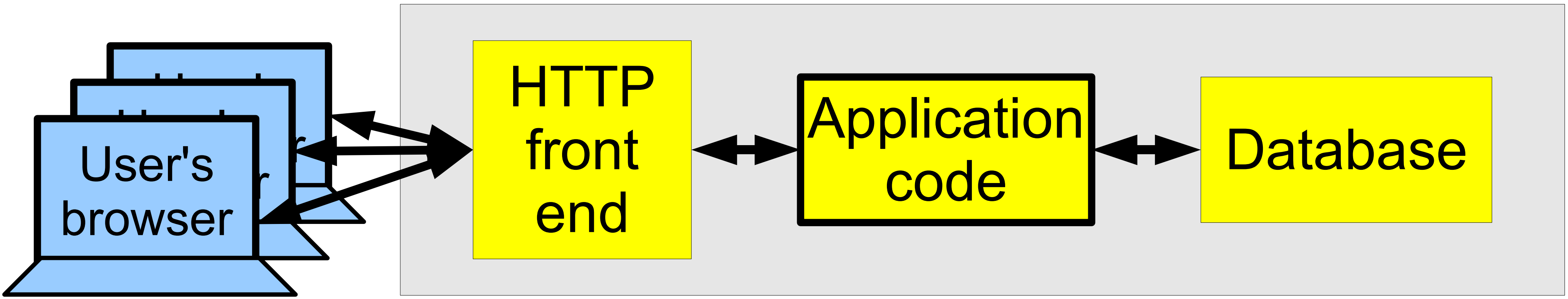
Securing distributed systems with information flow control

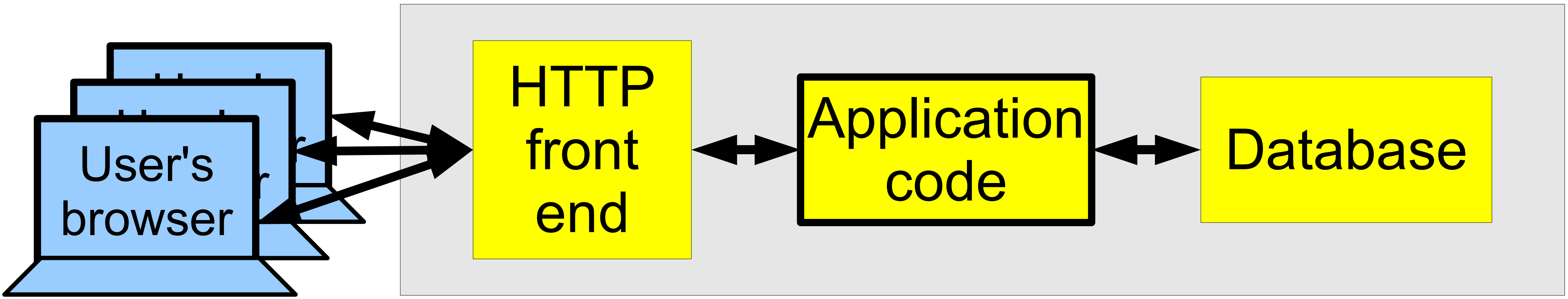
Nickolai Zeldovich
Silas Boyd-Wickizer
David Mazières

Traditional web applications: lots of trusted (yellow) code



- Application is typically millions of lines of code
- Lots of third-party libraries from SourceForge
- Application has access to entire user database

Traditional web applications: lots of trusted (yellow) code



- Application is typically millions of lines of code
- Lots of third-party libraries from SourceForge
- Application has access to entire user database
- Result: any bug allows attacker to steal all data!
  - PayMaxx app code exposed 100,000 users' SSNs

Recent work: information flow control

- Don't try to eliminate all application bugs (**hard**!)
- OS'es like Asbestos, HiStar, Flume keep user data secure even if application is malicious
  - Track flow of user's data through system
  - Only send user's data to that user's browser
  - No need to audit/understand application code!

Recent work: information flow control

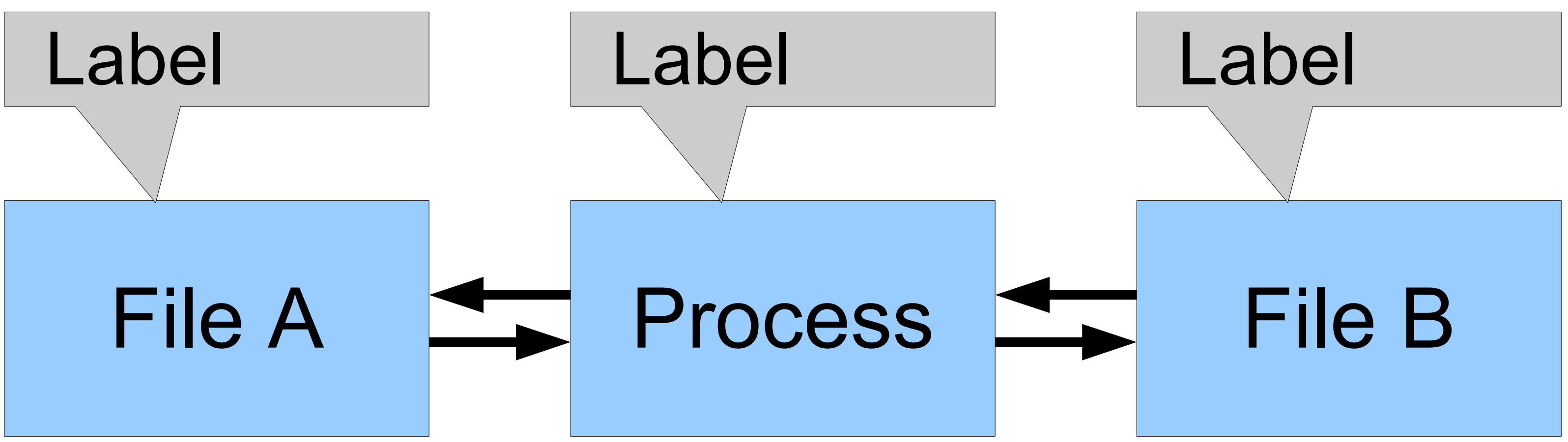
- Don't try to eliminate all application bugs (**hard**!)
- OS'es like Asbestos, HiStar, Flume keep user data secure even if application is malicious
  - Track flow of user's data through system
  - Only send user's data to that user's browser
  - No need to audit/understand application code!
- Limitation: works only on one machine
  - Web applications need multiple machines for scale

This talk: extending information flow control to distributed systems

- Outline:
  - Review of information flow control (IFC) in an OS
  - Challenges in distributed IFC and our solution
  - Apps: web server, incremental deployment, ...

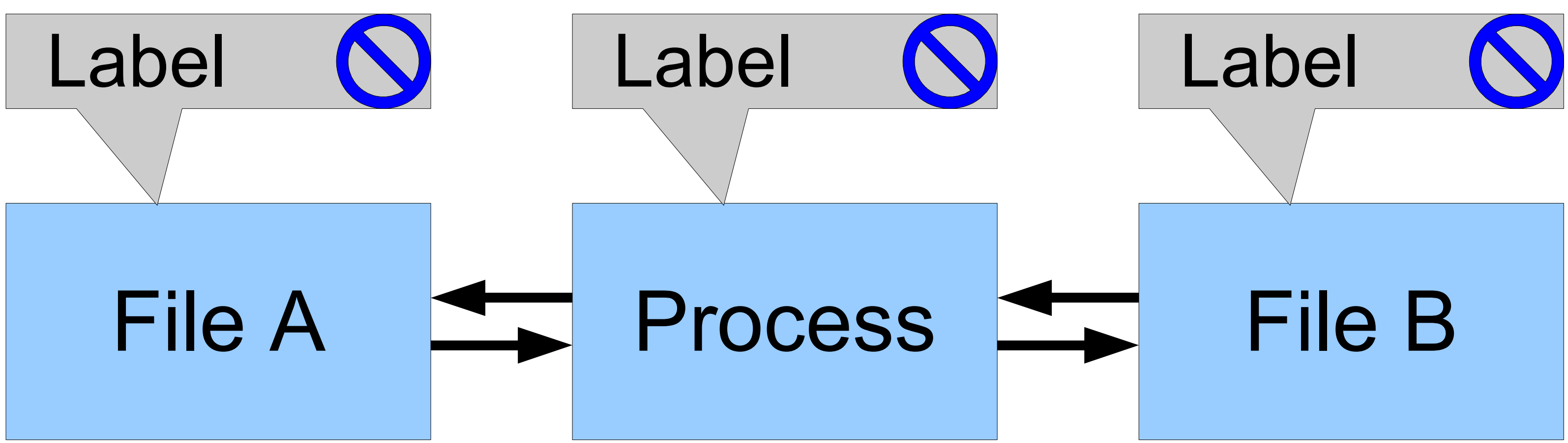
- Results:
  - Can control information flow in distributed system
  - Key idea: self-certifying category names
  - Enforce security of scalable web server in 6,000 lines

Labels control information flow



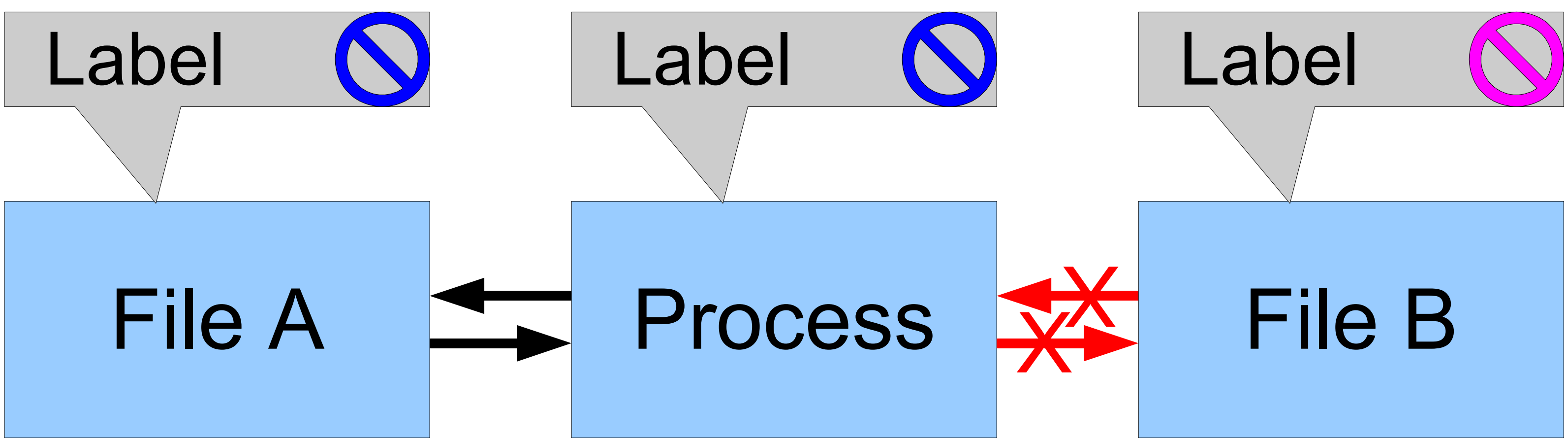
Labels control information flow

- Color is category of data (e.g. my files)
- Blue data can flow only to other blue objects



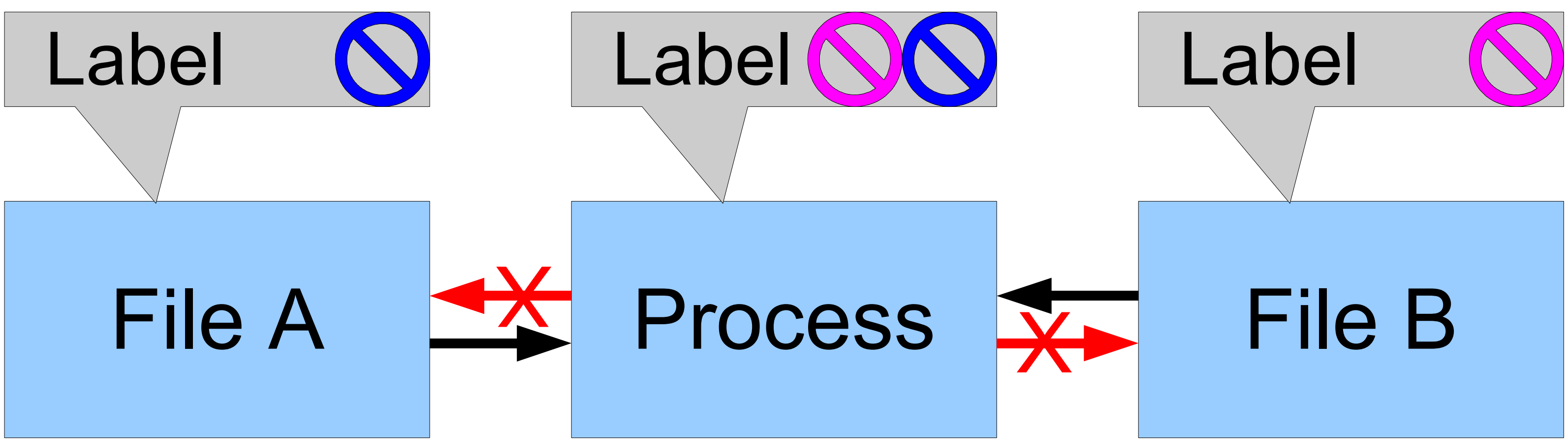
Labels control information flow

- Color is category of data (e.g. my files)
- Blue data can flow only to other blue objects



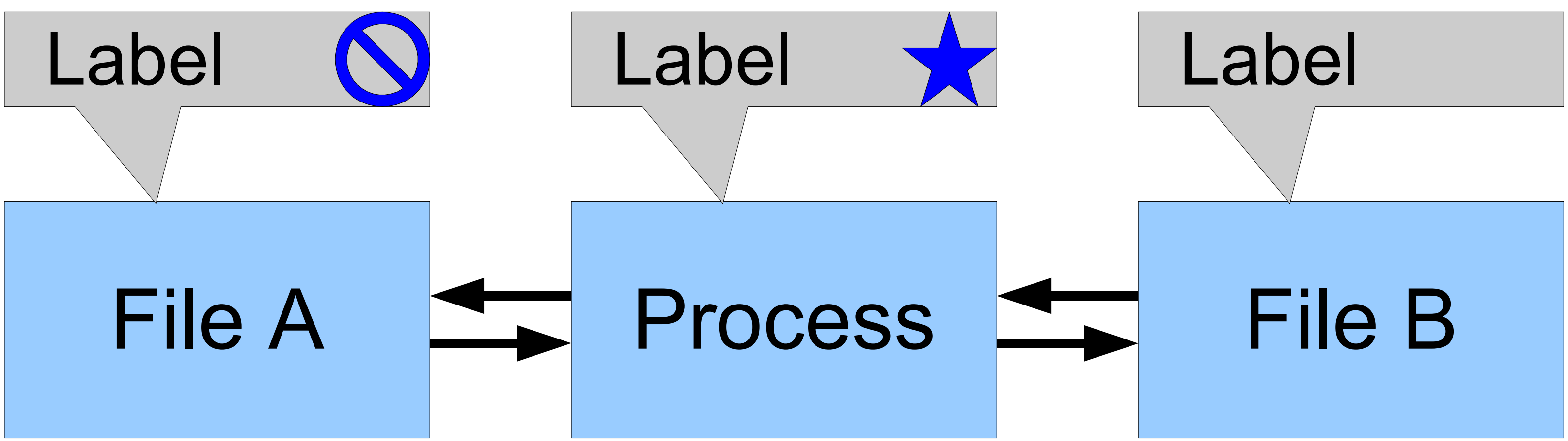
Labels control information flow

- Color is category of data (e.g. my files)
- Blue data can flow only to other blue objects



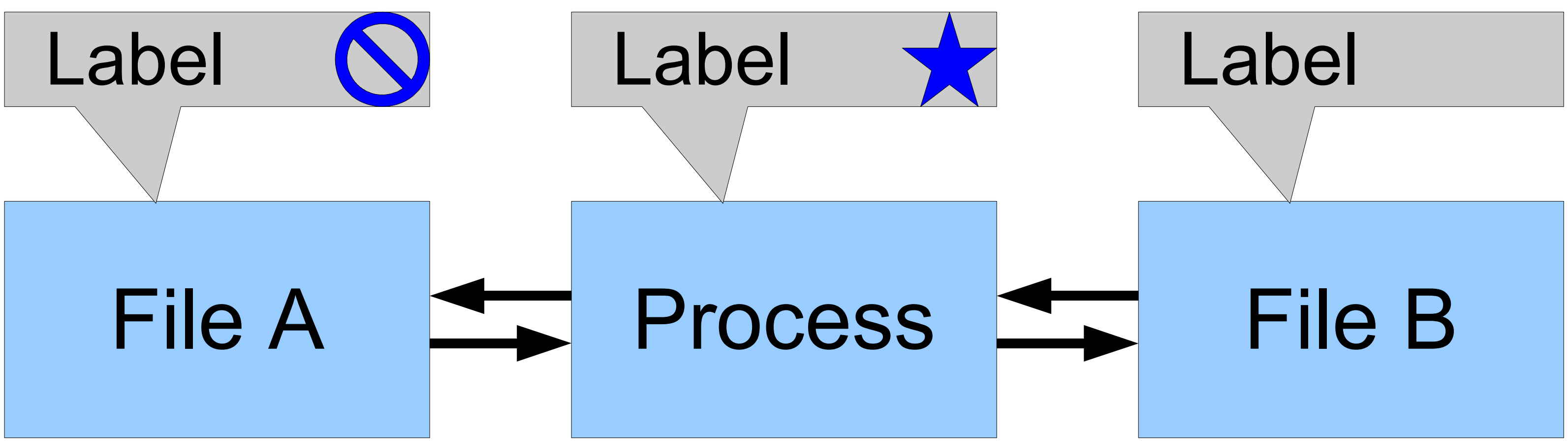
Labels control information flow

- Color is category of data (e.g. my files)
- Blue data can flow only to other blue objects
- Owns blue data, can remove color (e.g. encrypt)

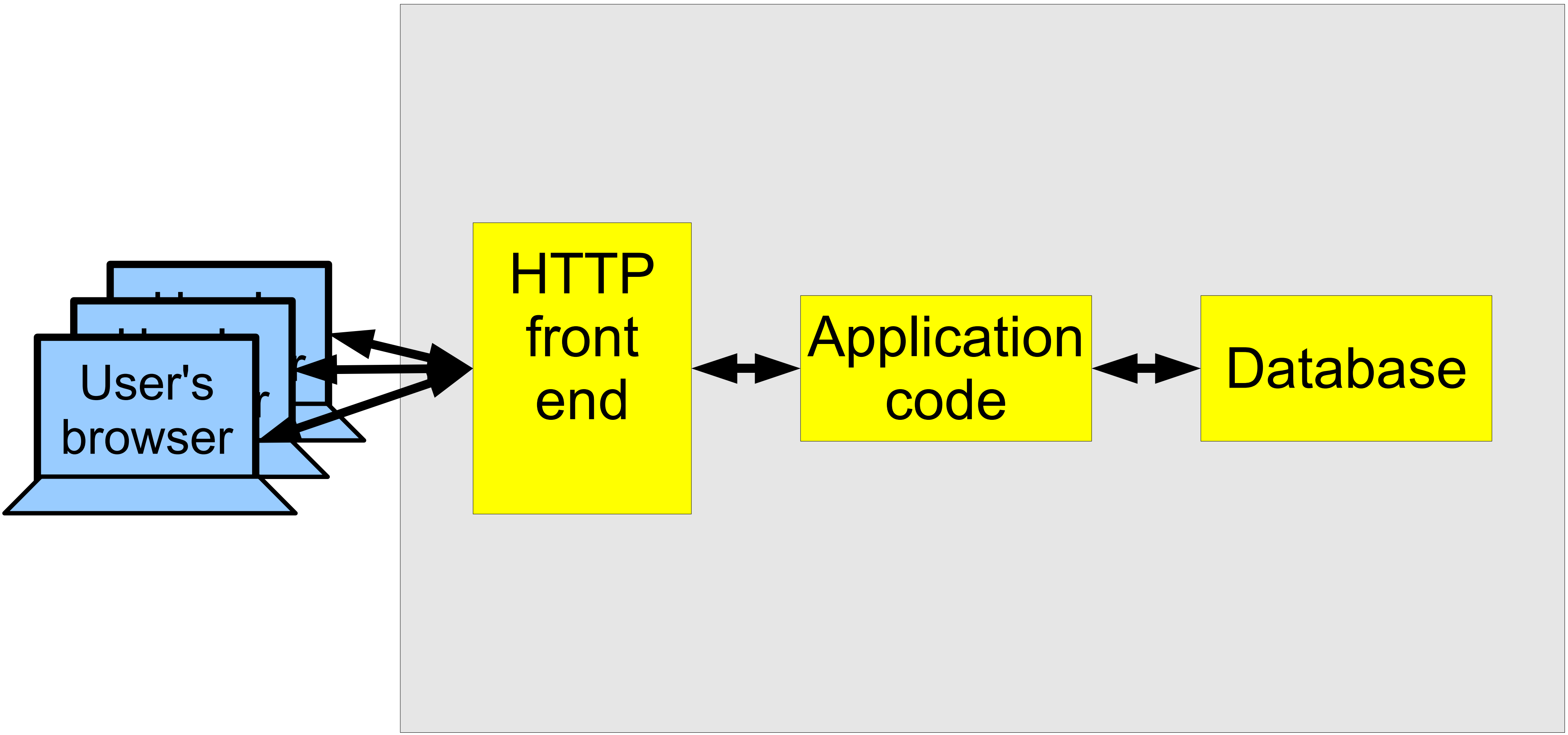


Labels are egalitarian

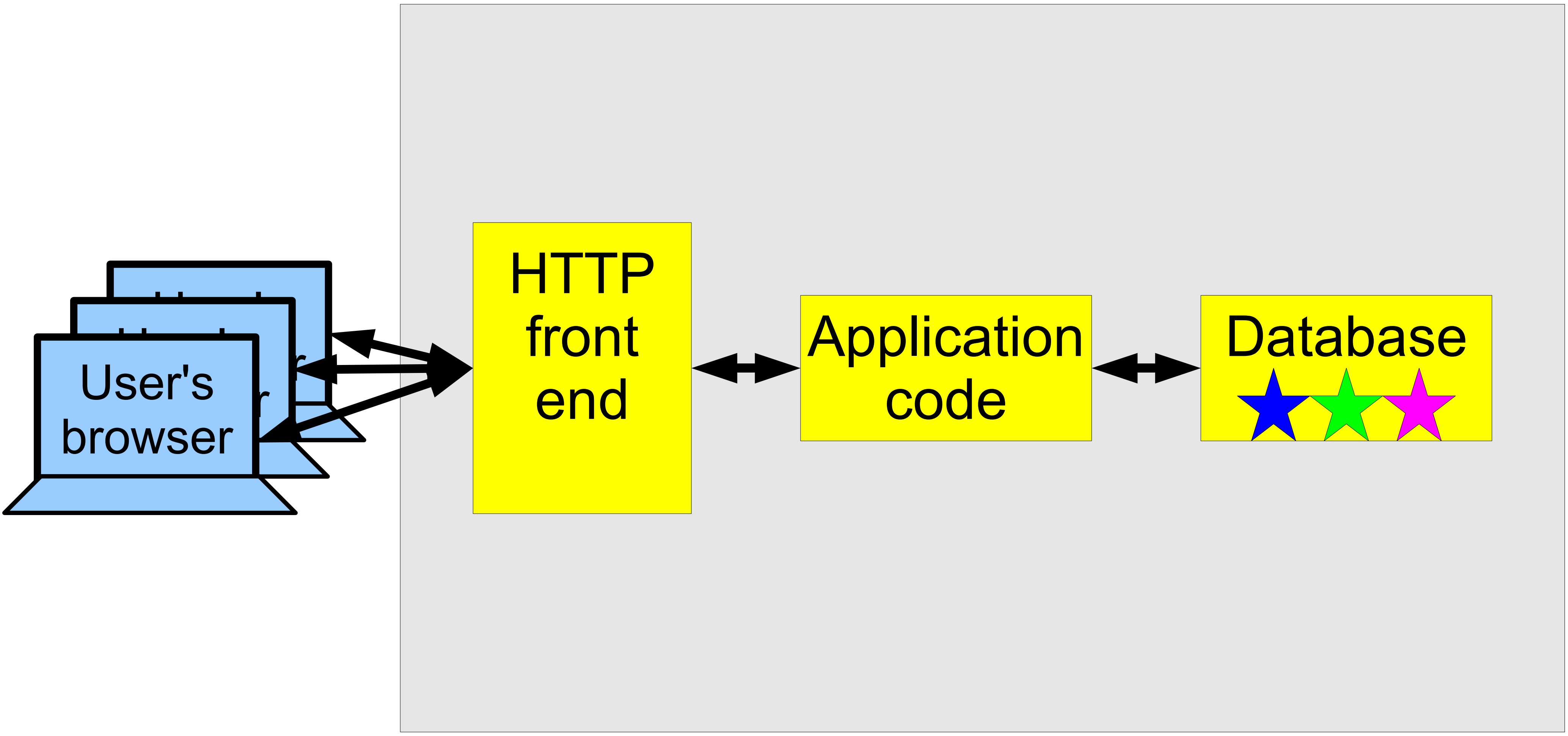
- Any process can request a new category (color)
  - Gets ownership of that category (★)
  - Uses category in labels to control information flow
  - Can grant ownership to others



