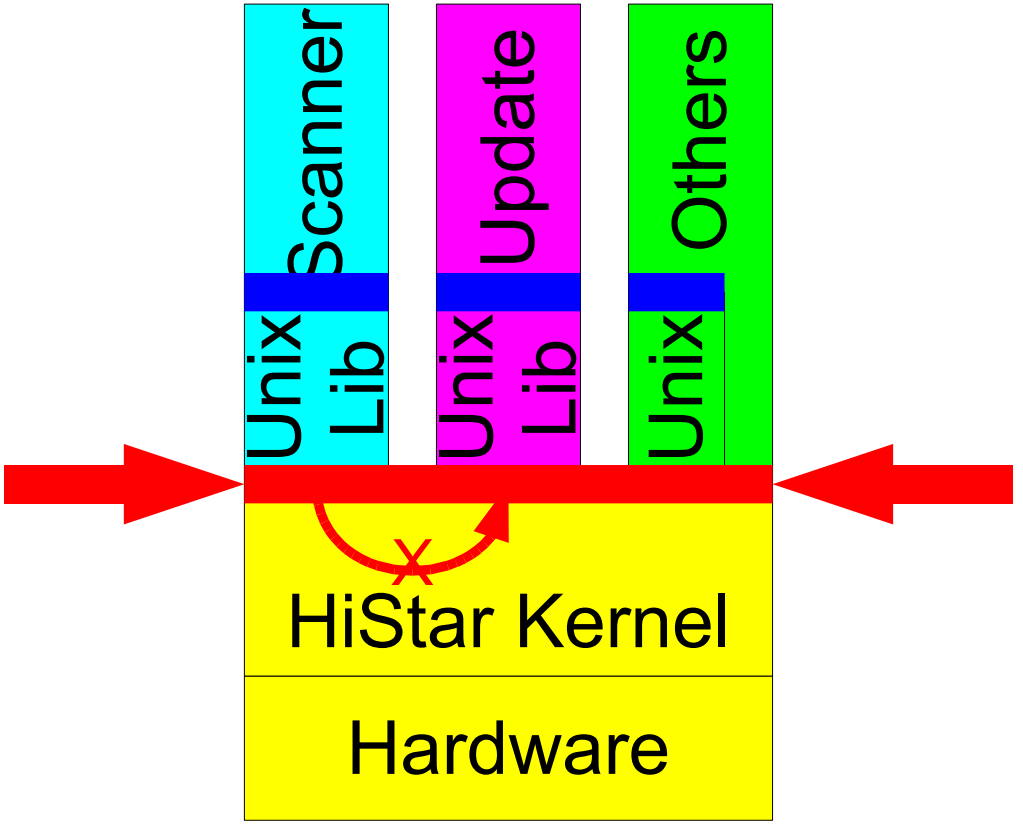
HiStar

Challenge:
How to design kernel mechanism?

Goal: minimal trusted code needed for functionality



Unix



HiStar

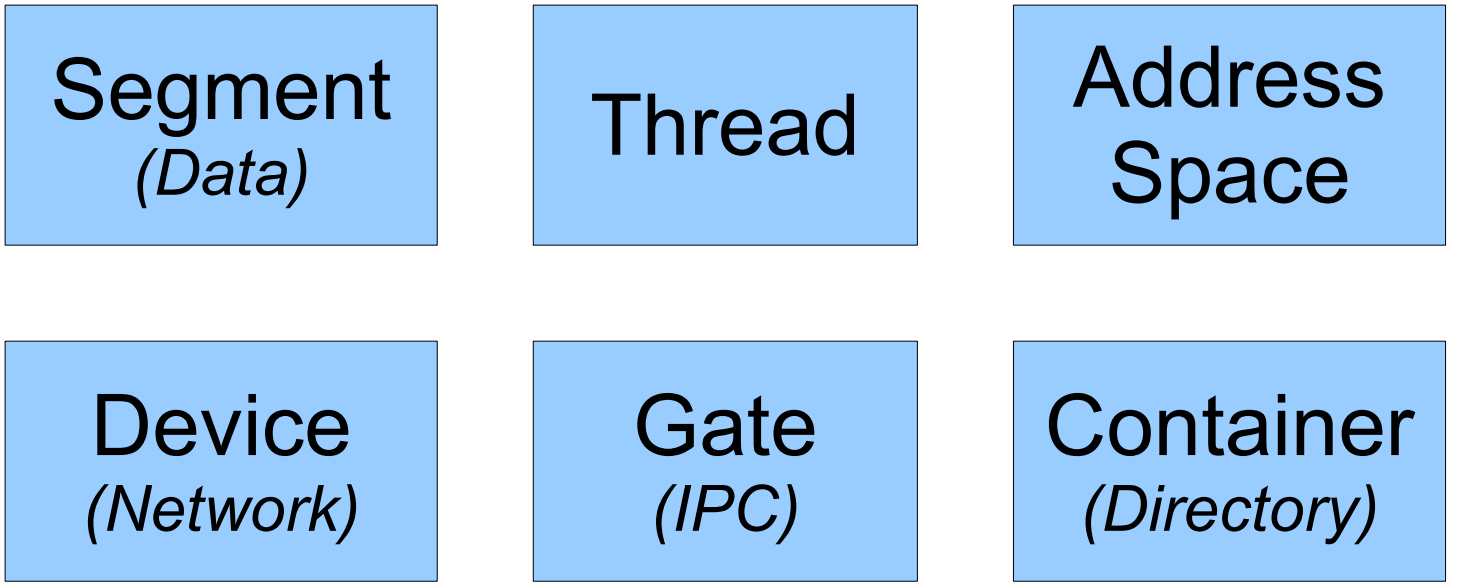
HiStar approach

- Simple interface – 6 types of kernel objects
  - Expressive enough to build a Unix-like environment
- Security mechanism: information flow control
  - Egalitarian – any process can use it
- Build everything else from these primitives
  - Same mechanisms seem to solve a lot of problems
  - Suggests this might be a good mechanism design

HiStar outline

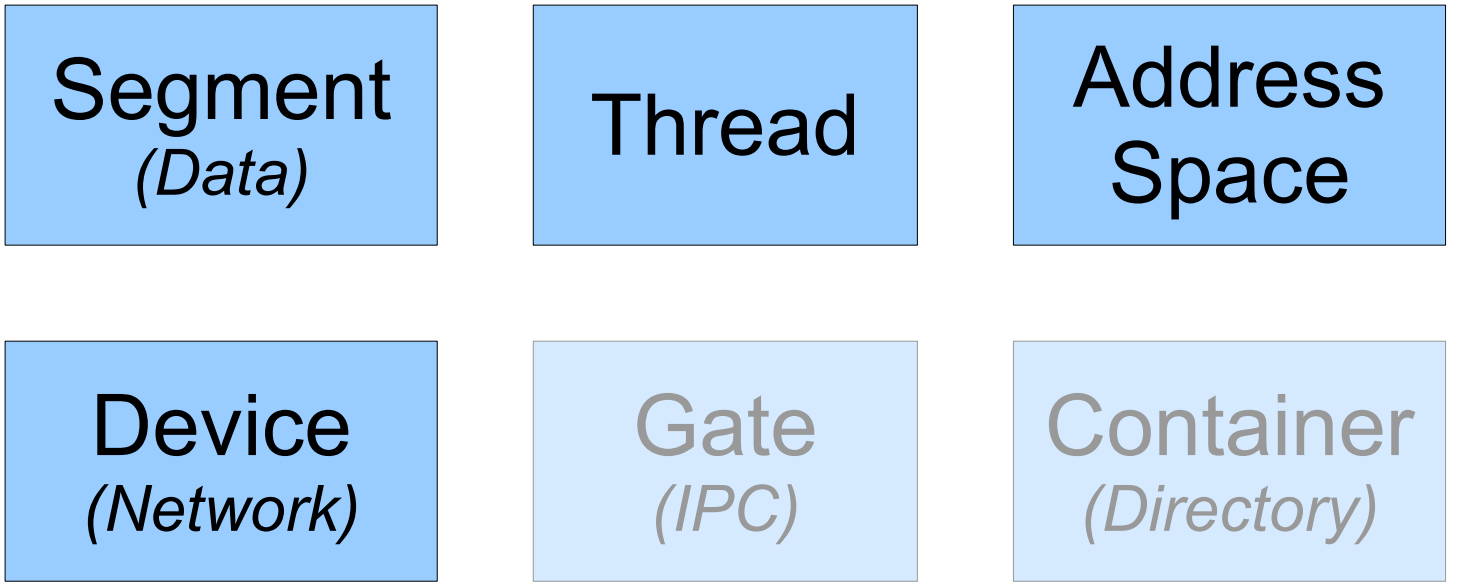
1. Kernel mechanisms: objects and labels
2. Example uses of these mechanisms
3. How these mechanisms improve security
4. Applications

HiStar kernel objects

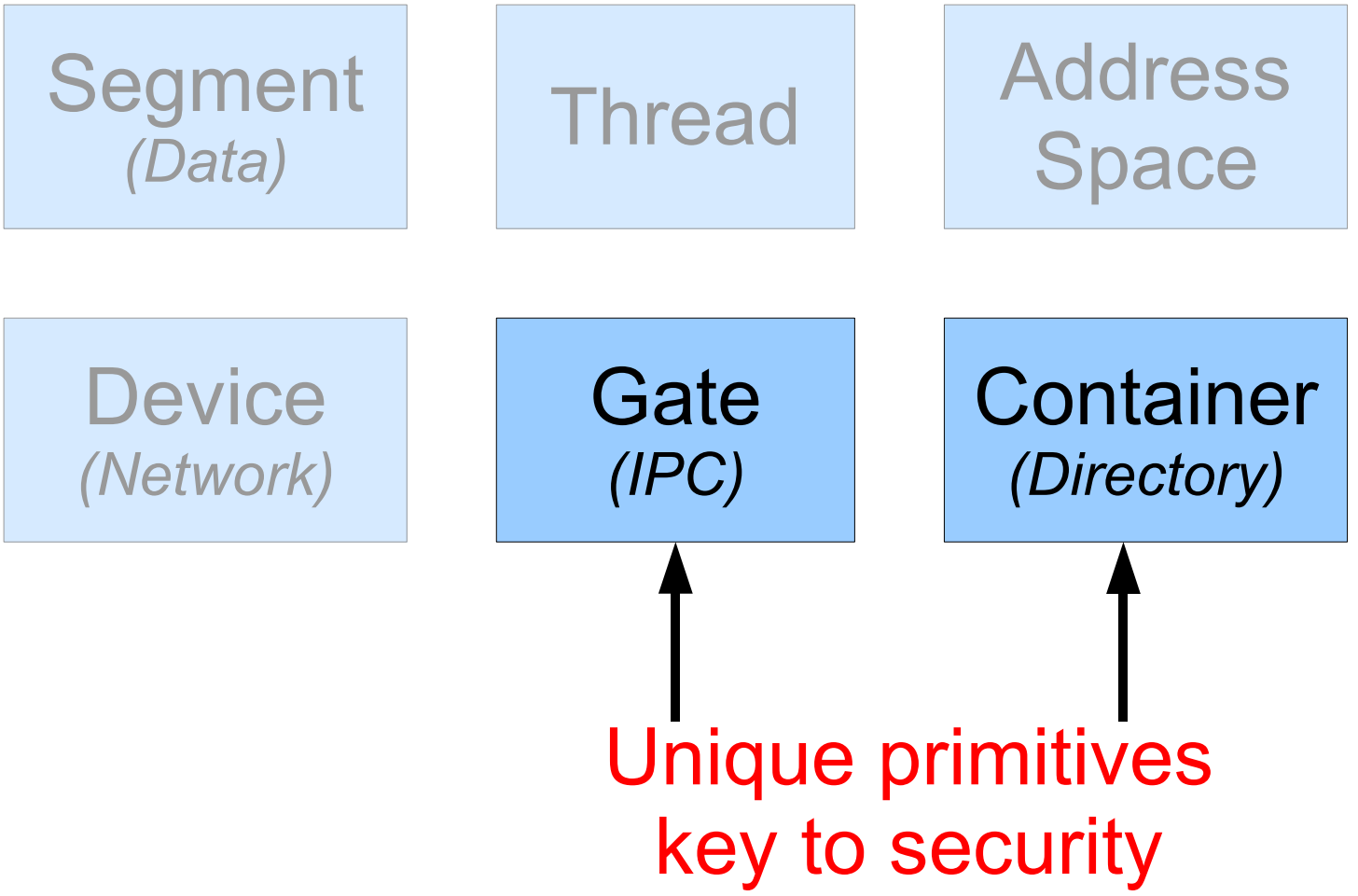


HiStar kernel objects

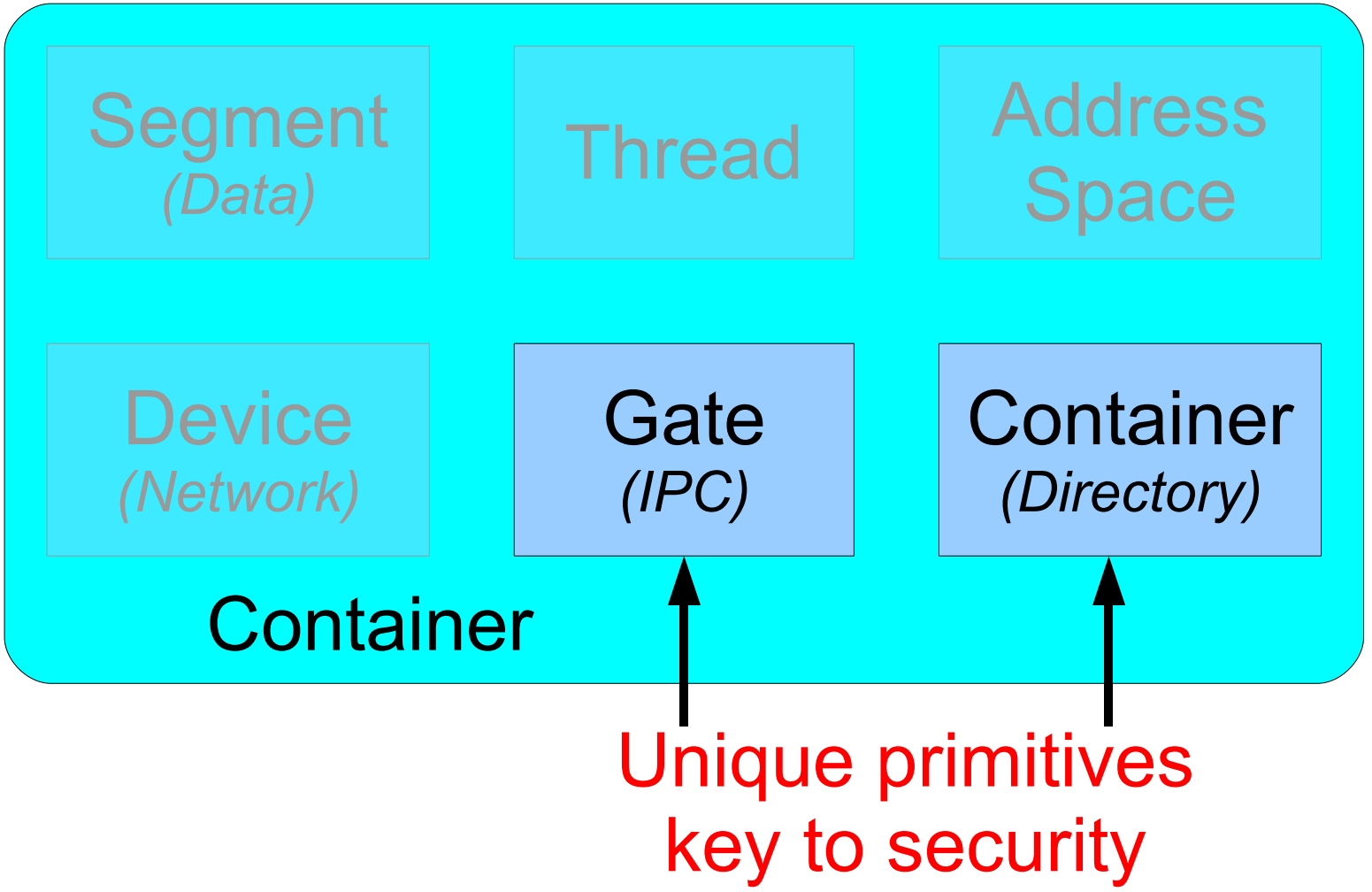
Familiar low-level primitives, can do the usual things



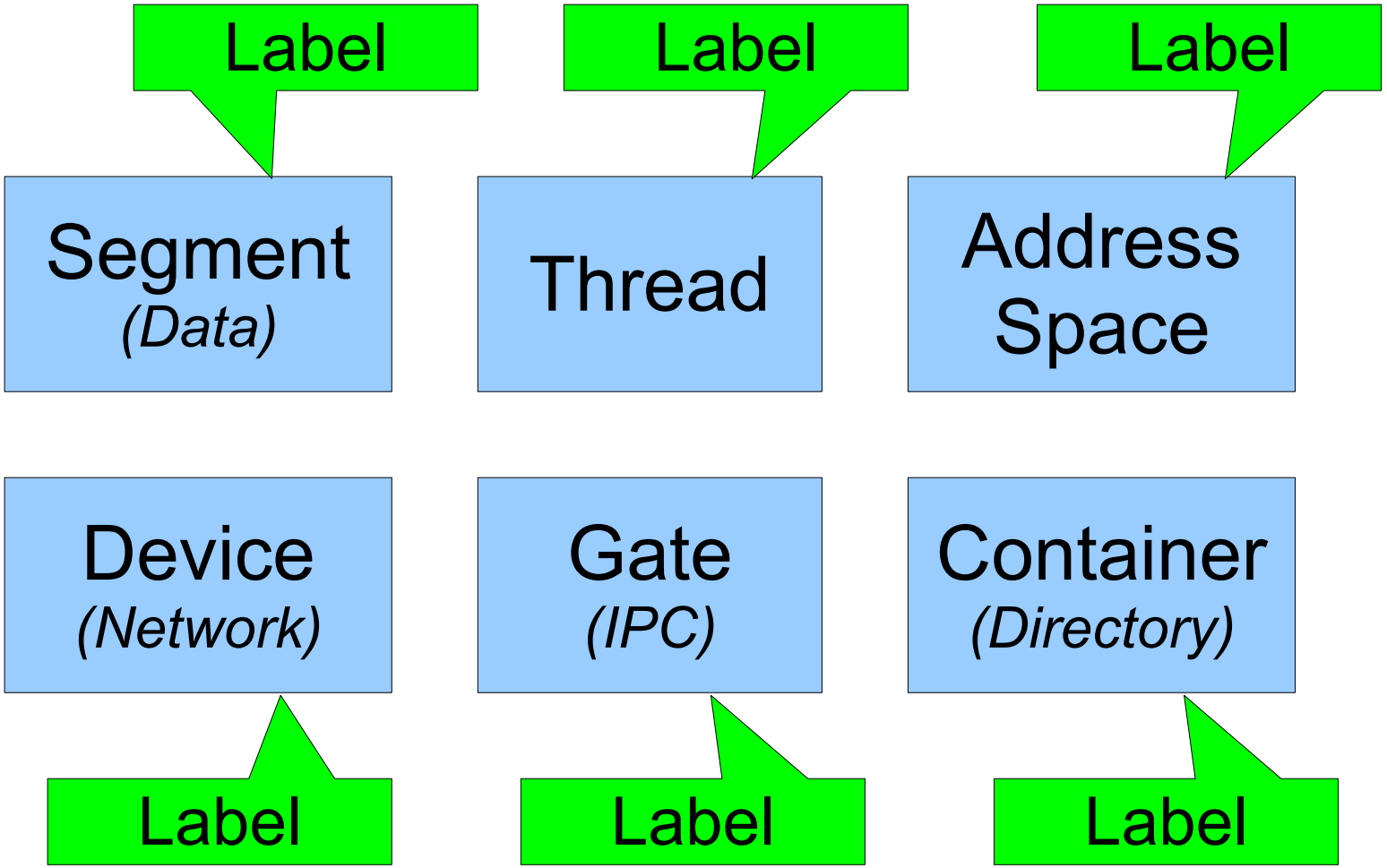
HiStar kernel objects



HiStar kernel objects

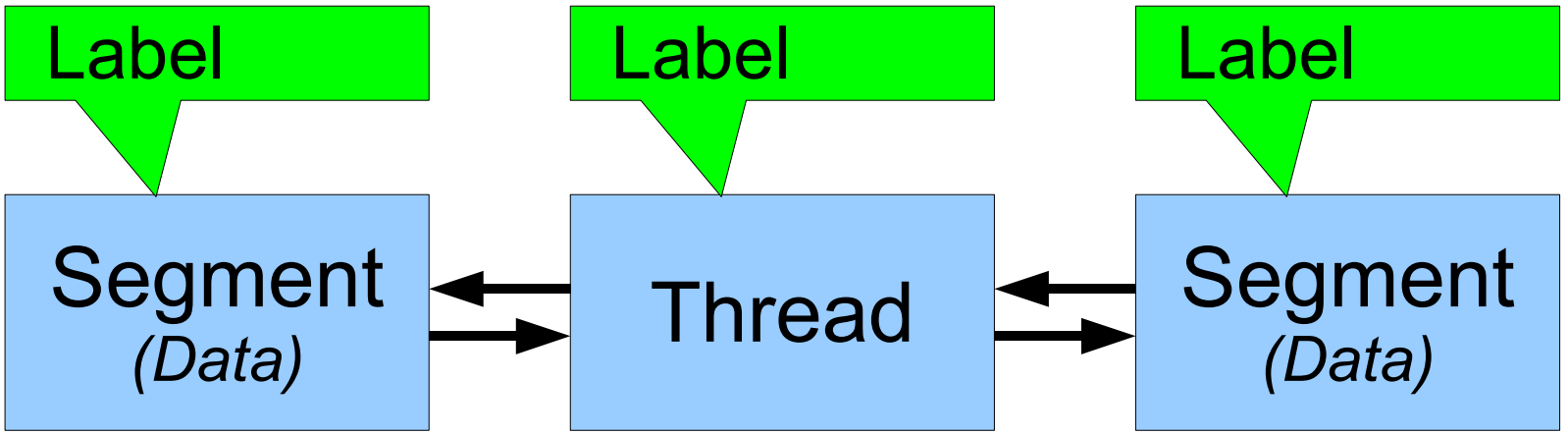


Protection mechanism: labels

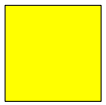


Labels control information flow

Color is category of data (e.g. my files)

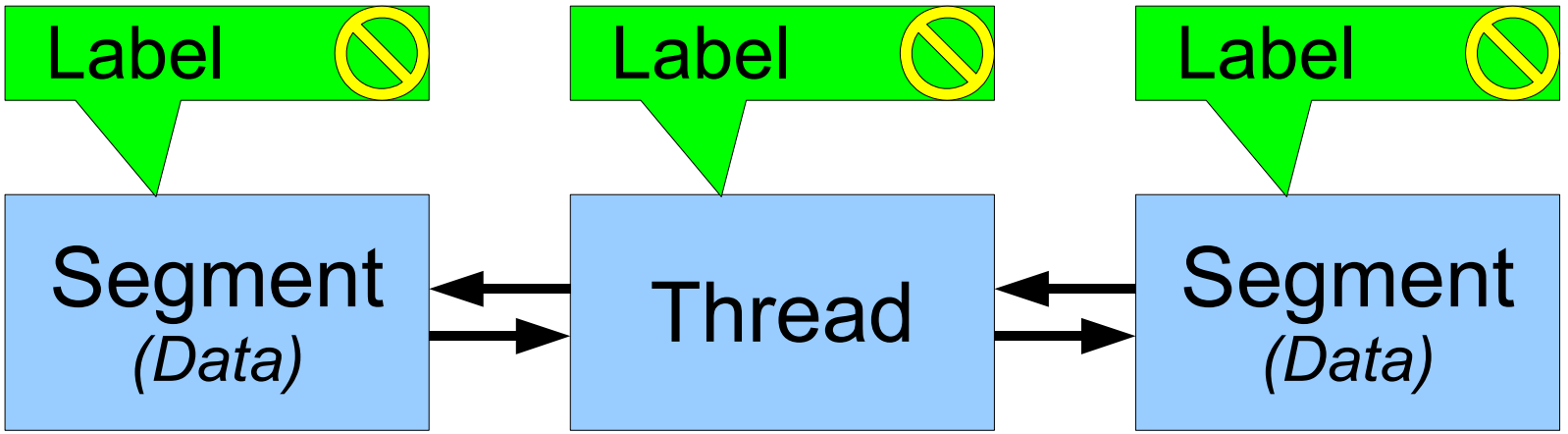


Labels control information flow



Color is category of data (e.g. my files)

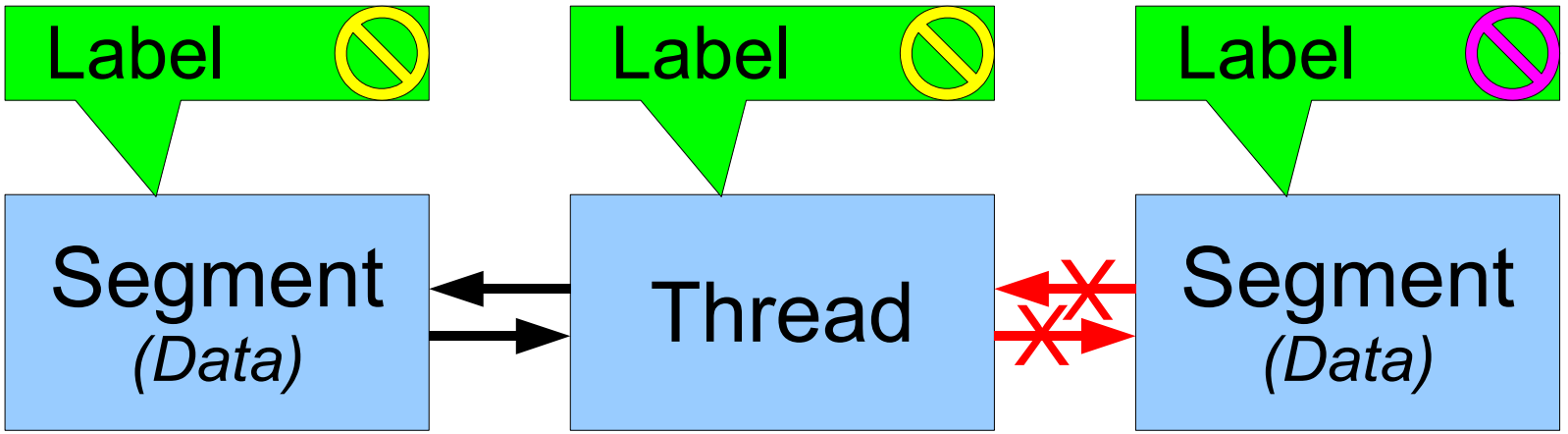
Yellow data can flow only to other yellow objects



Labels control information flow

Color is category of data (e.g. my files)

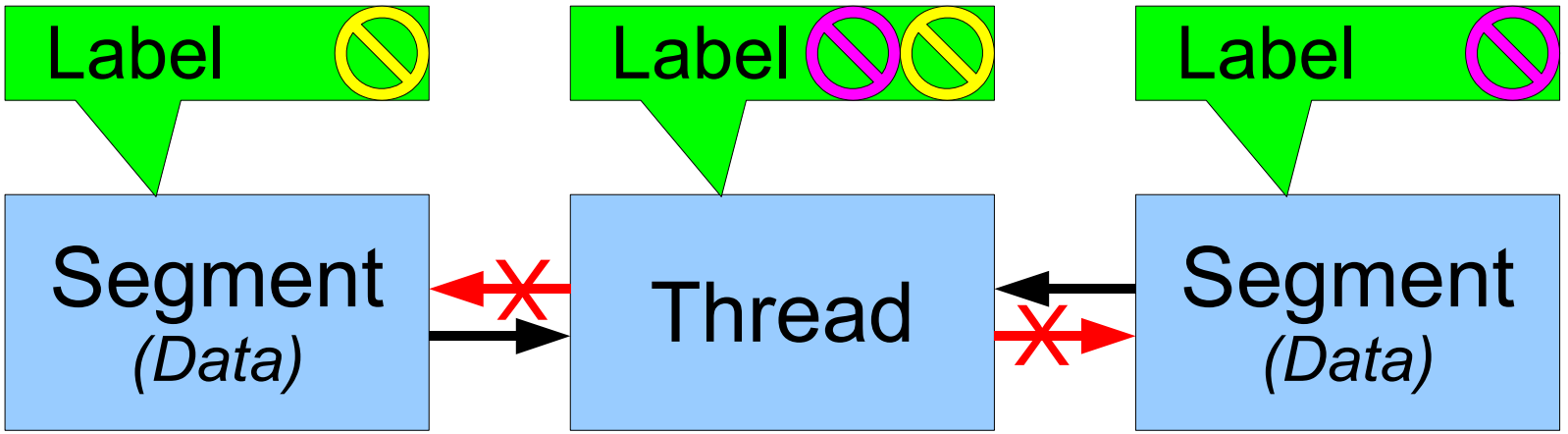
Yellow data can flow only to other yellow objects



Labels control information flow

Color is category of data (e.g. my files)