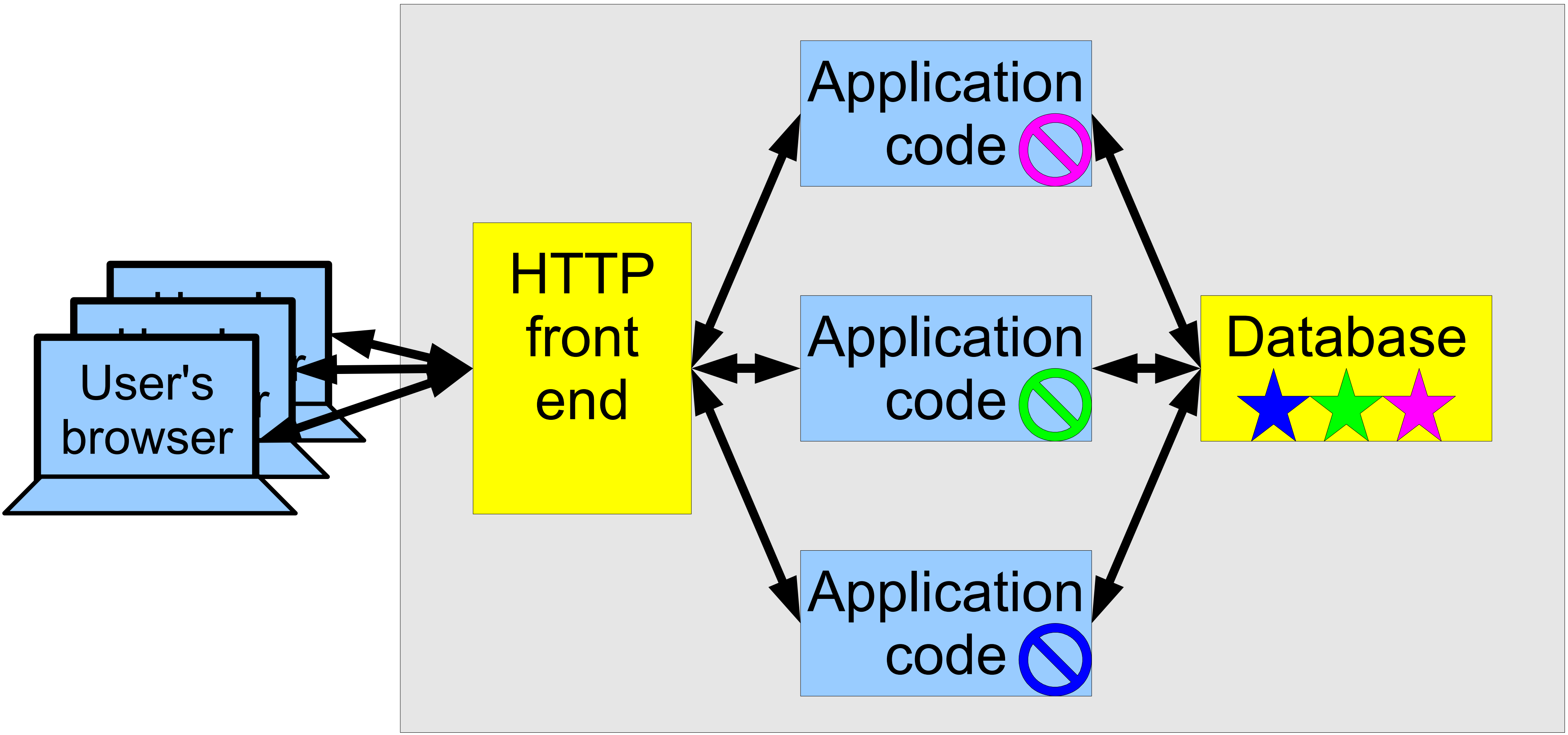
Traditional web server: lots of trusted (yellow) code



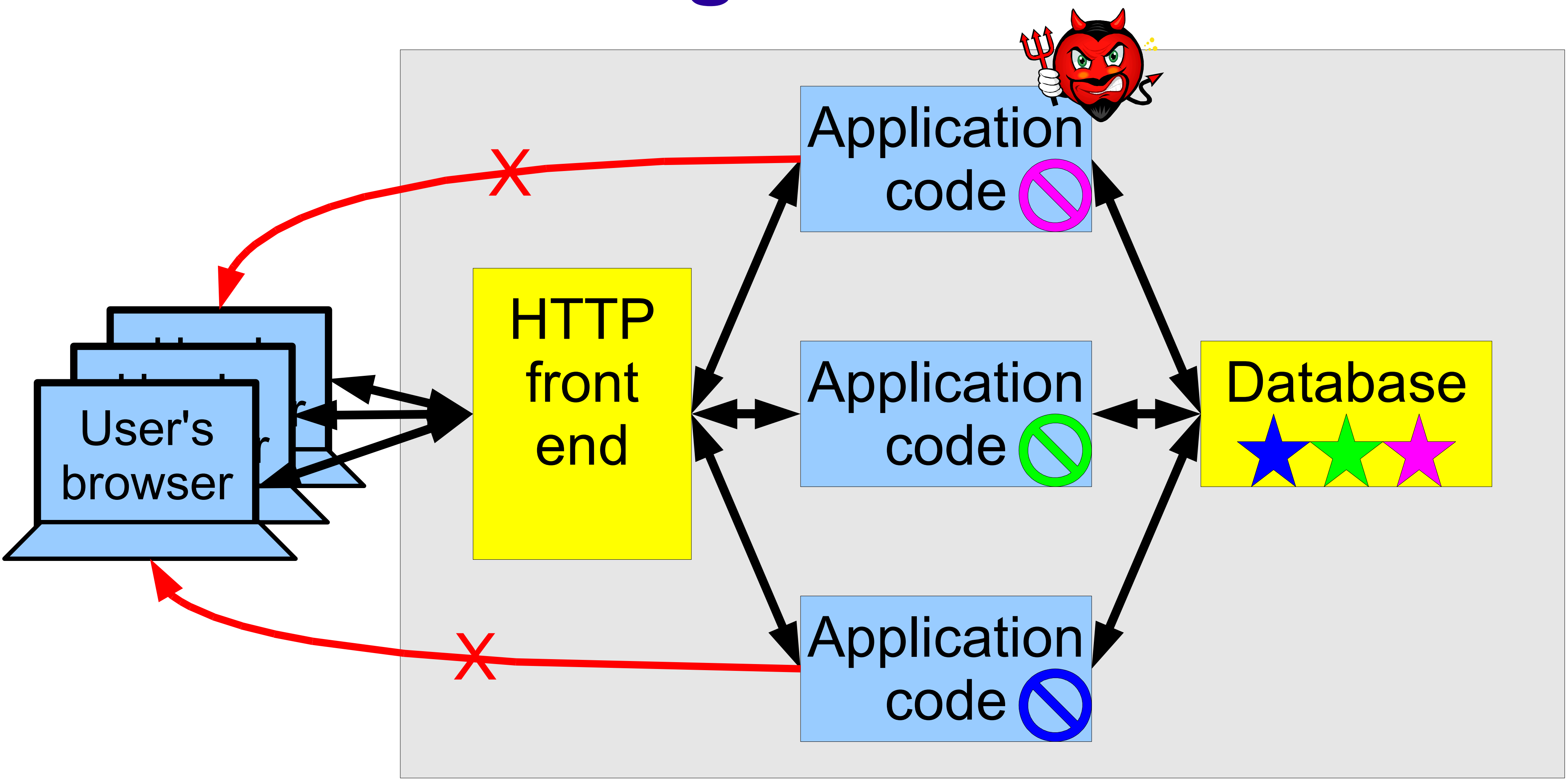
Information flow control: separate color for each user's data



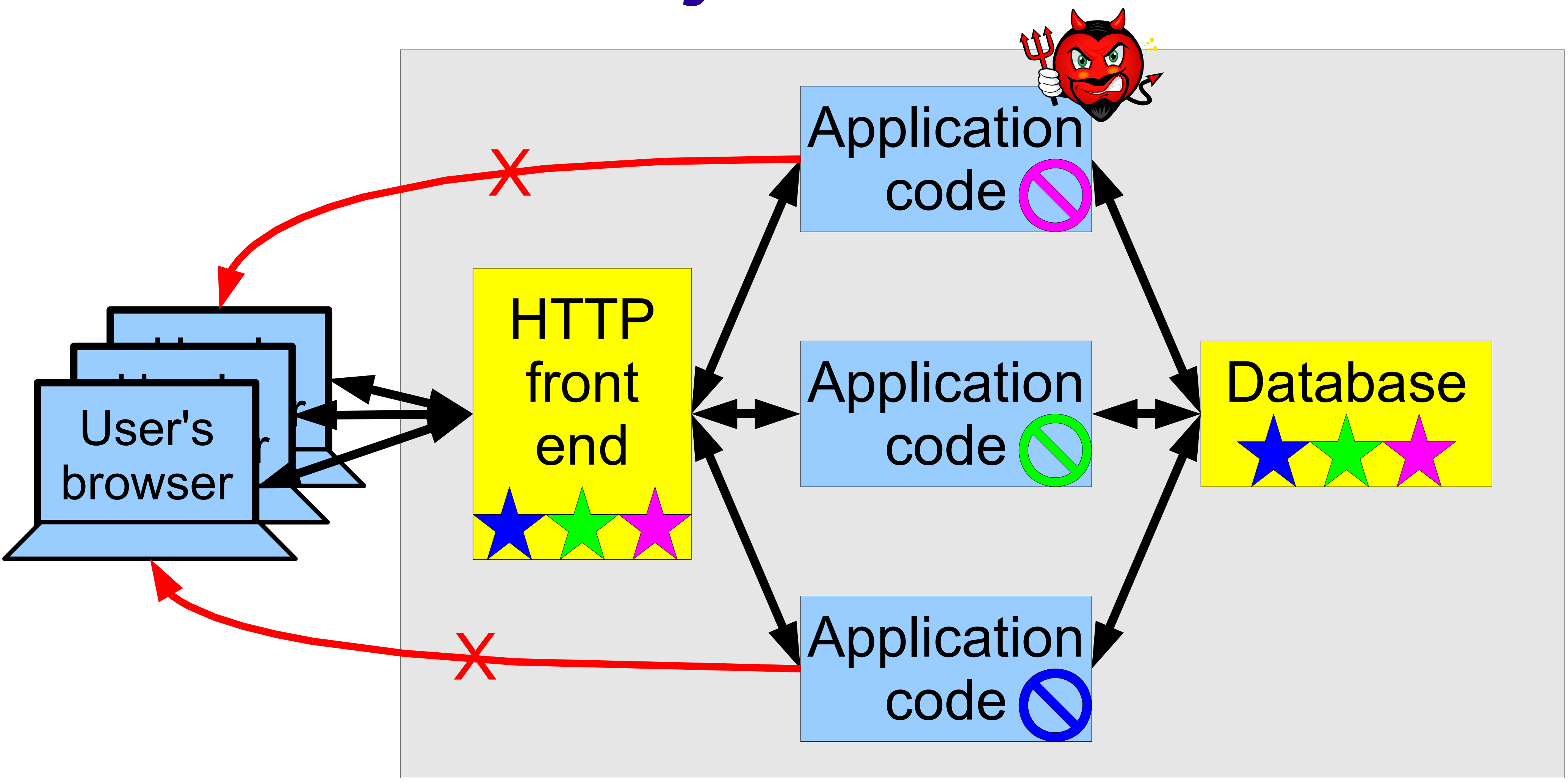
Information flow control: track each user's data in app



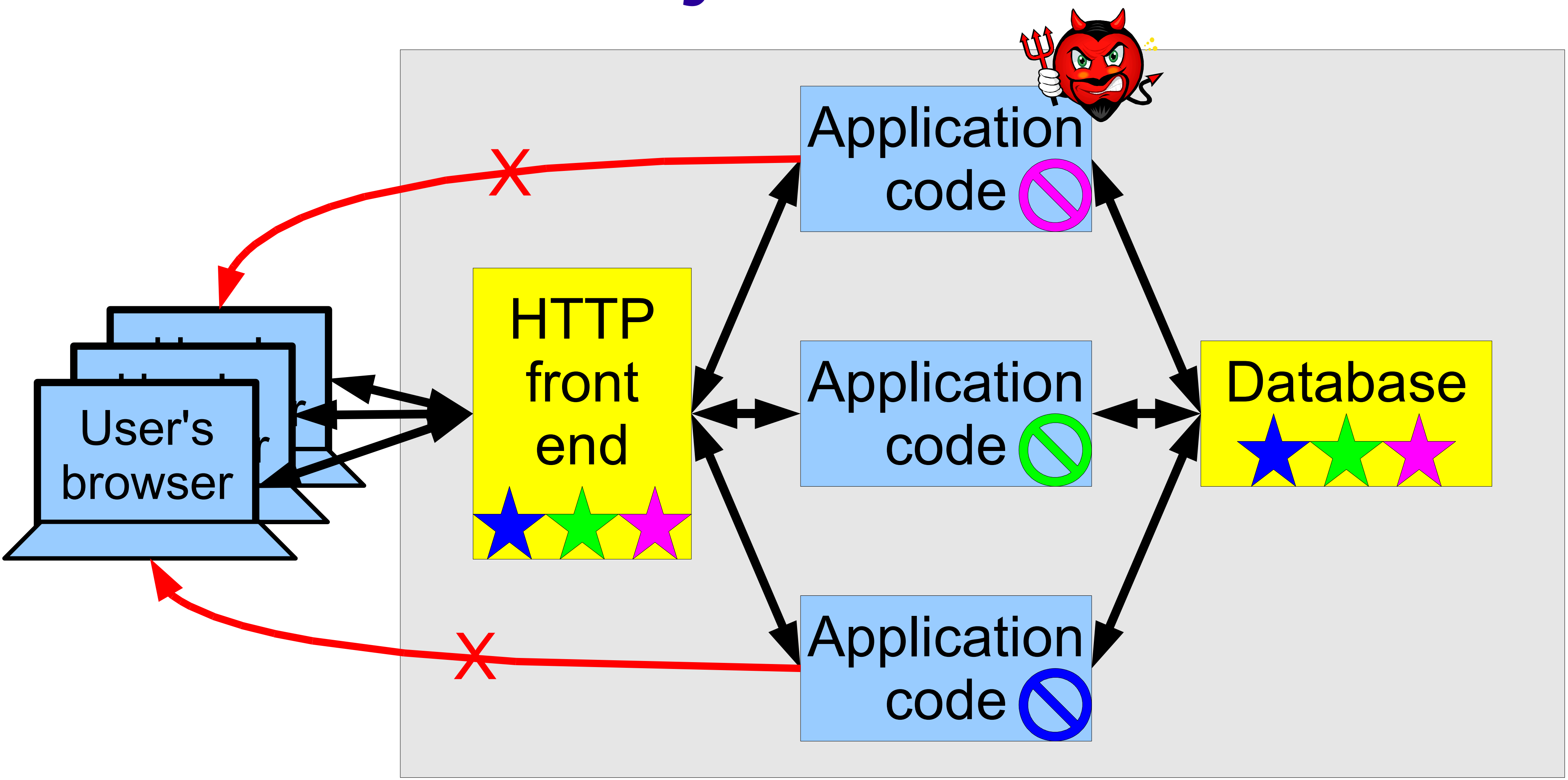
Labels prevent application code from disclosing data onto network



Front-end uses ownership to send data *only* to user's browser

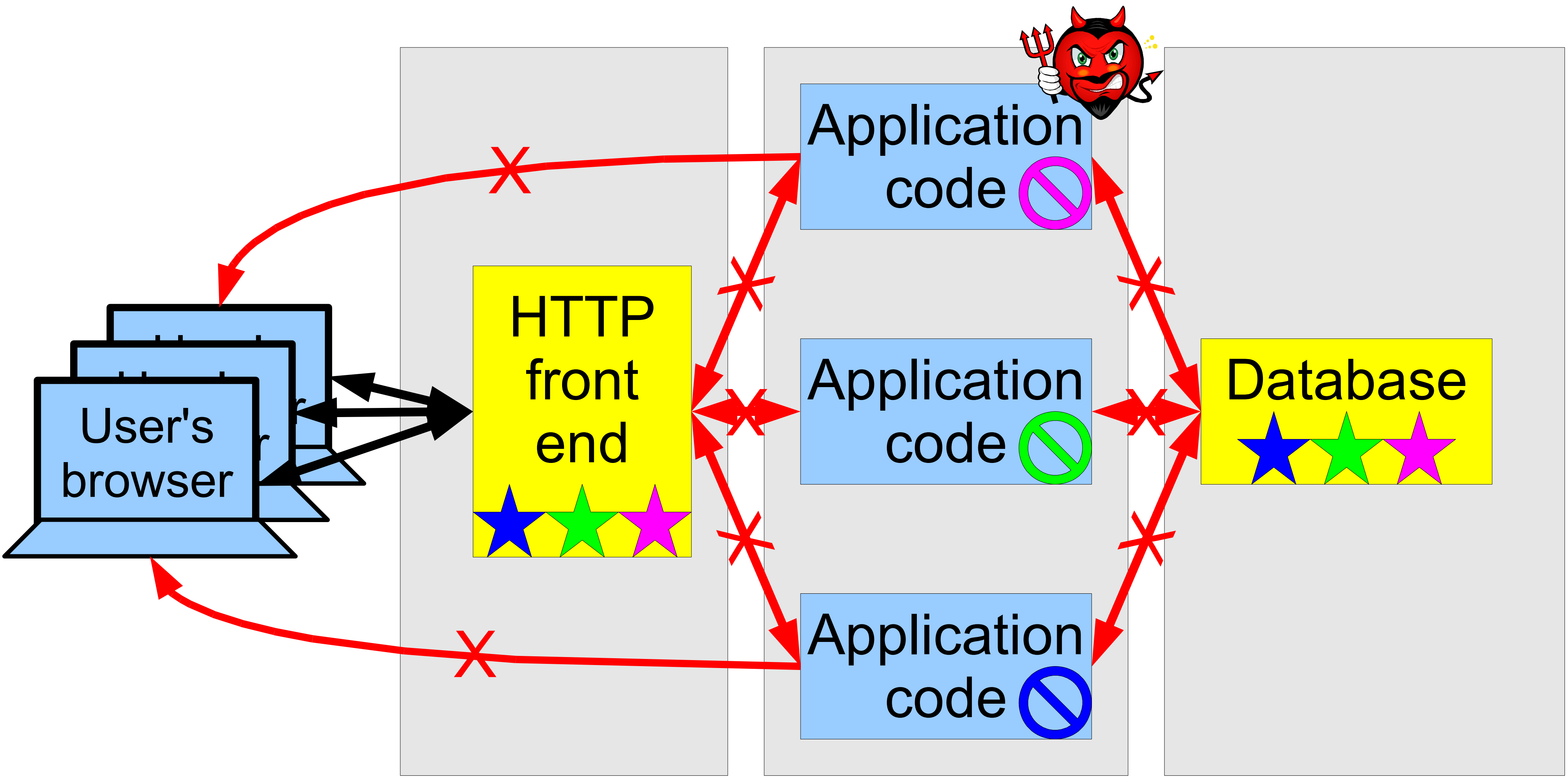


Front-end uses ownership to send data *only* to user's browser



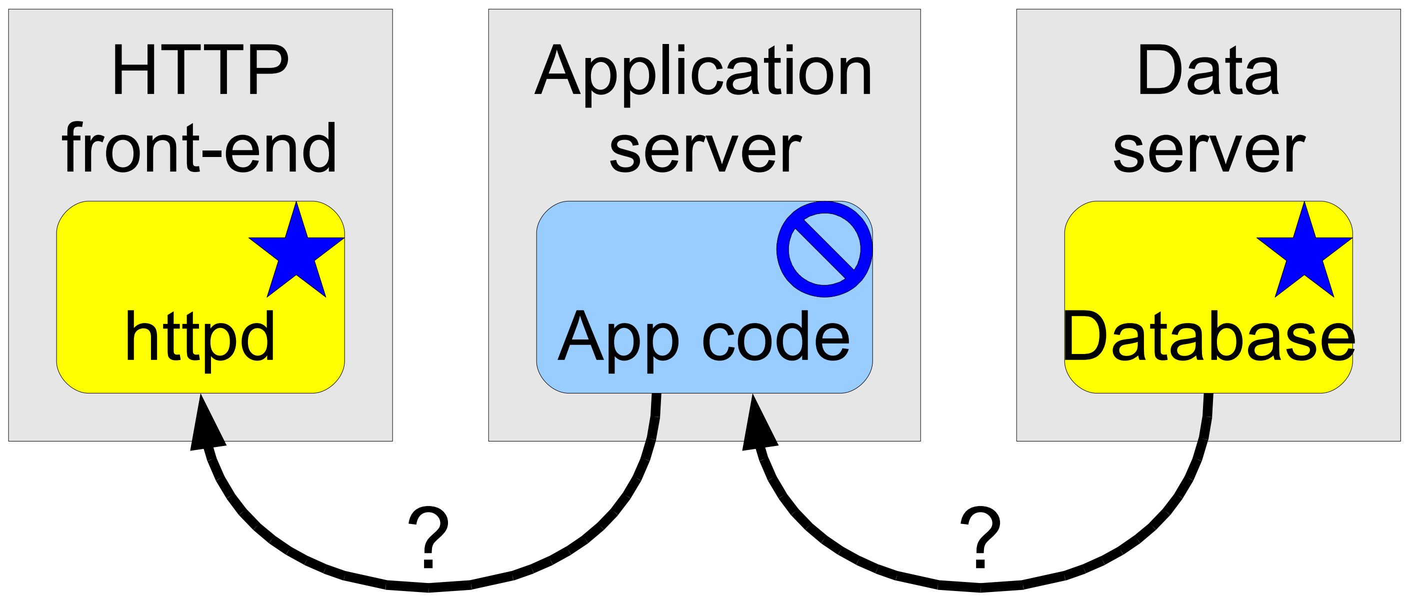
- What happens when the server gets overloaded?

Limitation: OS alone cannot control information flow in distributed system



Distributed challenge: when to allow processes to communicate?

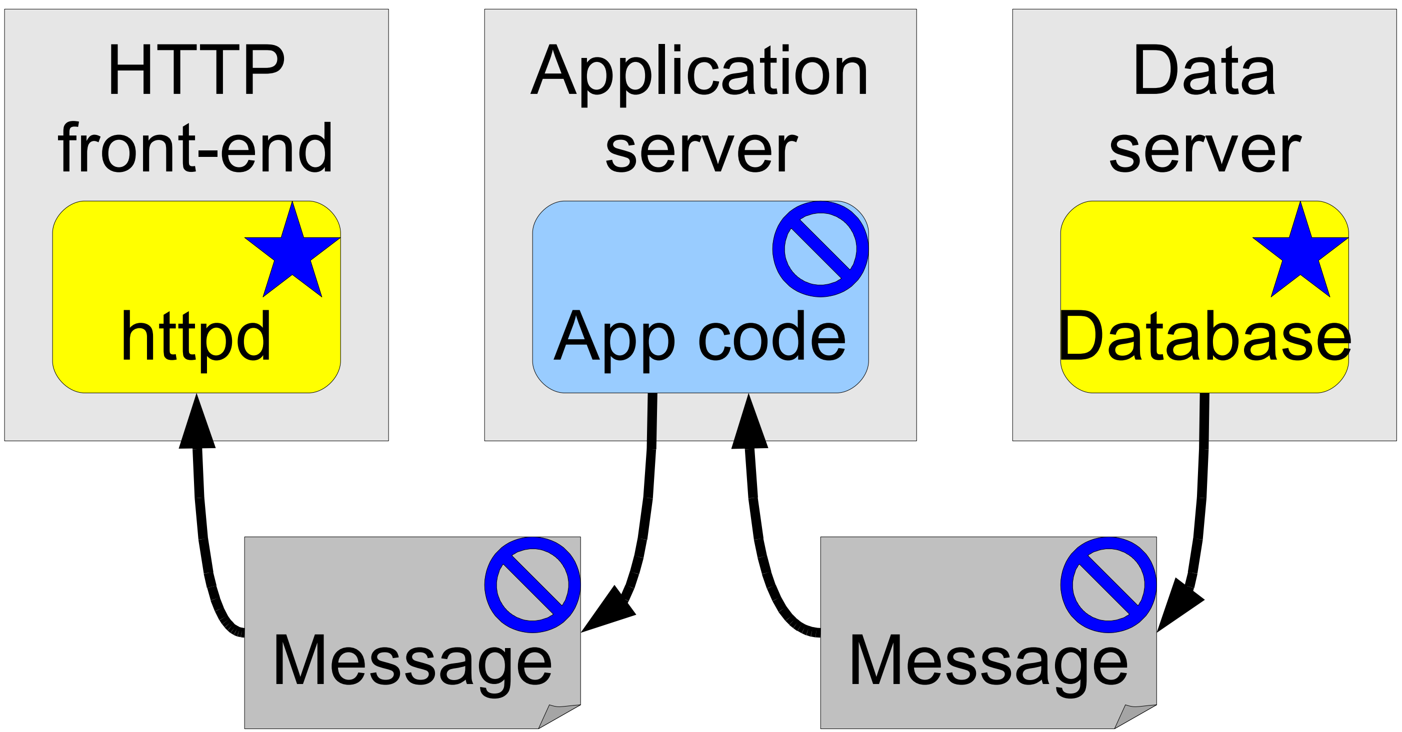
- Design goal: decentralized – no fully-trusted parts
  - (Not the usual meaning of decentralized IFC, or DIFC)



- Challenge: no equivalent of a fully-trusted OS kernel that can make all decisions

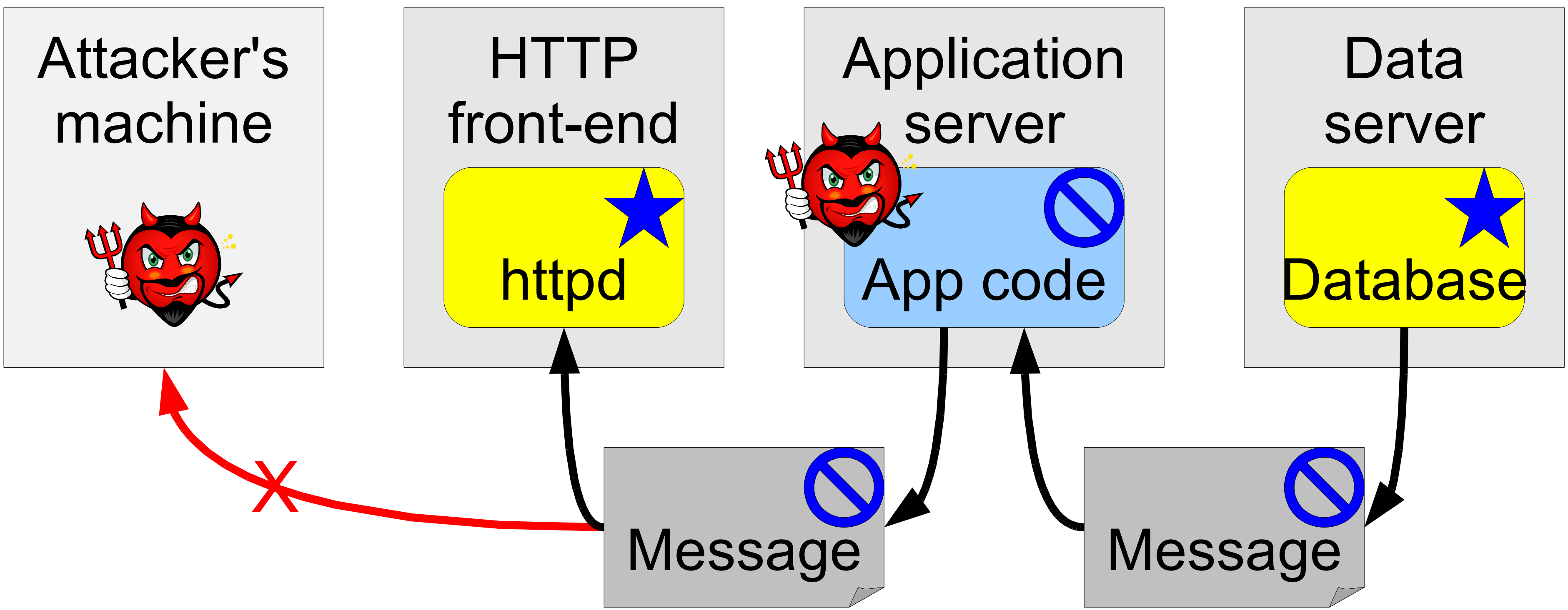
High-level approach: encode labels in messages

Each machine uses OS to enforce labels locally



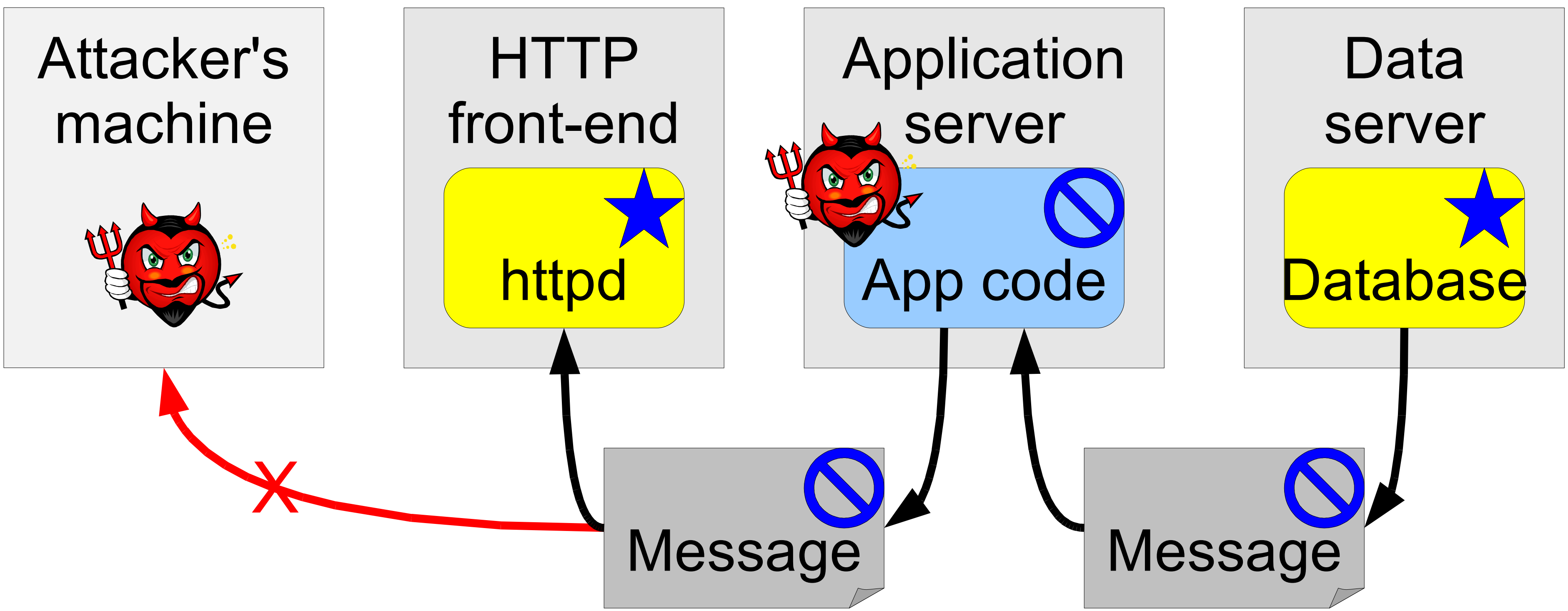
Problem: decentralized trust

- When can we trust the recipient with message?



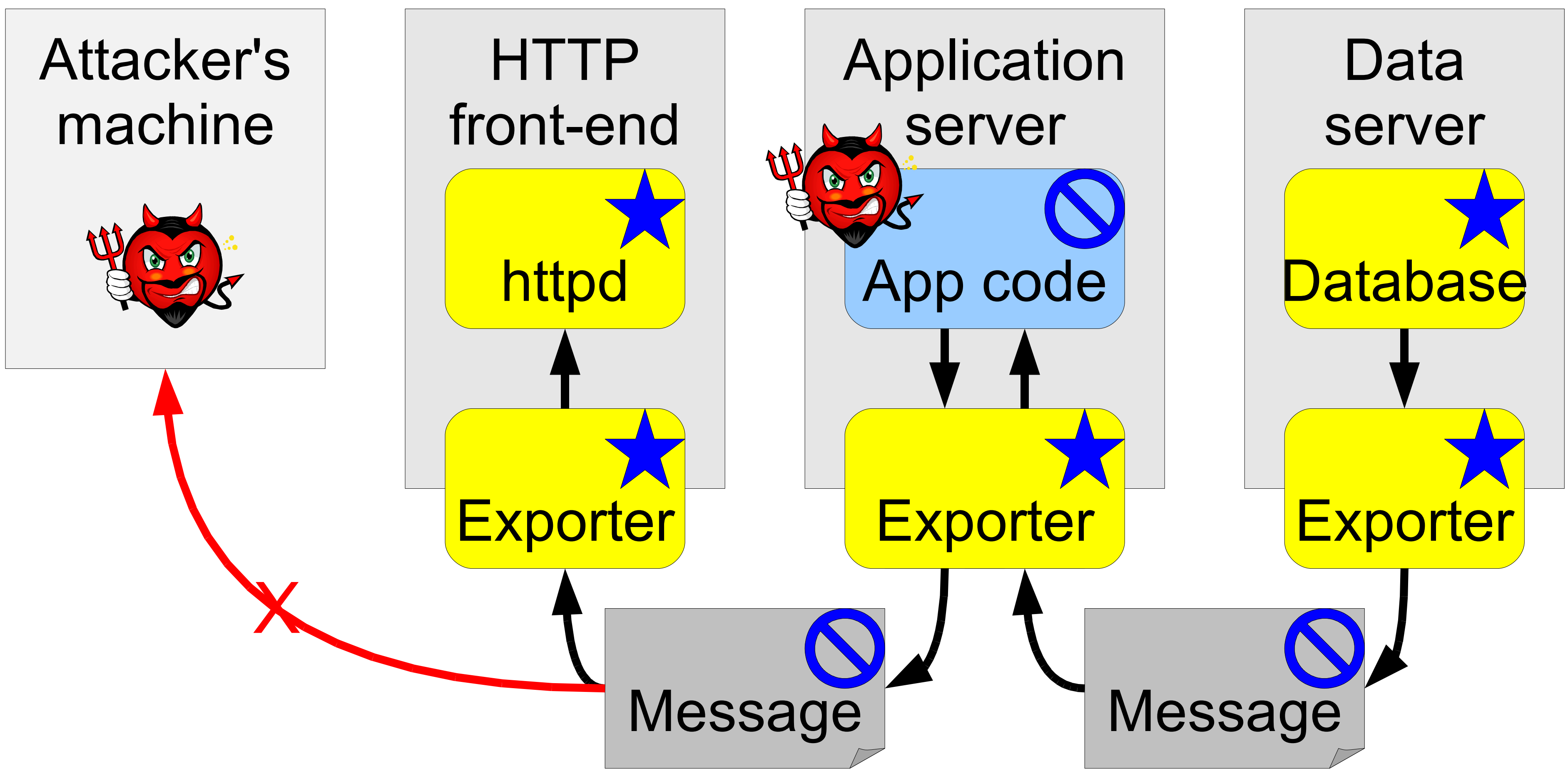
Solution: per-category trust

- DB trusts front-end, app servers with a particular user's data (e.g. messages labeled blue)
- But DB doesn't trust the app code...

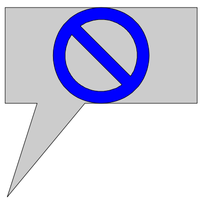


Exporters control information flow on each machine using local OS

- Database doesn't trust the app code, but trusts the app server's exporter to contain the app code



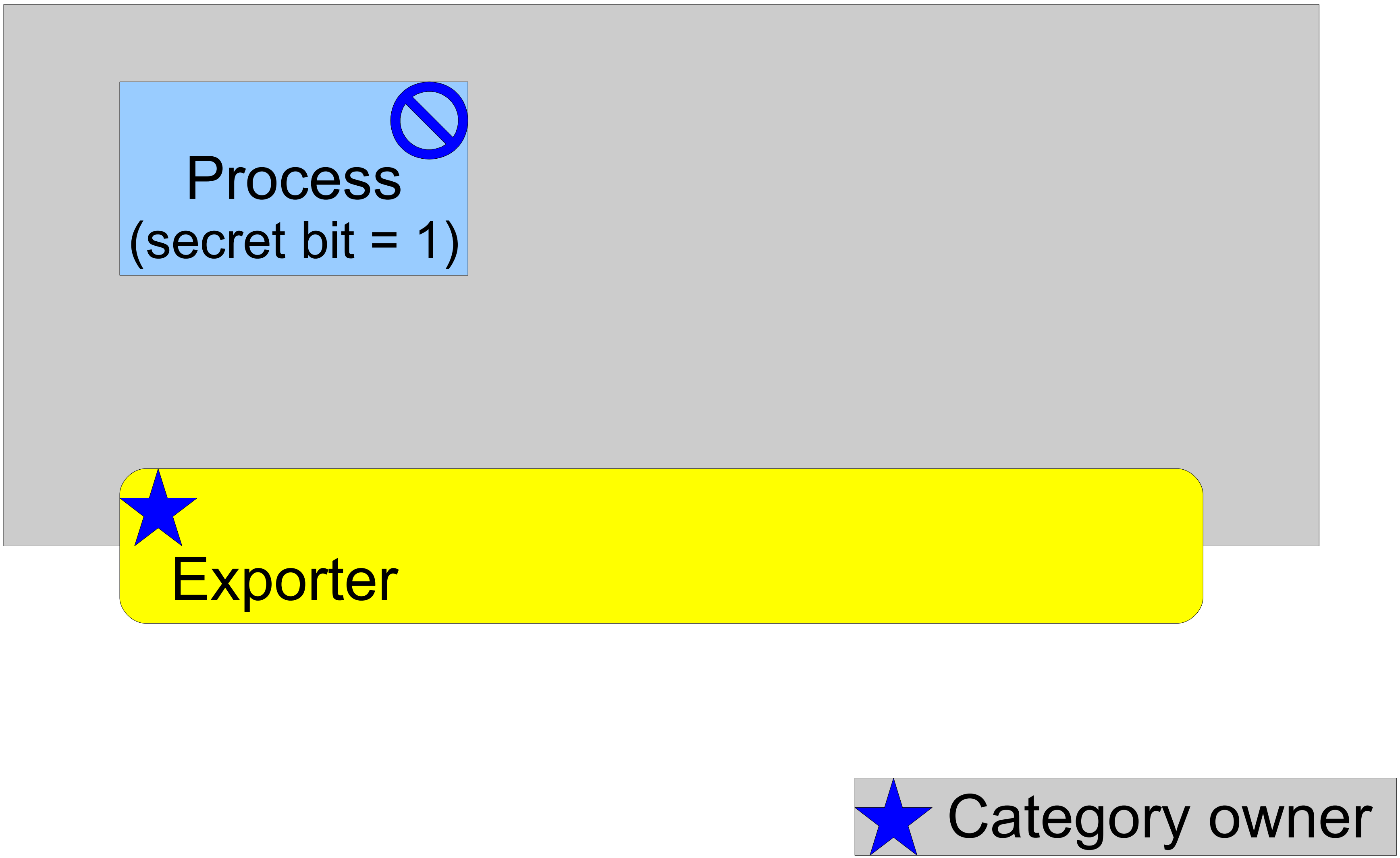
Exporter's API



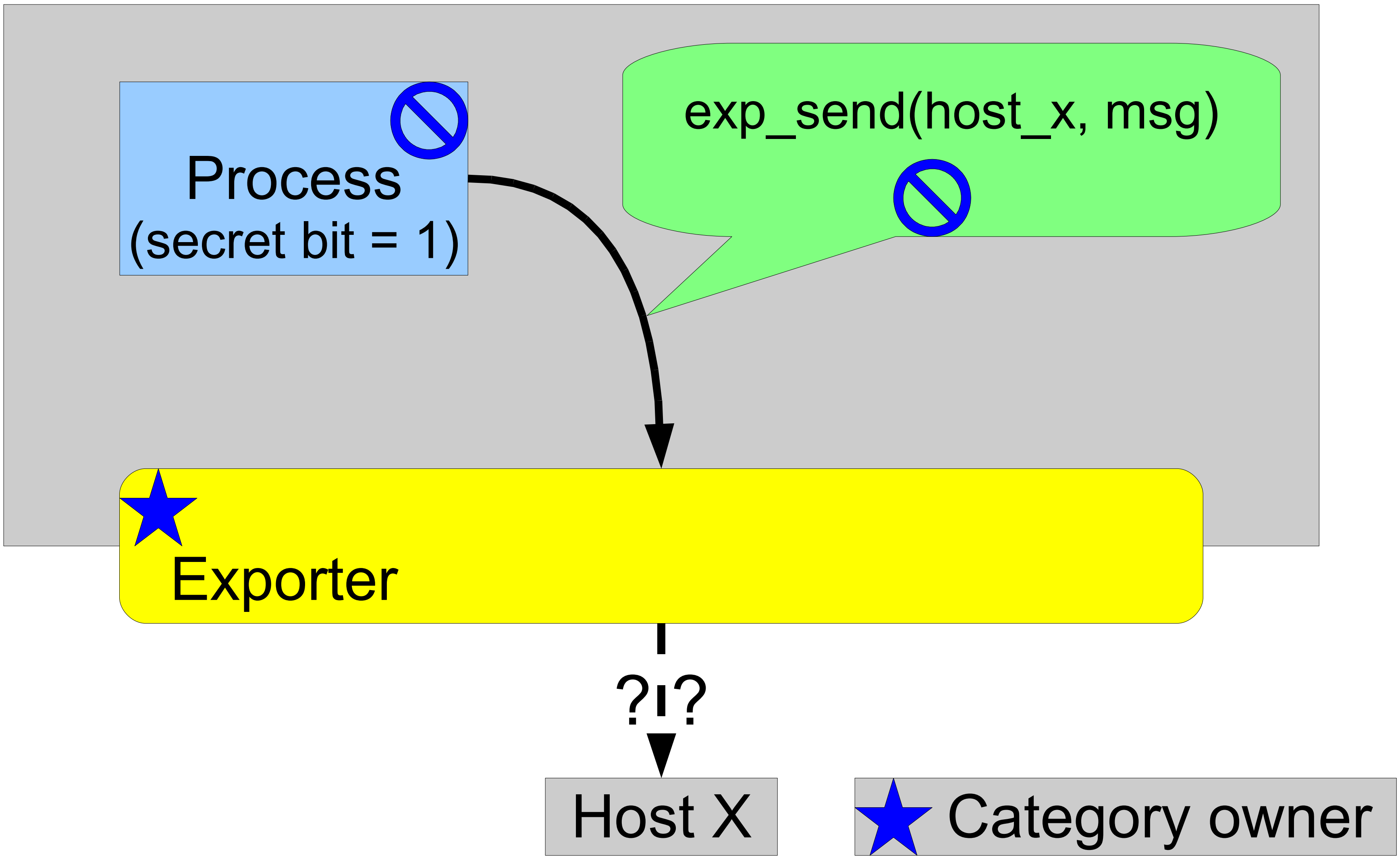
exp_send(*dest_host*, *dest_mbox*, *msg*, *label*)

- Exporter provides interface to send datagrams
- Message should only be sent if every category in *label* trusts the machine *dest_host*
- How does the exporter check for this trust?