Yellow data can flow only to other yellow objects

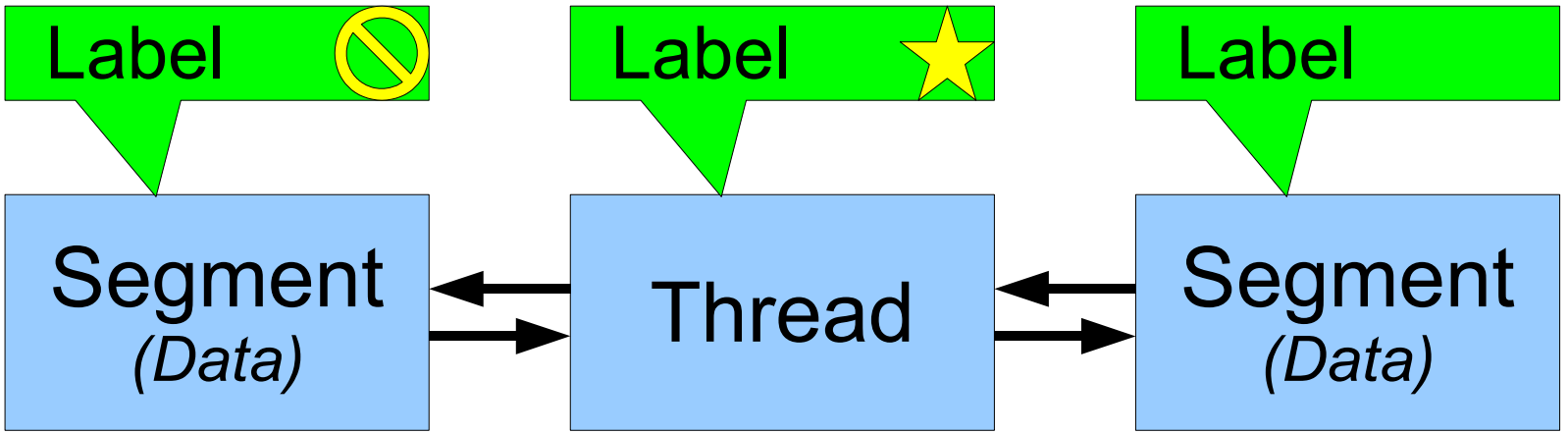


Labels control information flow

Color is category of data (e.g. my files)

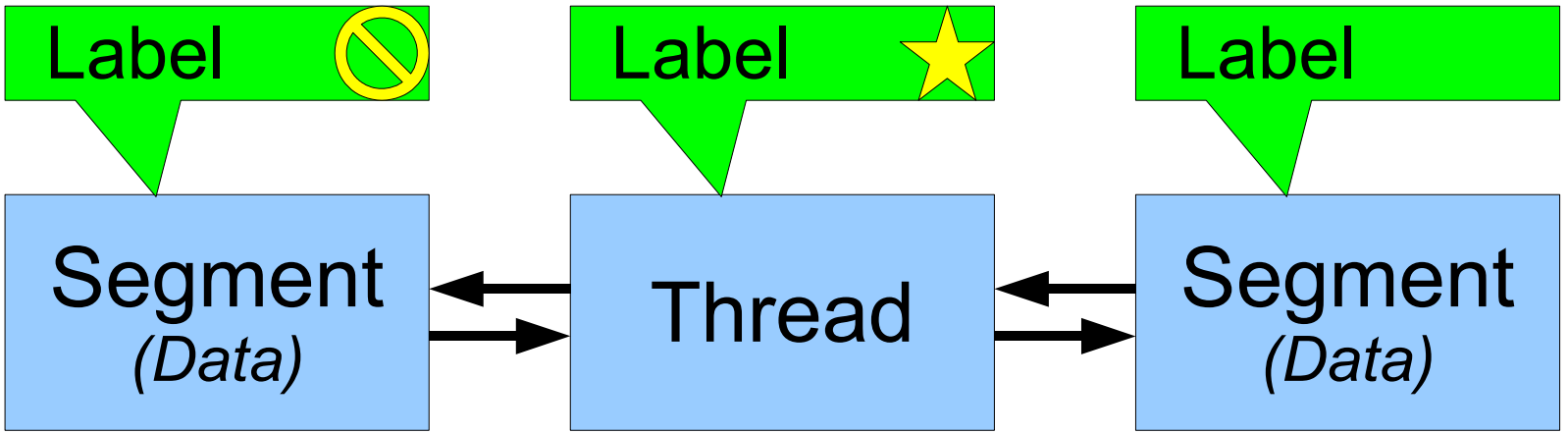
Yellow data can flow only to other yellow objects

Owns yellow data, can remove color (e.g. encrypt)



Labels are egalitarian

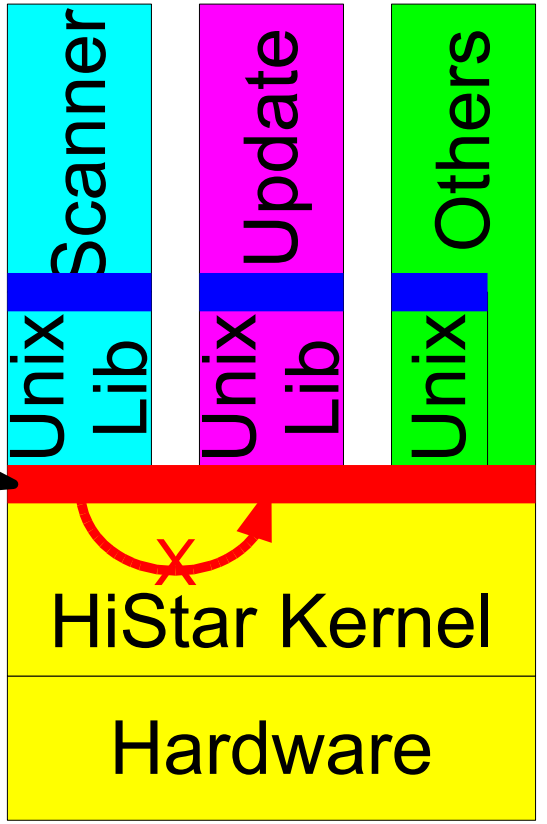
- Any thread can request a new category (color)
  - Gets ownership of that category ()
  - Uses category in labels to control information flow



HiStar mechanisms

Mechanisms:

6 kernel object types,
Labels for protection

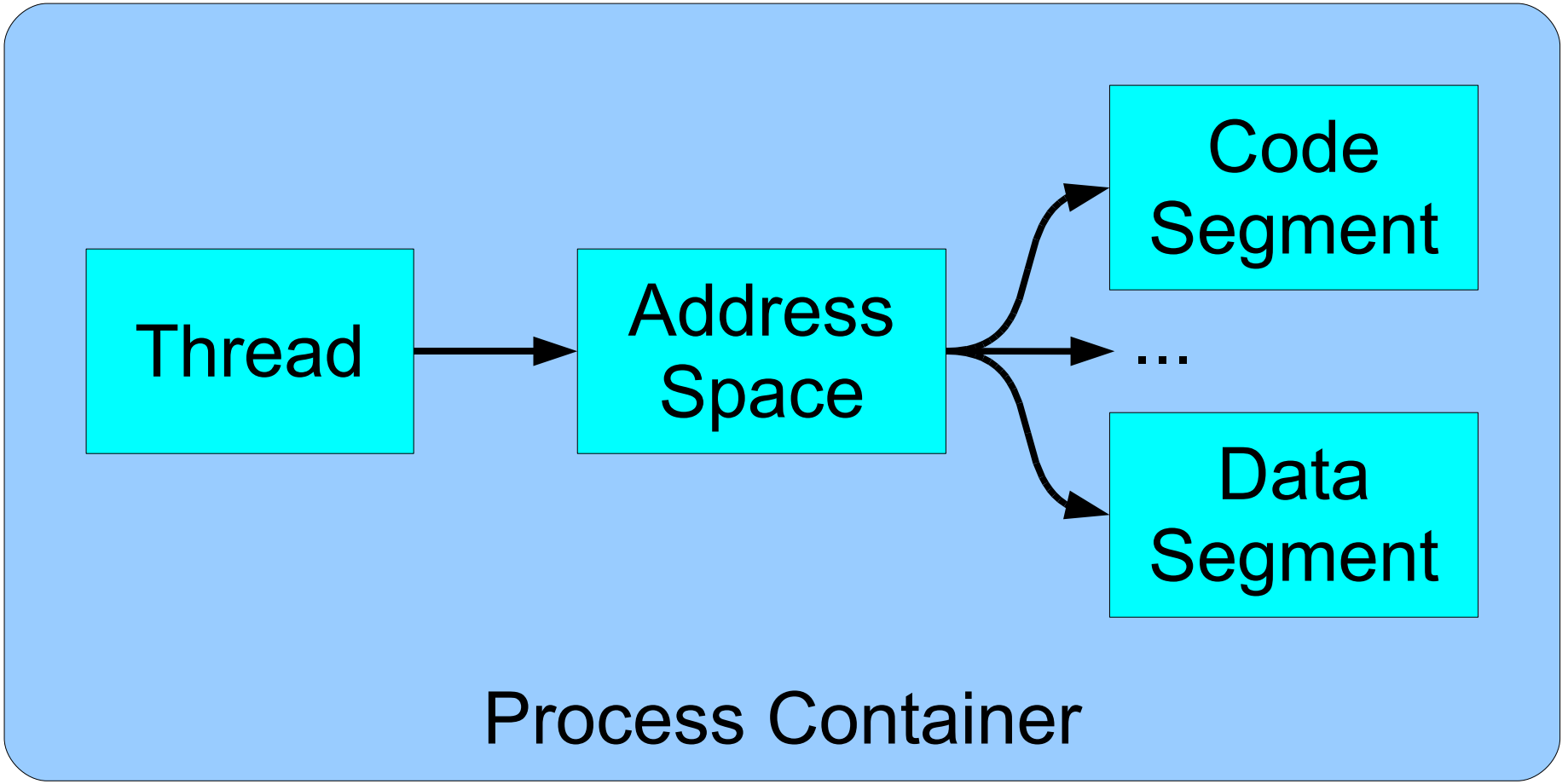


HiStar

HiStar outline

1. Kernel mechanisms: objects and labels
2. Example uses of these mechanisms
3. How these mechanisms improve security
4. Applications

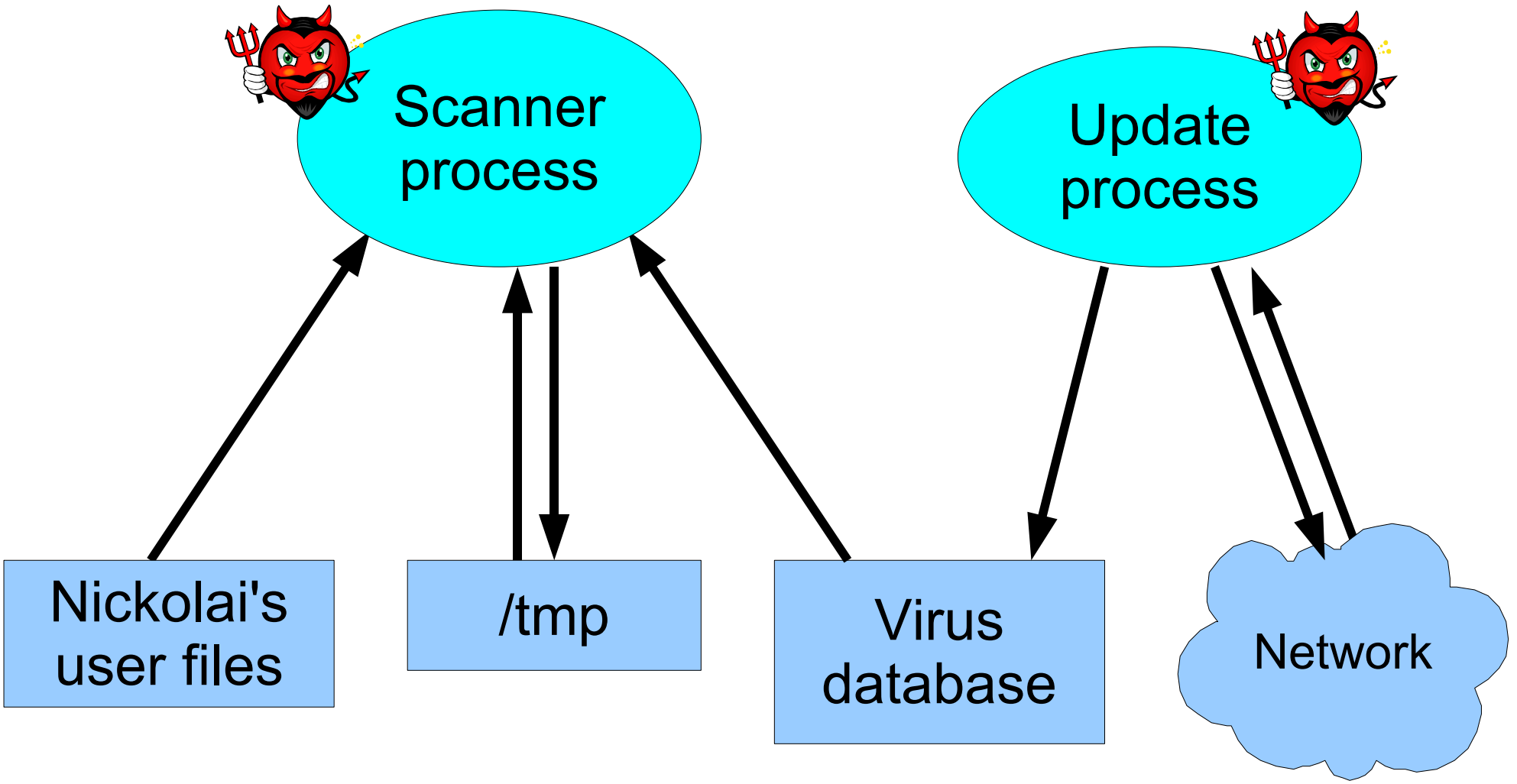
HiStar: Unix process



HiStar looks like a Unix system

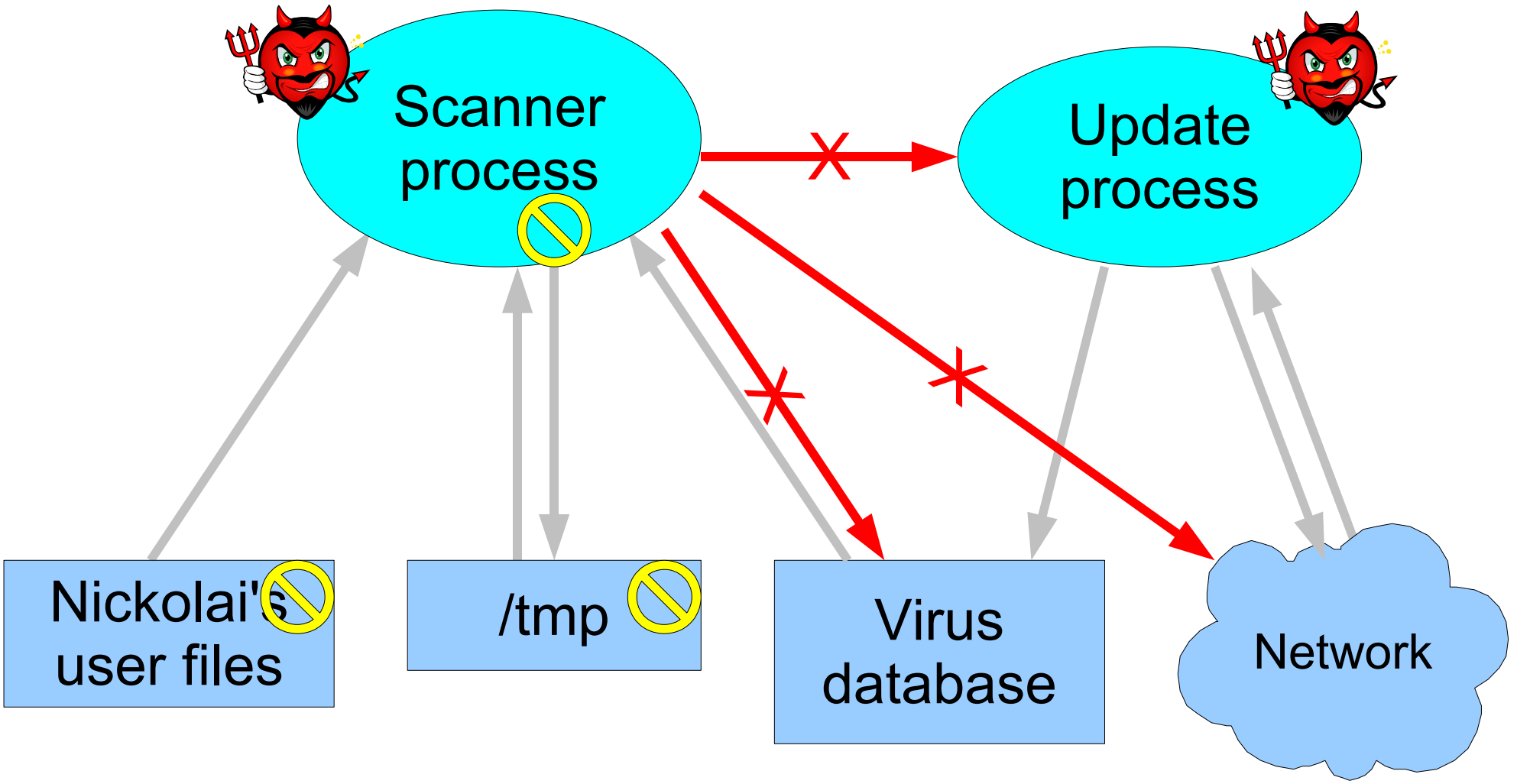
- Laptop running HiStar
- Quick demo
  - ls /, id, ps, ...

Recall:
Virus scanner example

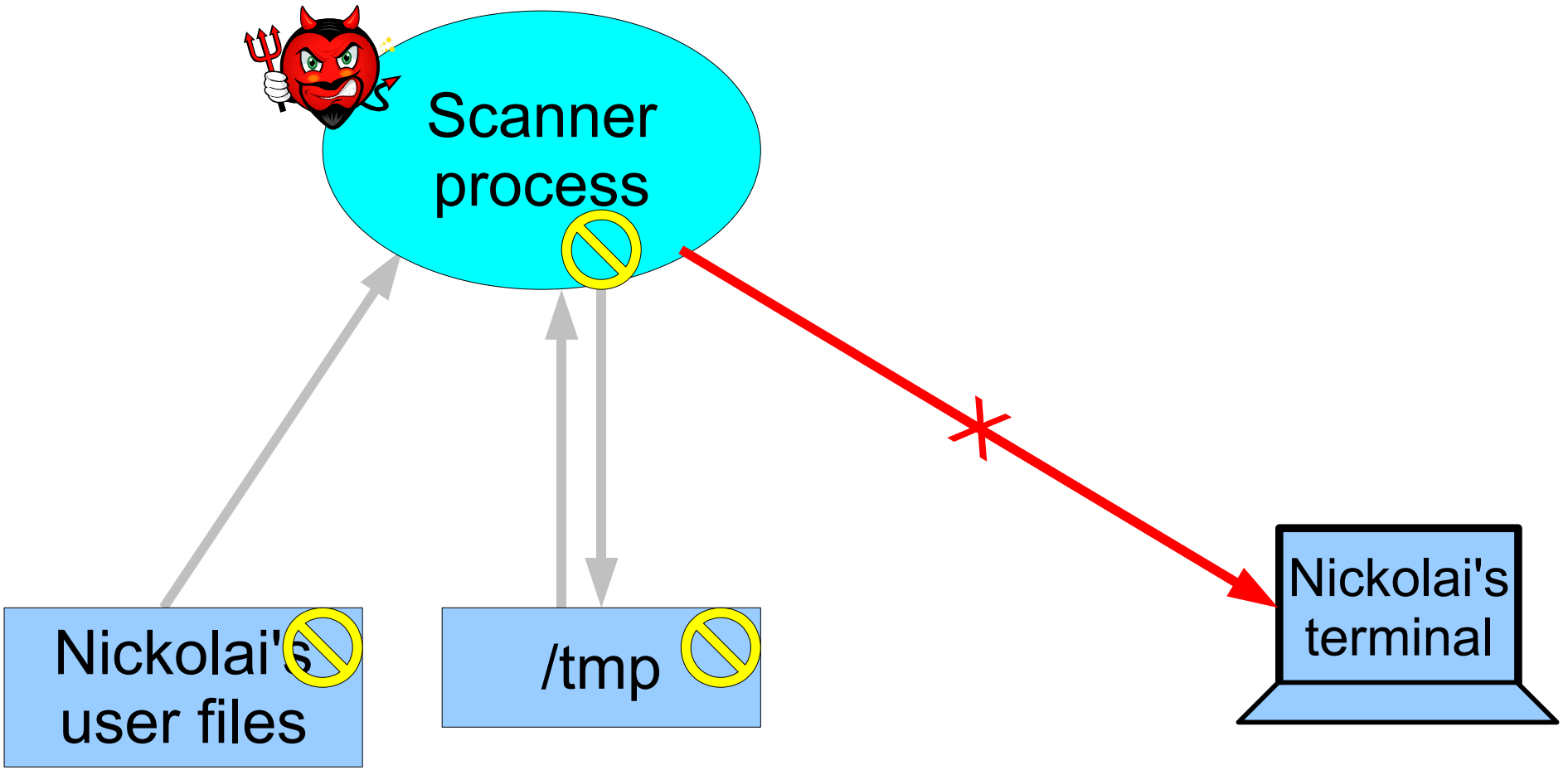


Malicious virus scanner cannot leak private data on HiStar

No need to audit code for security!

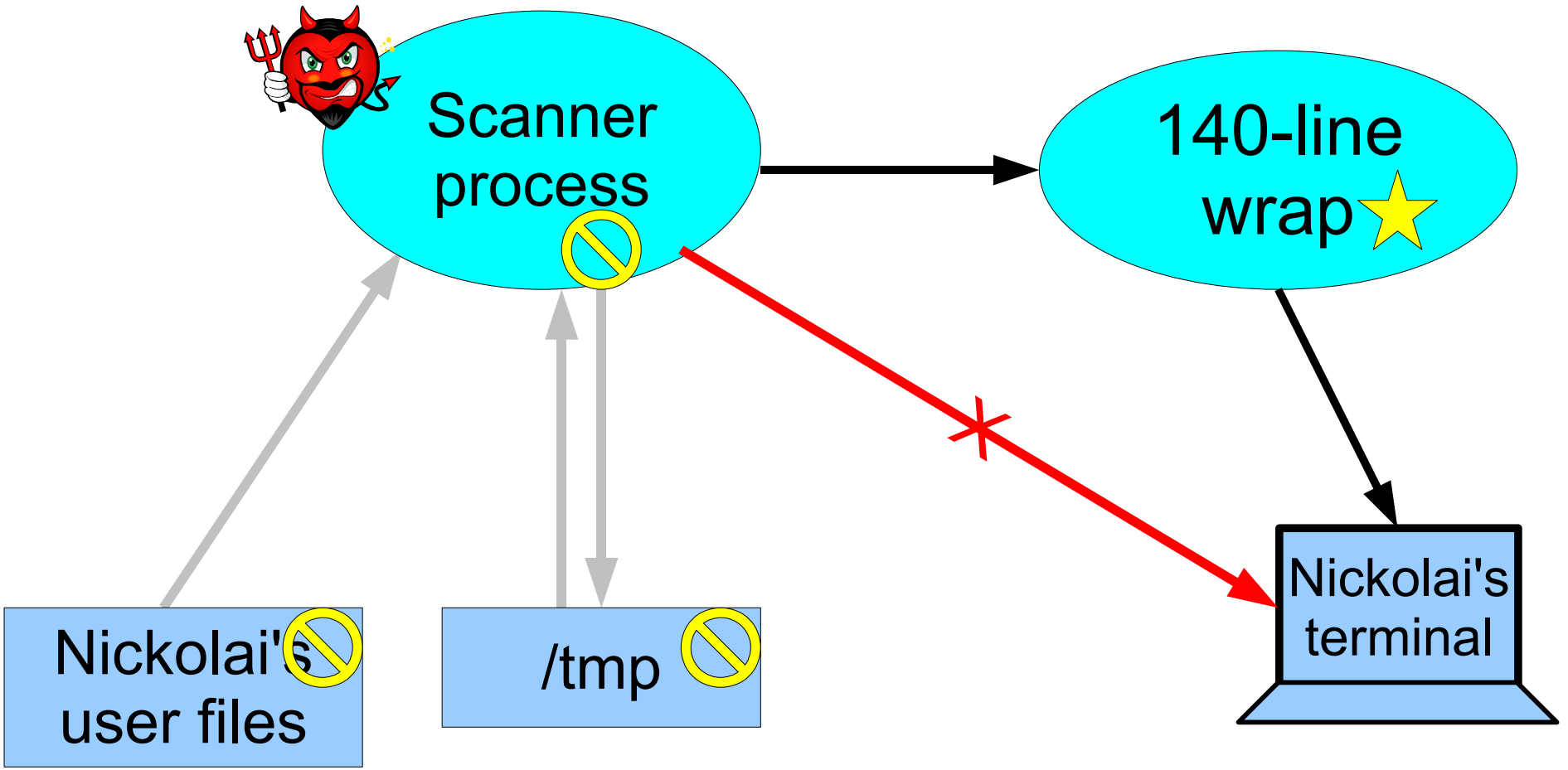


How do I get the output?



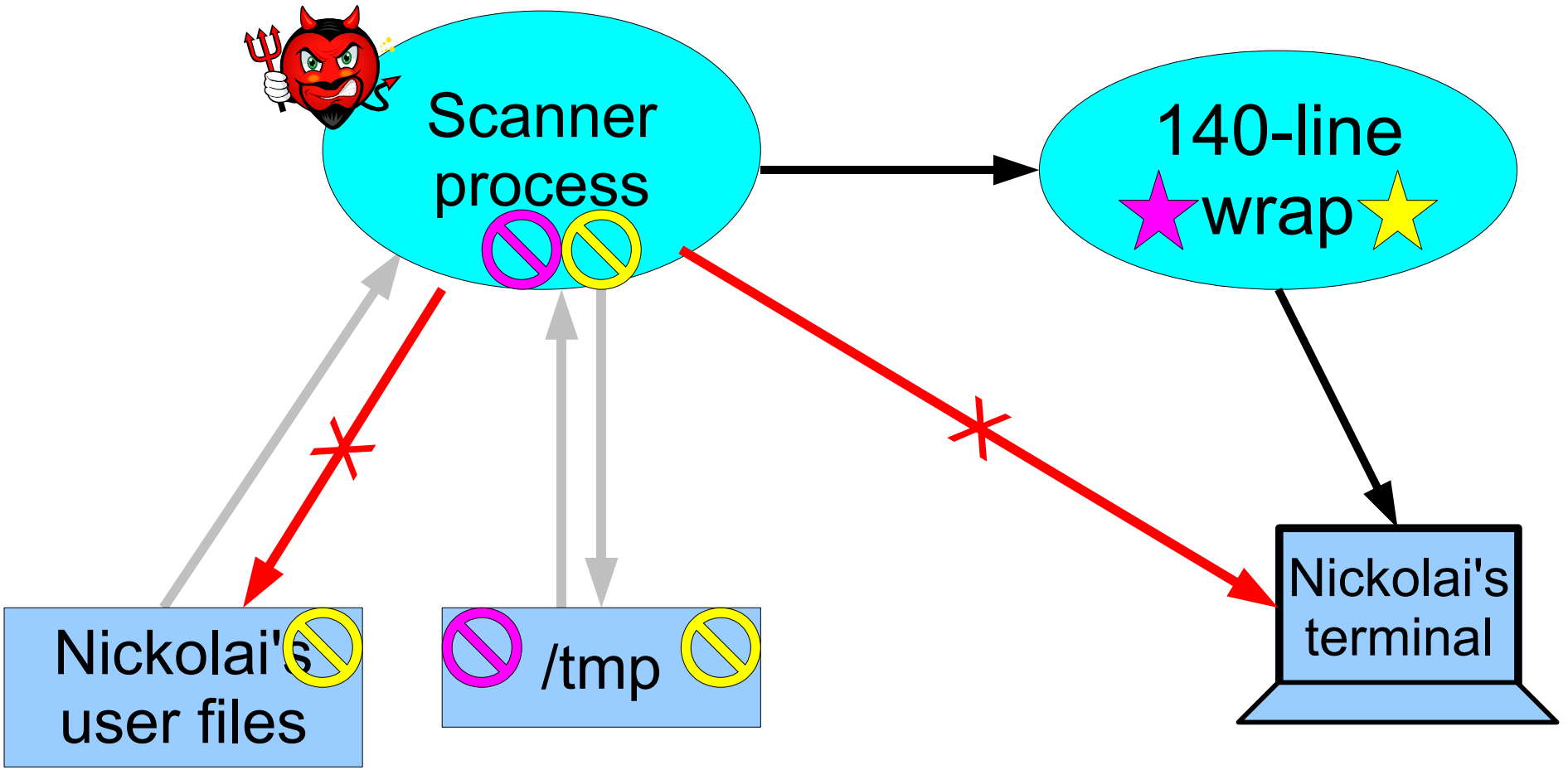
“wrap” sends data only to terminal

- 140-line trusted “wrap” can isolate a large, frequently-changing application



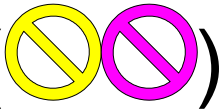
IFC also prevents file corruption

- 140-line trusted “wrap” can isolate a large, frequently-changing application



Another quick demo

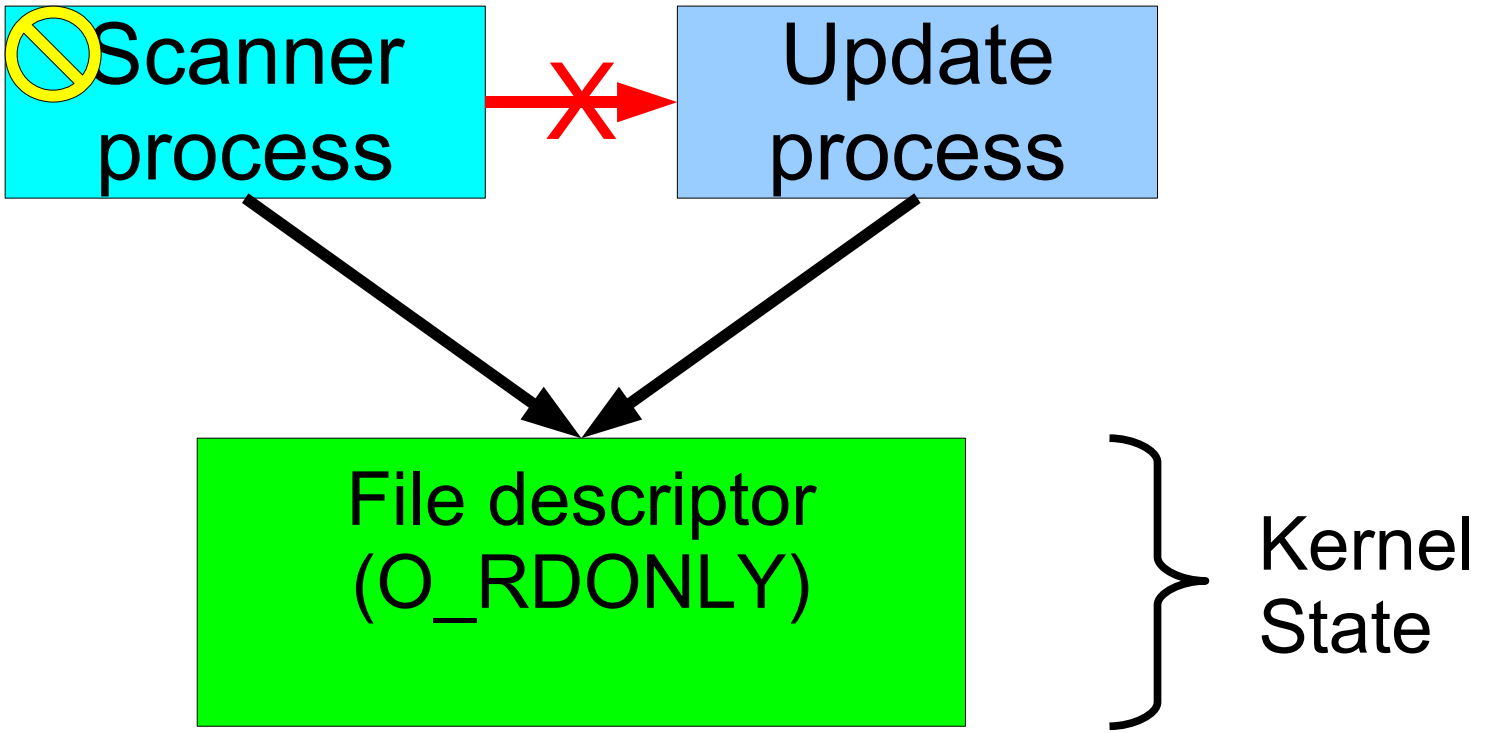
- “wrap” program runs standard Unix apps, but:
  - App cannot corrupt my files
  - App cannot divulge my data over the network