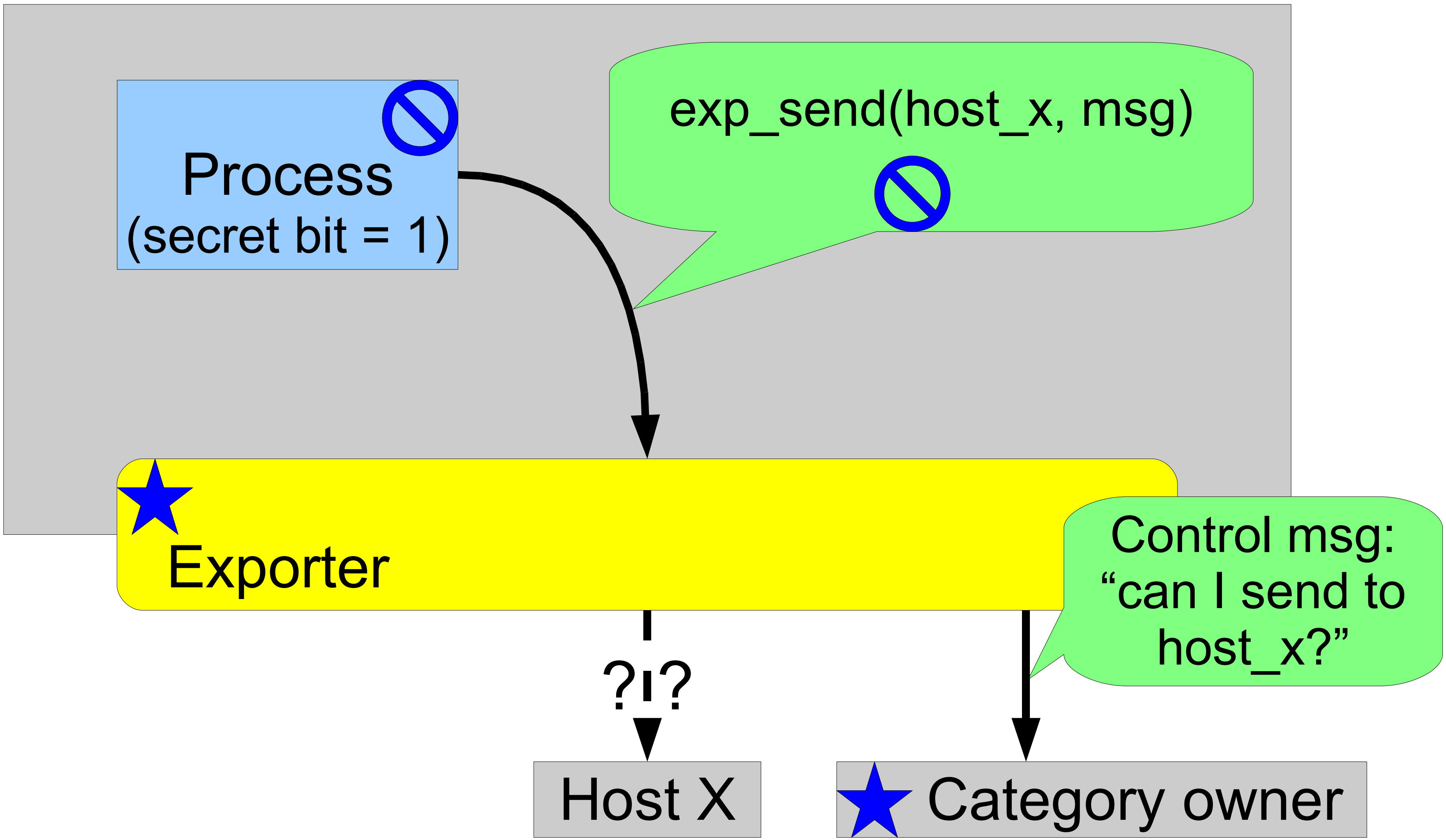
Strawman: check trust by querying category owners



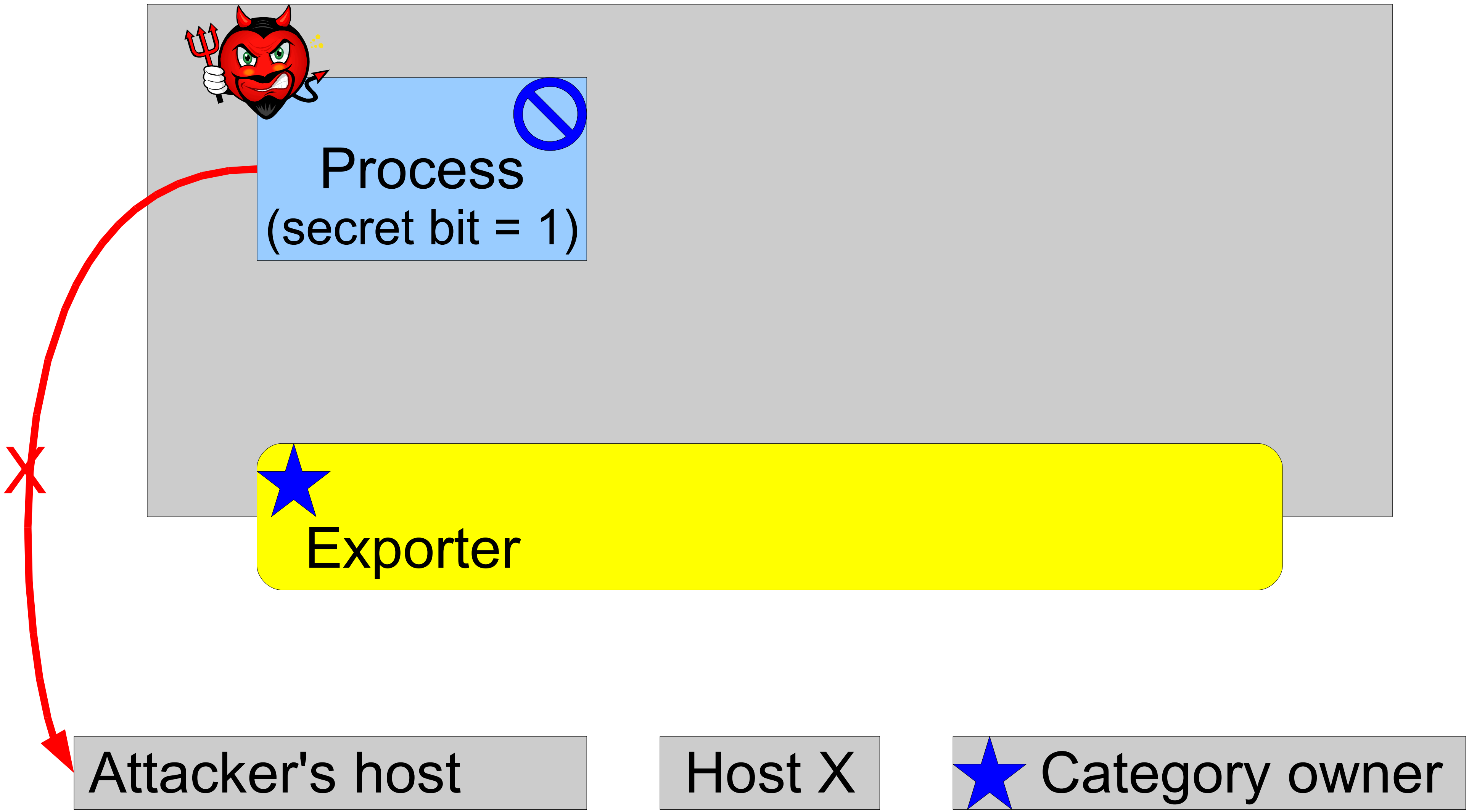
Strawman: check trust by querying category owners



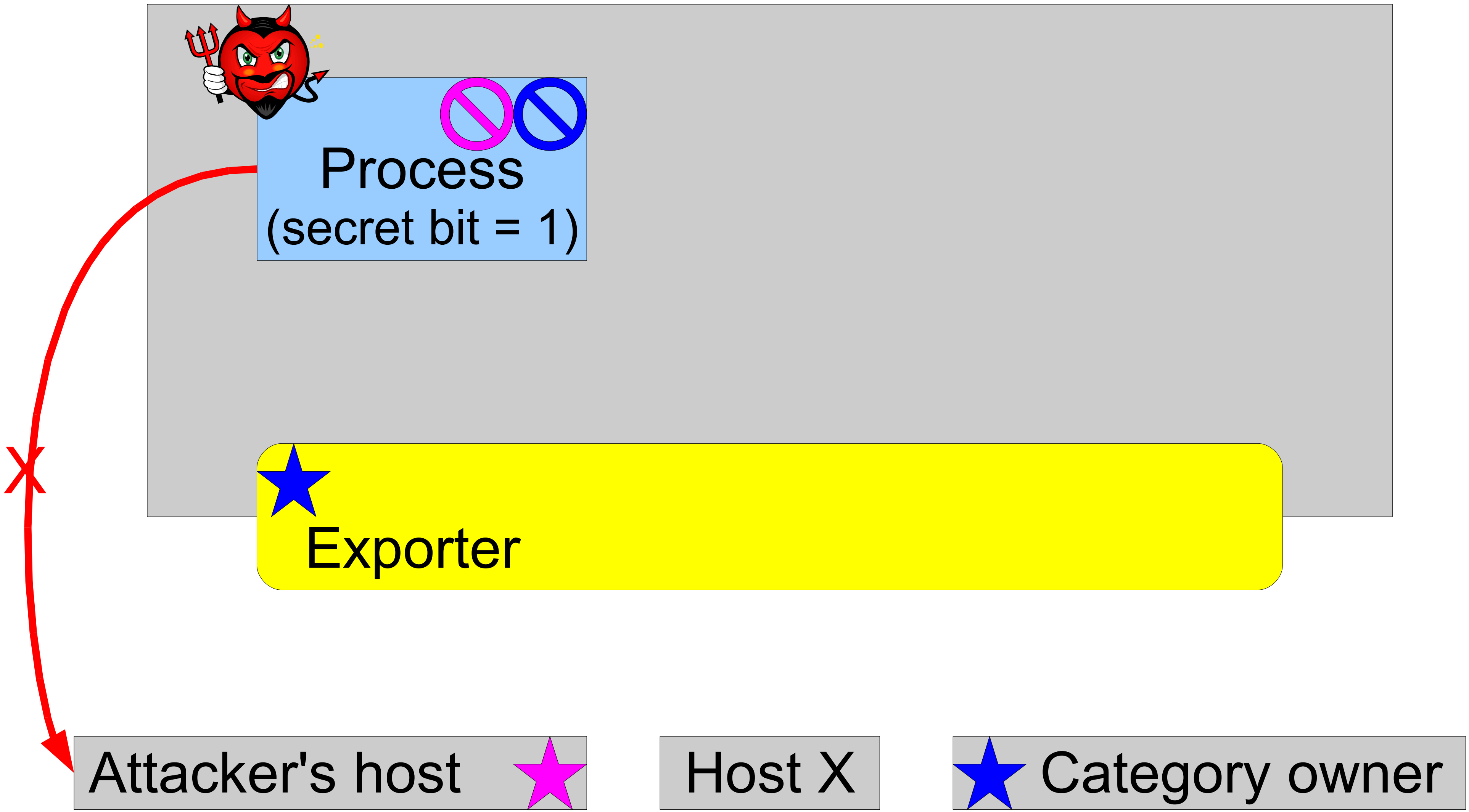
Strawman: check trust by querying category owners



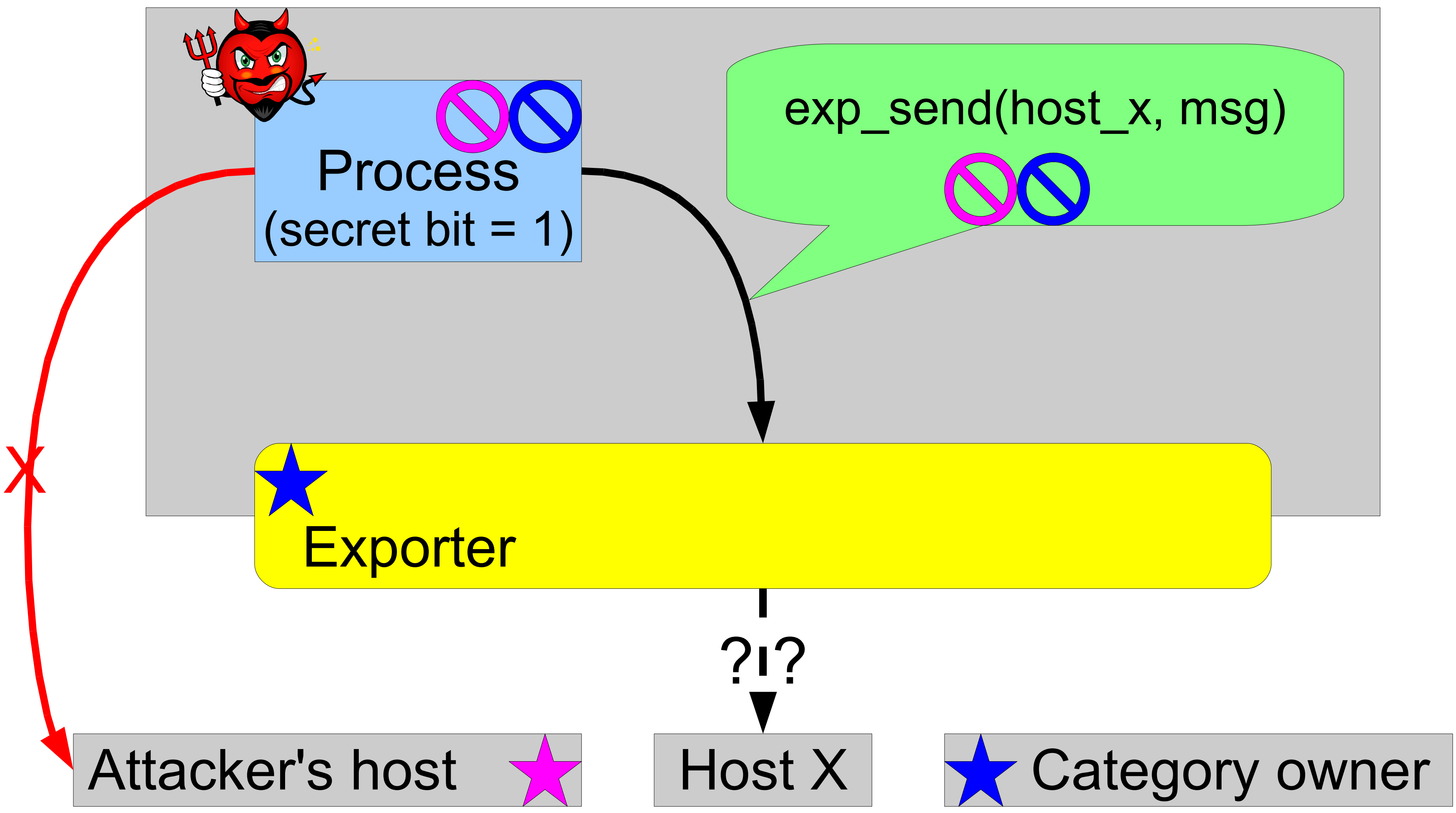
Querying category owners creates a covert channel in API



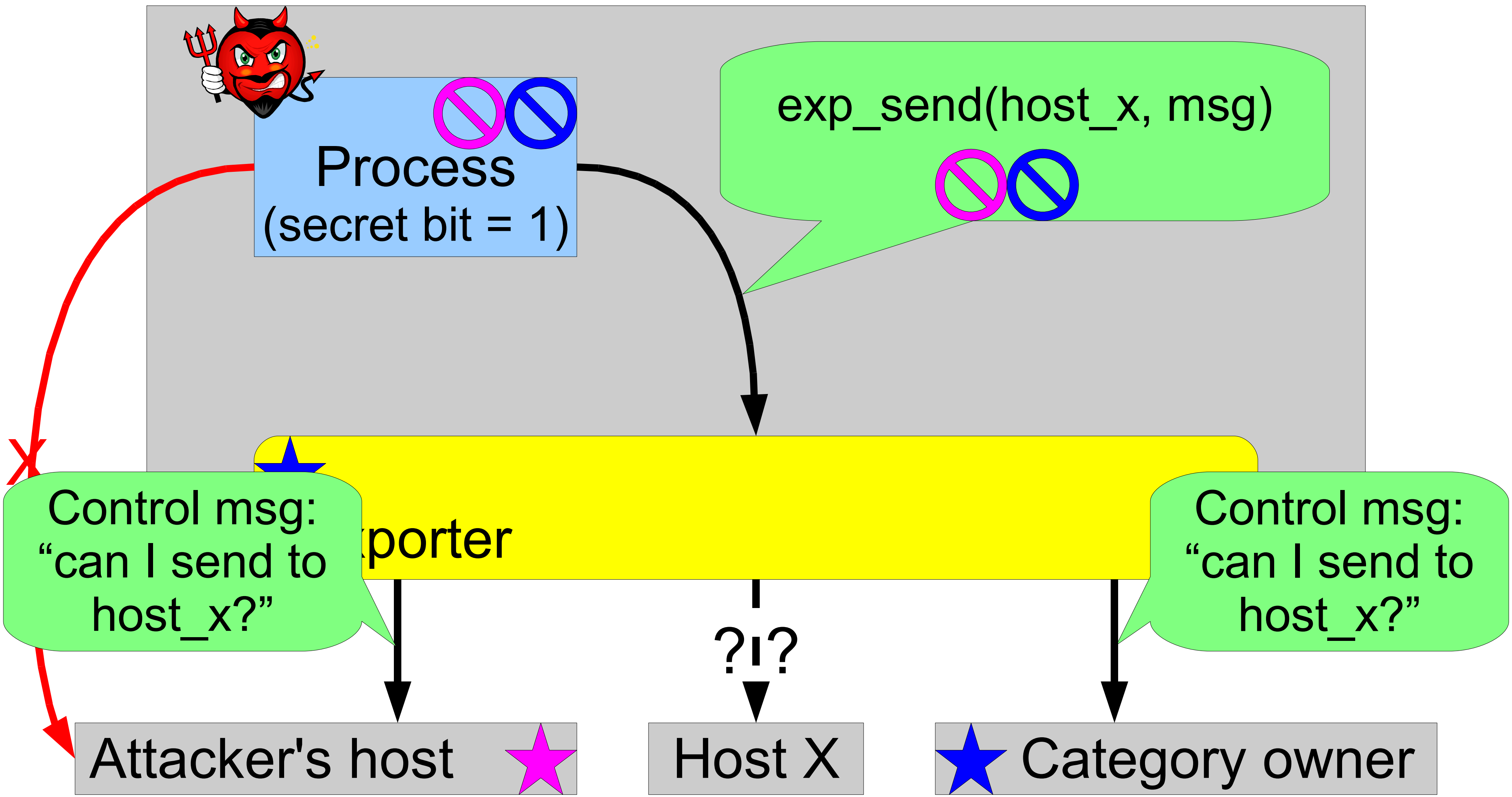
Querying category owners creates a covert channel in API



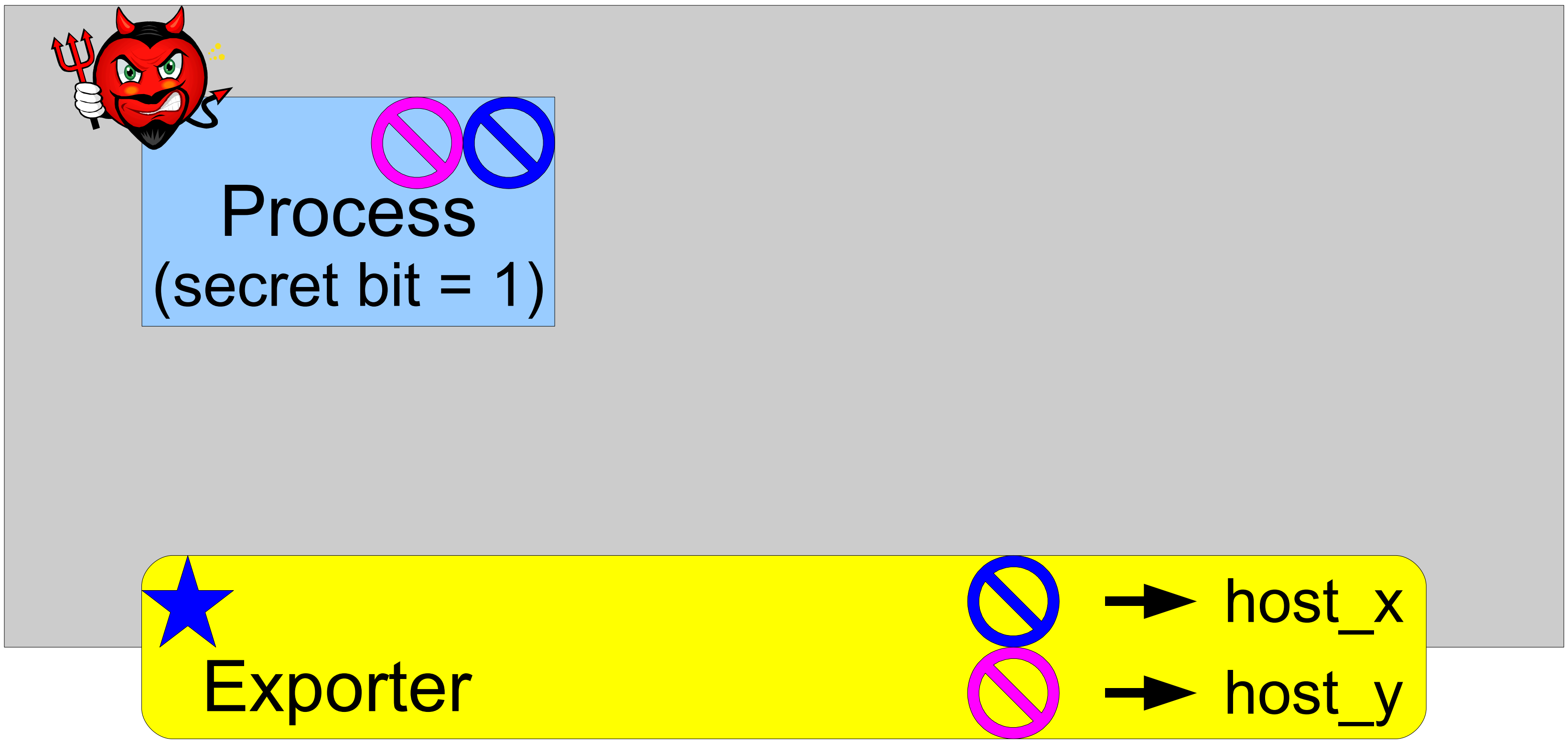
Querying category owners creates a covert channel in API



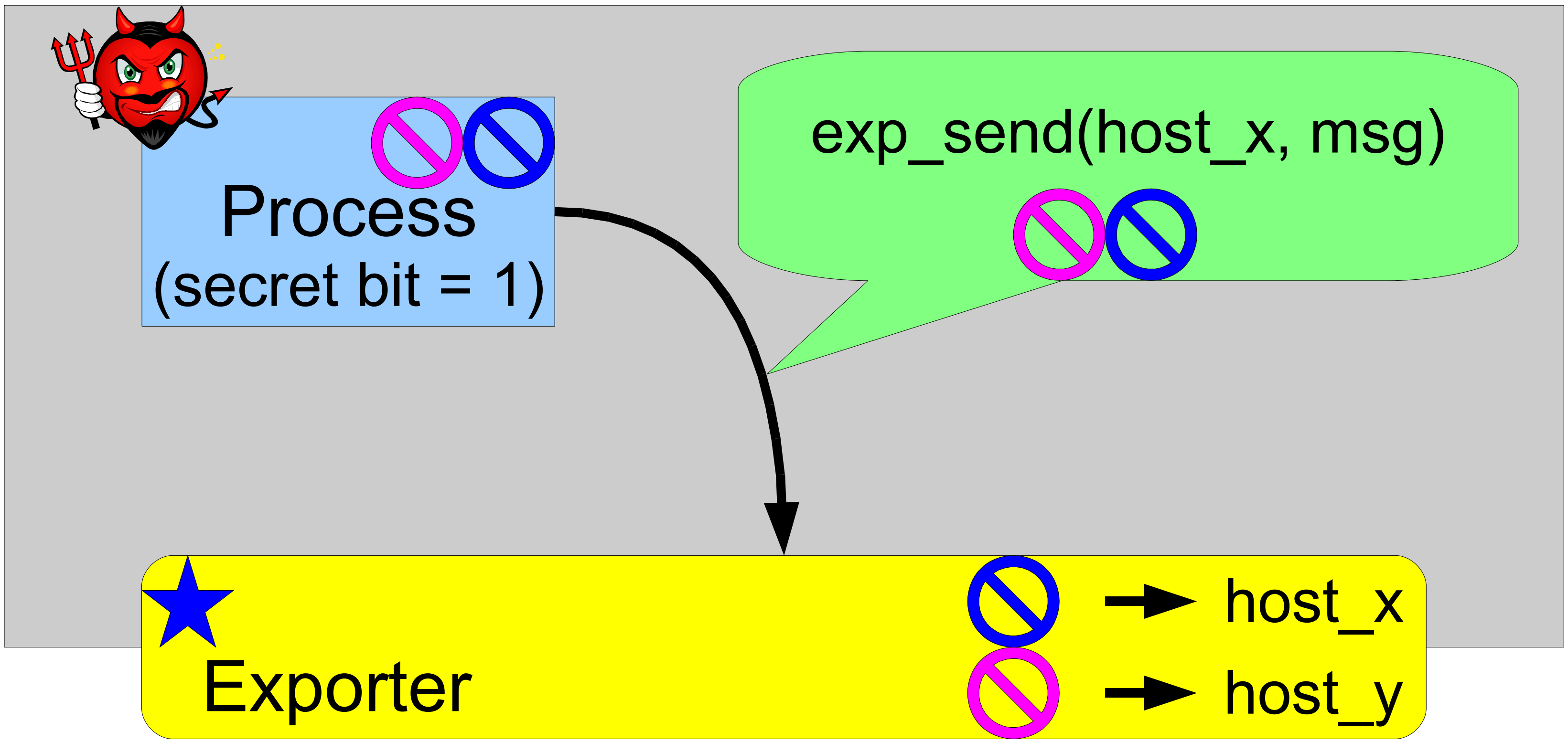
Querying category owners creates a covert channel in API