- Sets label () and runs app: 140 lines of code
- Kernel enforces policy at low-level interface

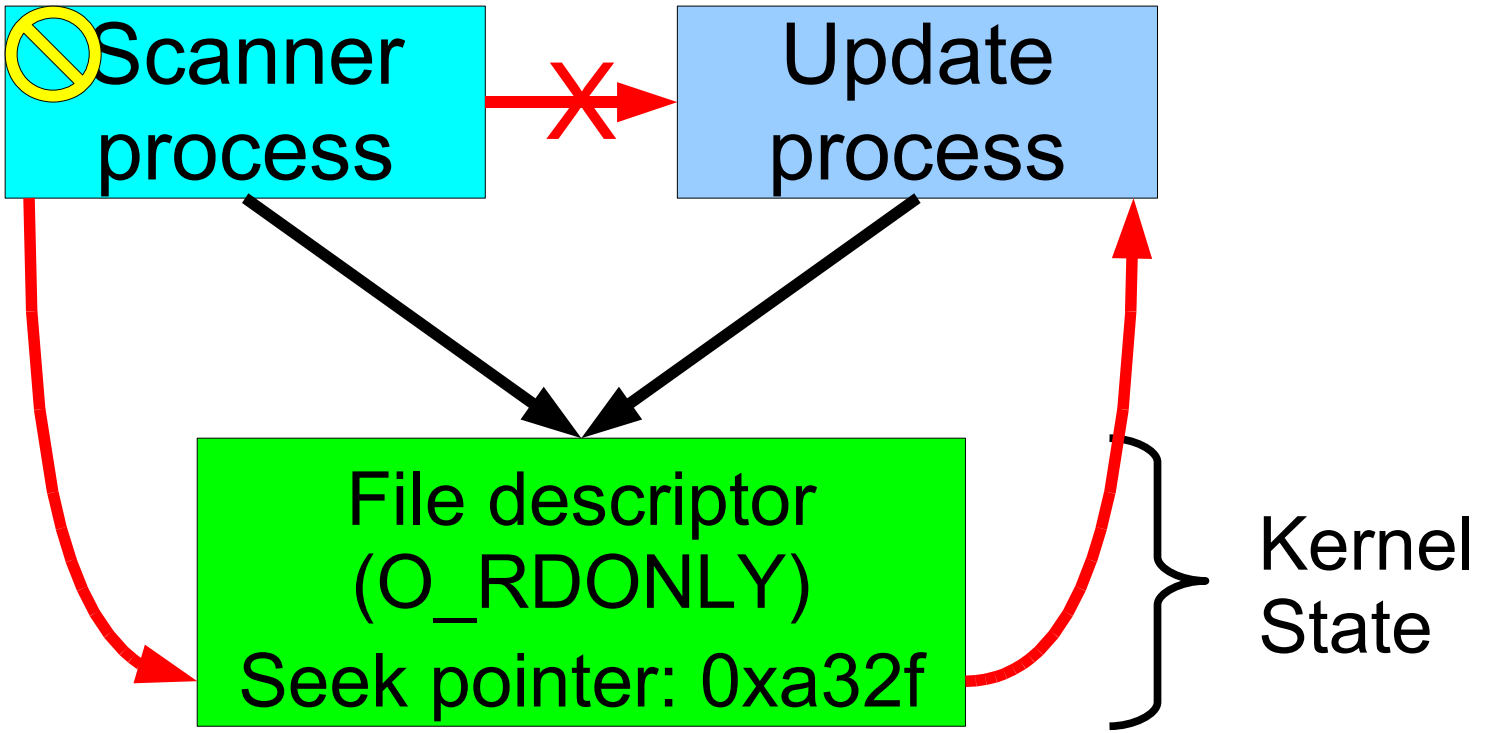
HiStar outline

1. Kernel mechanisms: objects and labels
2. Example uses of these mechanisms
3. How these mechanisms improve security
4. Applications

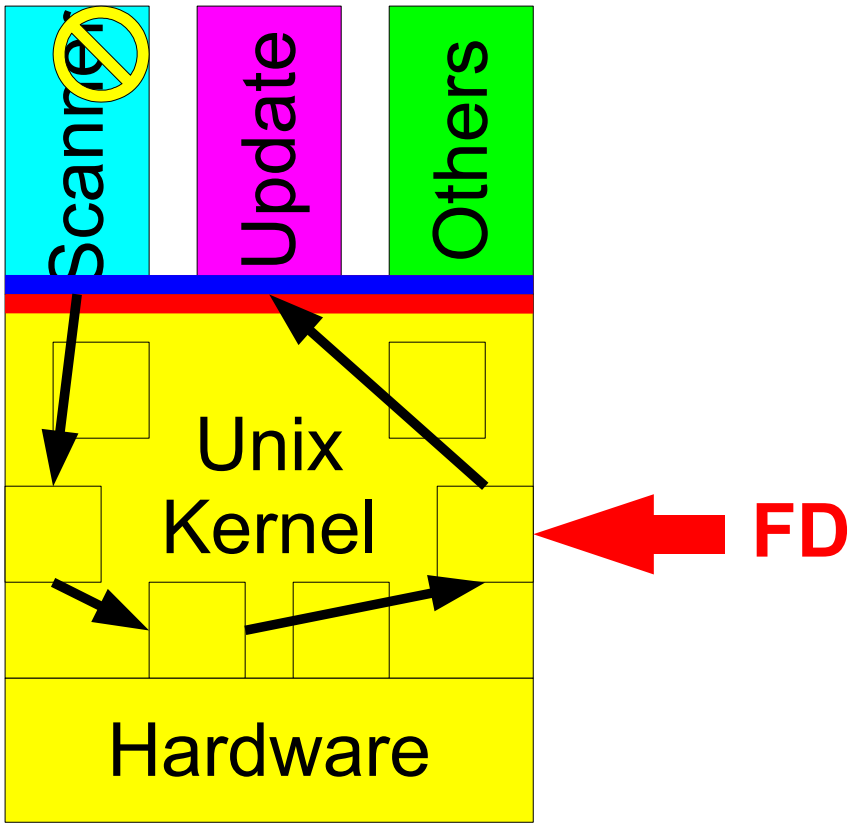
Example: Unix file descriptors



Unix file descriptor seek pointer leaks data



Unix implements FDs in kernel

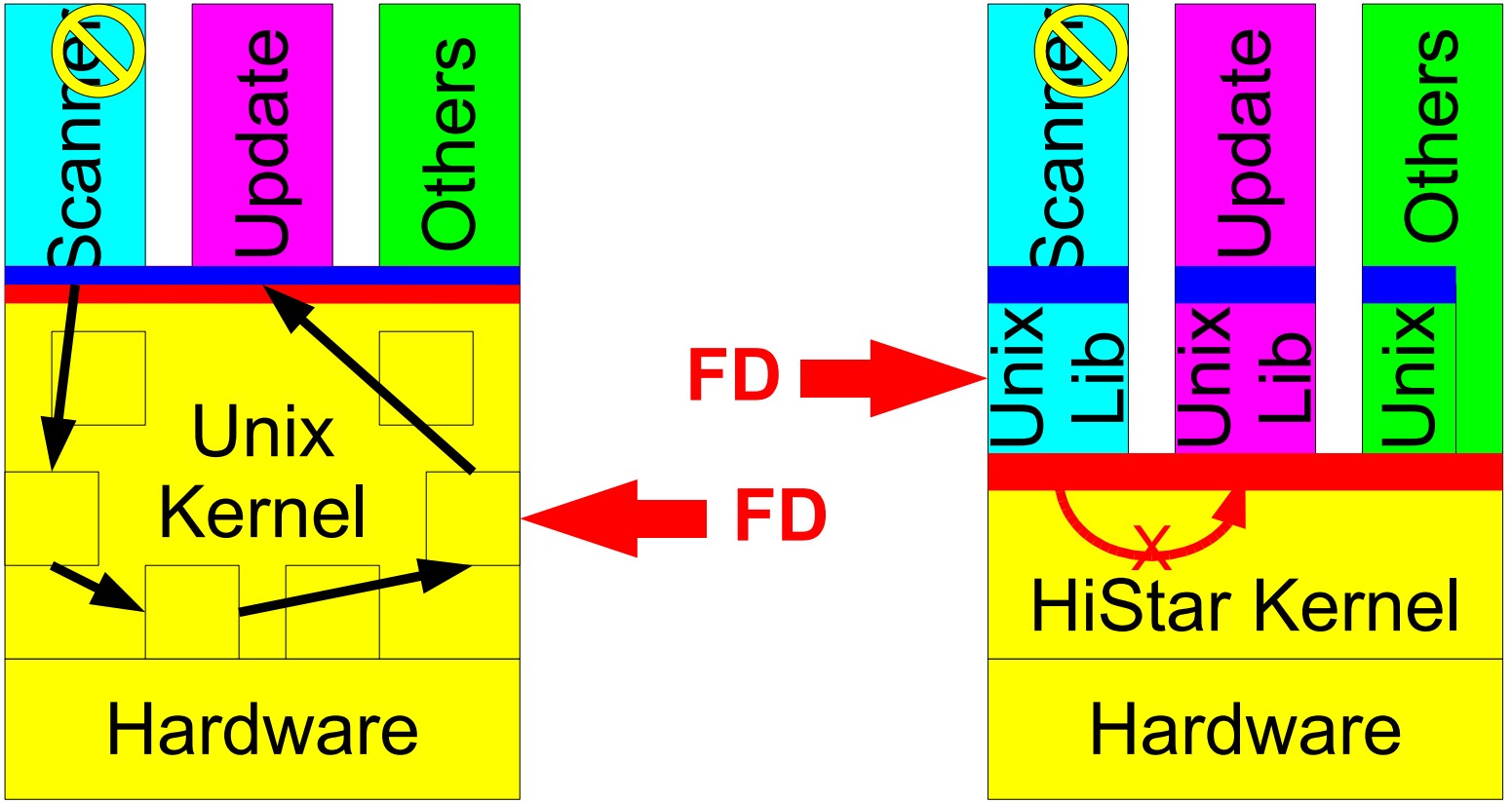


Unix

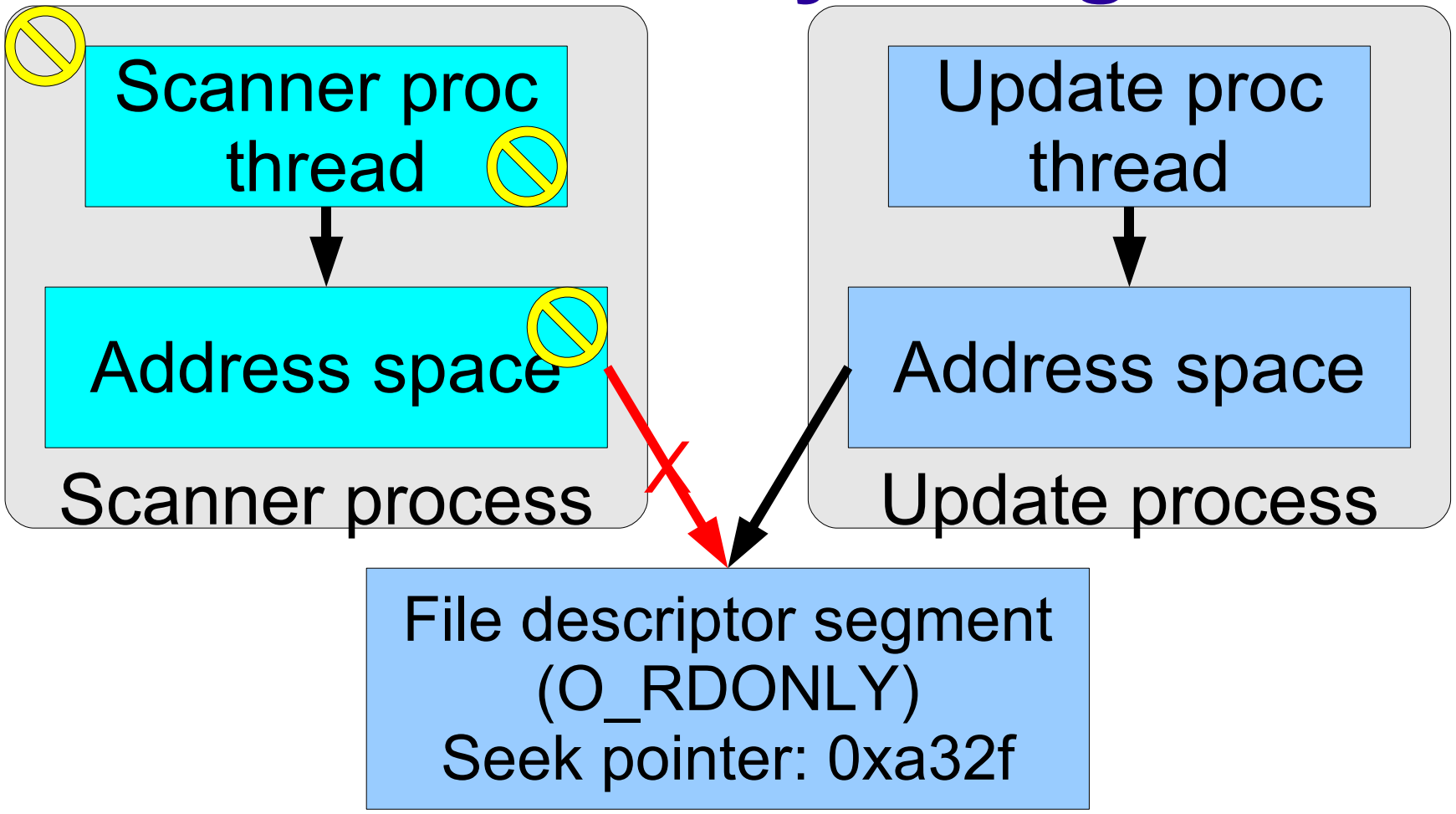
- Unix has lots of shared state, easy to miss some
- Hard to enforce data security at Unix API level

HiStar implements FDs in library

- HiStar FDs above security boundary (red line)



No analogous problem on HiStar by design



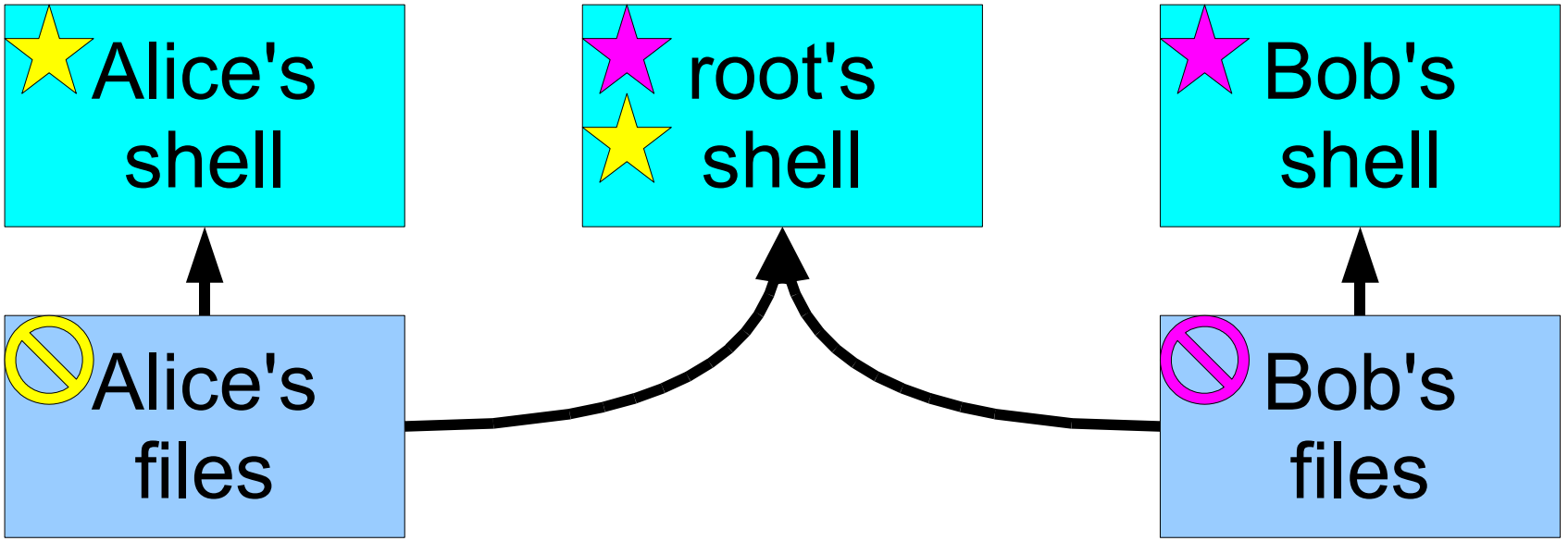
Only one mechanism: object R/W checks!

Labels are the security building block

- Low-level mechanism
  - Controls information flow between objects
- Expressive
  - Used to implement Unix user IDs, groups, etc
  - Can be also used for other policies (wrap)
- Egalitarian
  - Anyone can allocate a new color, gets star
  - No inherent superuser rights for administrator

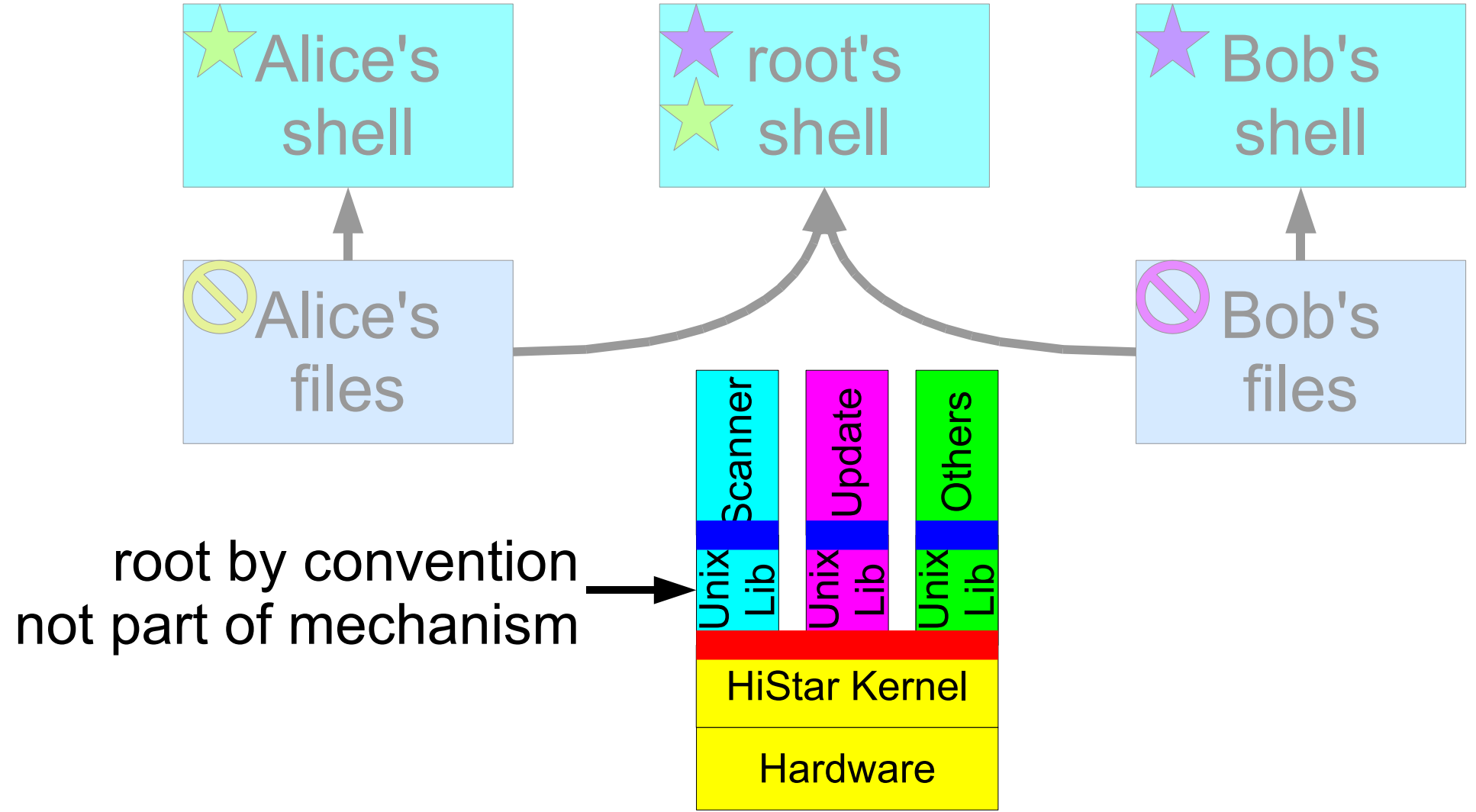
HiStar has no inherent superuser privileges

- By convention, root gets stars for backup, etc



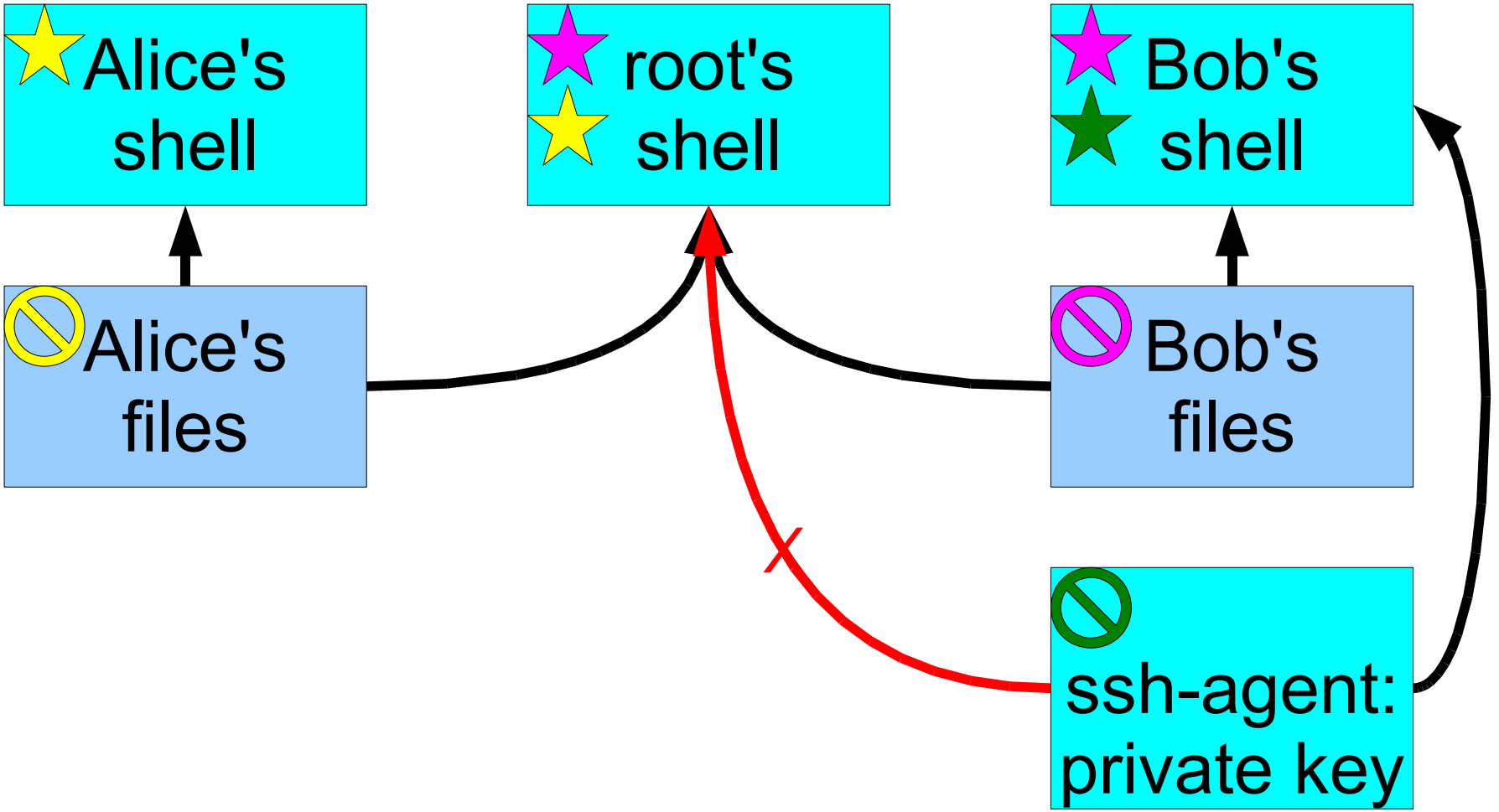
HiStar has no inherent superuser privileges

- By convention, root gets stars for backup, etc



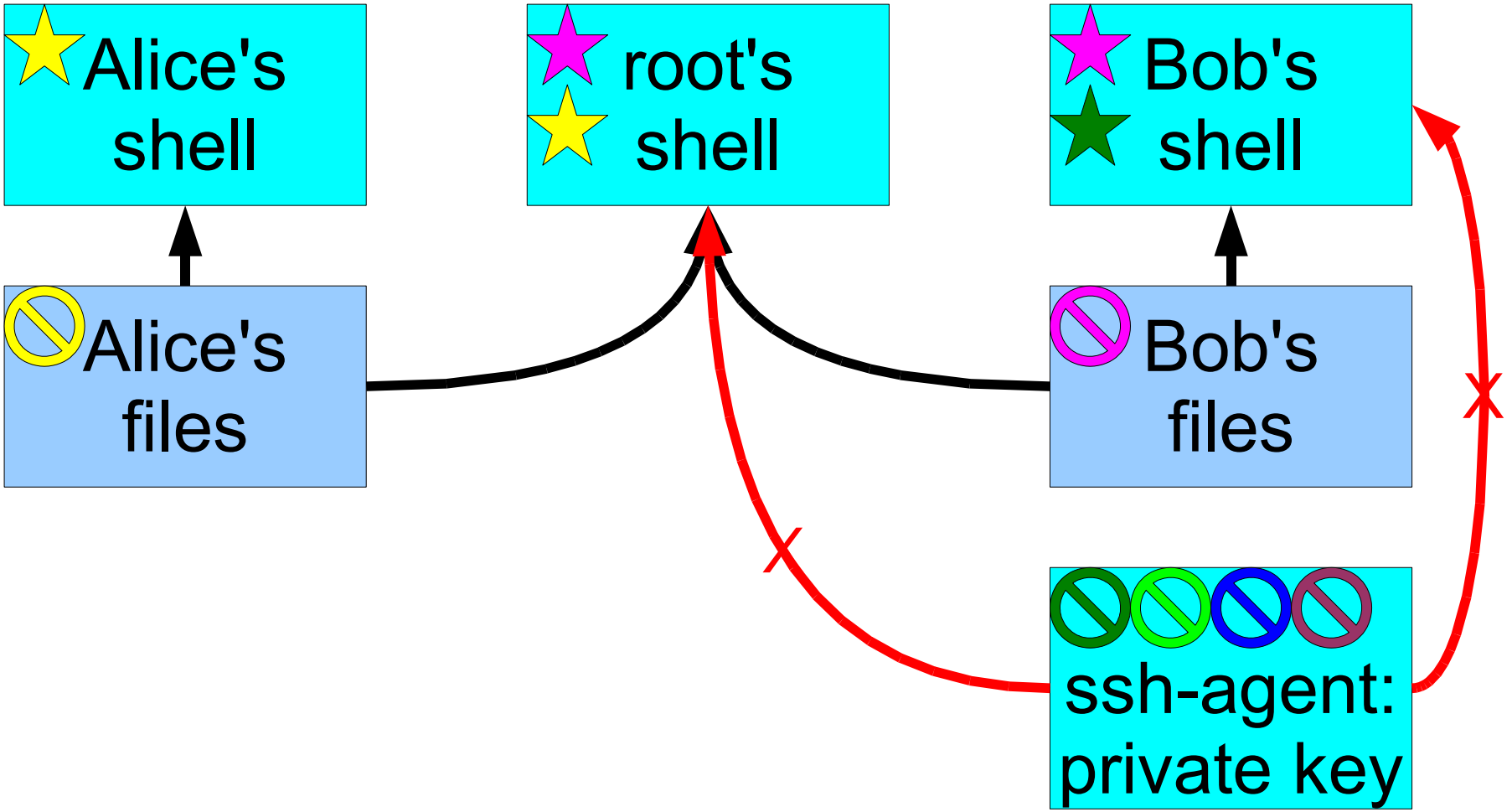
HiStar has no inherent superuser privileges

- Users can keep data inaccessible to root

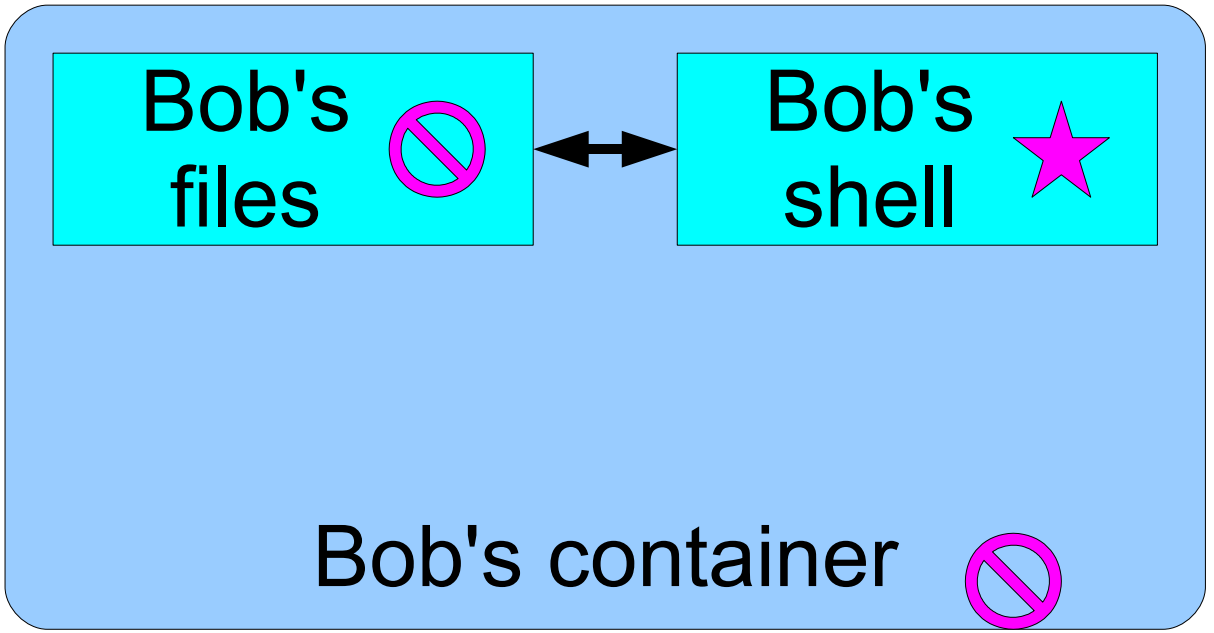


Runaway process wasting CPU?