Strawman 2: store trust in exporter

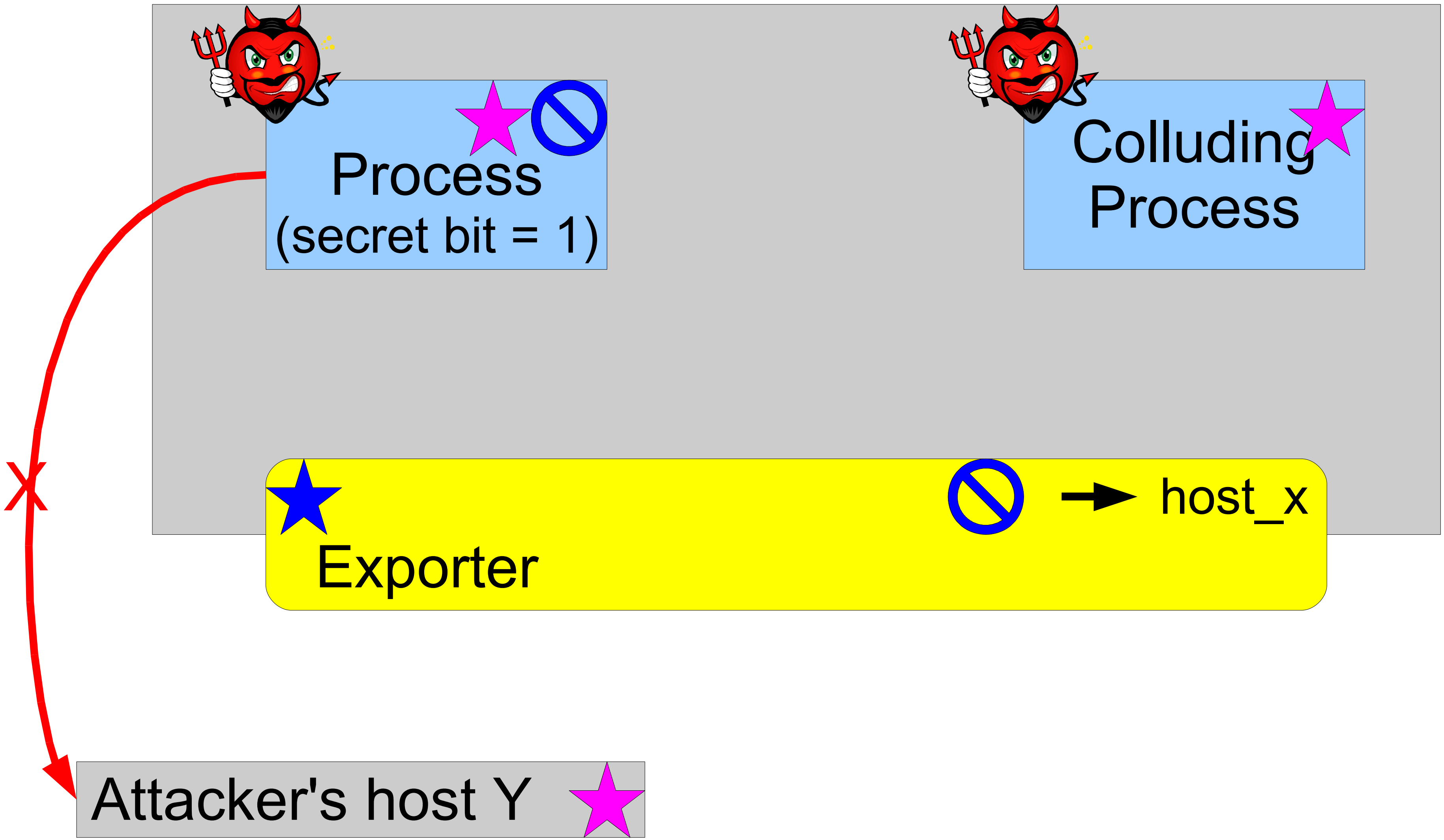


Strawman 2: store trust in exporter

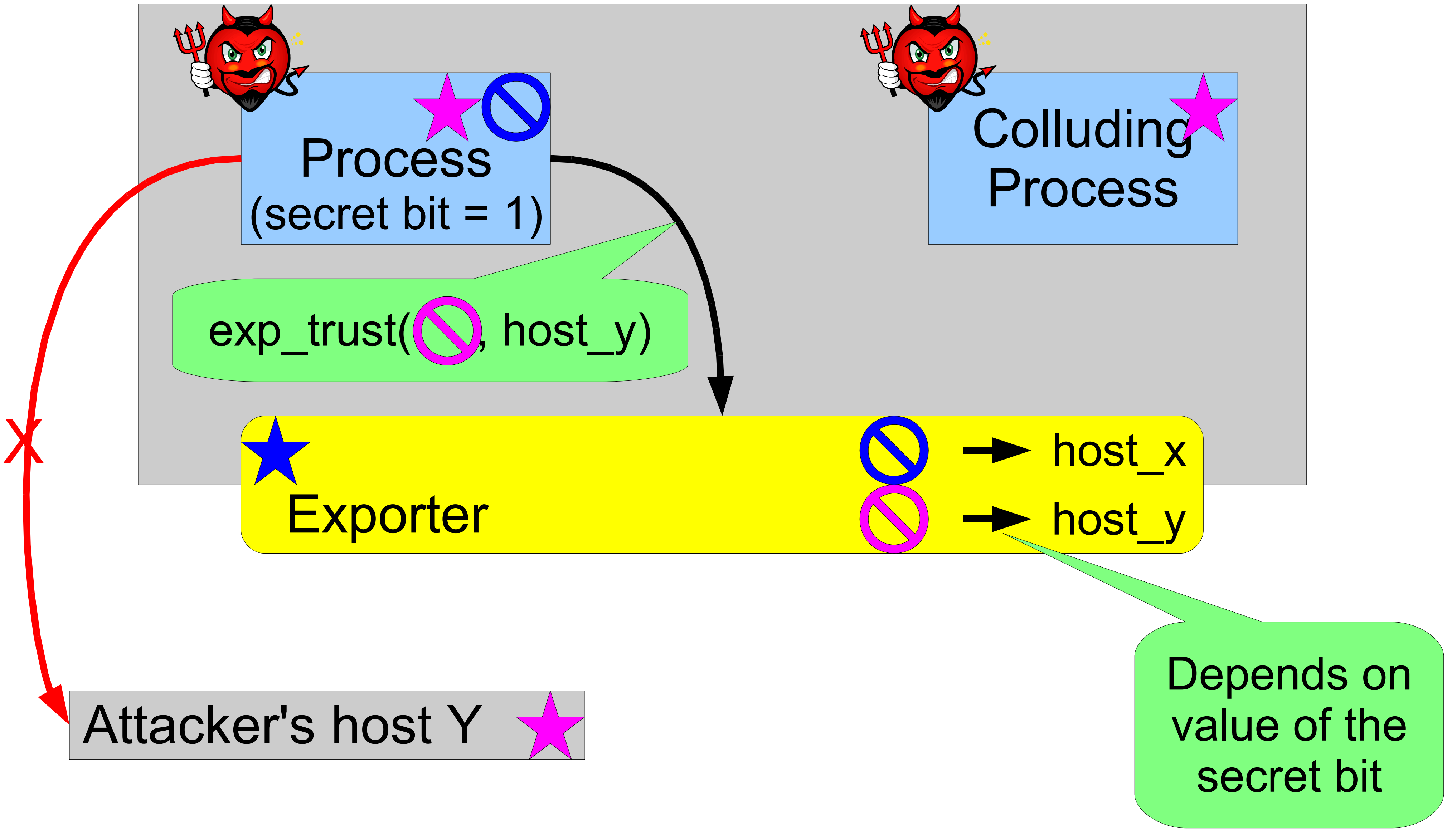


- Exporter sends no queries that could leak data

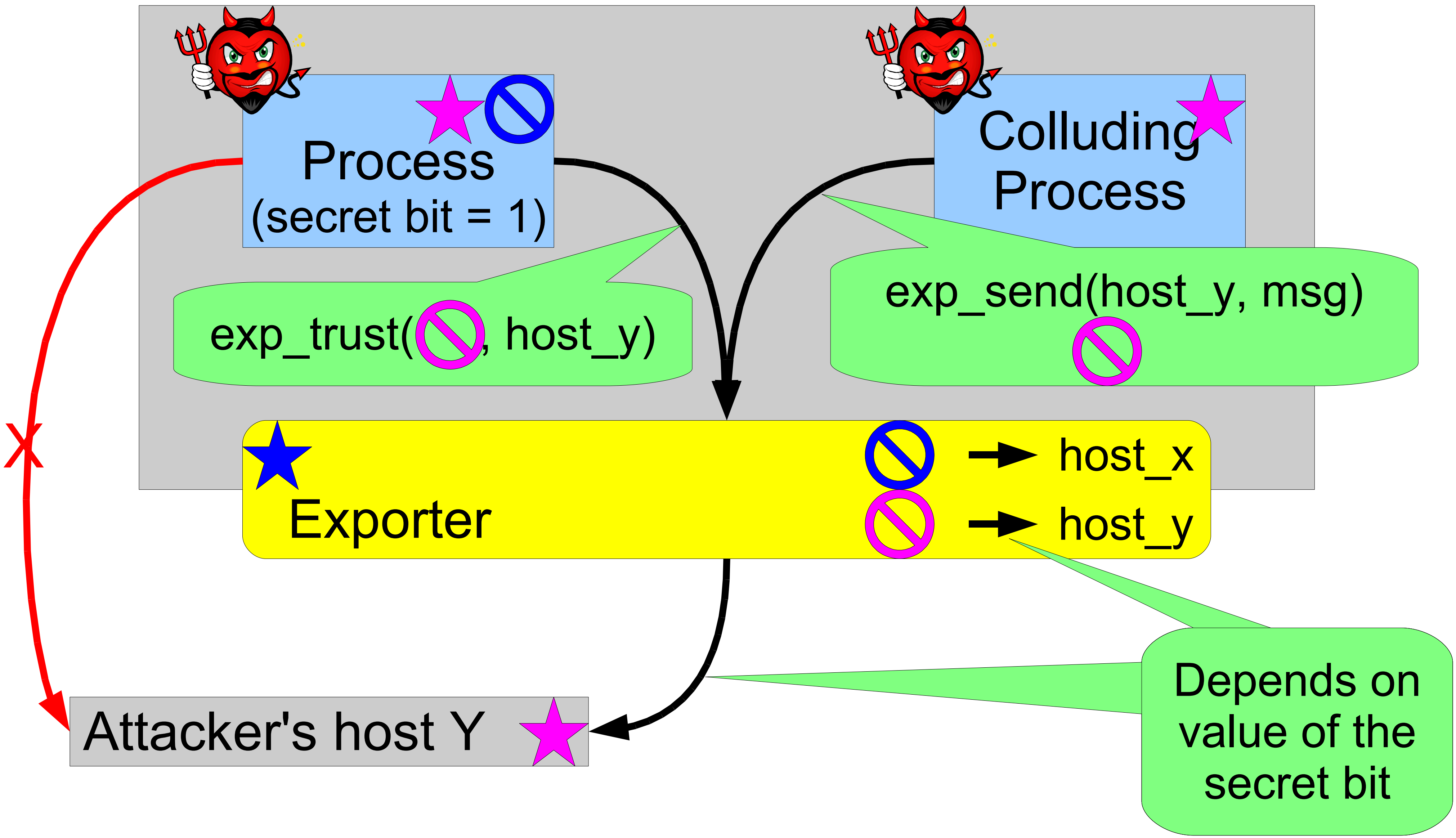
Storing trust in exporter also creates a covert channel in API



Storing trust in exporter also creates a covert channel in API



Storing trust in exporter also creates a covert channel in API



Problem:
What to do with covert channels?

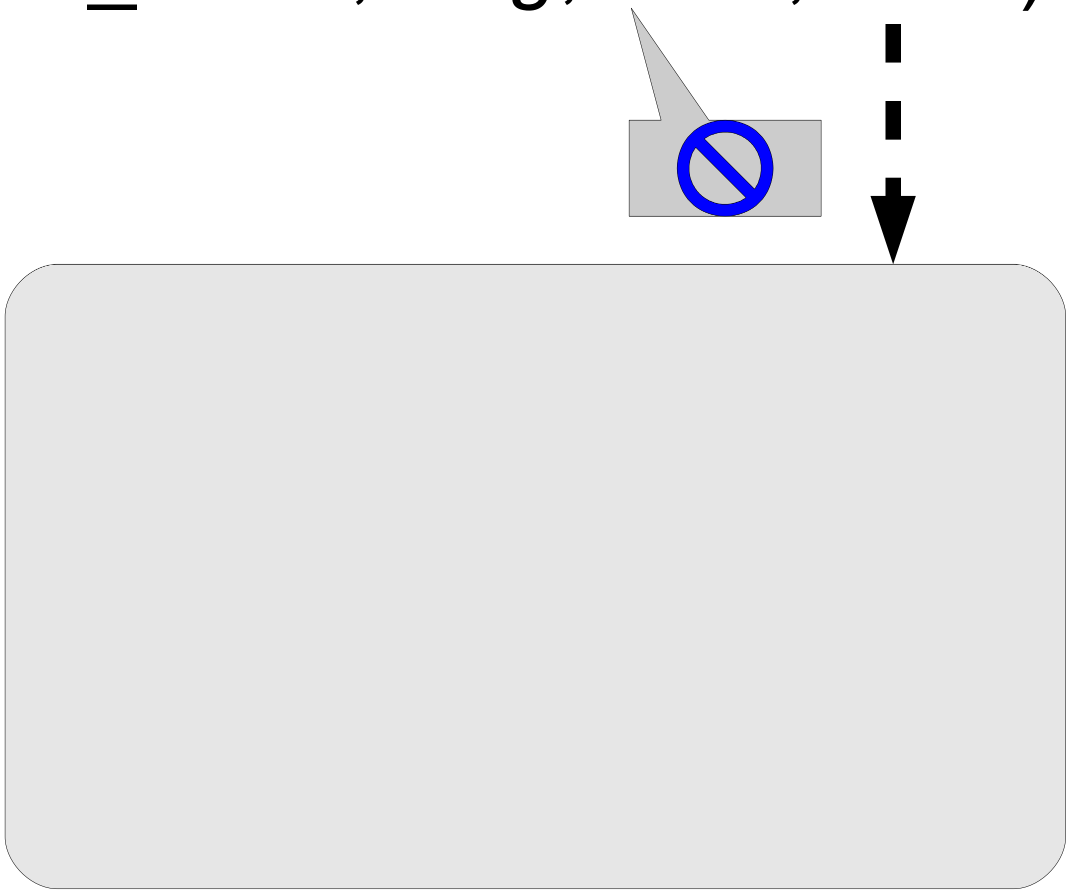
- Non-goal: eliminate all covert channels
  - Not practical
- Goal: avoid covert channels in interface
  - Allow trading off performance to mitigate covert channels without changing the API

Solution:
Self-certifying category names

- Categories named by public key
- Trust for a category defined by certificates signed by that category's private key
- Caller supplies all certificates to exp_send()

Caller supplies all certificates needed by exporter

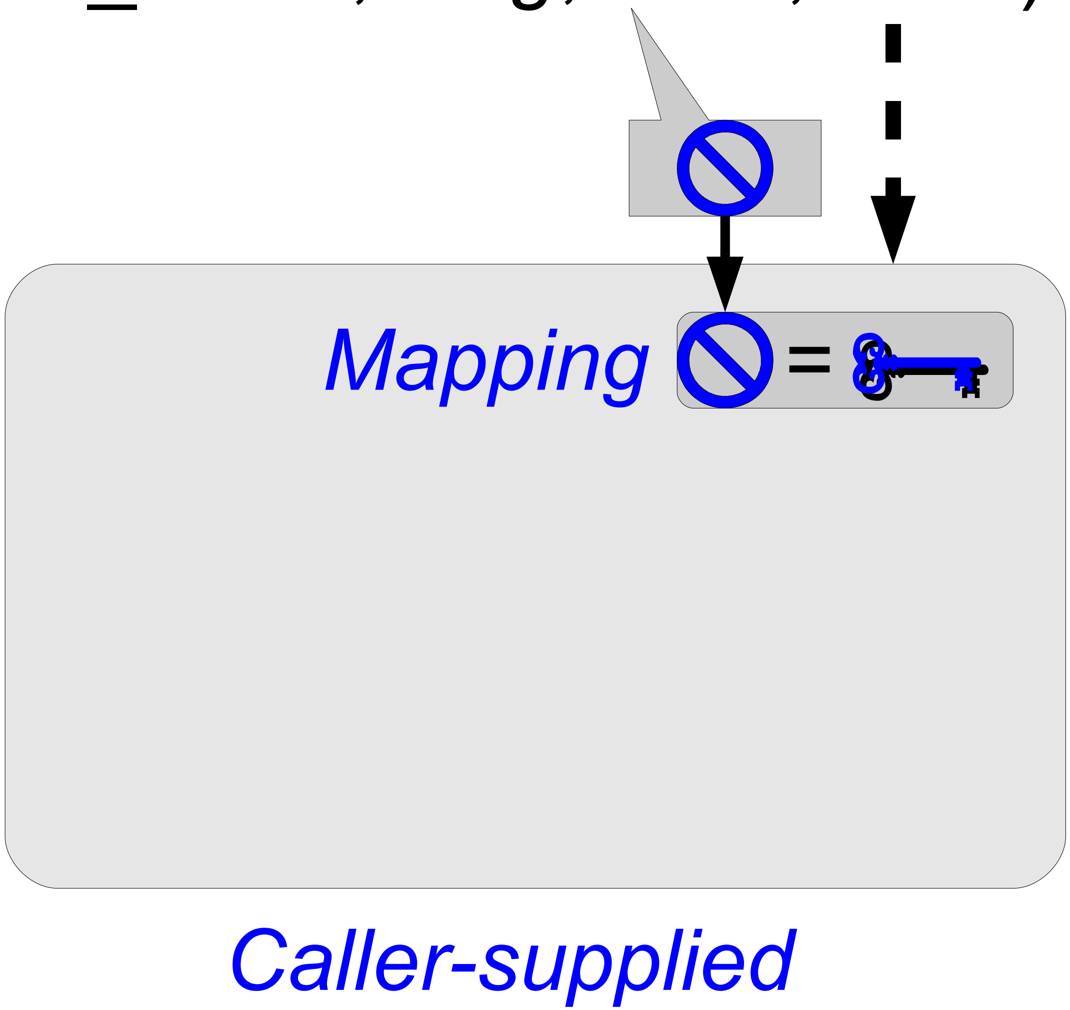
exp_send(*dest_host*, *dest_mbox*, *msg*, *label*, *certs*)



Caller-supplied

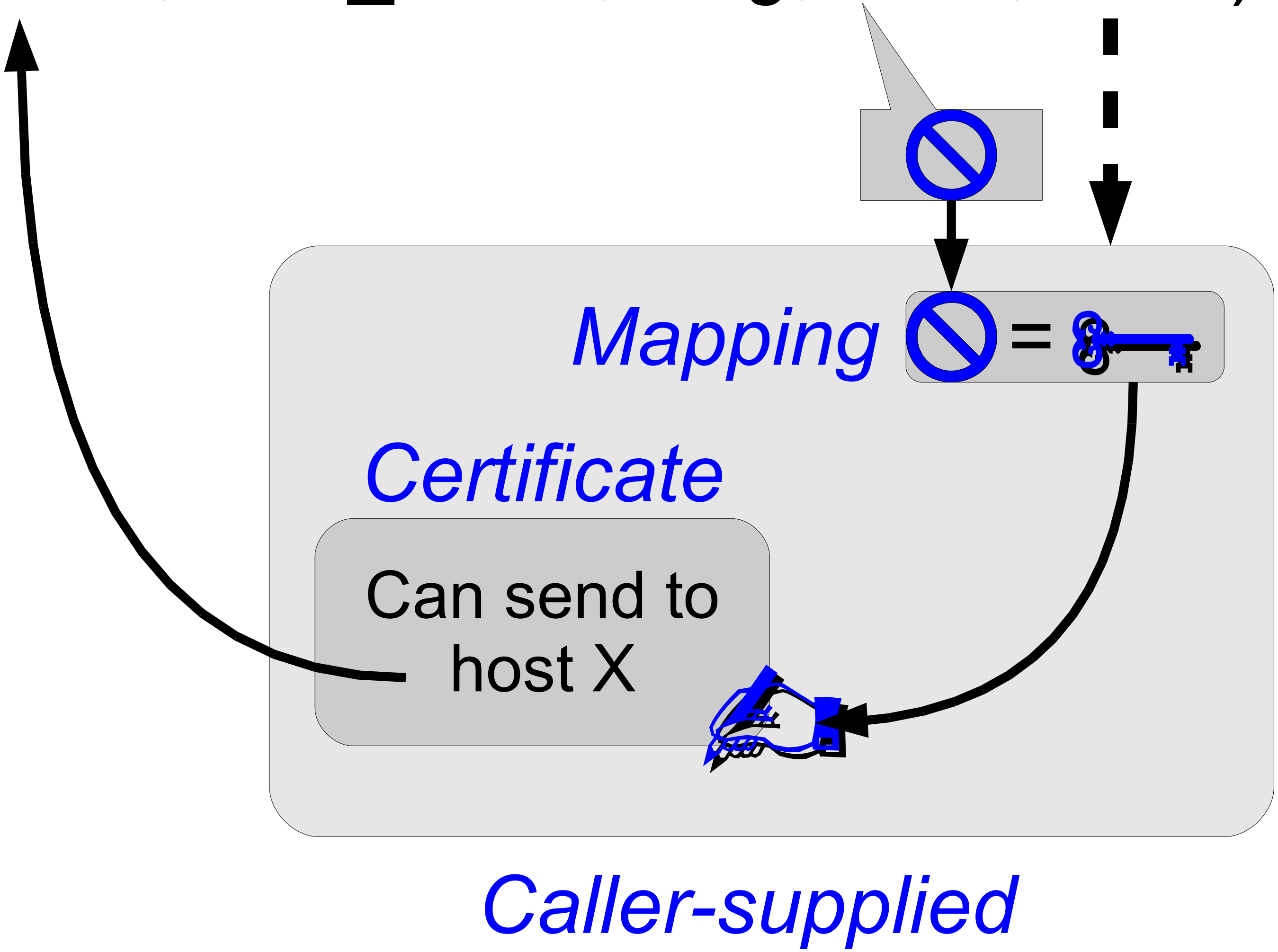
Caller supplies all certificates needed by exporter

exp_send(*dest_host*, *dest_mbox*, *msg*, *label*, *certs*)



Caller supplies all certificates needed by exporter

exp_send(*dest_host*, *dest_mbox*, *msg*, *label*, *certs*)



No covert channels to determine trust:

- No external communication
- No shared state

Exporter API design summary

- Self-certifying categories allow exporter to be stateless – just verify caller-supplied certificates
  - Stateless exporter design avoids covert channels
- exp_send() sends labeled datagrams
  - Also allows granting ownership (stars) across network
  - By design, only depends on caller-supplied args!
- Small trusted exporter: 3,700 lines + libs (crypto)

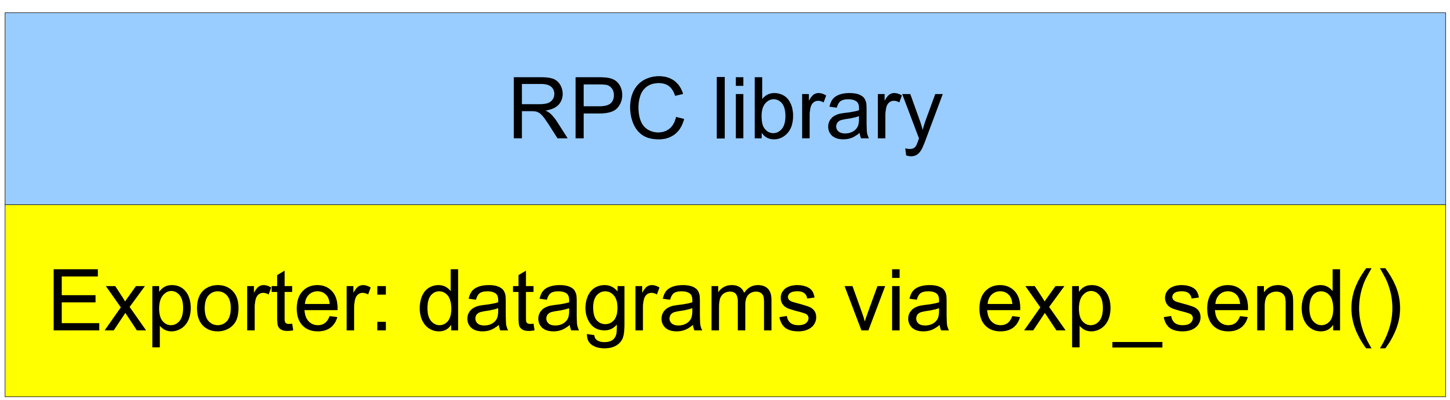
exp_send() enforces security policies specified by labels

- Higher-level functionality will not be trusted

Exporter: datagrams via exp_send()

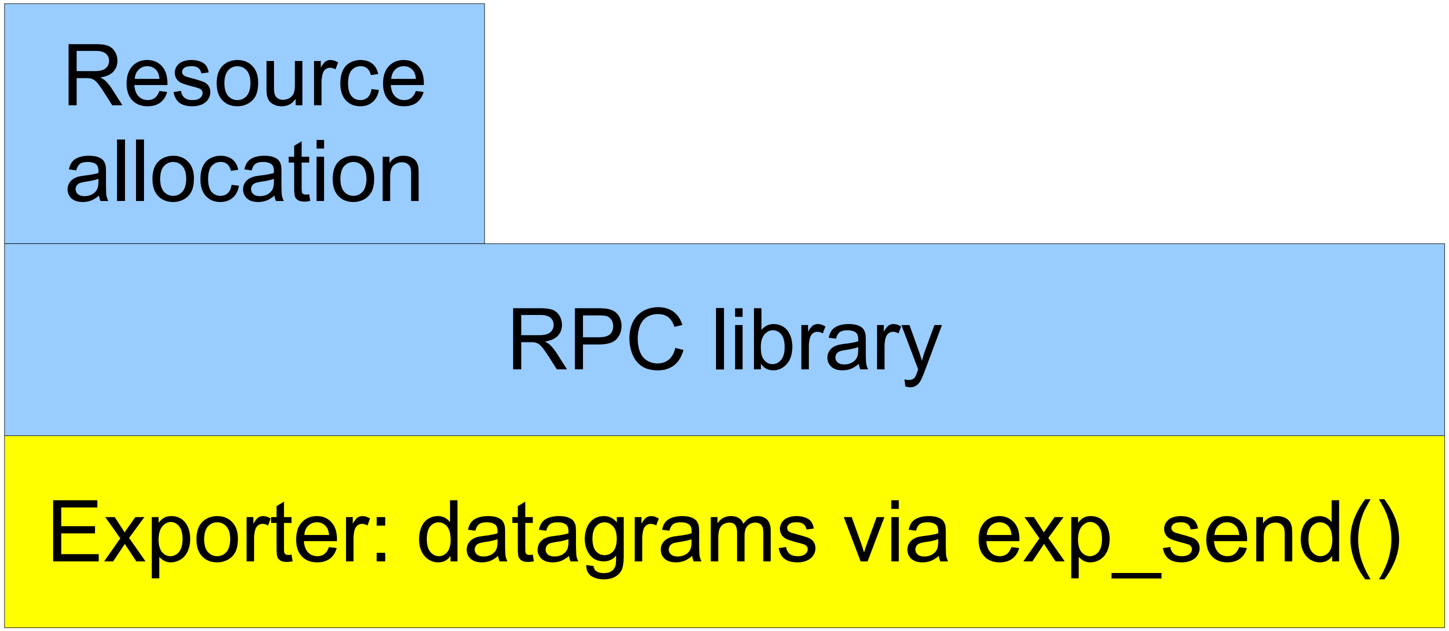
Building distributed applications on top of exp_send()

- RPC implemented on top of exp_send's datagrams, much like RPC over UDP



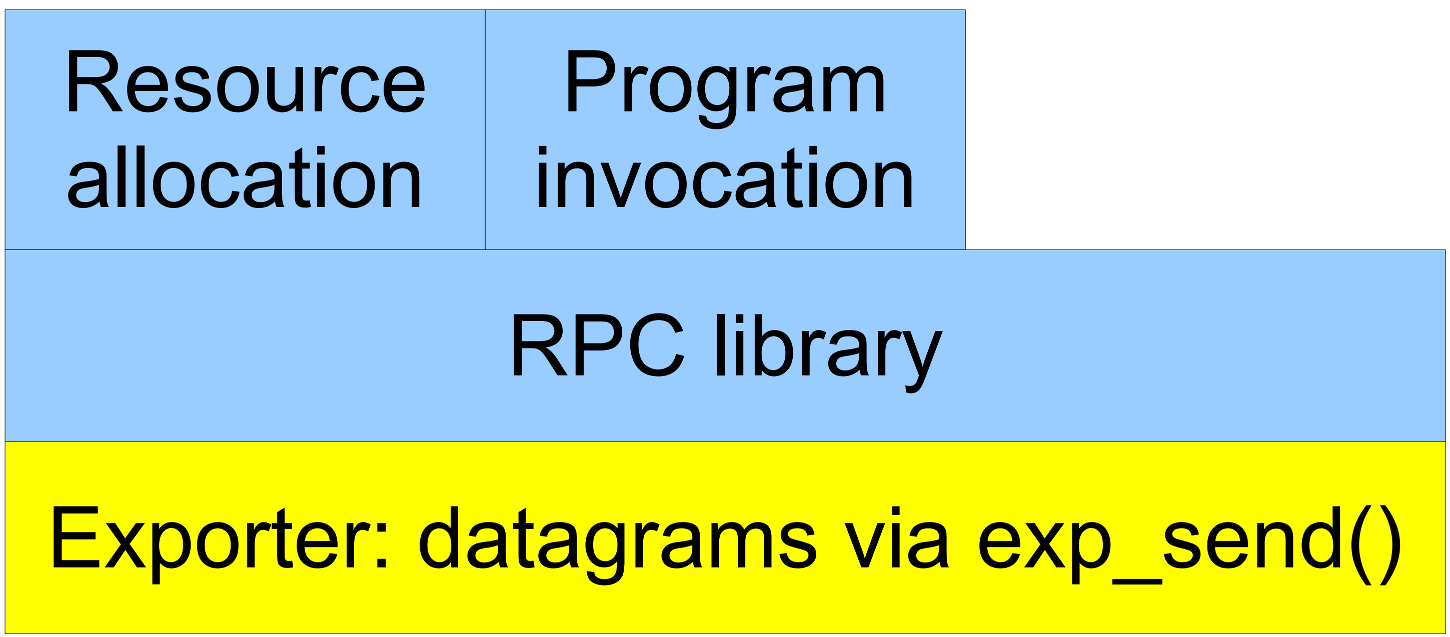
Building distributed applications on top of exp_send()

- Resource allocation RPC server (manages access to CPU, memory)

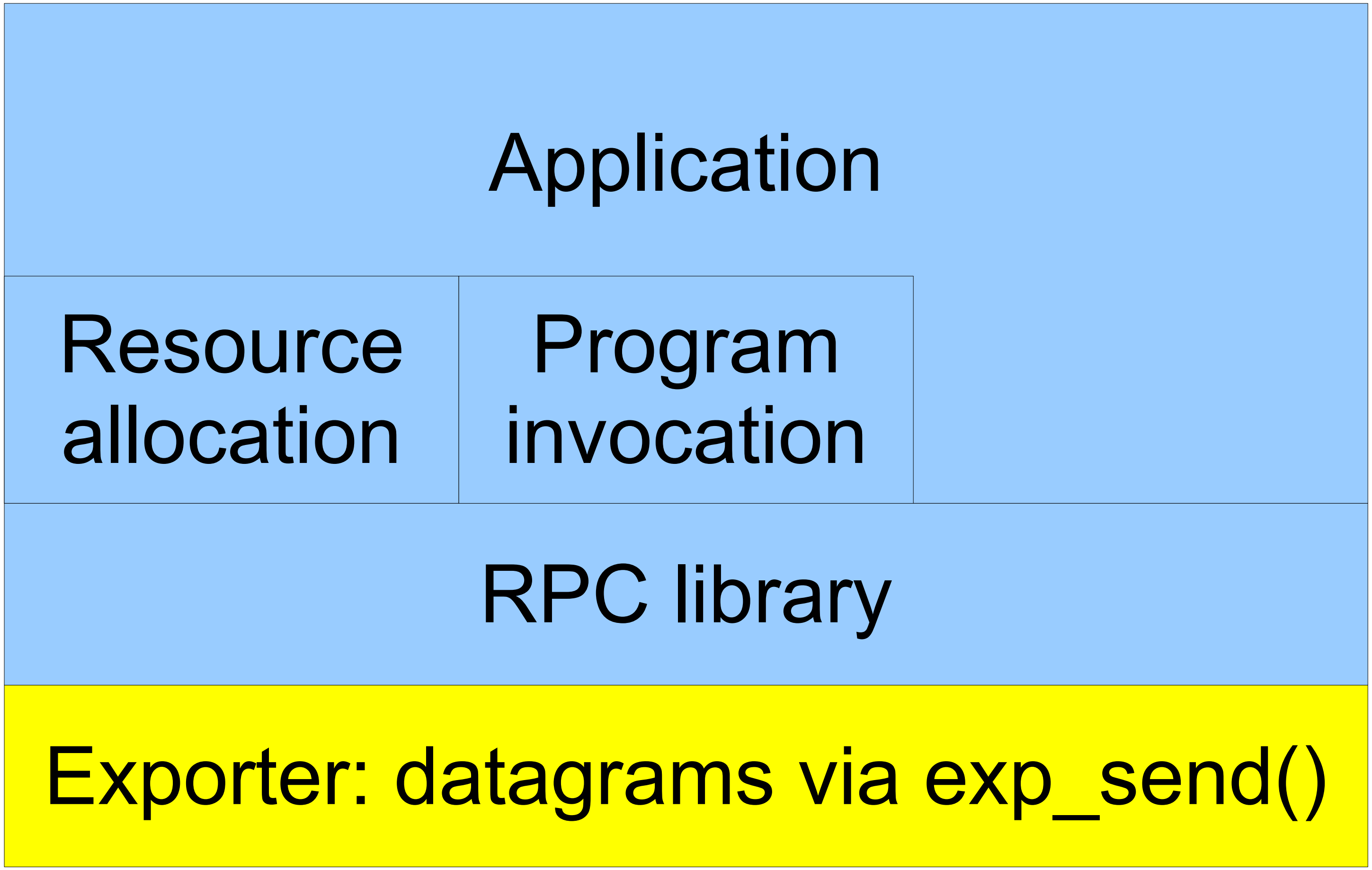


Building distributed applications on top of exp_send()

- Program invocation: starts a process using previously-allocated resources

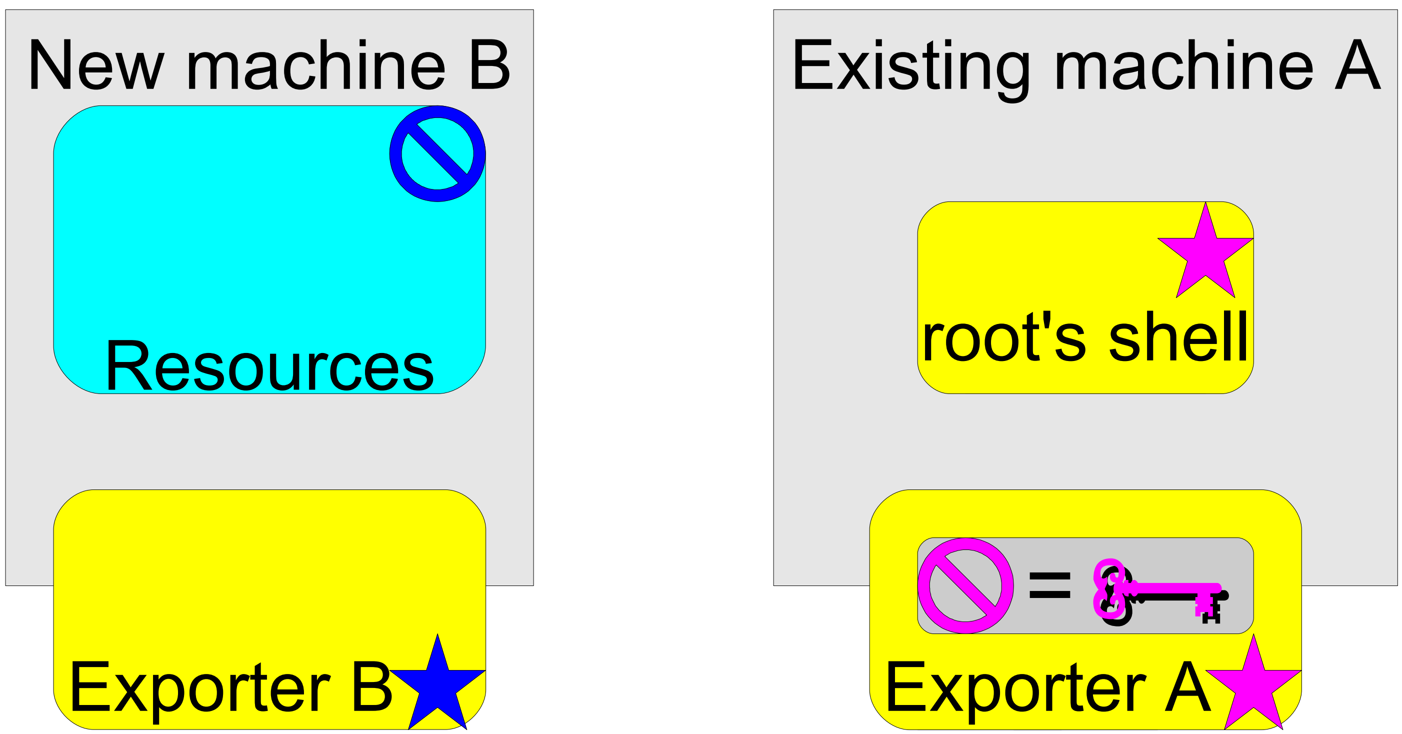


Building distributed applications on top of exp_send()



Bootstrapping a new machine

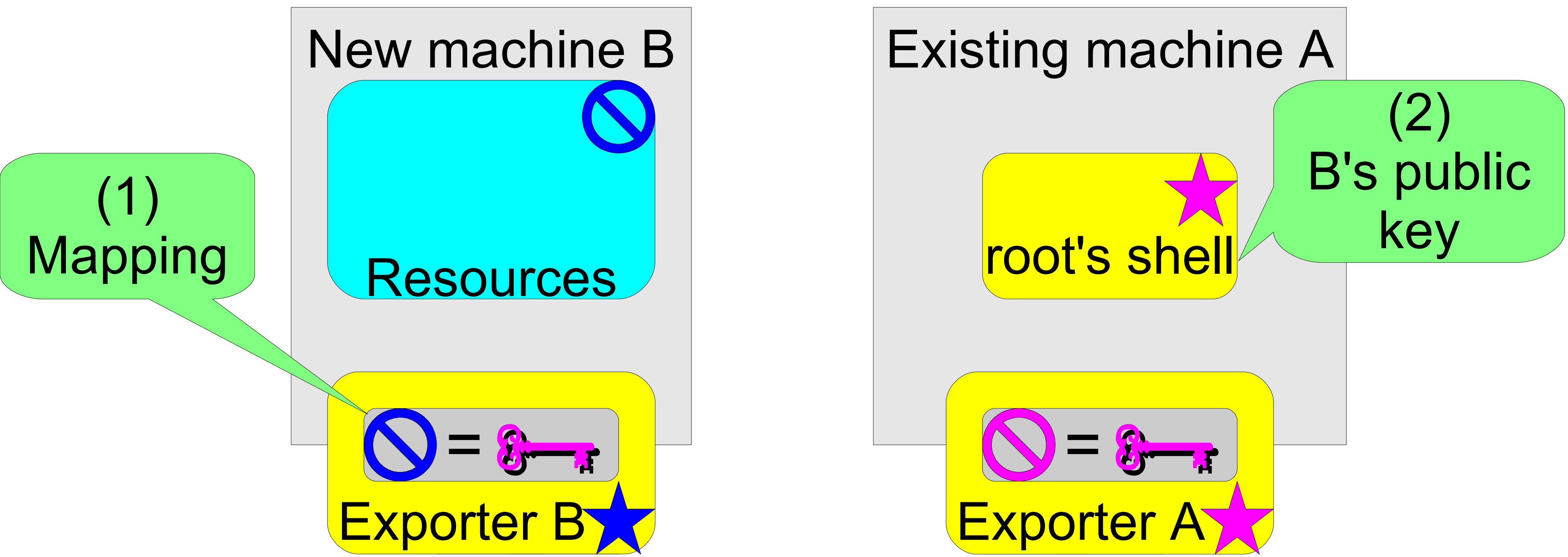
- Goal: gain access to new machine's resources using admin's privileges on existing machine



Bootstrapping a new machine

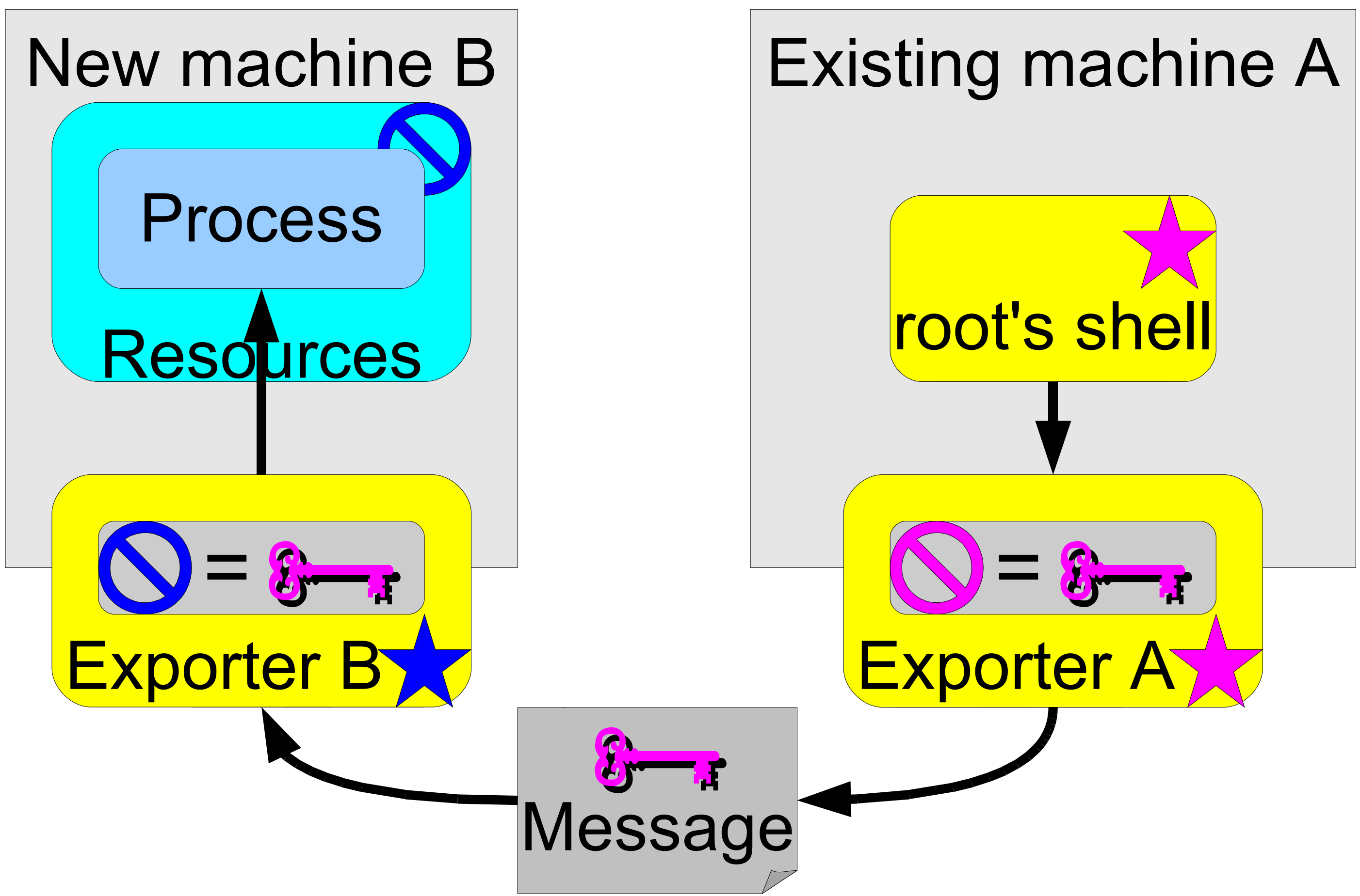
(1) Create mapping on new machine to bridge its protection domain with existing machine's