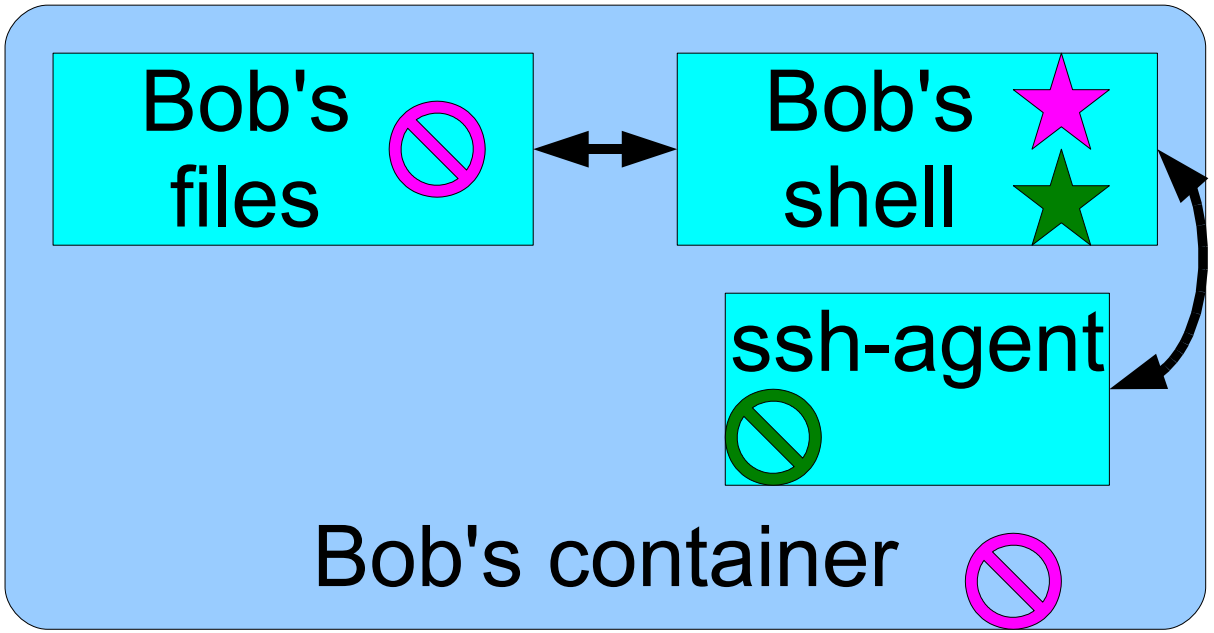
- Nobody has privilege to access ssh-agent now!



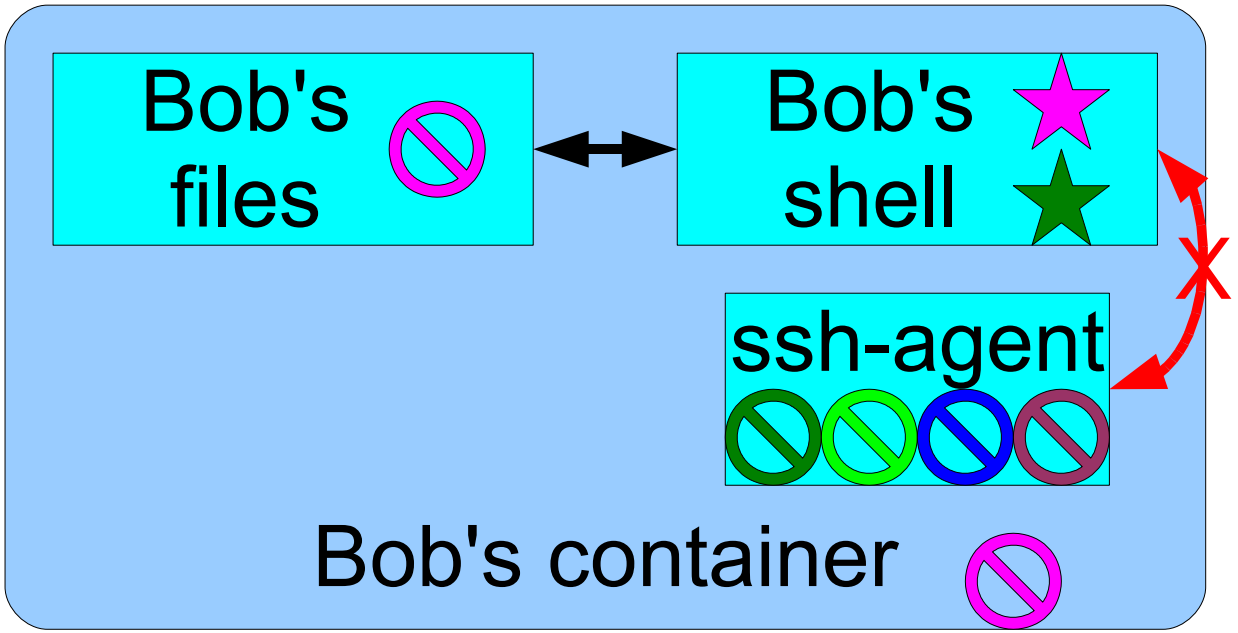
HiStar resource allocation



HiStar resource allocation

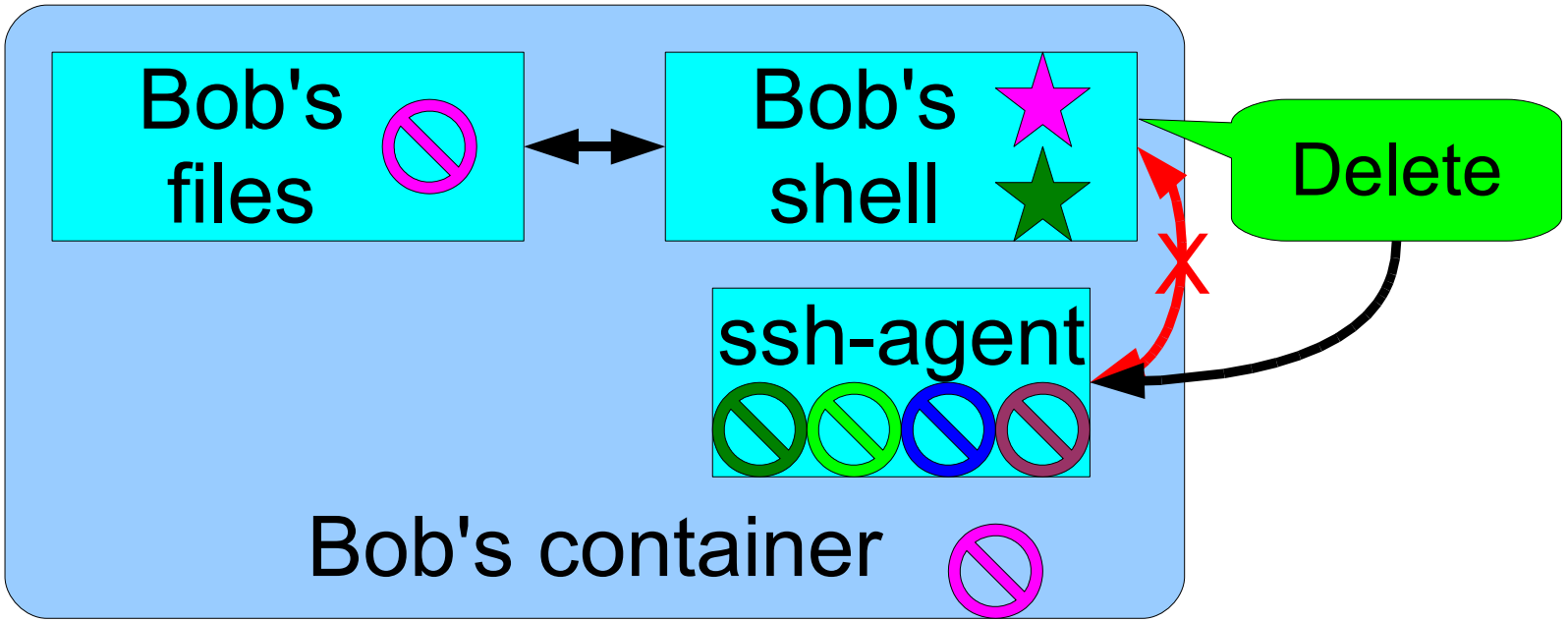


HiStar resource allocation



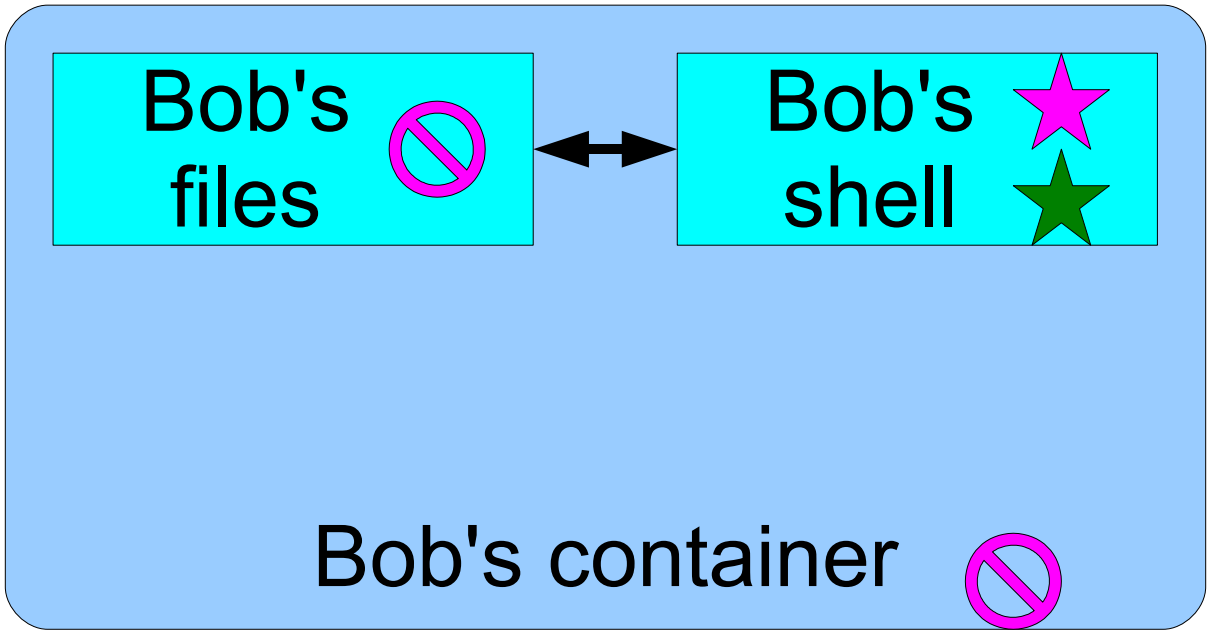
HiStar resource allocation

- Bob can delete key even if he cannot otherwise access it!



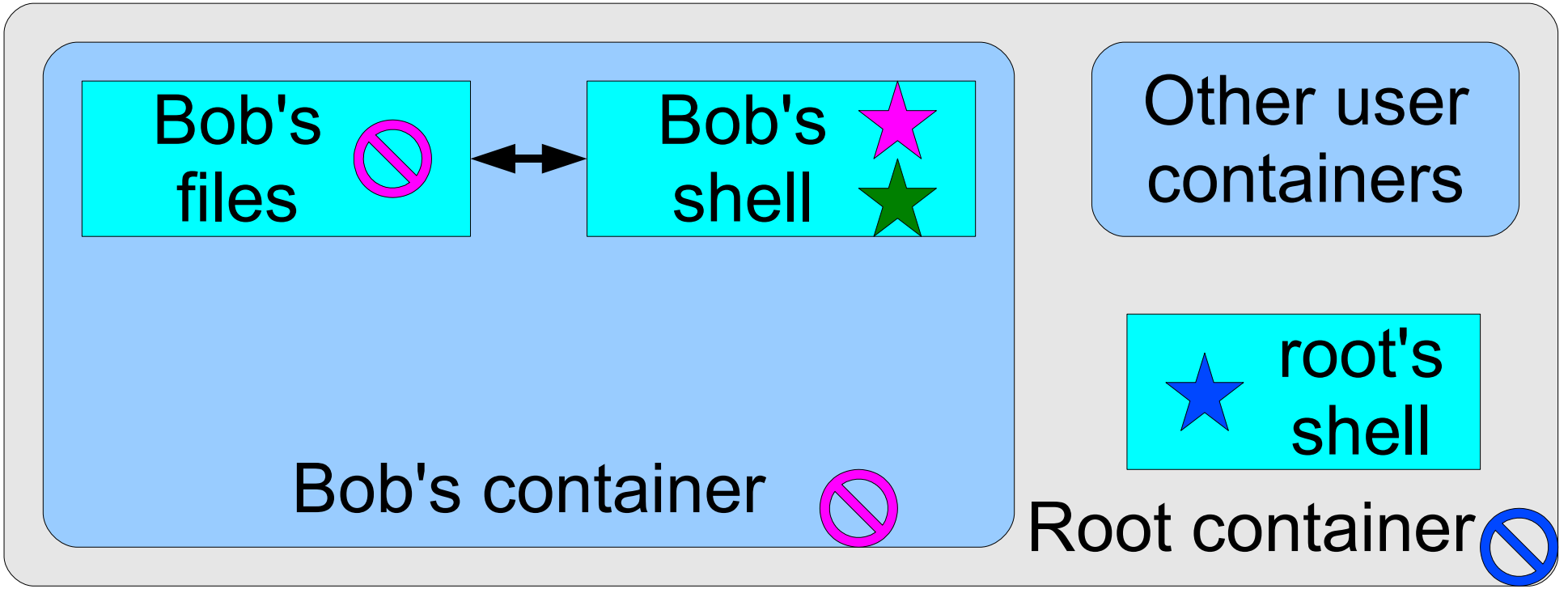
HiStar resource allocation

- Bob can delete key even if he cannot otherwise access it!



HiStar resource allocation

- Root controls resources without data access
  - Compromised sysadmin can't access all user data
  - Can revoke resources of compromised or bad users

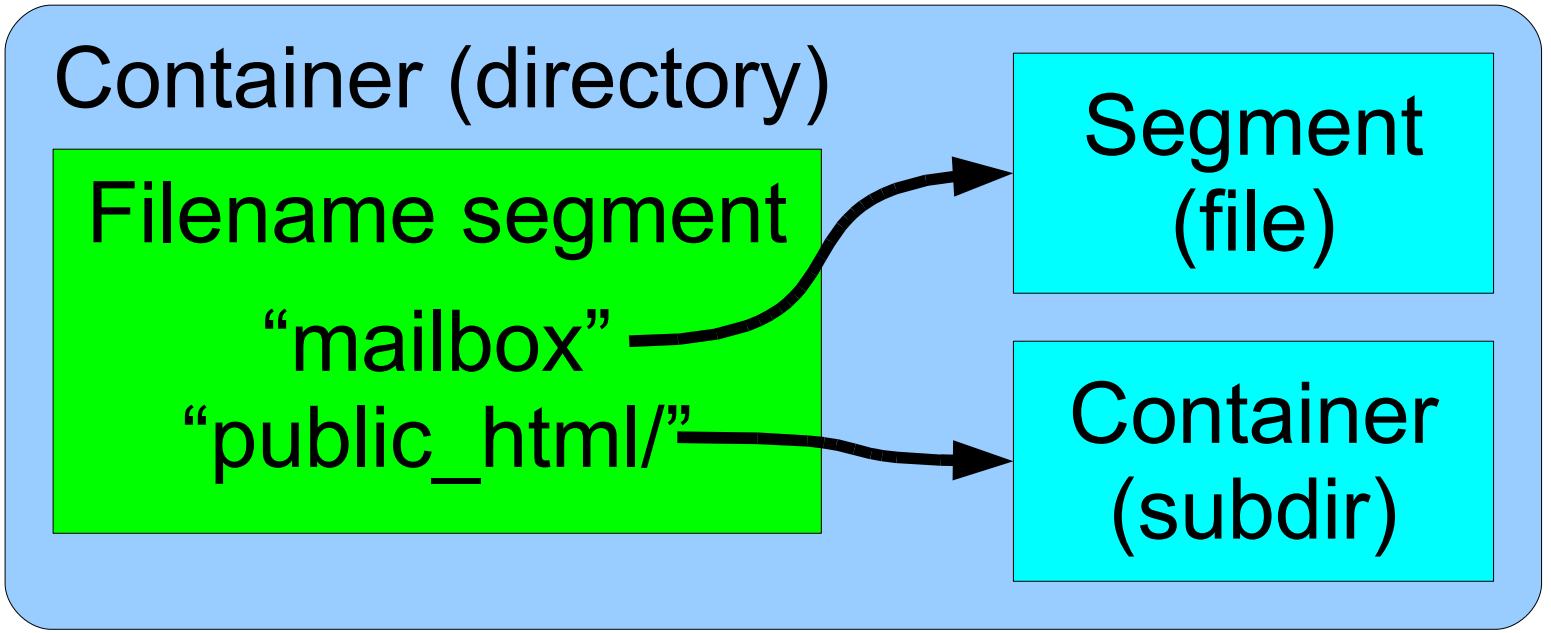


File system mechanisms

- File system requires persistent store mechanism
- Unix: separate mechanism for disk access
  - Kernel provides two levels of storage: memory, disk
  - Different way of specifying security for memory, disk
- HiStar: existing mechanisms are sufficient
  - Kernel provides a single level of storage [Multics, EROS]
  - All kernel objects stored on disk; memory is a cache

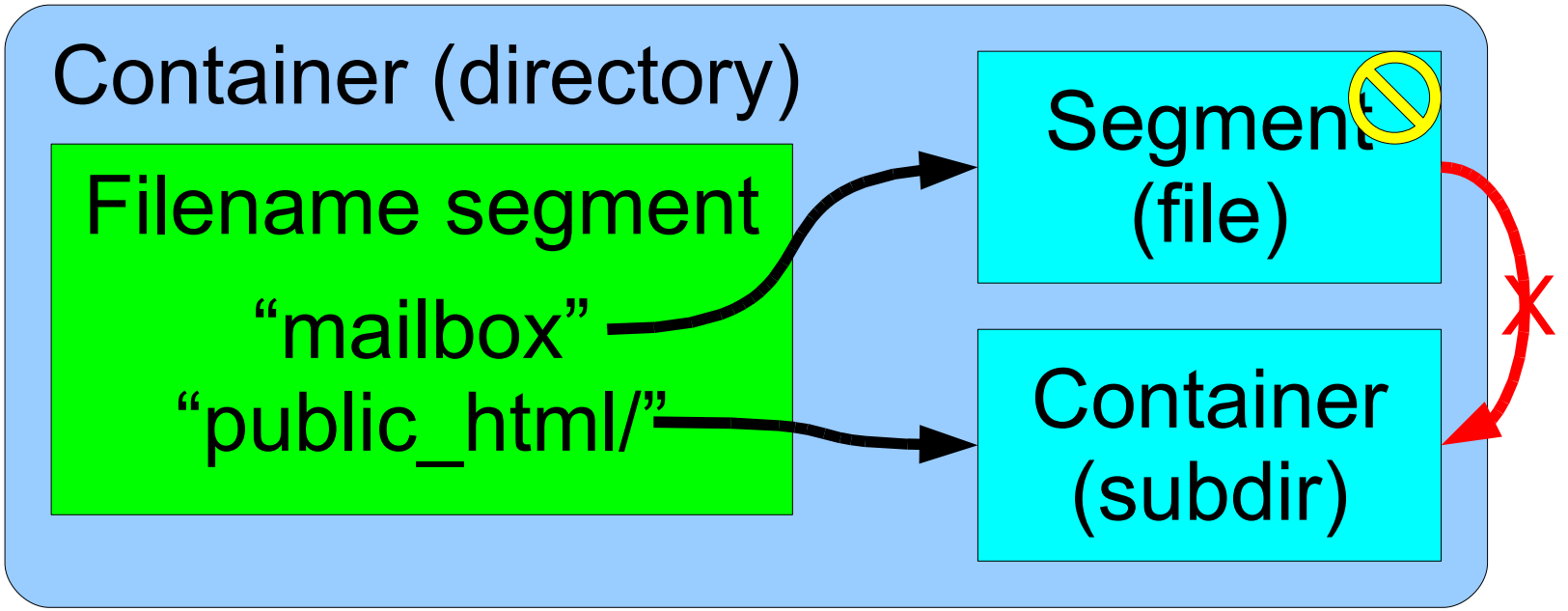
HiStar file system reuses kernel mechanisms

- Implemented at user-level, using same objects



HiStar file system reuses kernel mechanisms

- Implemented at user-level, using same objects
- Uniform protection for memory and file system



Single-level store avoid superuser

- Unix: root must start everything at bootup
  - Must have superuser privileges
- HiStar: processes don't notice hardware reboot
  - Another quick demo
  - Bob's ssh-agent continues running,
    does not trust root to restart it after reboot

How to *really* reboot (software)?

- Separate command called "`ureboot`"
  - Deletes all process containers, keeps FS containers
  - Start a new `init` process

- In a few slides:
  HiStar's *gates* allow `init` to not require superuser

How do we know if HiStar enforces IFC correctly?

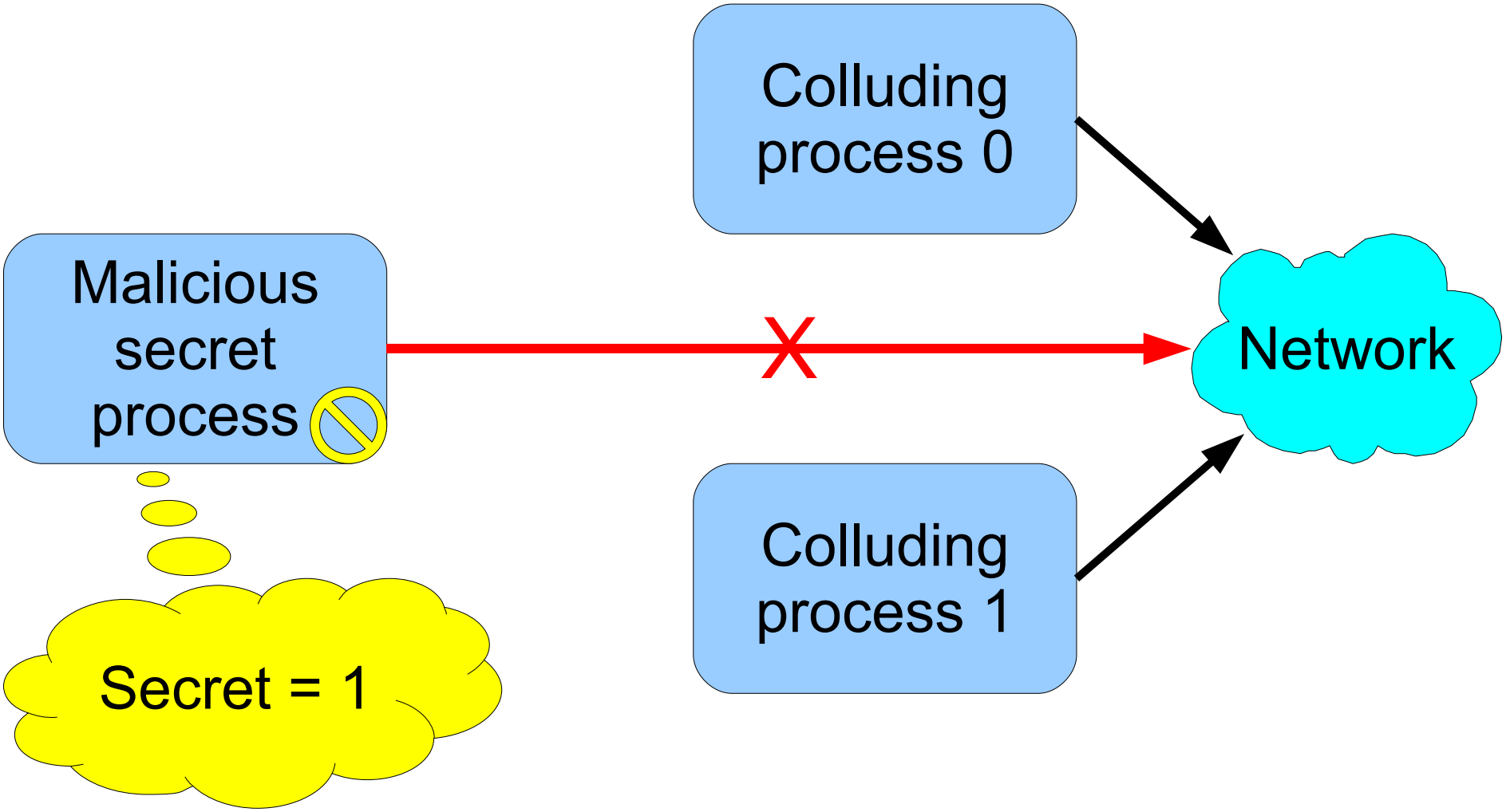
- Information flow precisely defines goal (labels)
  - Security check: can information flow from A to B?
- Long-term goal: verify implementation security
  - At least we know what to verify (unlike in Unix)
- Challenge: how to ensure design is sound?
  - Information flow – how to avoid “covert channels”?