(2) Write down new machine's public key

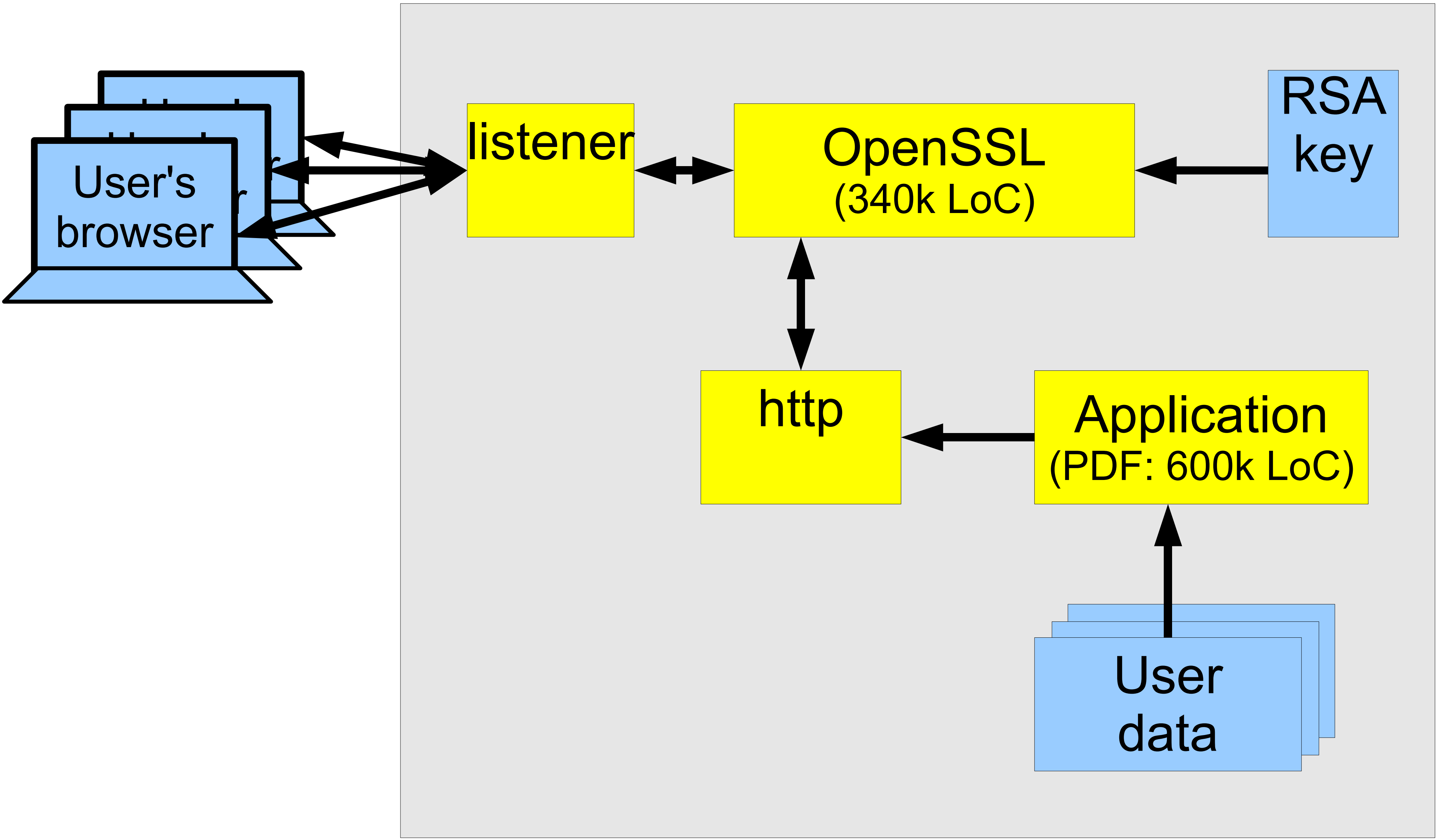


Bootstrapping a new machine

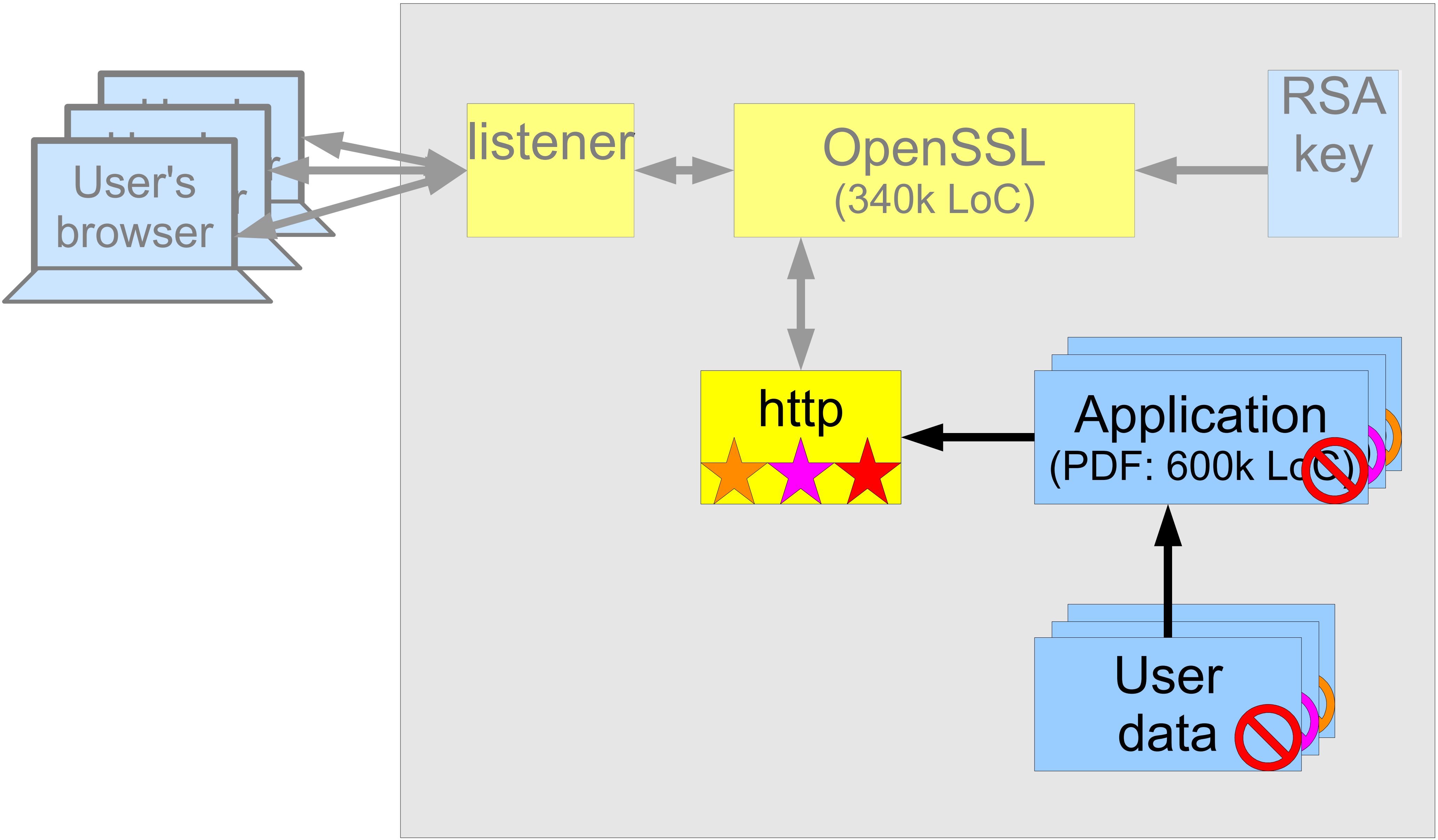
- Use process invocation and ownership of root's category to start running code on new machine



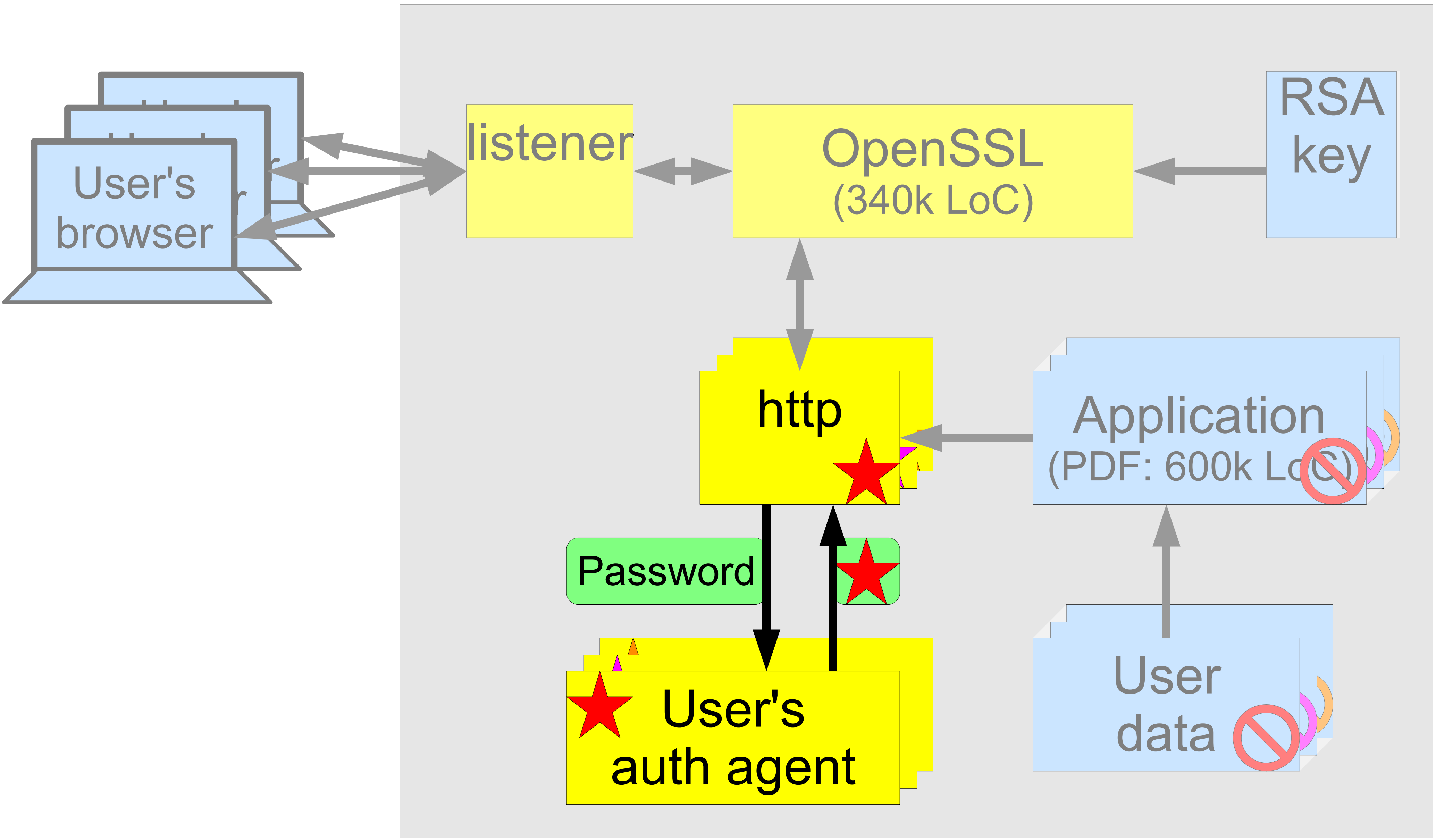
Traditional web server (like Apache): 1M+ lines of trusted code



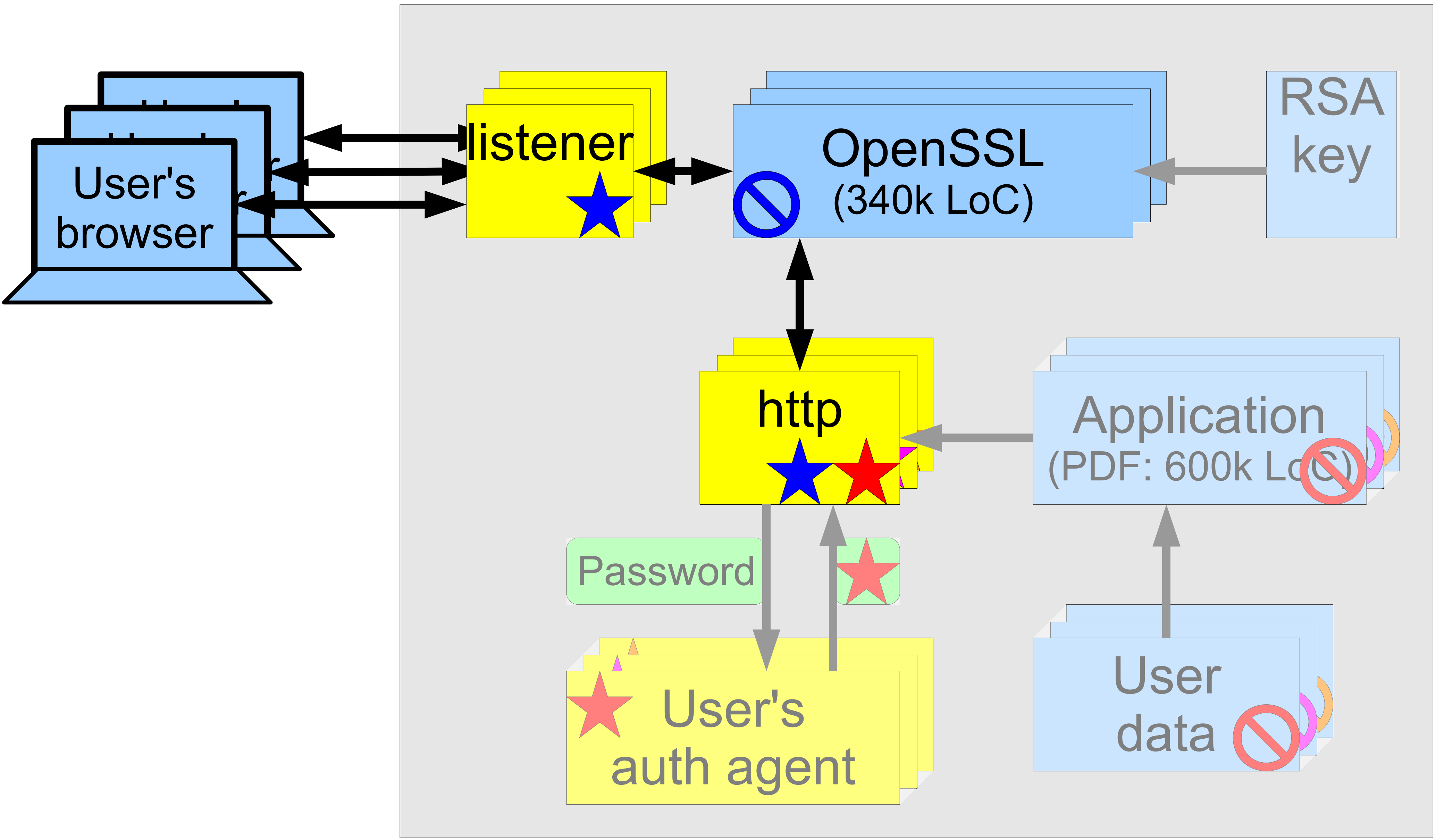
Application code cannot disclose user data



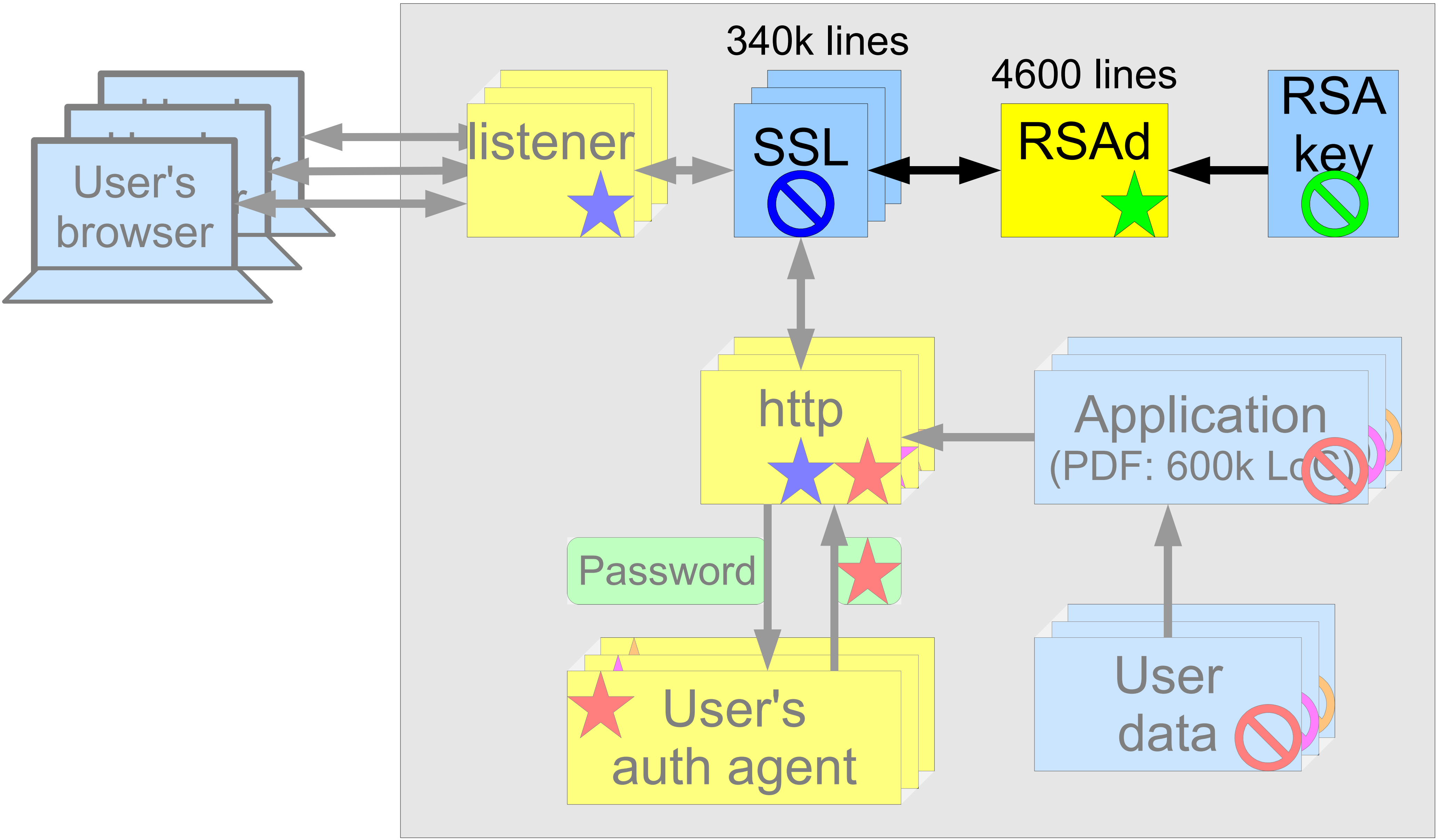
Per-user authentication agents, no fully-privileged code



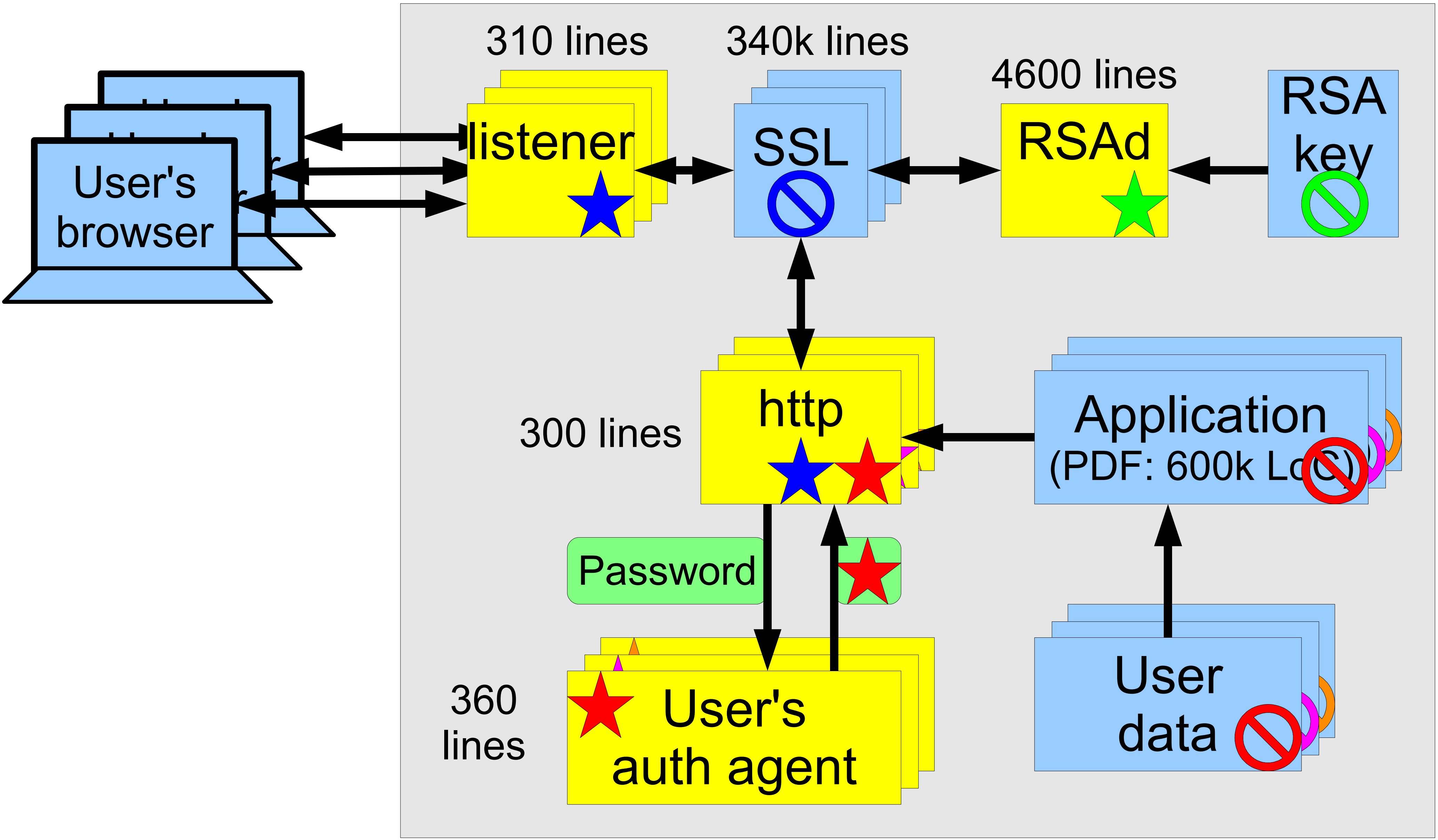
SSL library cannot send data to attacker