Two kinds of covert channels

- Implementation covert channels: not in spec
  - Apps don't depend on these behaviors
  - E.g. timing channels: almost impossible to eliminate
  - Can mitigate bit-rate by introducing noise [Hu '91]

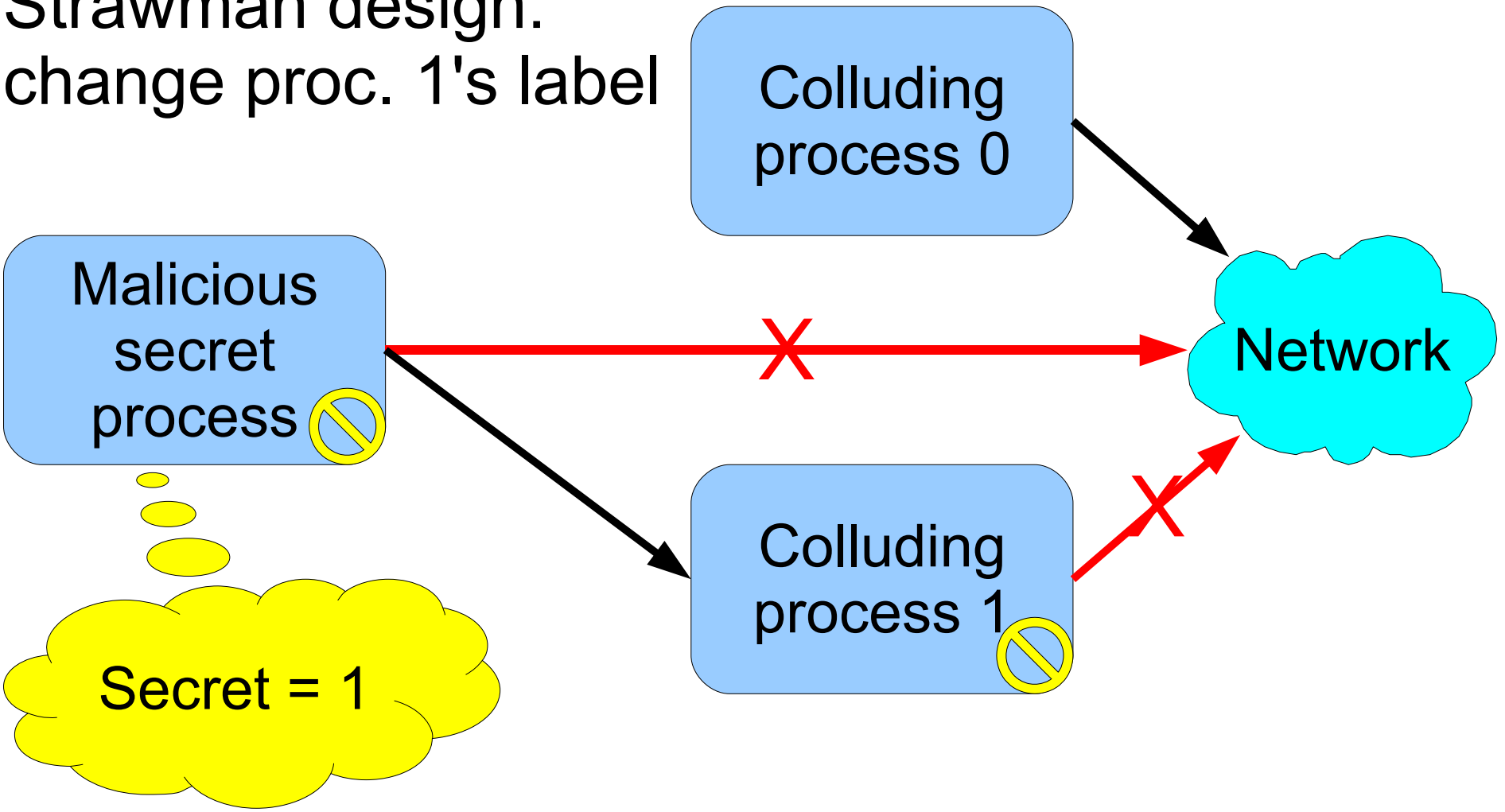
- Design covert channels: inherent in spec
  - Cannot fix or mitigate without breaking apps!
  - HiStar: explicitly label everything in spec
  - Challenge: what about the labels themselves?
    (Similar to the Unix FD seek pointer problem shown earlier)

Example of covert channel through labels



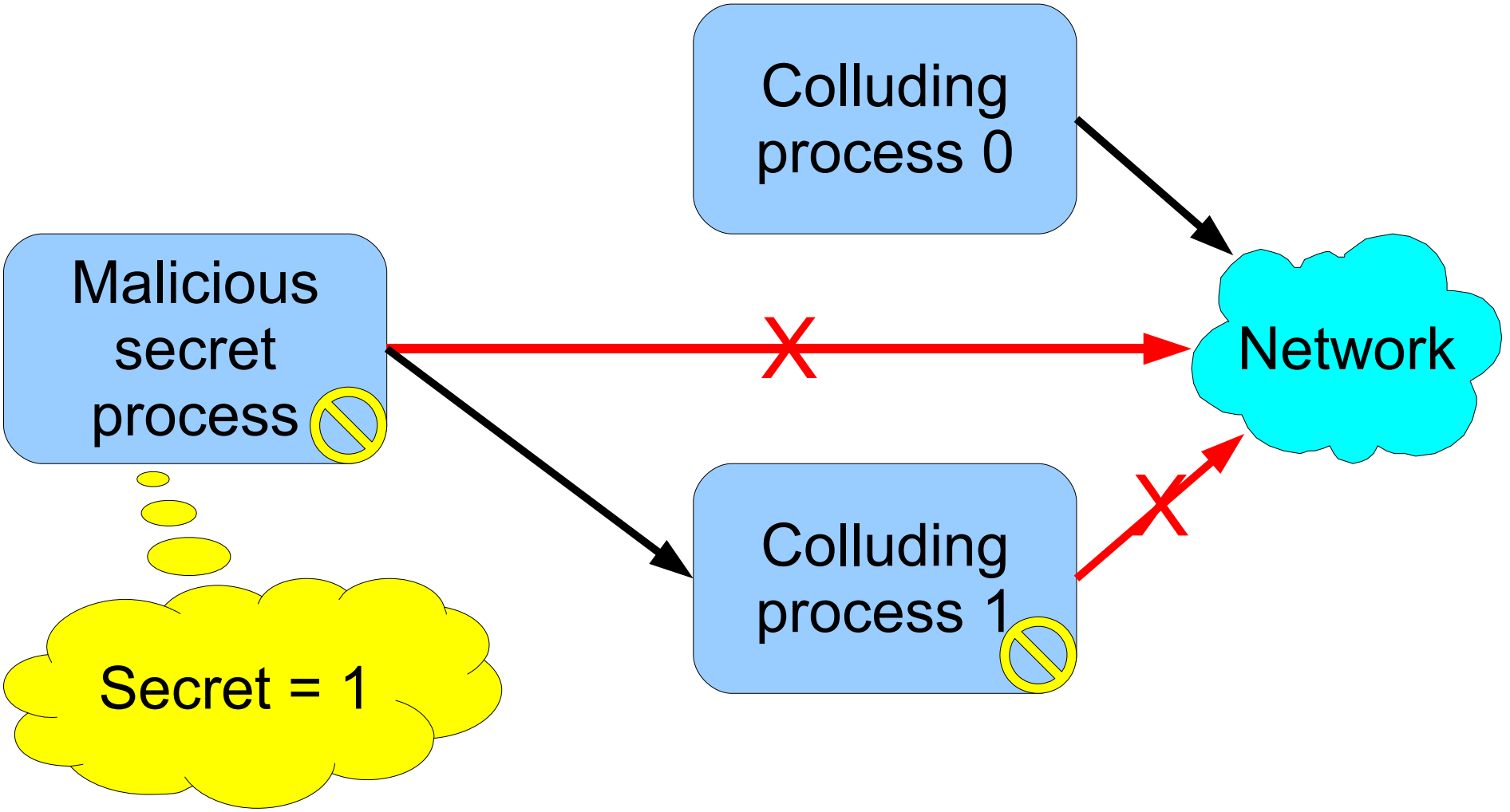
Example of covert channel through labels

- Malicious process sends message to process 1
- Strawman design: change proc. 1's label



Process 1's label change leaks data in strawman

- The fact that process 1 can no longer talk to network is observable



Previous approaches to this problem

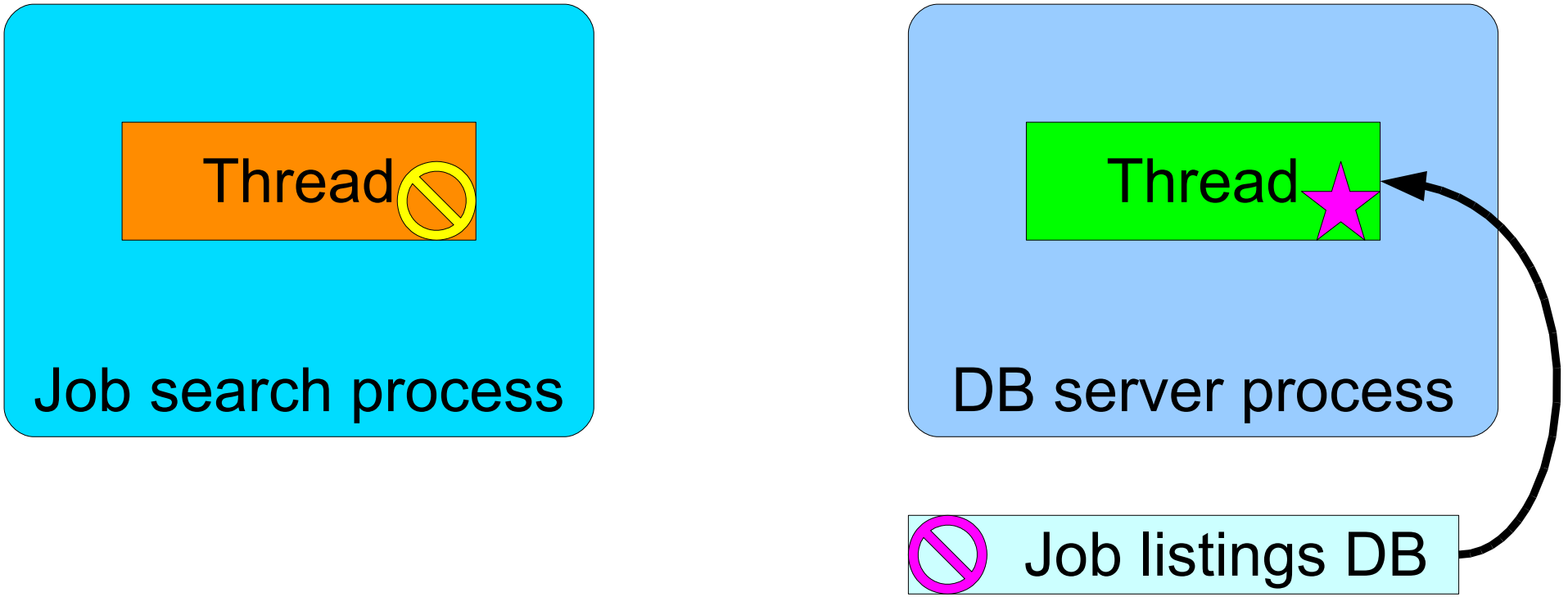
- Programming languages: compile-time check [Jif]
  - Cannot do “wrap” – requires dynamic runtime labels
- Some OSes gave up, didn't find an answer [IX, Asbestos]
  - Claims in literature this covert channel is inevitable
- Military systems: fixed labels [Adept-50, VMM SecKern, KeySafe]
  - Few labels (secret, top-secret) – cannot do “wrap”

Key idea to avoid covert channels

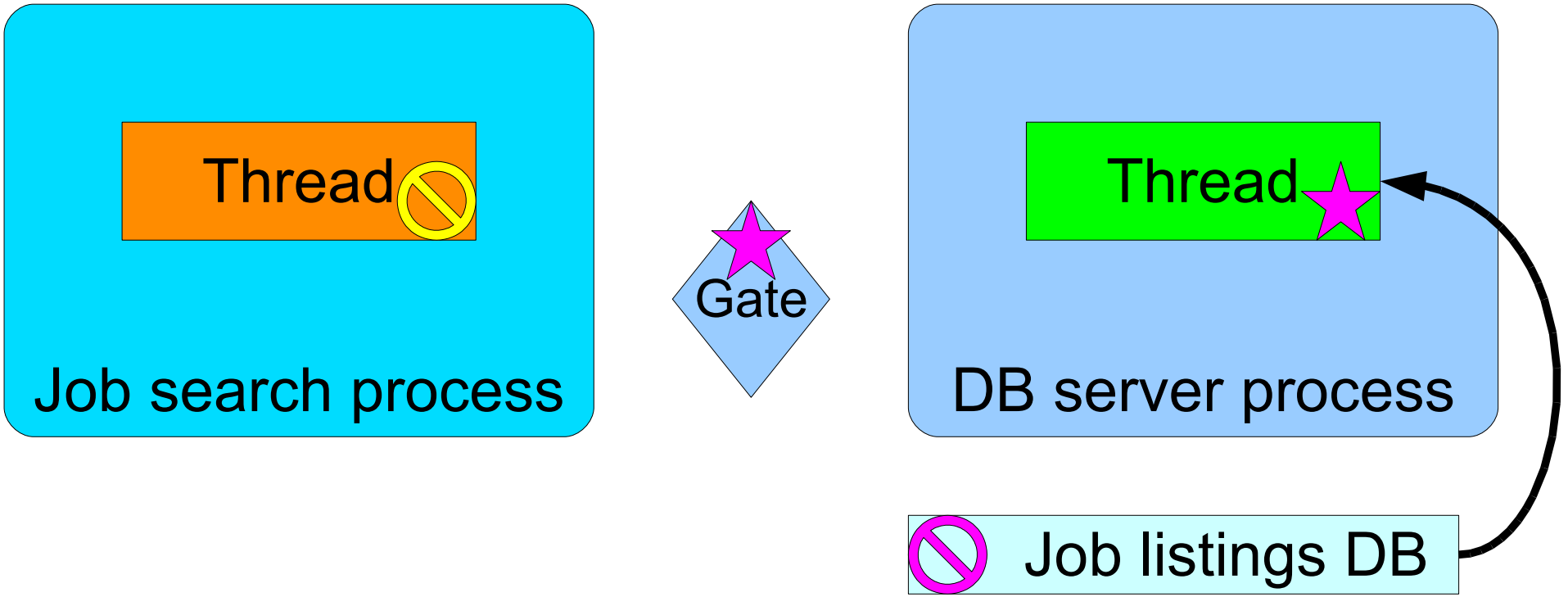
- Non-thread objects have immutable labels
  - Attacker cannot communicate via labels
  - But this is not dynamic enough, so...
- Threads can only change their own label
  - Does not leak any data thread didn't already have
    - (Of course, can only add, not remove categories)

Inter-process communication with immutable labels

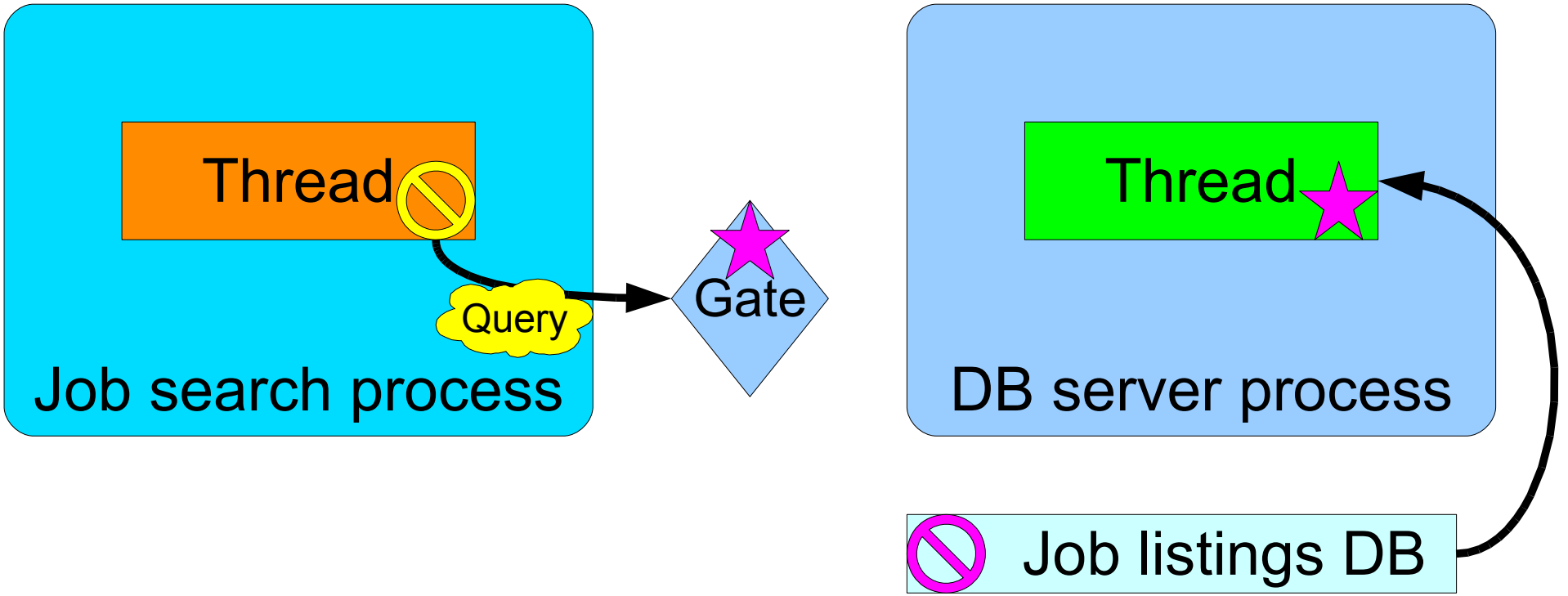
- Job search site: query DB for matching listings
  - Goal: privacy of queries and DB, even if search code is bad
  - Can two differently-labeled processes communicate?



HiStar gates
change label of client thread

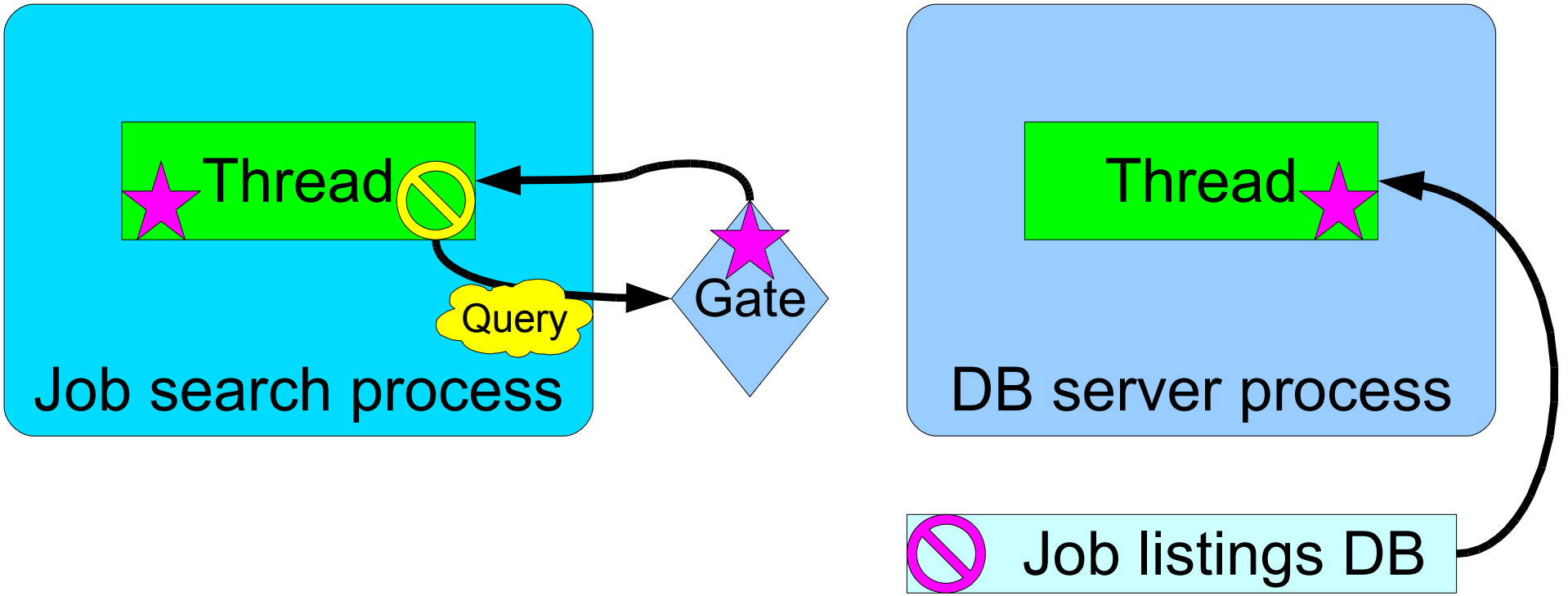


HiStar gates
change label of client thread



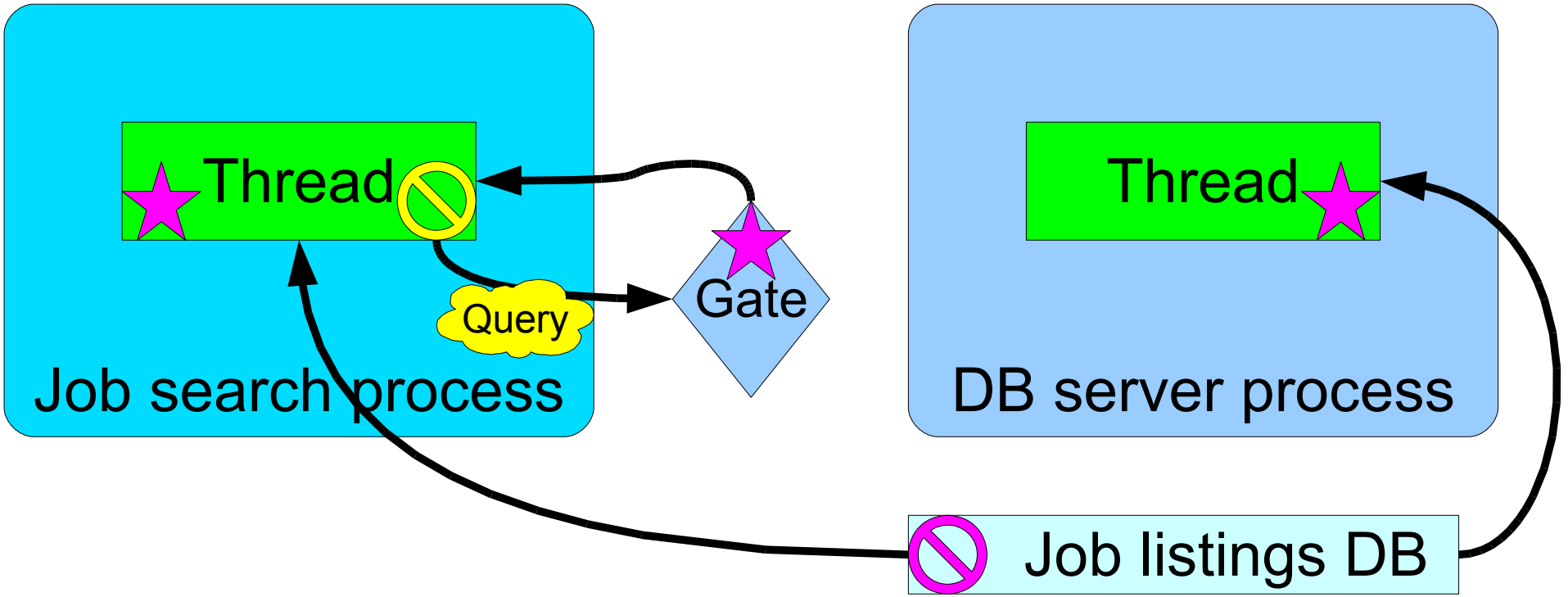
HiStar gates change label of client thread

- Gate uses client thread to execute server code
  - With server privileges but without server resources
  - Just like before, threads change their own label



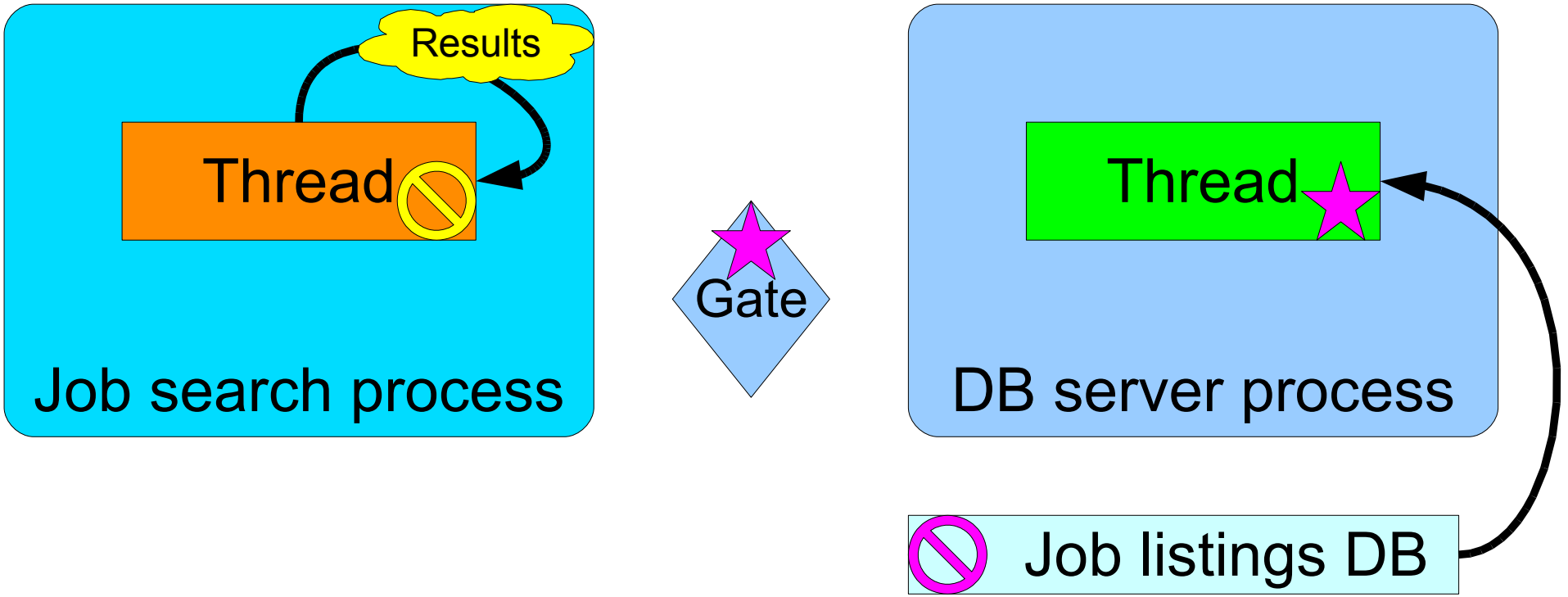
HiStar gates change label of client thread

- Gate uses client thread to execute server code
  - With server privileges but without server resources
  - Just like before, threads change their own label



HiStar gates change label of client thread

- Gates enable inter-process communication with immutable labels (+ threads change own label)



Gates help avoid superuser

- Gates store privileges across restarts
  - Database gate persists across reboot & ureboot
  - No need for superuser to restart database, even though ureboot kills all threads

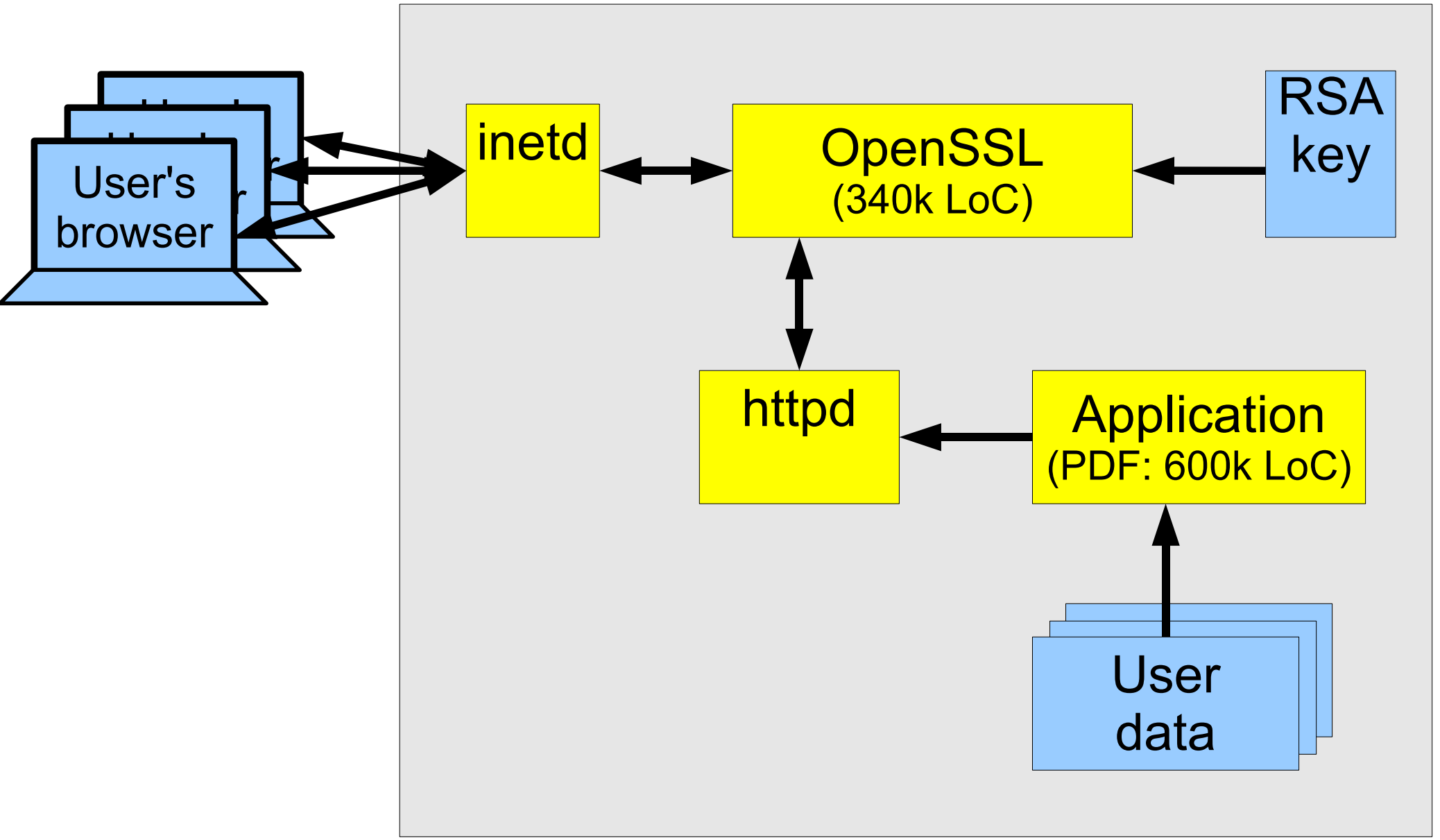
HiStar kernel design summary

- Few mechanisms solve many problems
  - Containers: resource control, FS, label discovery
  - Gates: IPC with immutable labels, avoid superuser
  - Labels: Unix users, wrap, DB security, web server, ...
- 20,000-line kernel provides these mechanisms
  - Everything else built on top