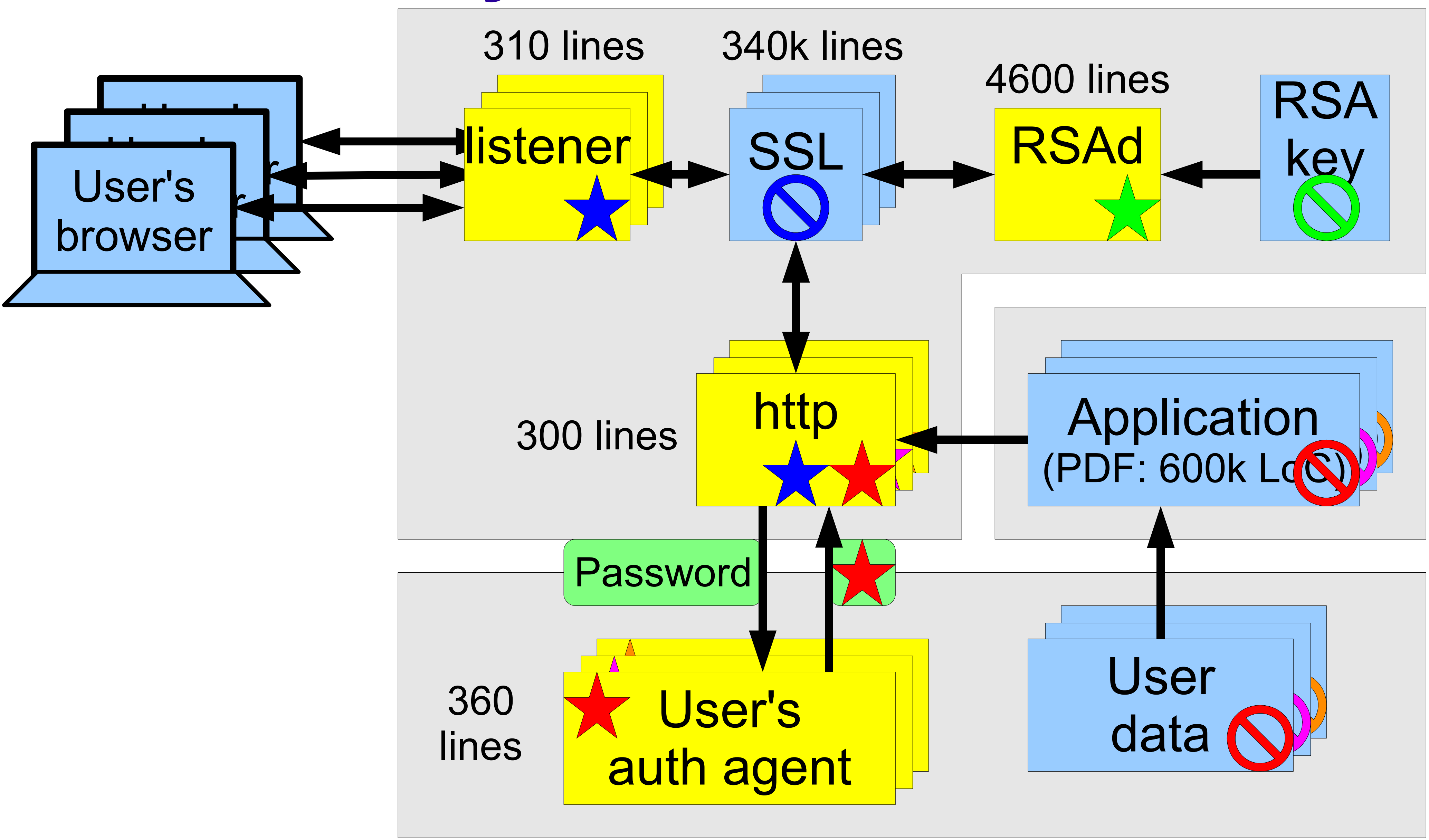
SSL library cannot disclose private key



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

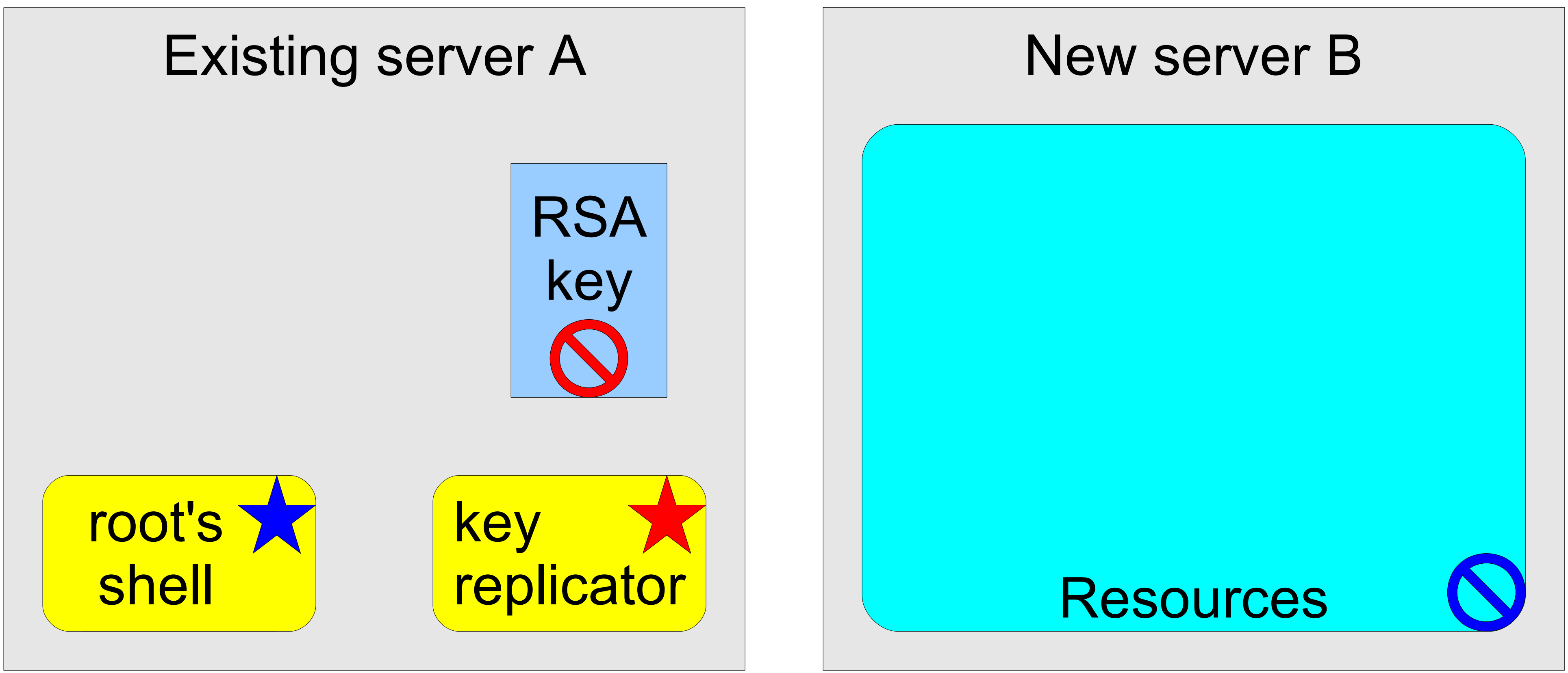


Scalable web server, no fully-trusted machines



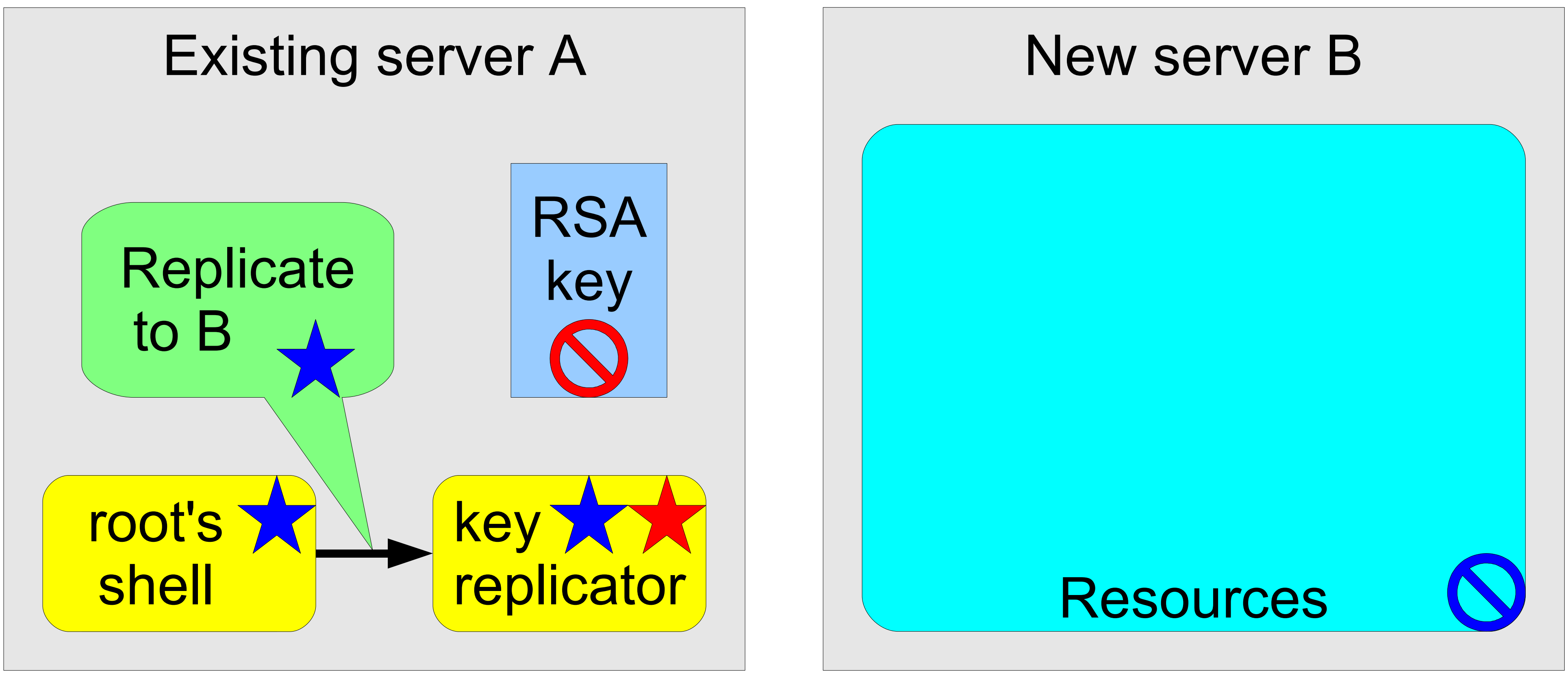
Replication

- Goal: ensure certificate private key is protected while minimizing trusted code



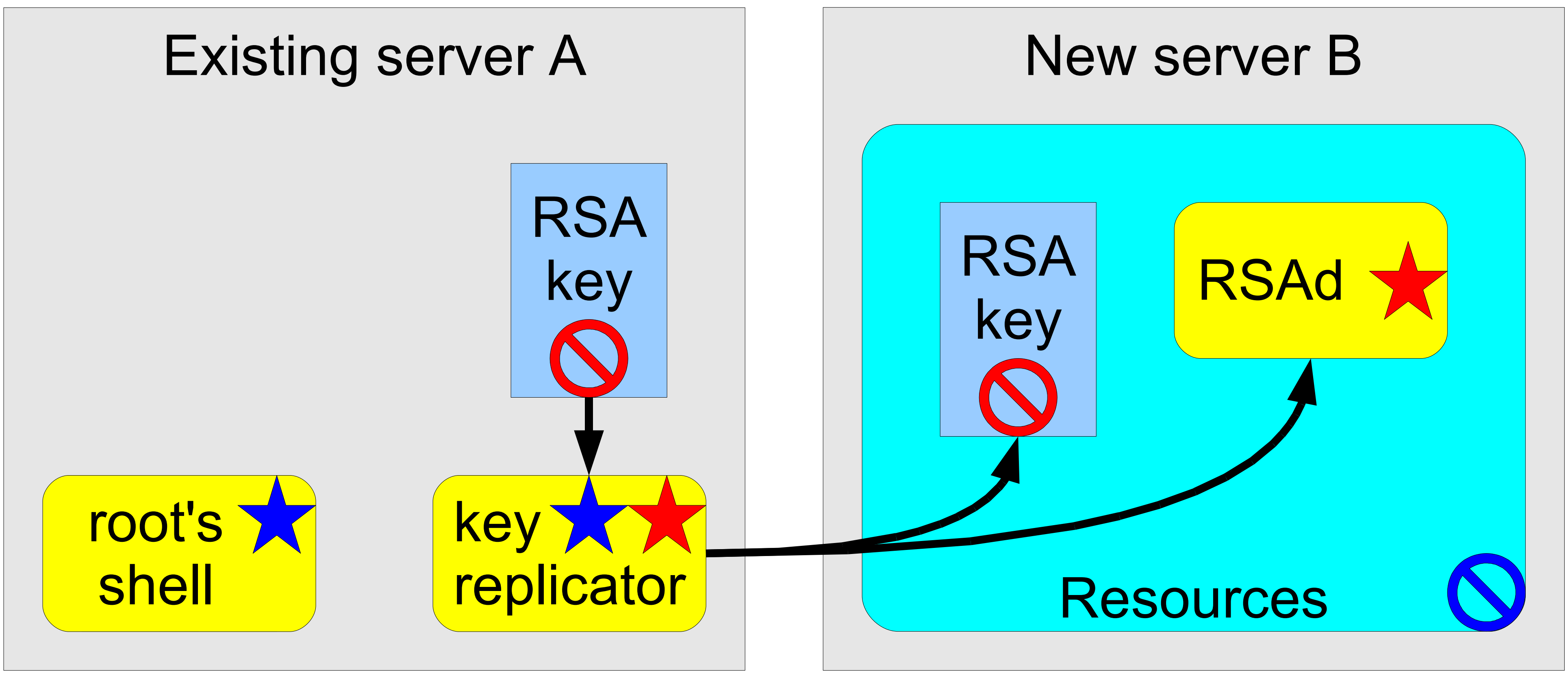
Replication

- Admin gives key replicator access to resources (blue star) and name (public key) of new server



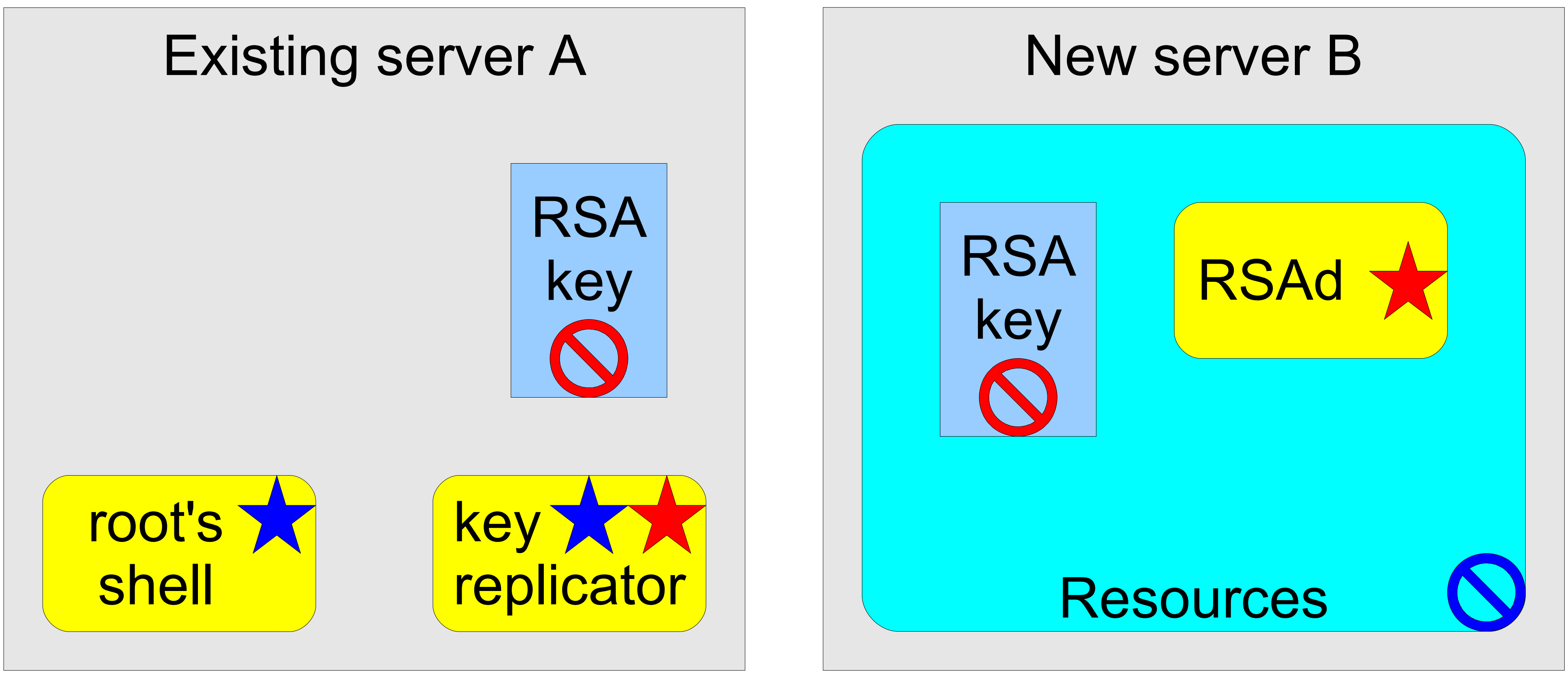
Replication

- Replication daemon sends over key and starts RSAd (using program invocation RPC service)

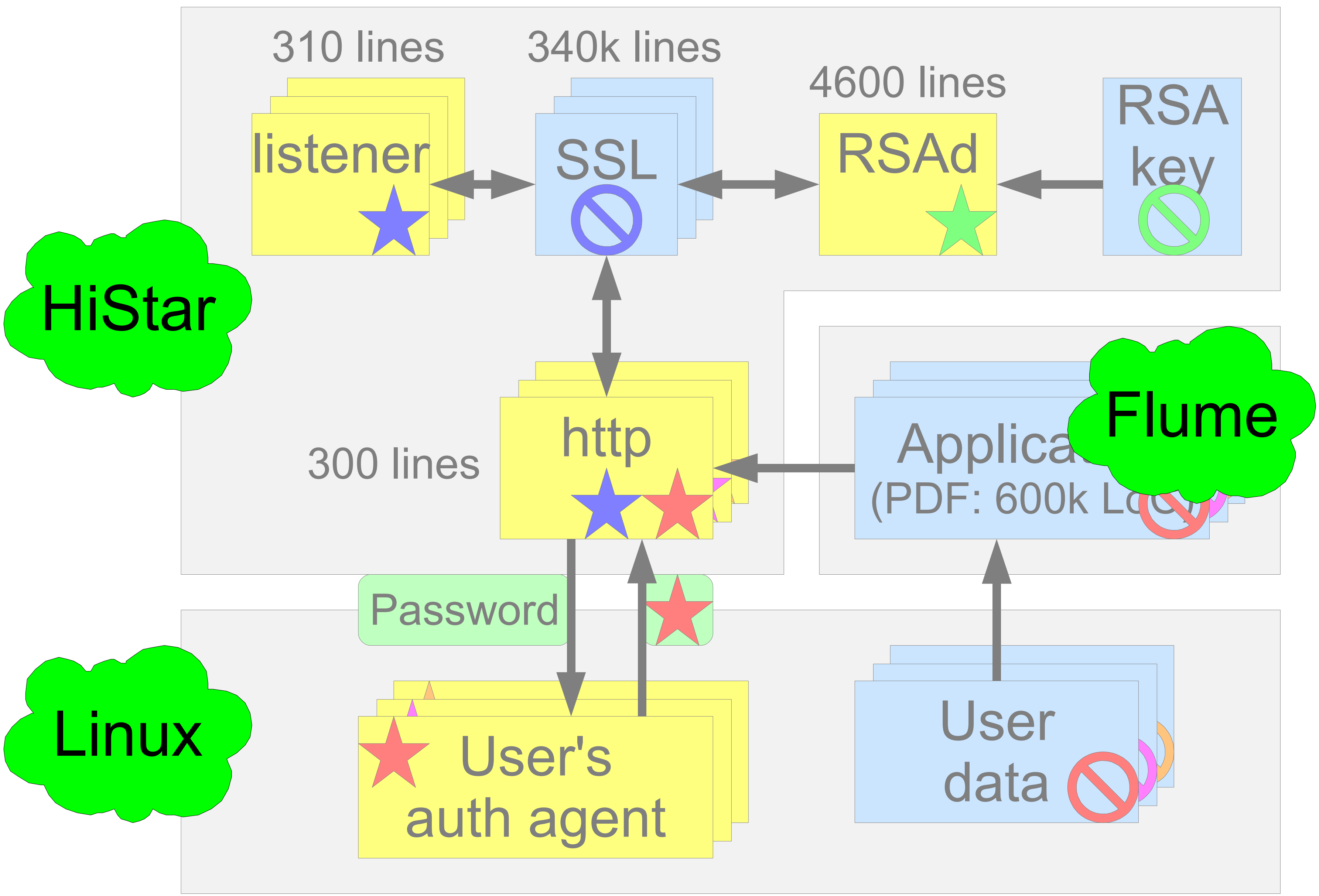


Replication

- Admin provides resources, but does not get access to RSA key itself

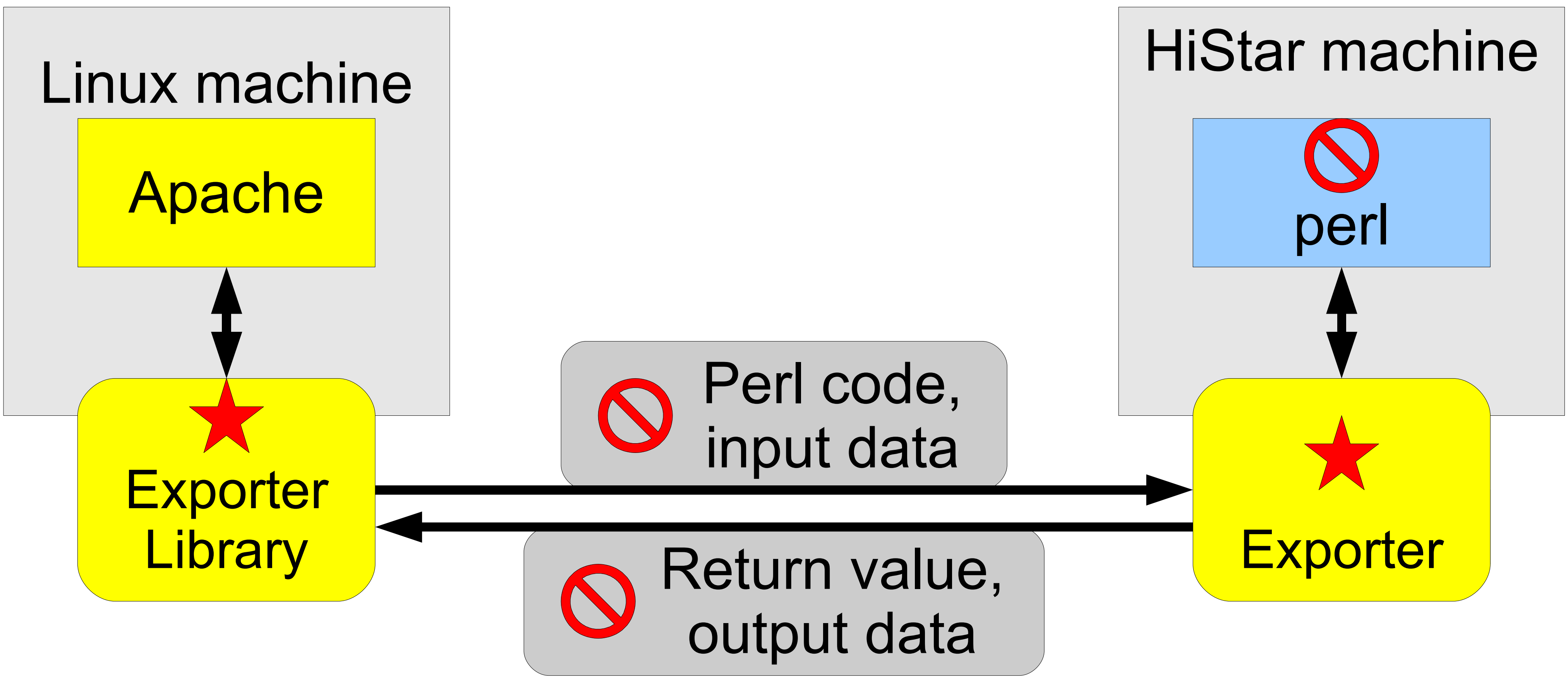


Network protocol works with multiple OS'es



Incremental deployment example: Run untrusted perl on HiStar

- Security policy specified by label
- Lower overhead, richer policies than VM/sandbox



Scaling untrusted app code to multiple compute clusters

- Extend the idea of untrusted application code to third-party compute clusters
- Earlier: untrusted app code handles user data
  - Limitation: had to use web site's trusted servers
  - Cannot mix Facebook+MySpace: no common server
- Now: users can explicitly trust compute clusters
  - Secure mash-ups can combine data from many sites
  - No need for fully-trusted common application platform

Summary

- Shown how to use information flow control for security in decentralized distributed systems

- Key idea: self-certifying category names
  → stateless checks
  → no implicit shared state
  → avoids covert channels in design

- Build everything on top of datagrams with IFC