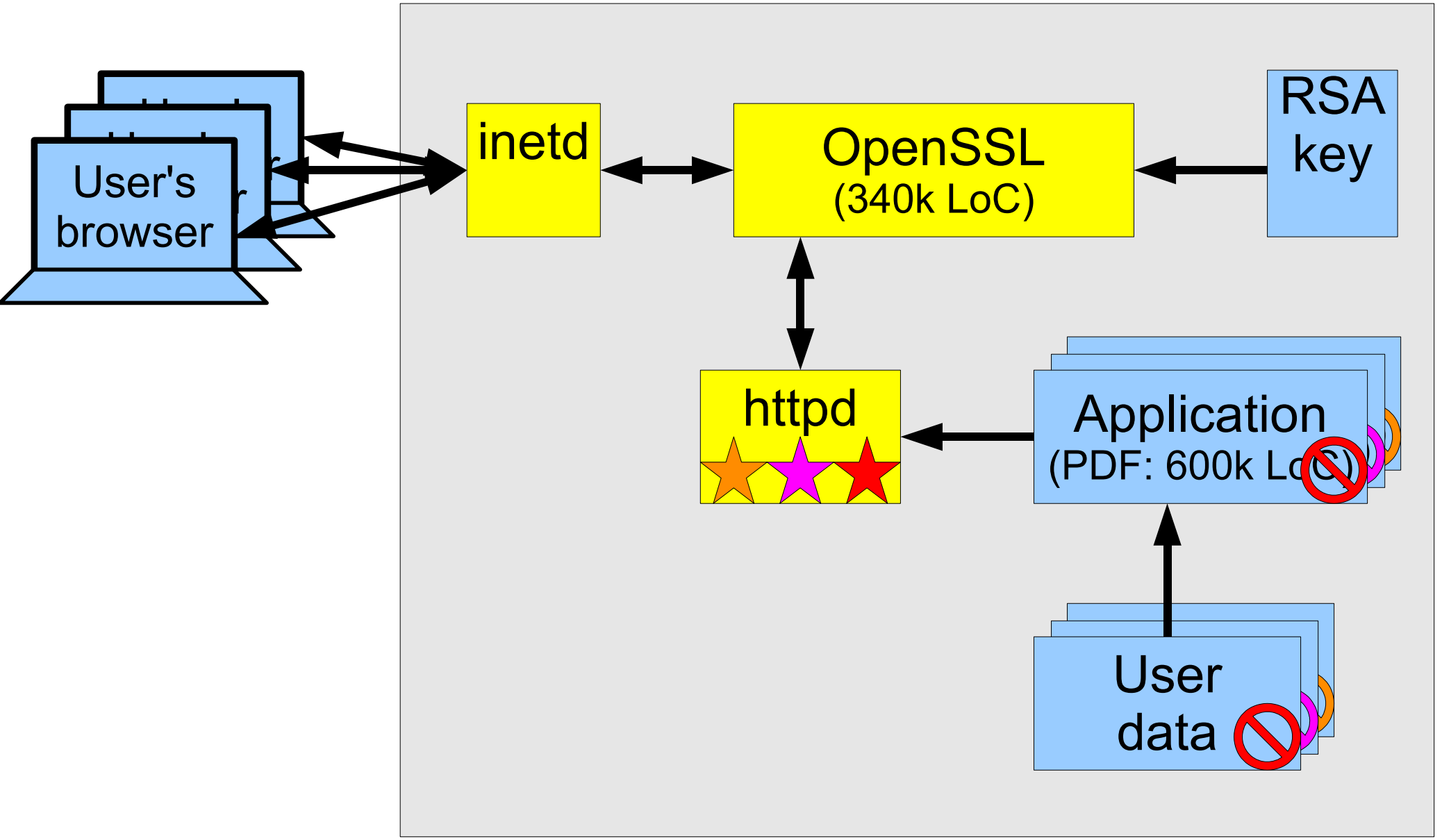
HiStar outline

1. Kernel mechanisms: objects and labels
2. Example uses of these mechanisms
3. How these mechanisms improve security
4. Applications
   - Shown earlier: wrap
   - Web server built out of largely untrusted code
   - Unix login with user-supplied password checking code

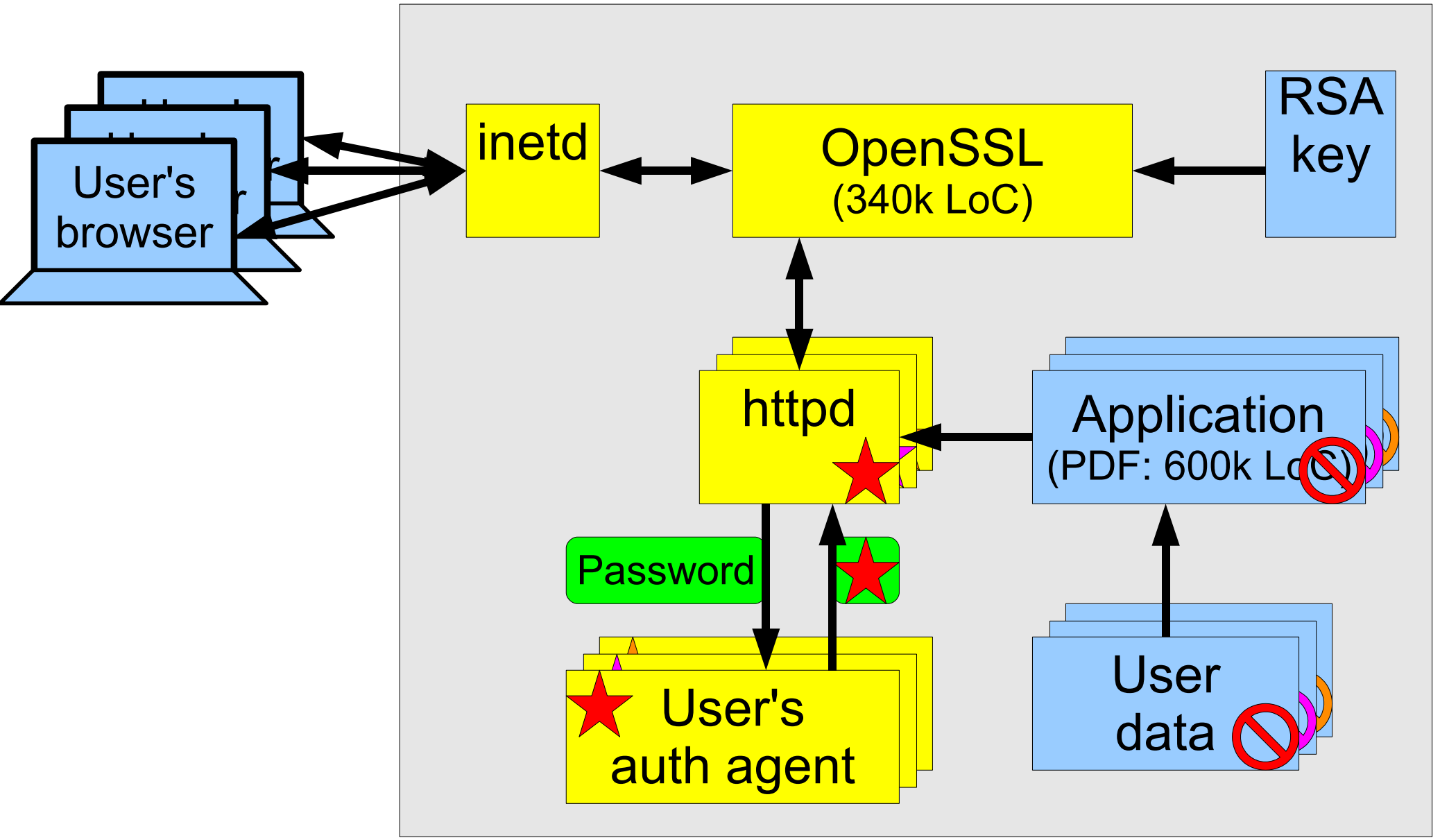
Traditional web server (like Apache): 1M+ lines of trusted code



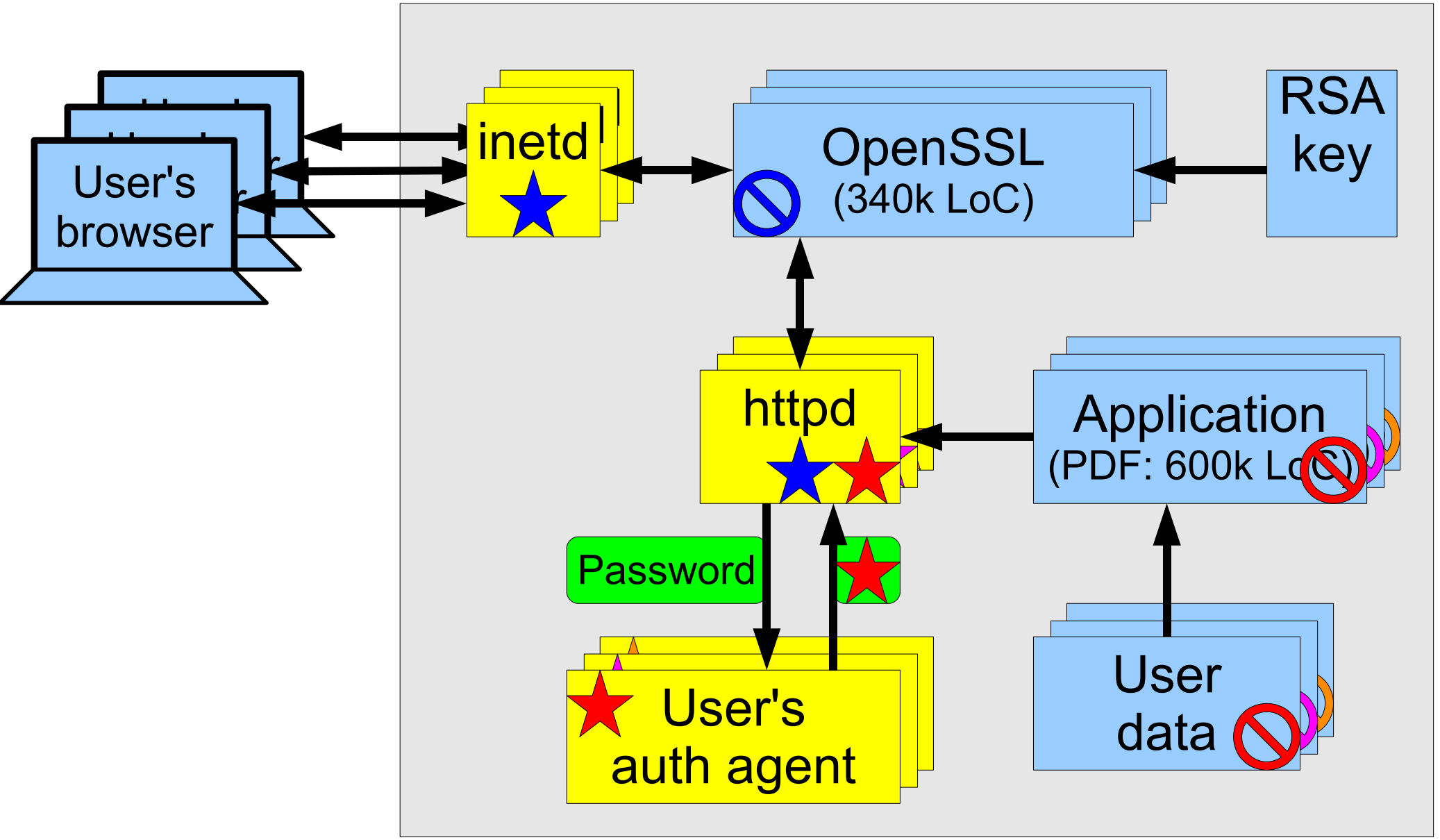
HiStar: Application code cannot disclose user data



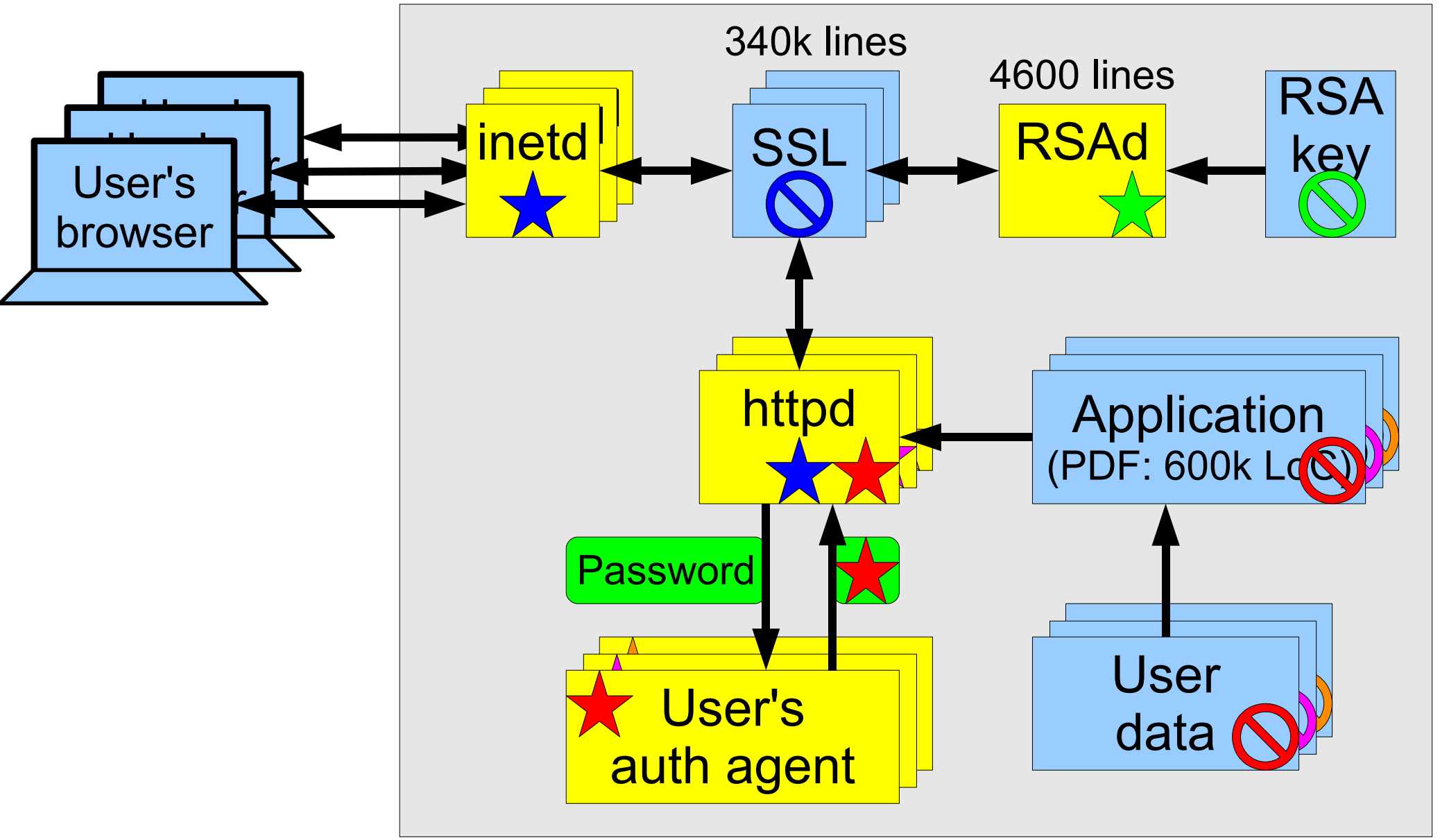
HiStar: Per-user authentication agents, no fully-privileged code



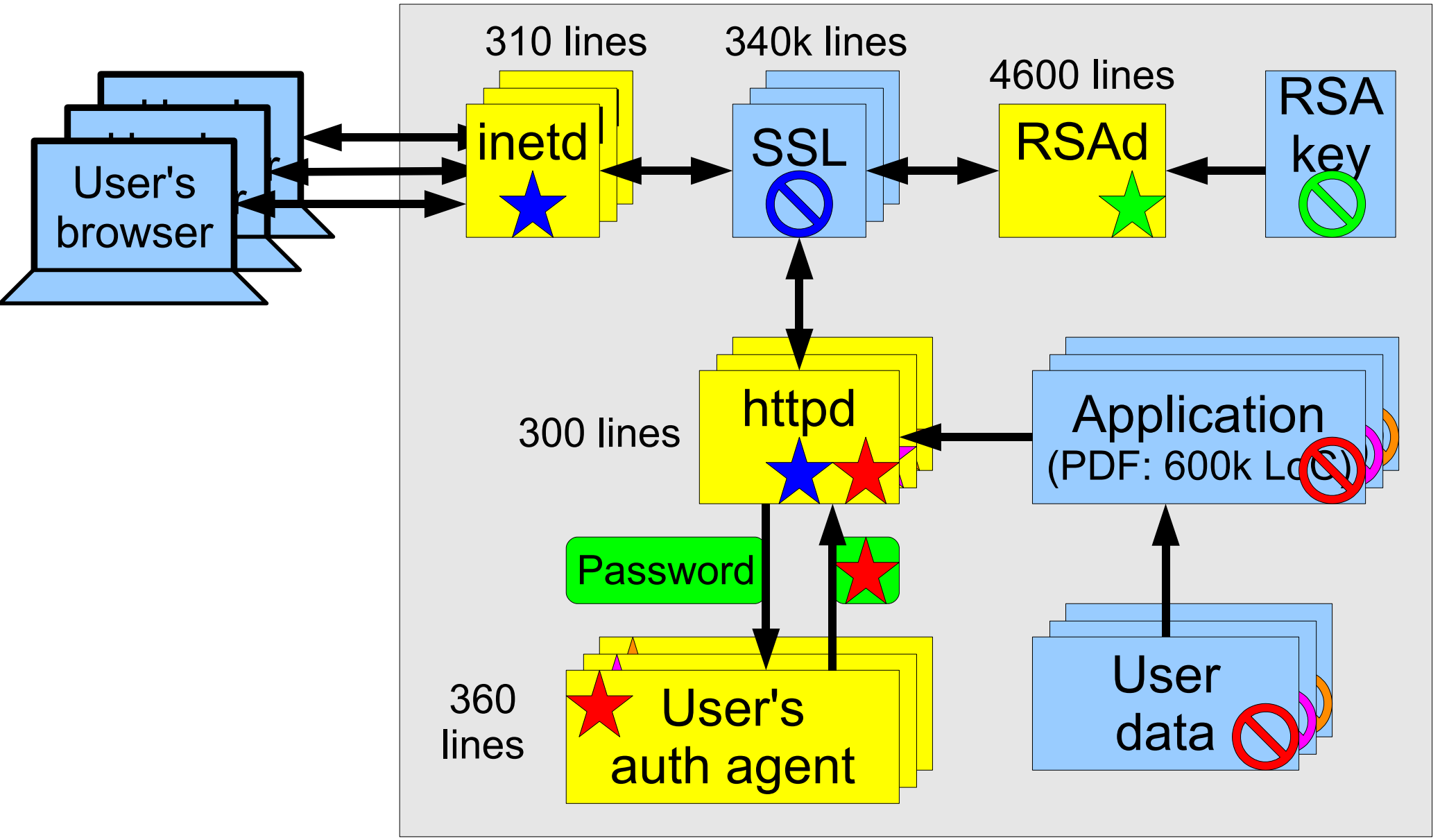
HiStar: SSL library cannot send data to attacker



HiStar: SSL library cannot disclose private key



Security enforced by ~6,000 lines of code (yellow)



Egalitarian labels enable authentication code reuse

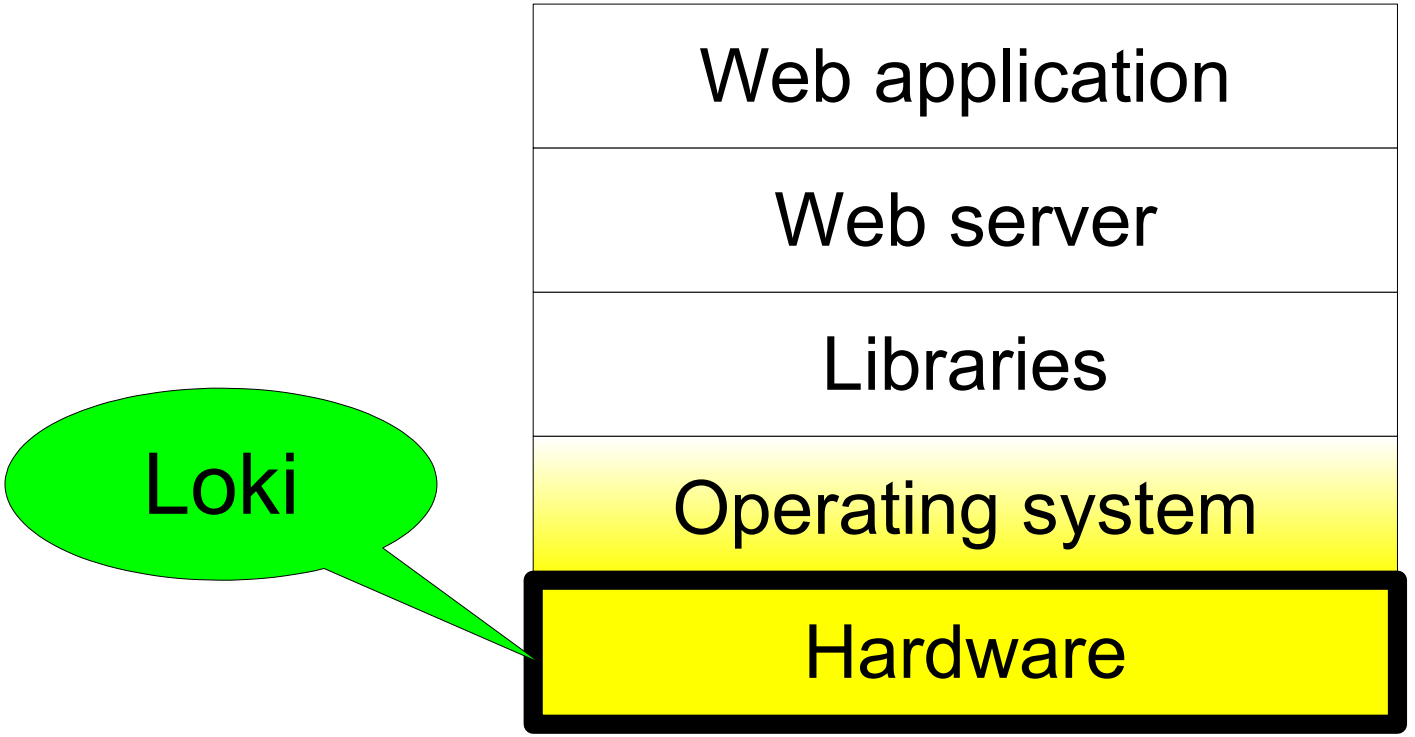
- Same exact code in web server and Unix login
  - Auth. agents don't care what stars they manage

- Egalitarian labels key for new functionality:
  - Each user controls their own authentication agent
    - Can add one-time passwords, challenge-response, ...
  - Login client uses IFC to ensure password secrecy
    - Even if password sent to evil agent (mistyped username)

Summary: IFC mechanism enforces application security

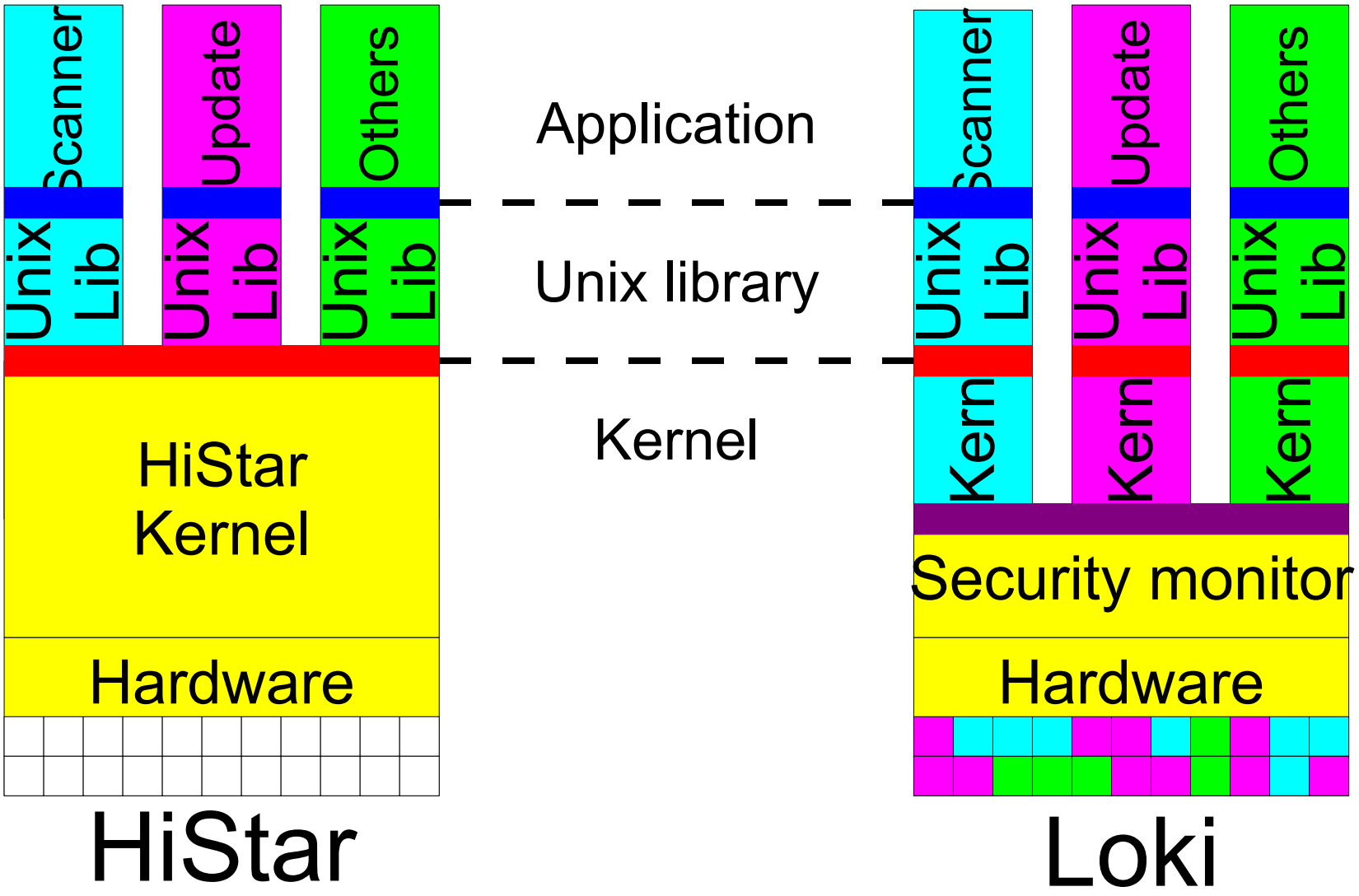
- Small part of application *specifies* security policy
  - 6,000 lines for web server, 140 lines for wrap
- 20,000-line kernel *enforces* security
  - Isolates virus scanner, mail search, ...
  - Password secrecy during login
  - User data privacy in web server
- Rest of application can be buggy!

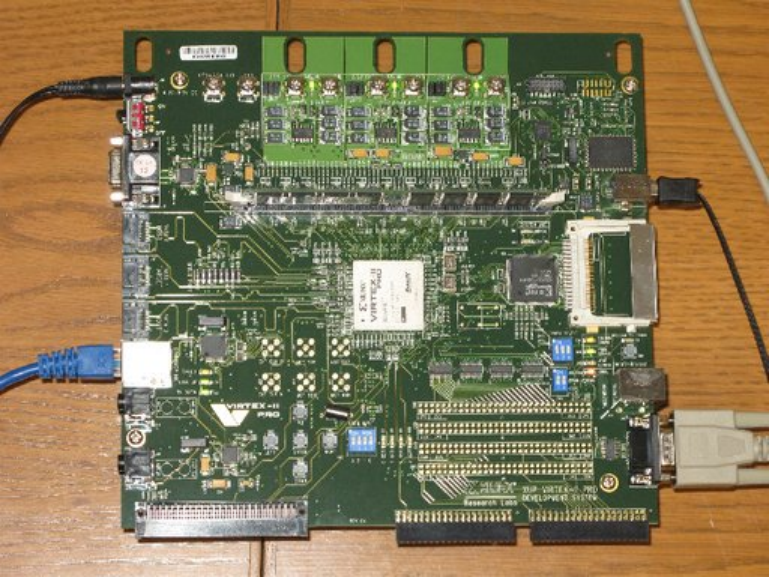
Information flow control in hardware



Loki pushes labels into hardware

Tagged memory, monitor translates labels to tags



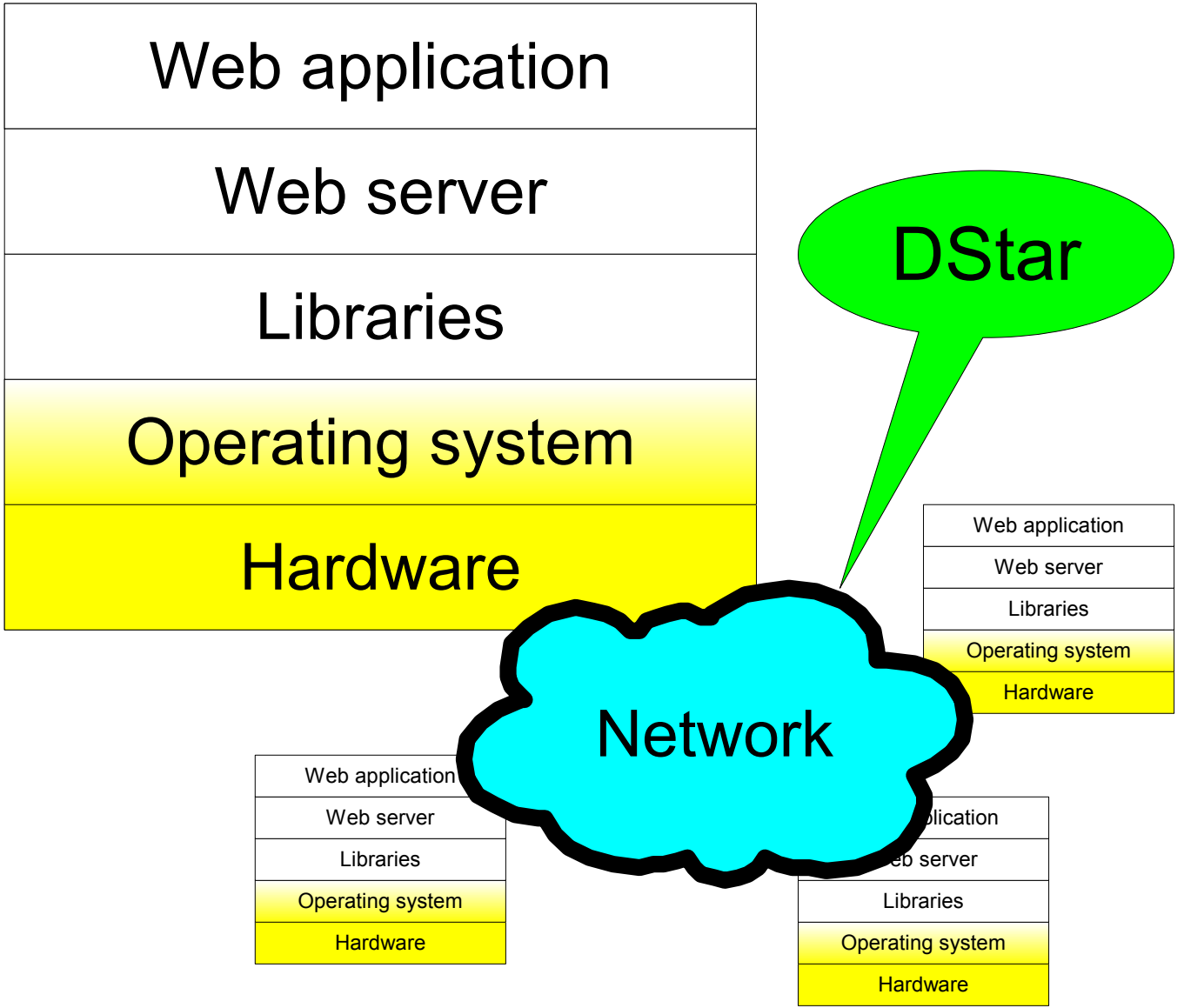


Loki results (modified SPARC processor)

	Unmodified SPARC	Loki
Trusted code lines	11,600 (kernel)	5,200 (monitor)
Clock speed	65 MHz	65 MHz
FPGA LUTs	13,858	15,929 (+15%)
FPGA BRAMs	46	51 (+11%)
Slowdown		~1%

- Runs HiStar, Unix library, web server, ...
- Security enforced in hardware + security monitor

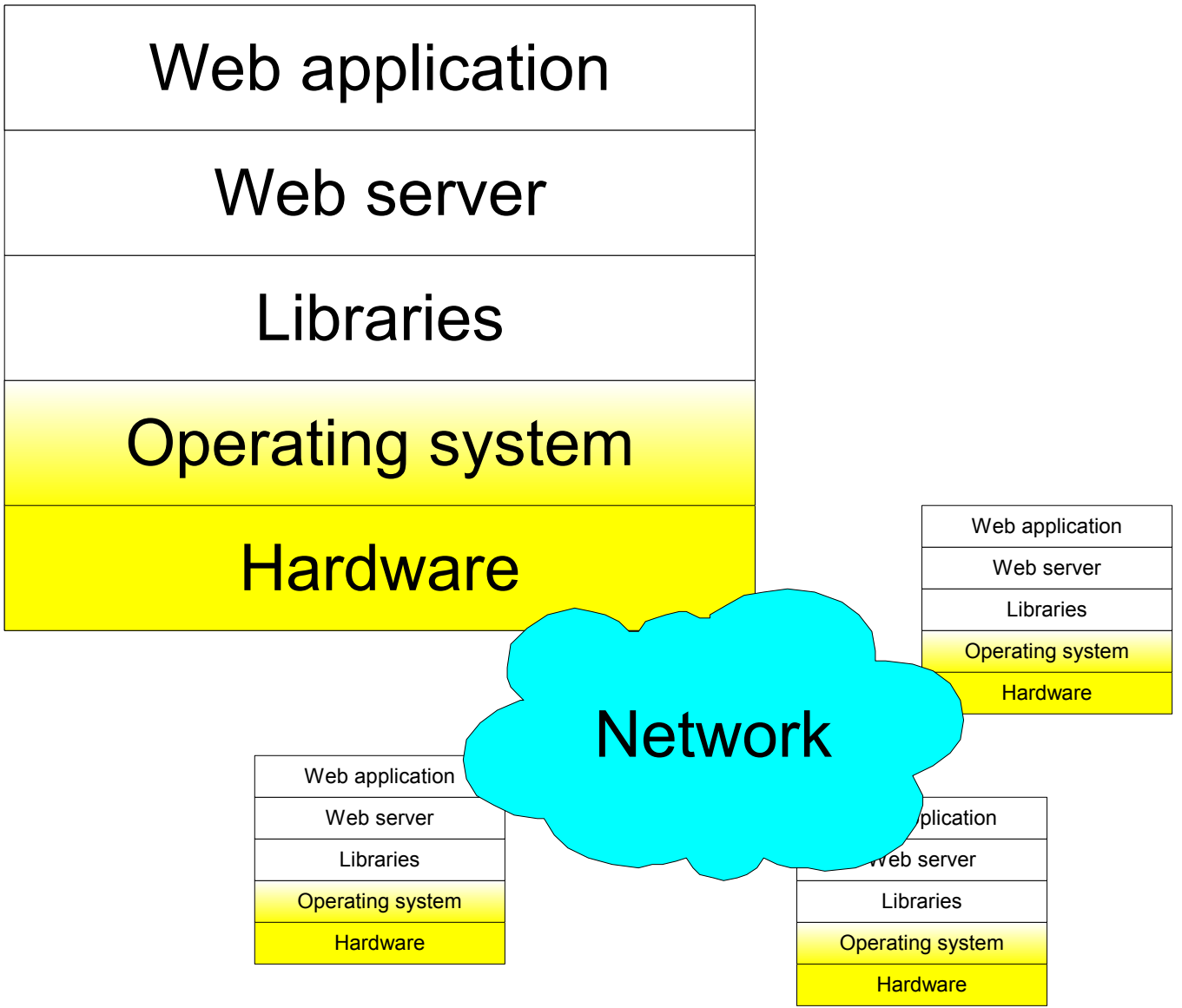
Distributed information flow control



Information flow control scales to distributed systems

- DStar encodes labels in messages
  - Each machine enforces labels using HiStar/Loki
- Requires only 5,000 more lines of trusted code
  - Plus crypto and other support libraries
- HiStar web server scales to many machines!

Summary: IFC allows building secure systems from buggy code



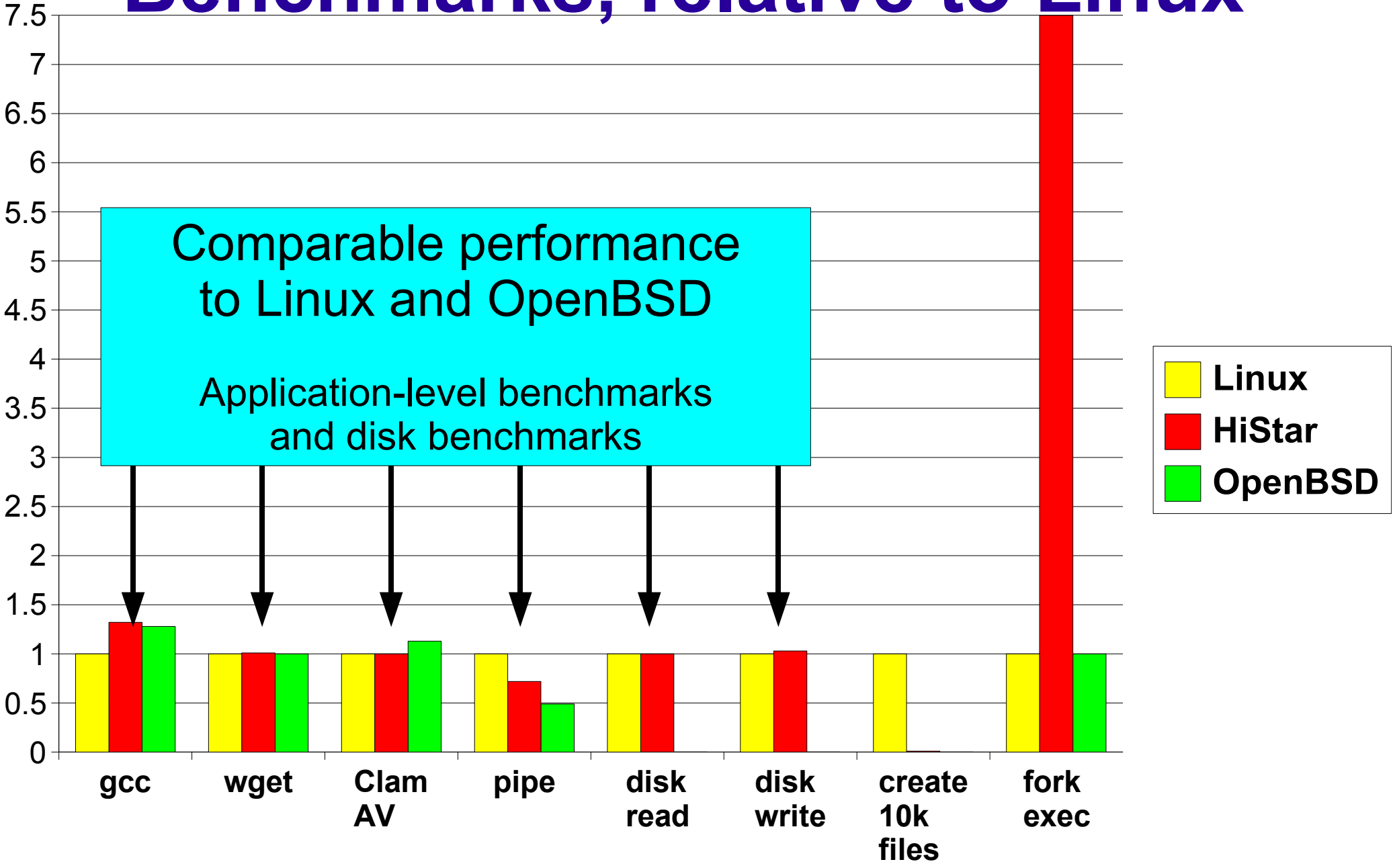
Future directions: mechanisms for security

- Hardware mechanisms for Linux, Windows
- Network mechanisms to improve app security
- Map data protection to cryptographic mechanisms
- More principled web *browser* security mechanisms

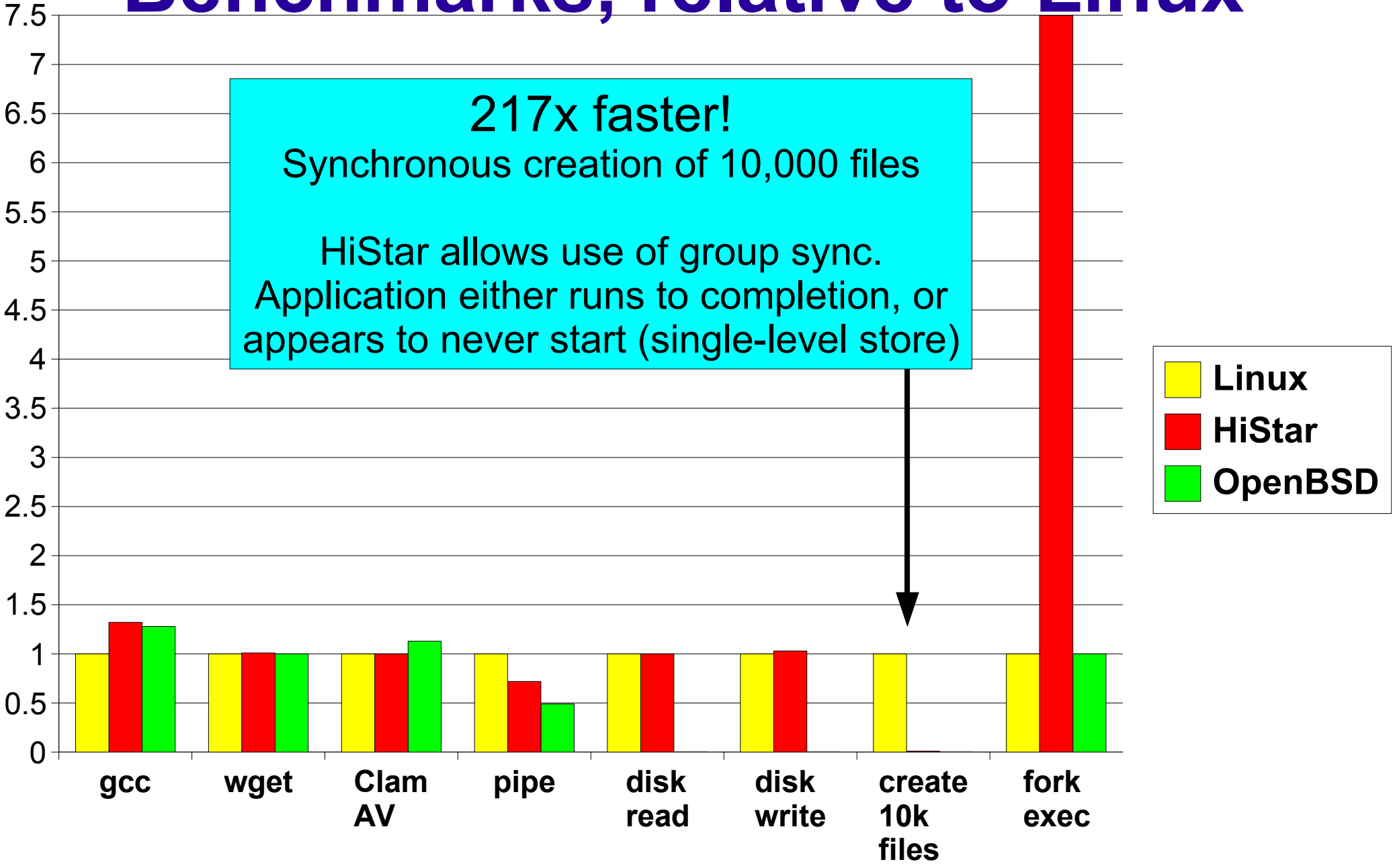
- Long-term goal: provable system security
  - So far: model checking, program analysis

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

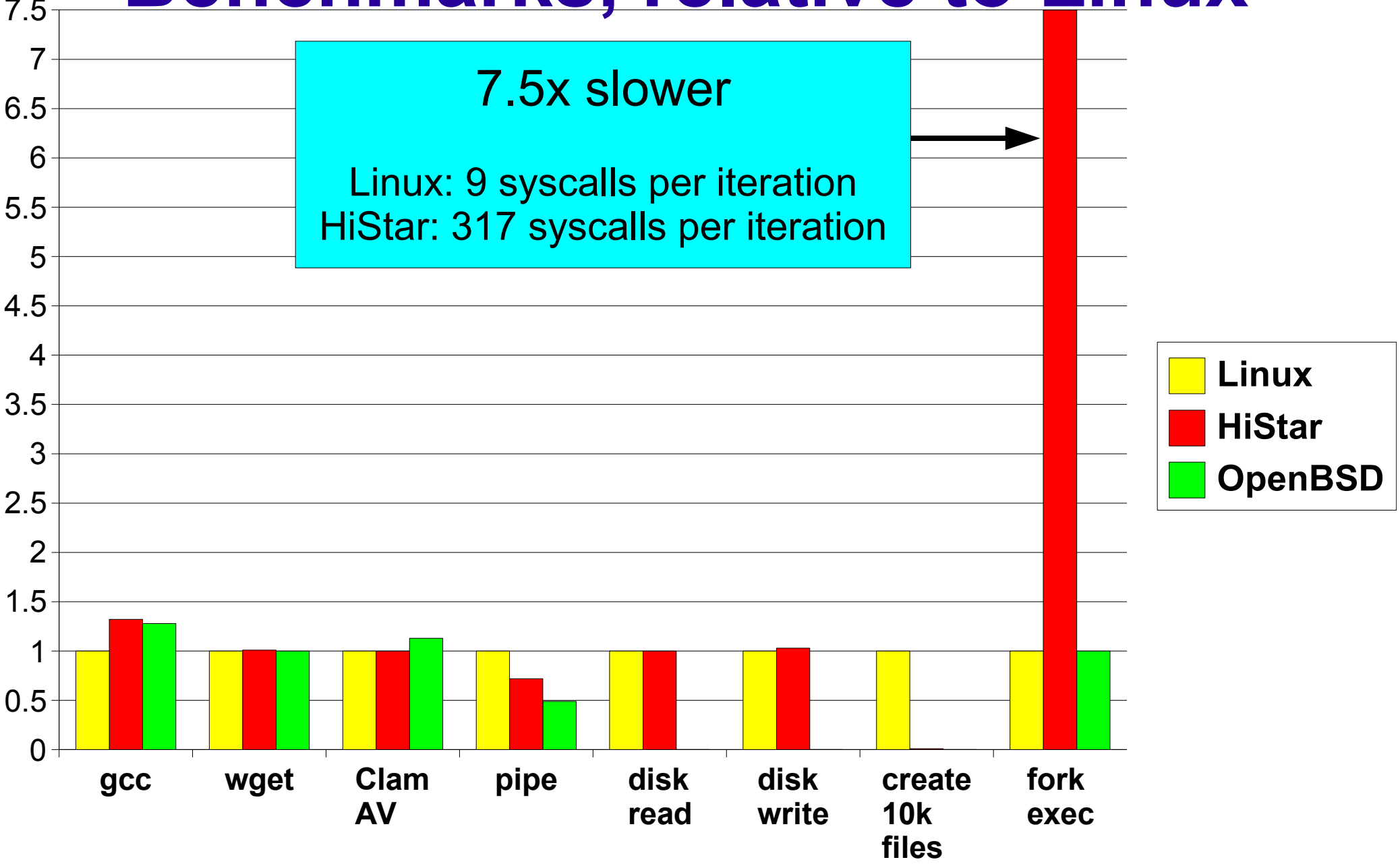
Benchmarks, relative to Linux



Benchmarks, relative to Linux

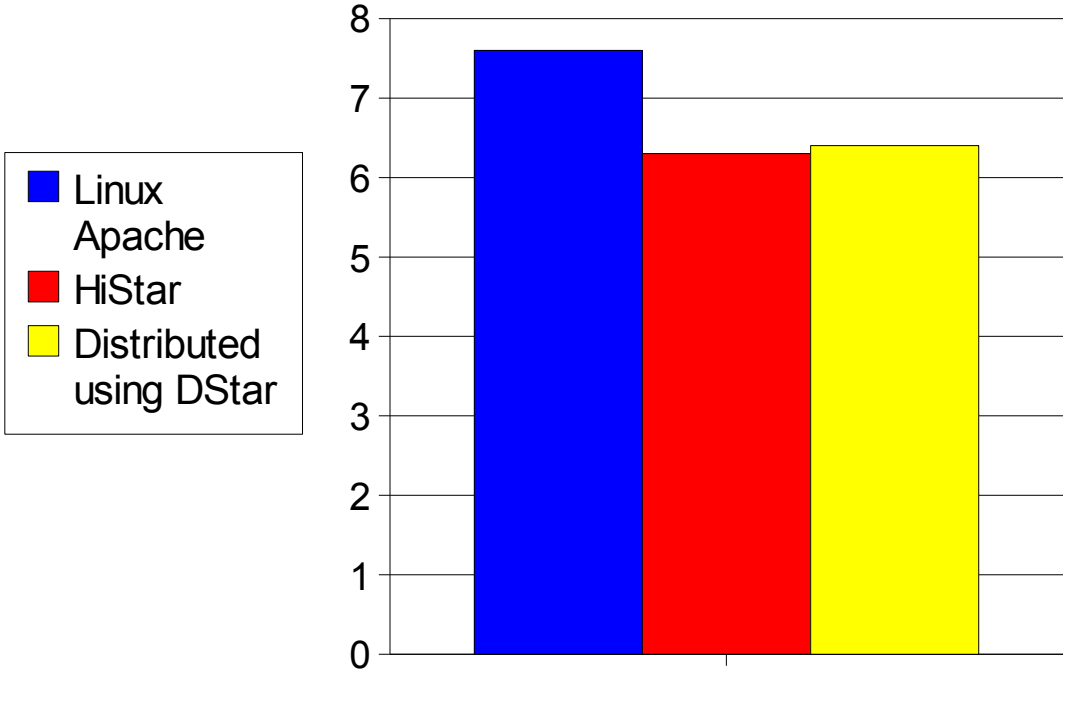


Benchmarks, relative to Linux

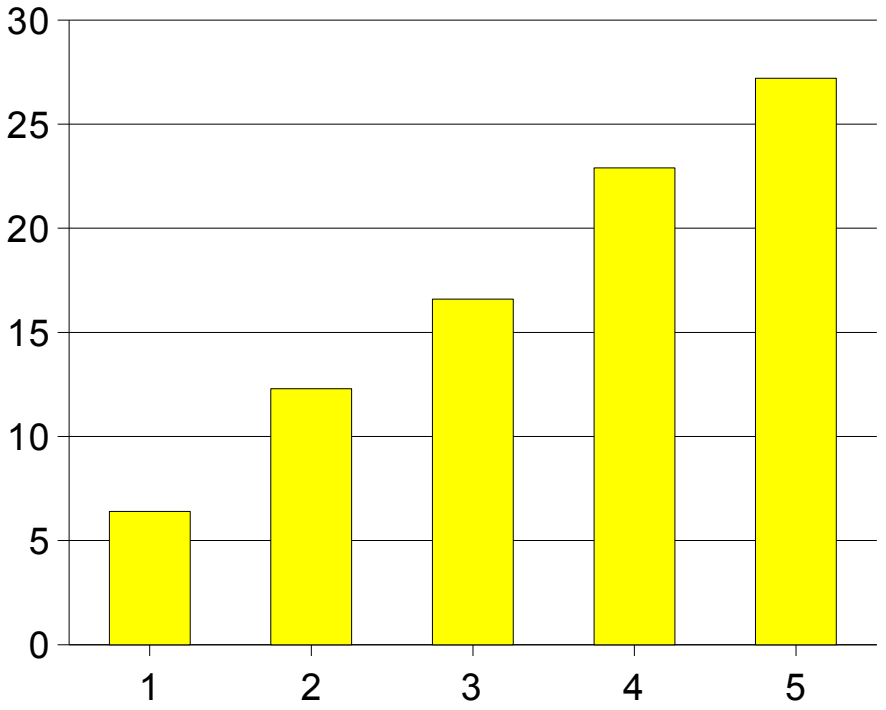


Web server: “PDF maker” app

Baseline throughput, req / second
(1 server, or 1 app server for distributed)



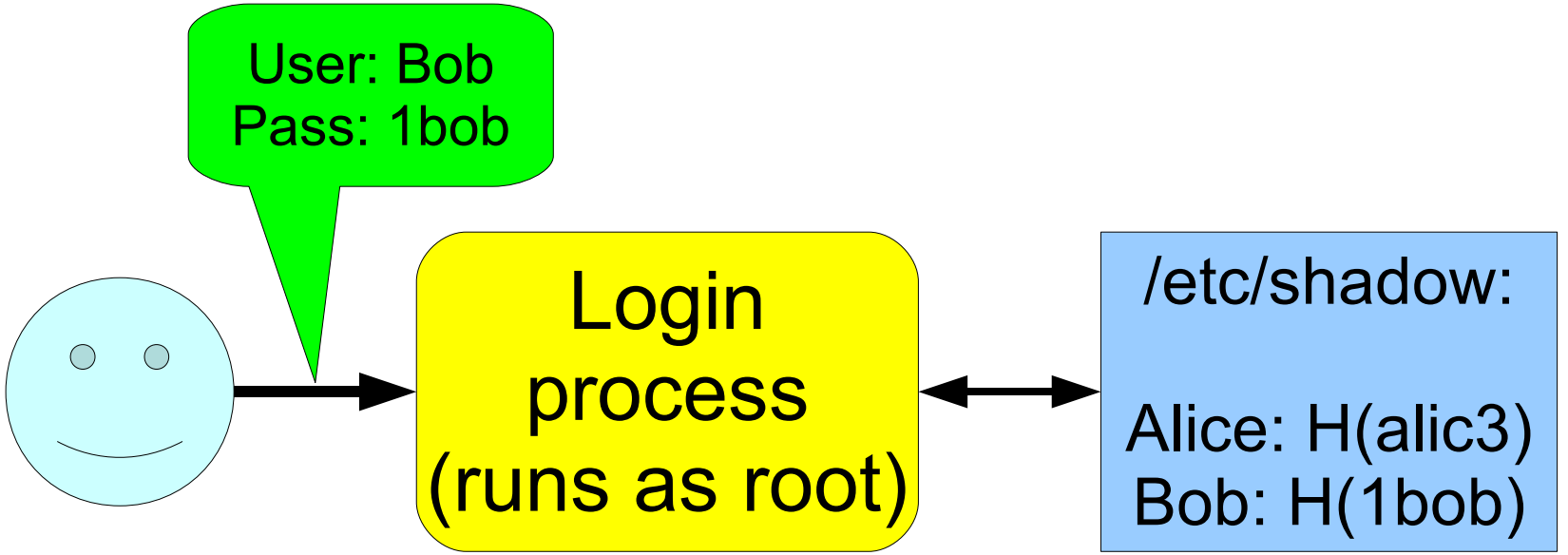
Scalability of application servers
(Fixed number of other servers)



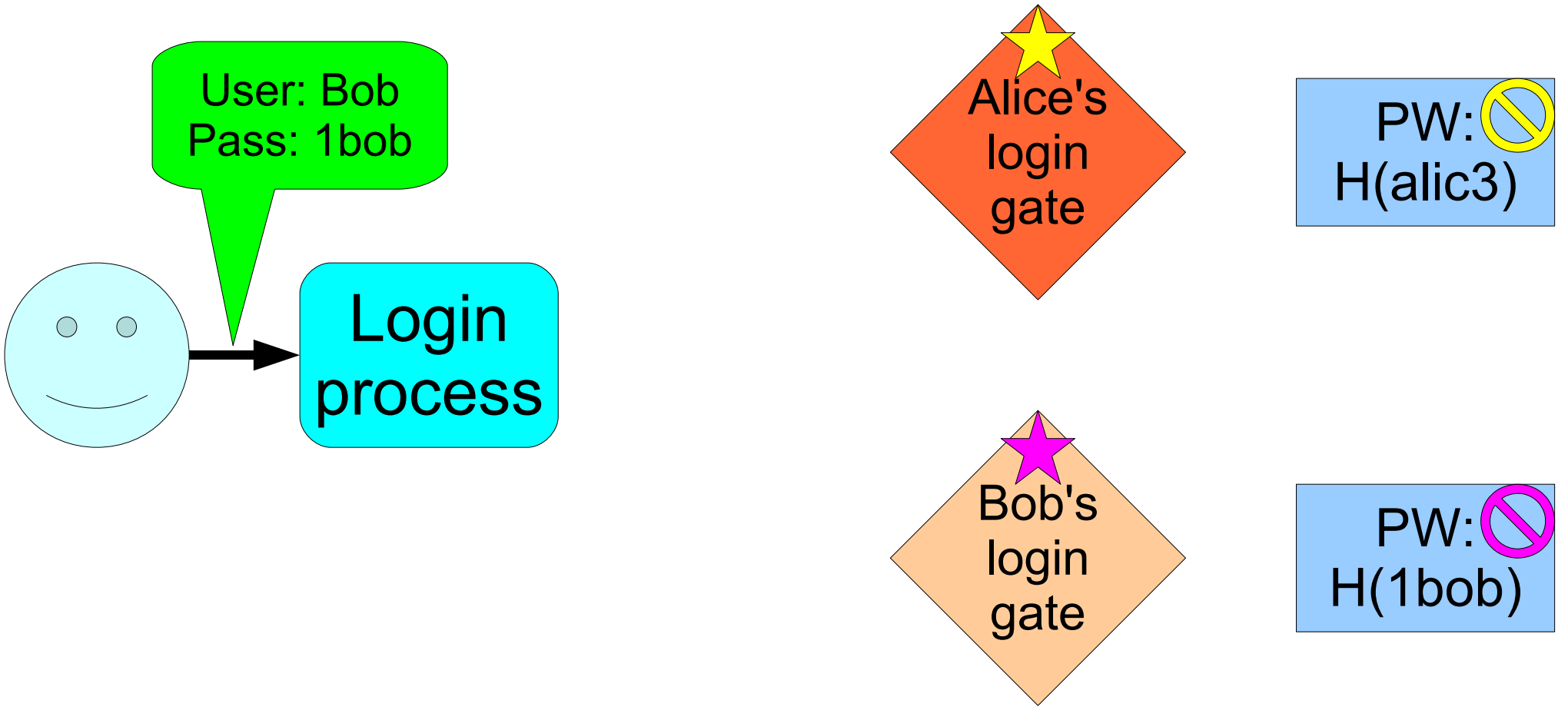
- Distributed across 3 types of servers: *front-end*, *application*, and *data* servers

Login on Unix: highly centralized

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

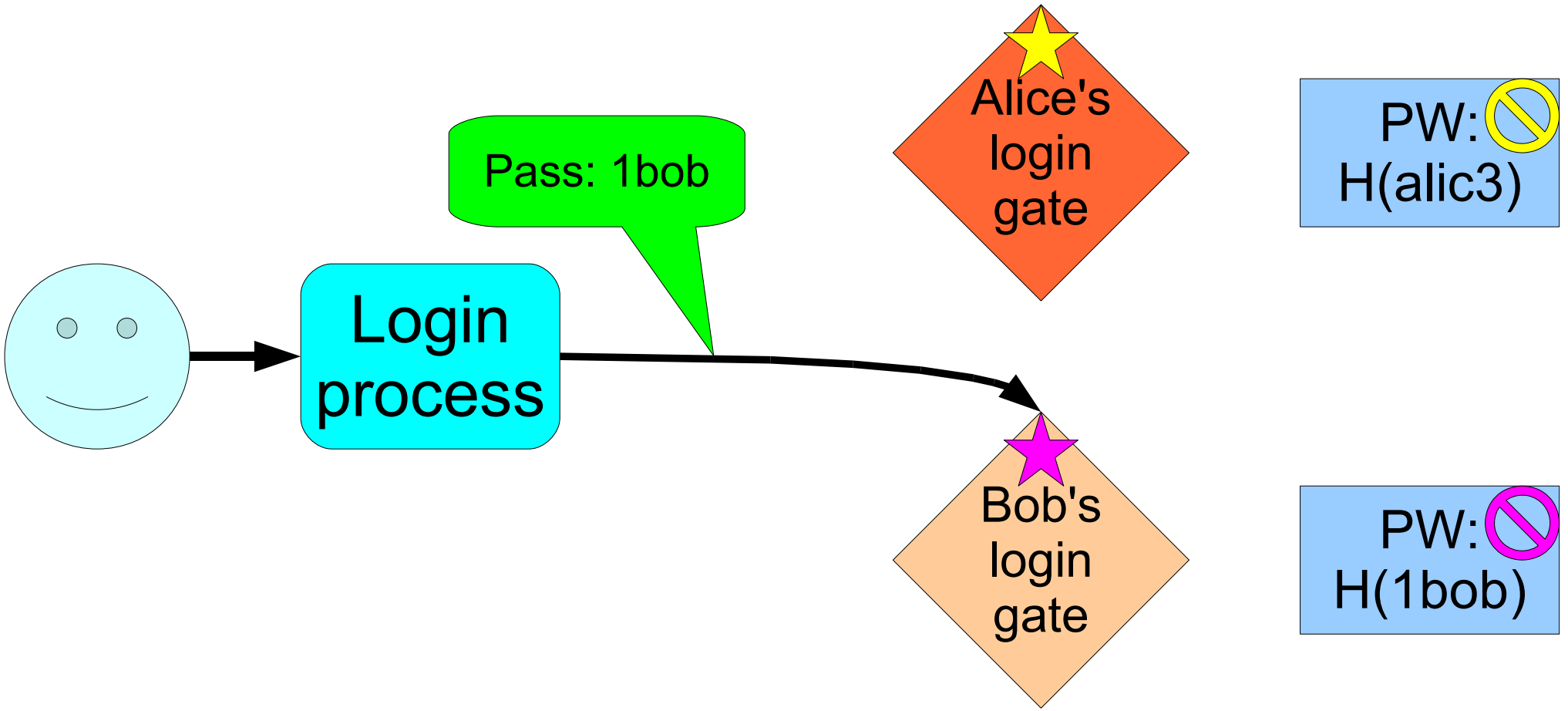


Login on HiStar: less trusted code



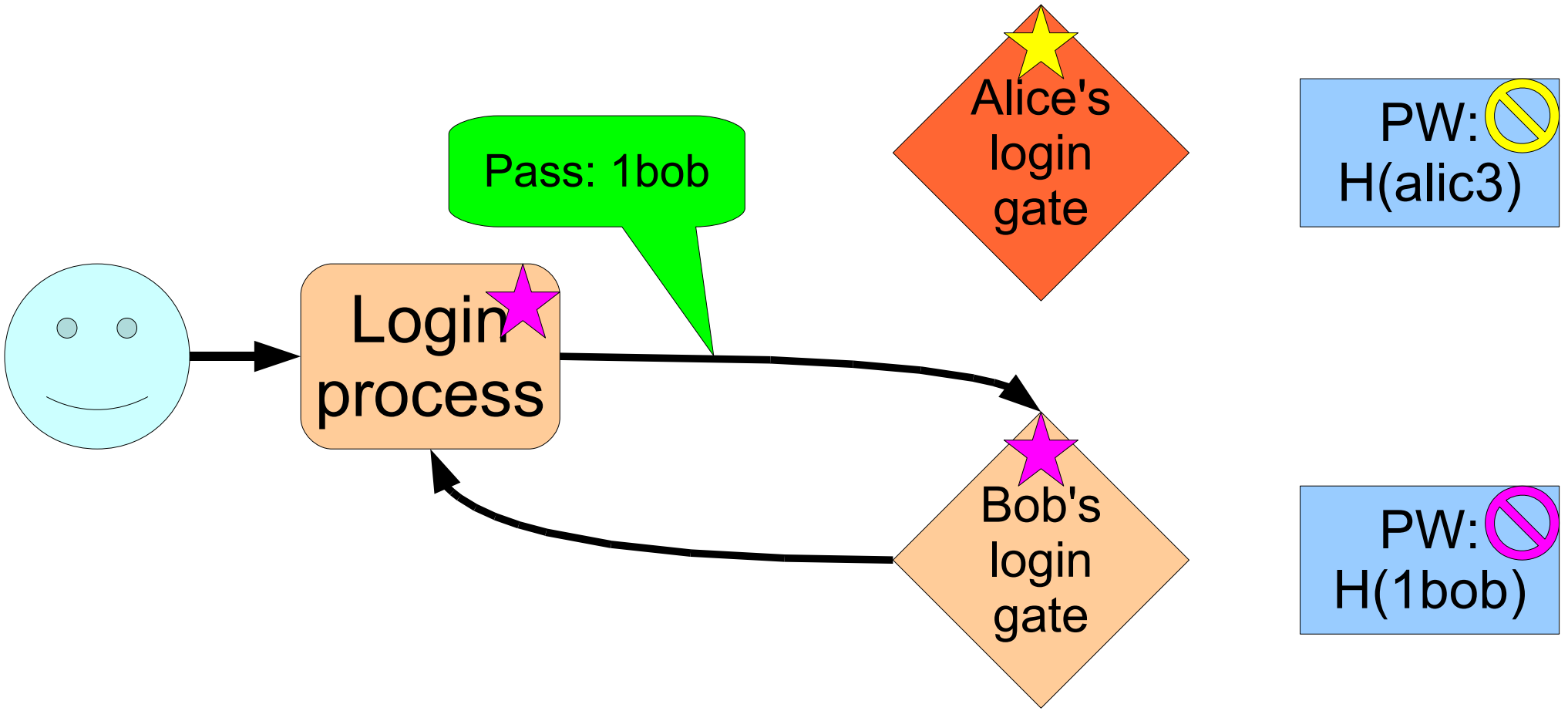
- No application runs with every user's privilege
- Users can supply their own login gate code

Login on HiStar: less trusted code



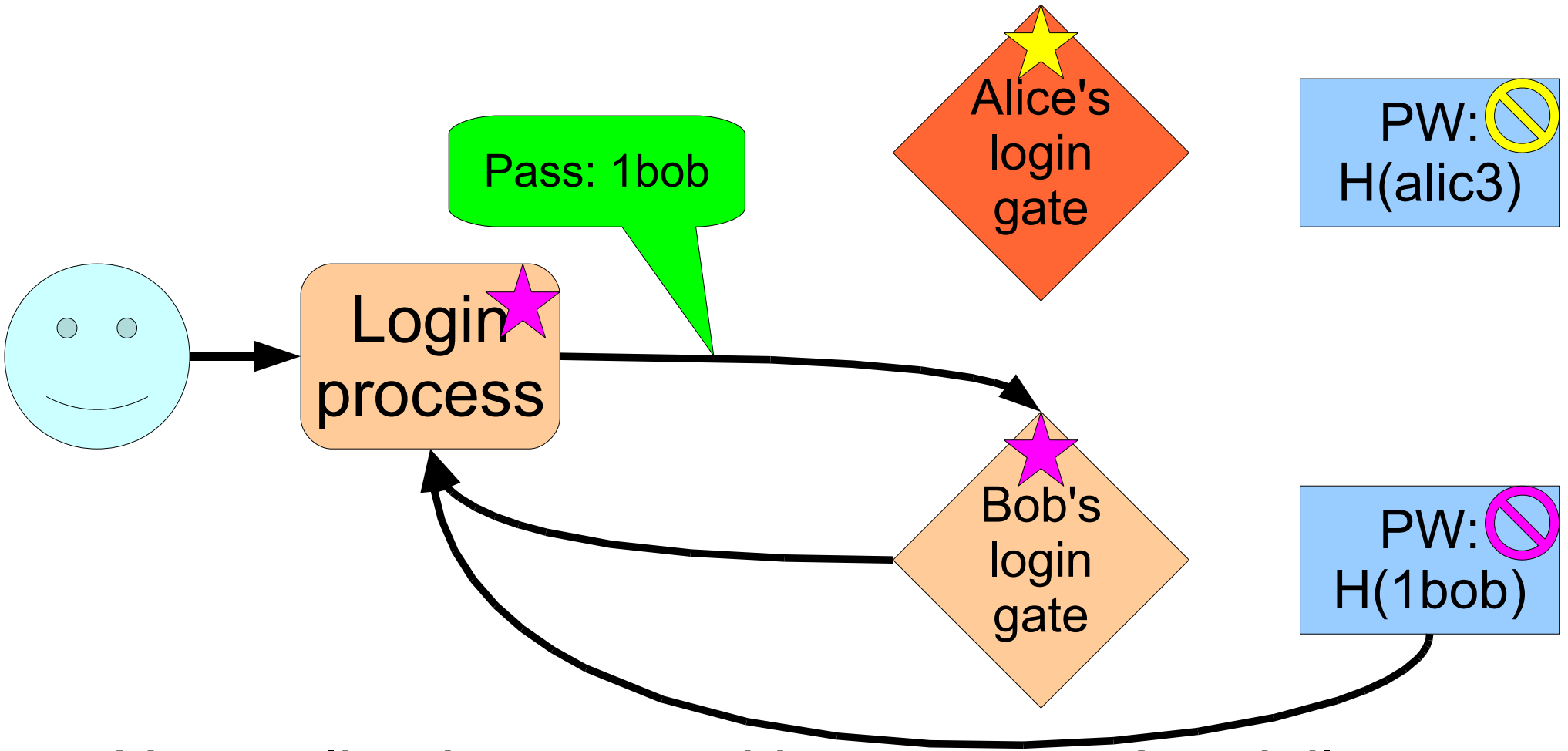
- No application runs with every user's privilege
- Users can supply their own login gate code

Login on HiStar: less trusted code



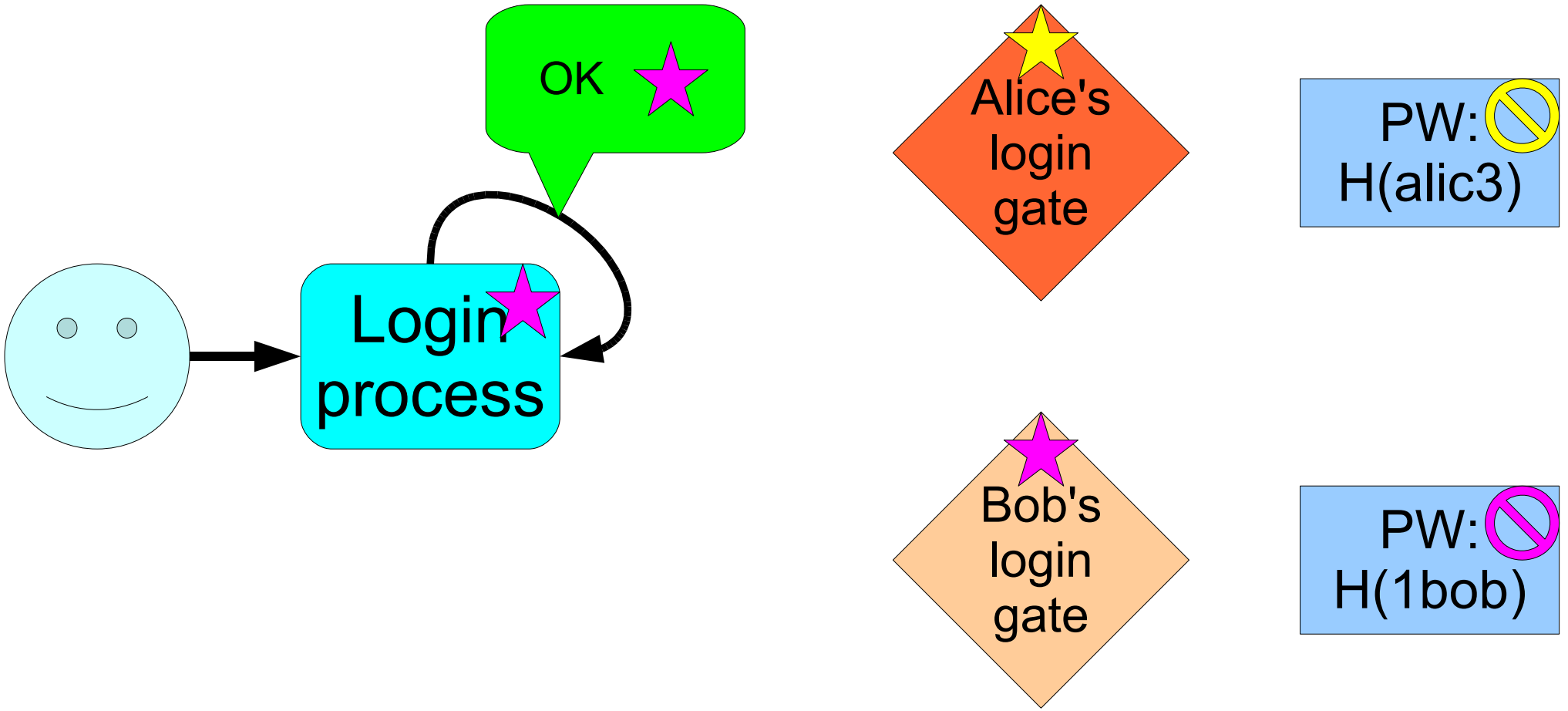
- No application runs with every user's privilege
- Users can supply their own login gate code

Login on HiStar: less trusted code



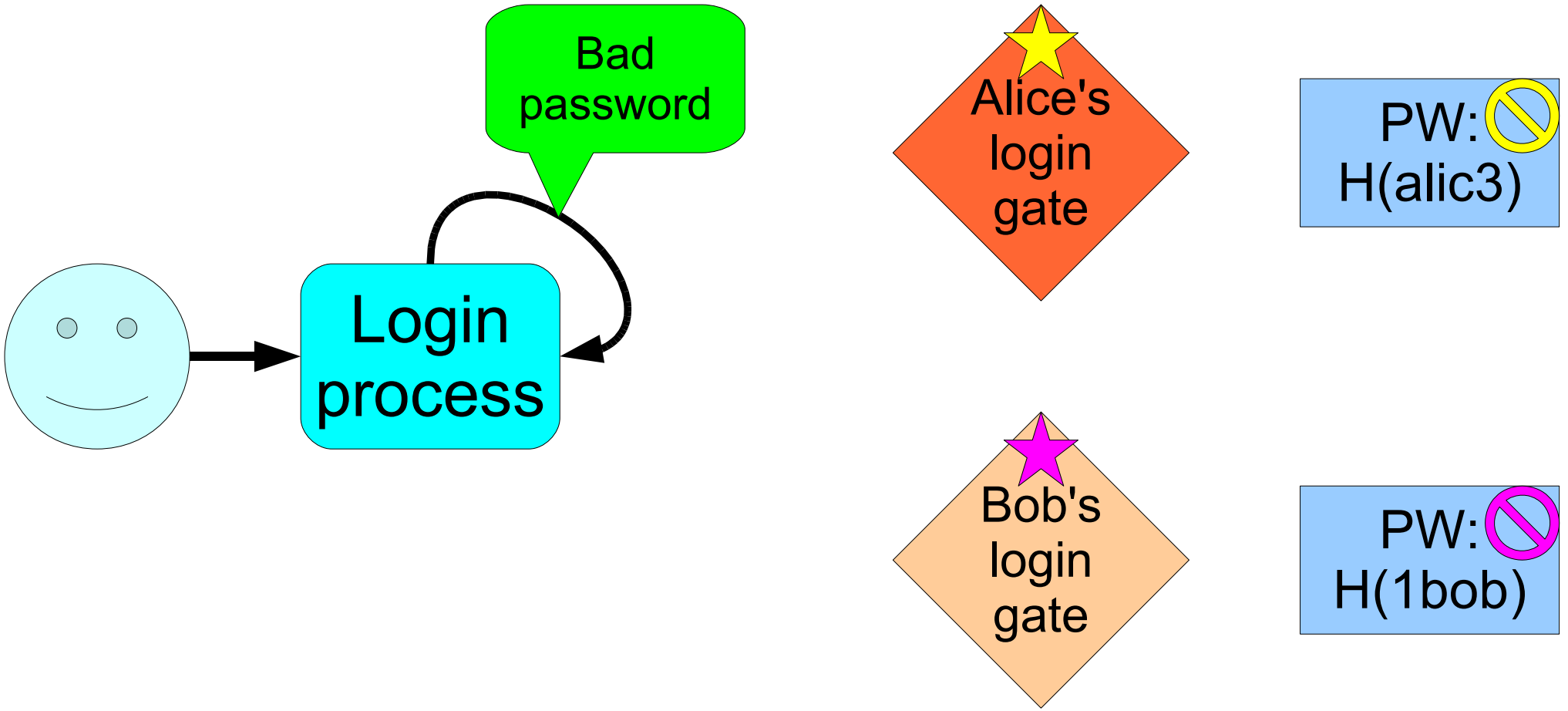
- No application runs with every user's privilege
- Users can supply their own login gate code

Login on HiStar: less trusted code



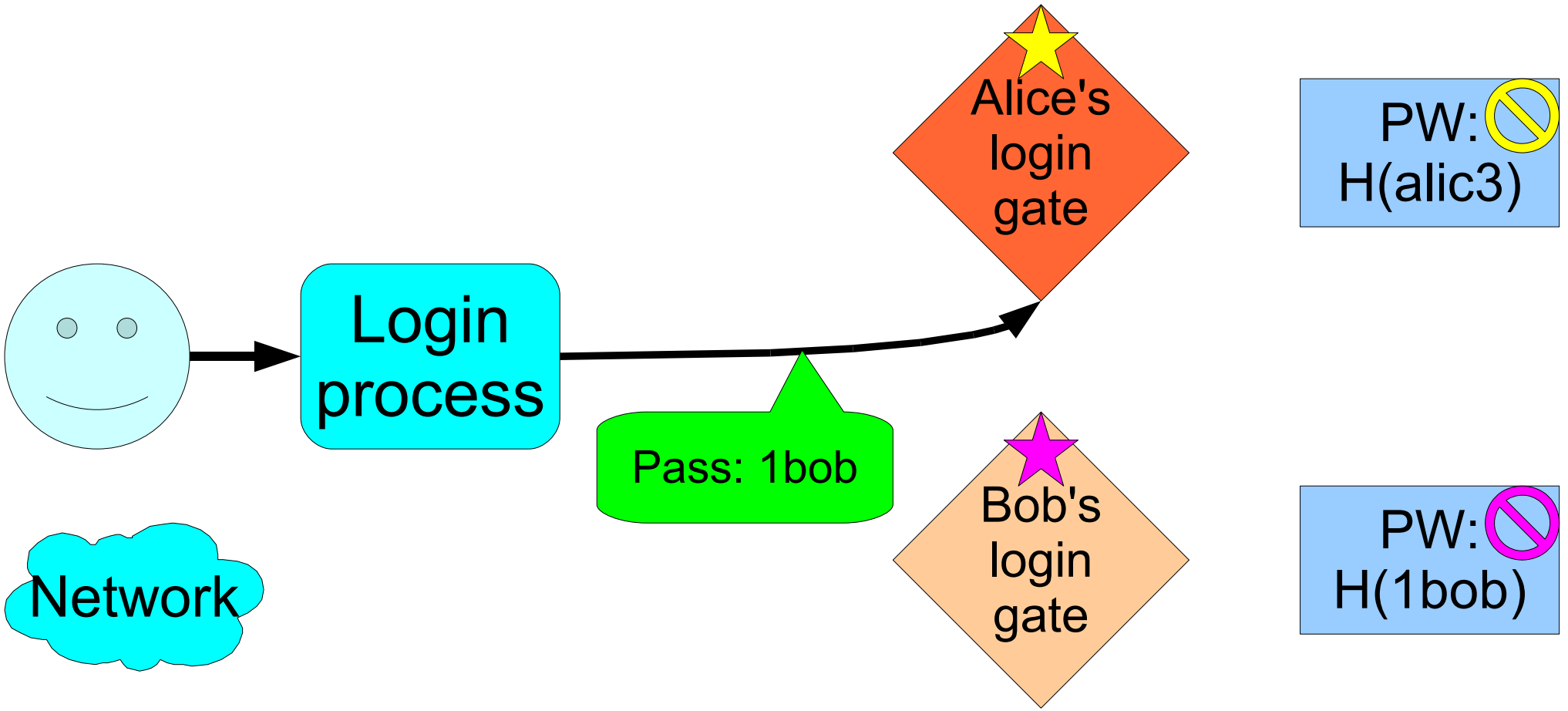
- No application runs with every user's privilege
- Users can supply their own login gate code

Login on HiStar: less trusted code

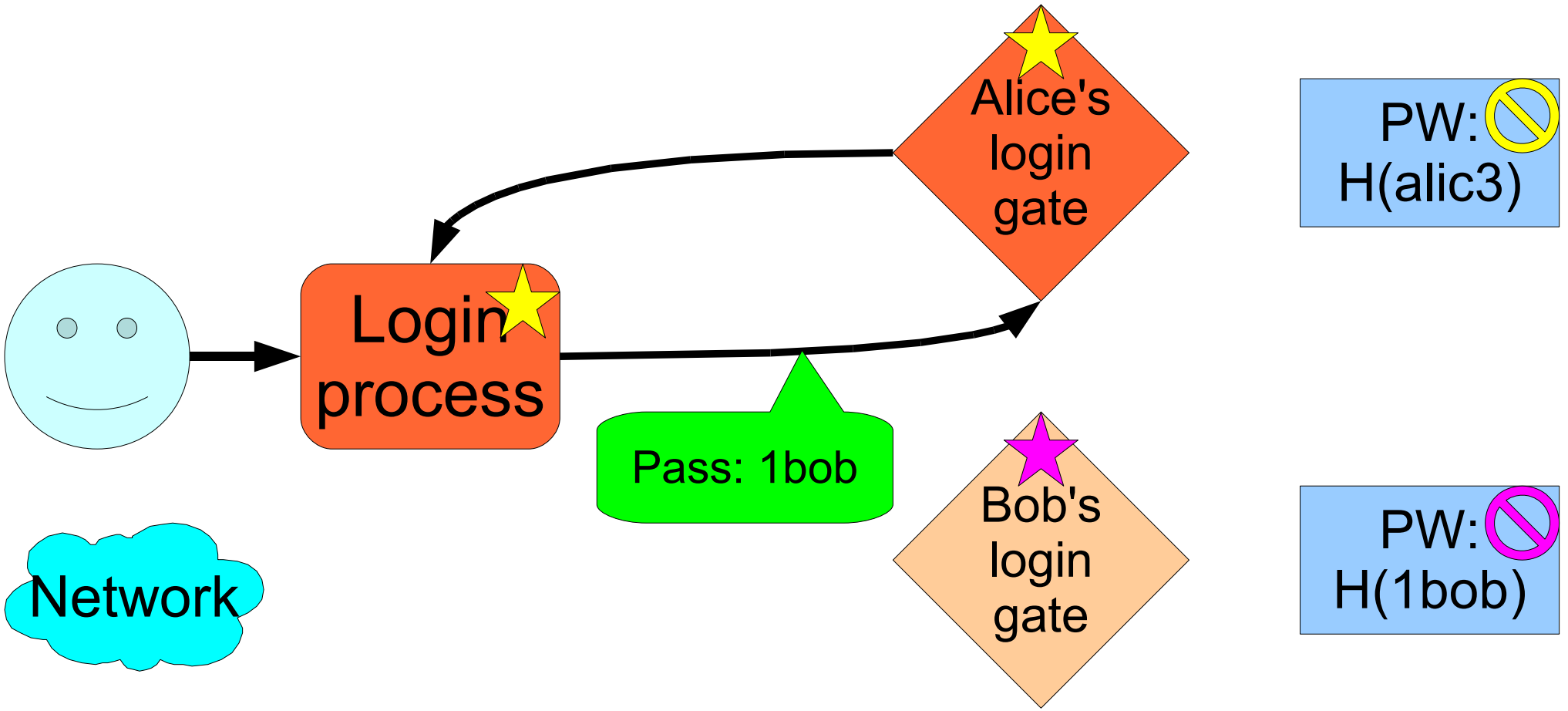


- No application runs with every user's privilege
- Users can supply their own login gate code

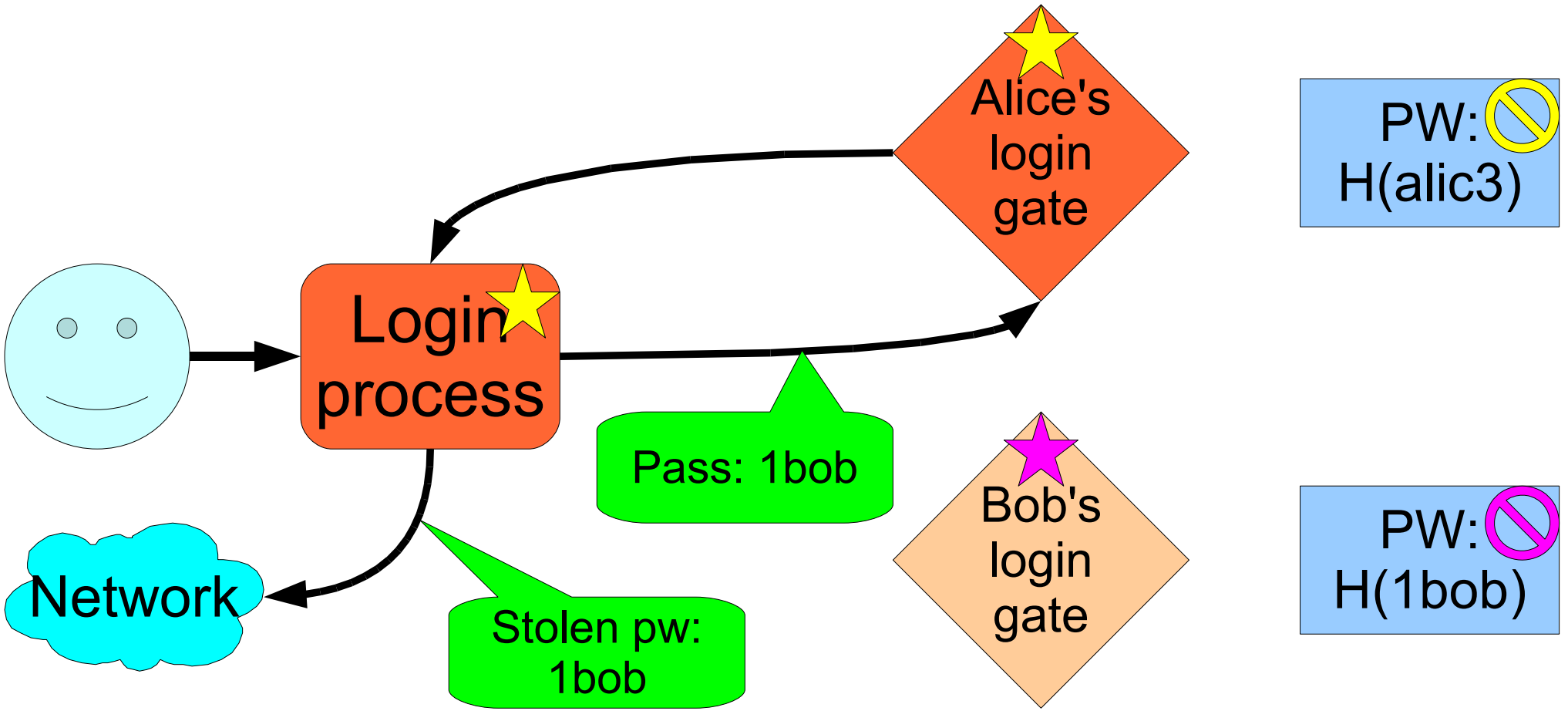
Problem: mistyped username
Malicious auth code steals password



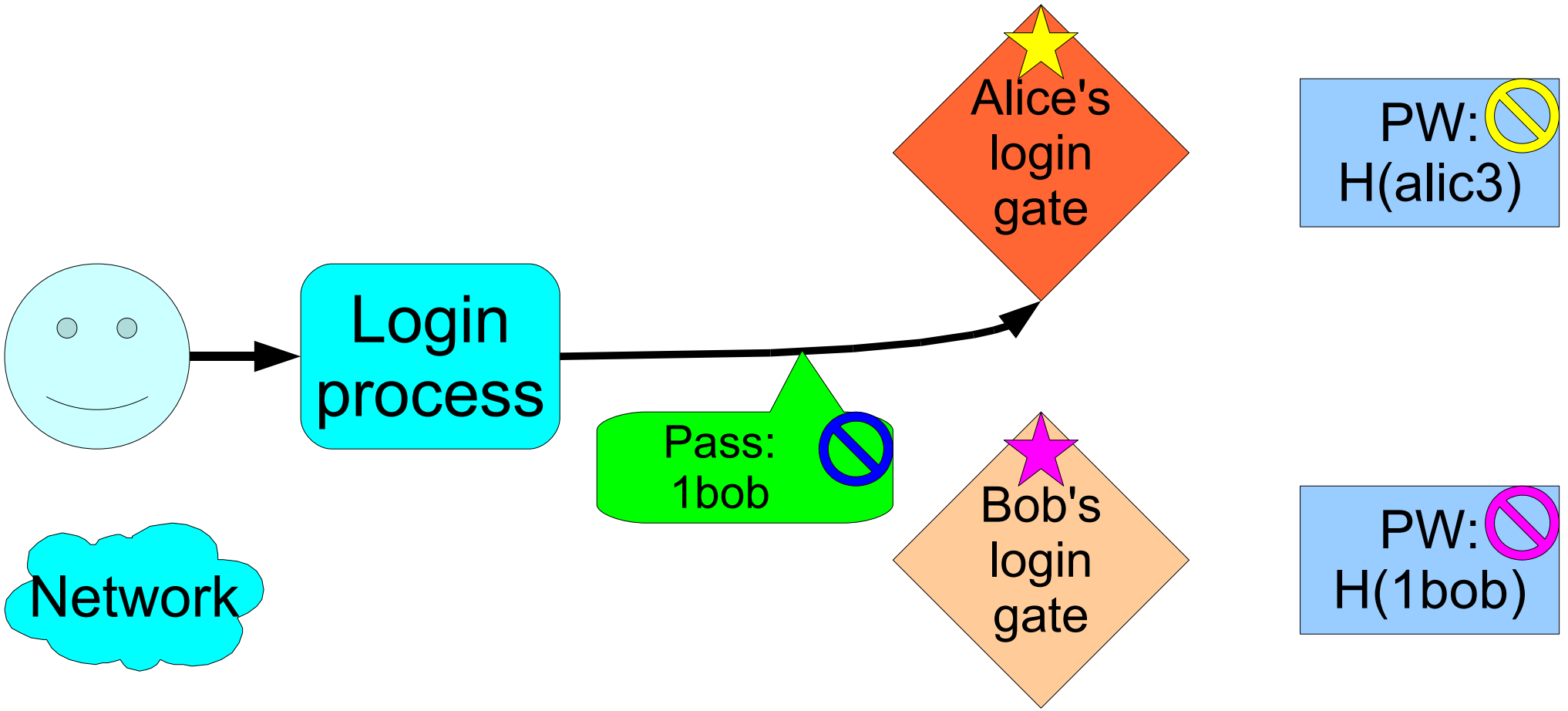
Problem: mistyped username
Malicious auth code steals password



Problem: mistyped username
Malicious auth code steals password



Information flow mechanism prevents password disclosure



Information flow mechanism prevents password disclosure