ECE590-03
Enterprise Storage Architecture

Fall 2019

Security

Tyler Bletsch
Duke University

What this lecture contains

- Included:
  - Basic definitions
  - Fundamental cryptography primitives
  - Where cryptography can be used in enterprise storage
  - Access control models applicable to storage
  - Secure deletion
- Not included:
  - Cryptography internals
  - How to program using cryptography primitives (it's easy to screw up!)
  - The many other uses of cryptography
  - Database security (e.g. SQL injection attacks)
  - Intrusion detection and prevention systems
  - Software security (bugs and exploits, e.g. buffer overflow)
  - Denial of service attacks
  - Too many other things to ever possibly list

Key Security Concepts

Confidentiality

- Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information

Availability

- Ensuring timely and reliable access to and use of information

Integrity

- Guarding against improper information modification or destruction, including ensuring information nonrepudiation and authenticity

From Computer Security: Principles and Practices by William Stallings and Lawrie Brown

Threat model

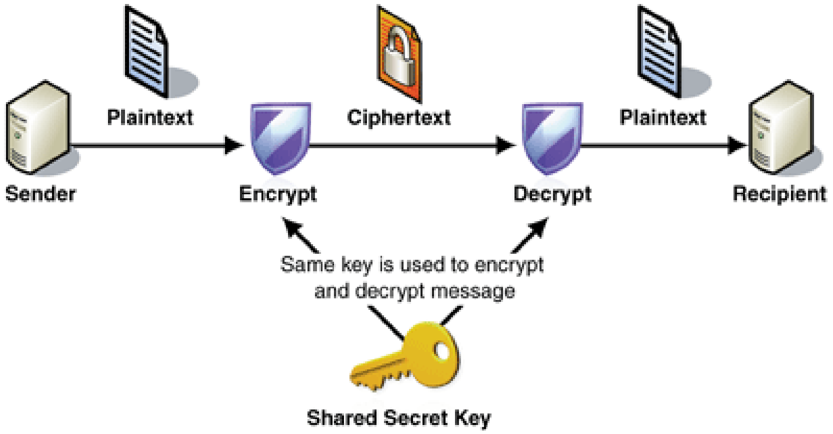
- Security is boolean:
  - If (ANY exploitable flaw exists): system can be compromised
    else: system cannot be compromised
- Can easily *prove* condition (existence proof);
  cannot easily *disprove* condition
- Result: Cannot determine if a system is secure
  - Scary/sad result
- To reason about security, need to identify **threat model**
  - What do we assume potential attacker can do?
  - Then, in that situation, what consequences can we prevent?
- Example: "Assume attacker can listen on this wire. Normally, they can intercept user data, but we if we use encryption, then they cannot."

Cryptography primitives

Cryptography basics: Symmetric encryption

(Also called shared-key encryption or secret-key encryption)

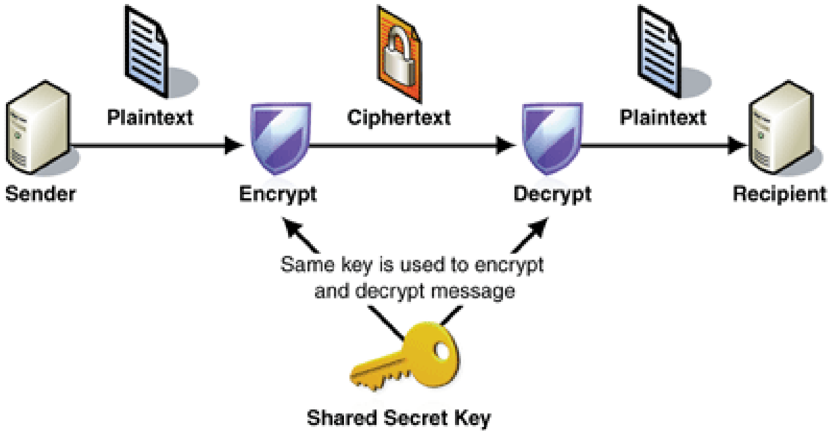
- Given:
  - Plaintext **p** (arbitrary size)
  - Secret key **k** (fixed size)
  - Encryption function **E**
  - Decryption function **D**
- Can produce ciphertext **c**:
  - **c = E(p,k)**
- Can recover plaintext:
  - **p = D(c,k)**



Cryptography basics: Symmetric encryption

- Ciphertext indistinguishable from random noise
- For a "good" algorithm, message cannot be recovered without key; attacker would need to try all possible keys
  - If k is big, that would take too long (longer than life of universe)
- Making a "good" algorithm is hard... a whole field of study
  - Never, ever make your own algorithm!
- Common algorithms: AES, Twofish, Serpent, Blowfish
  - If you're unsure, AES is a fine choice
    (unless these slides are old, then google it first...)

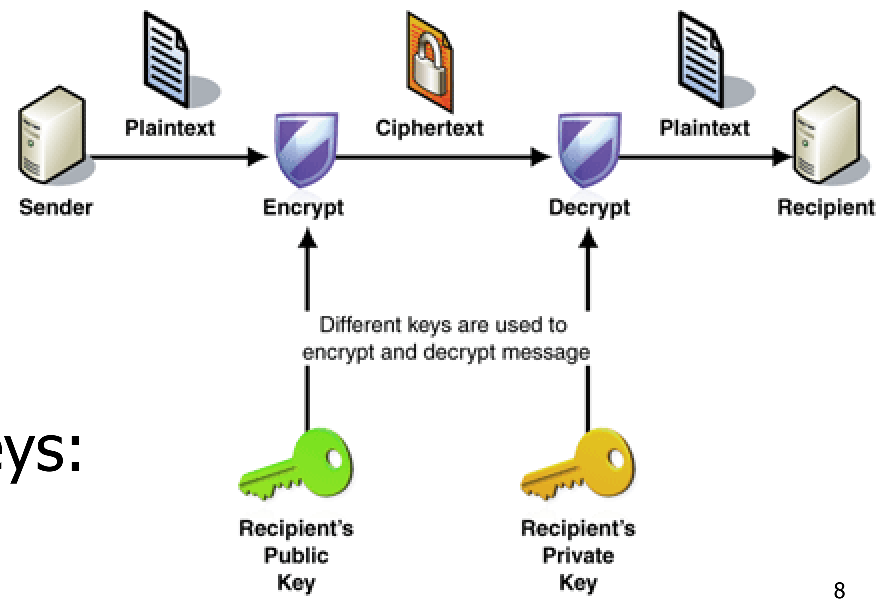
- **Problem with this?**
  - Need to pre-share the key!



Cryptography basics: Asymmetric encryption

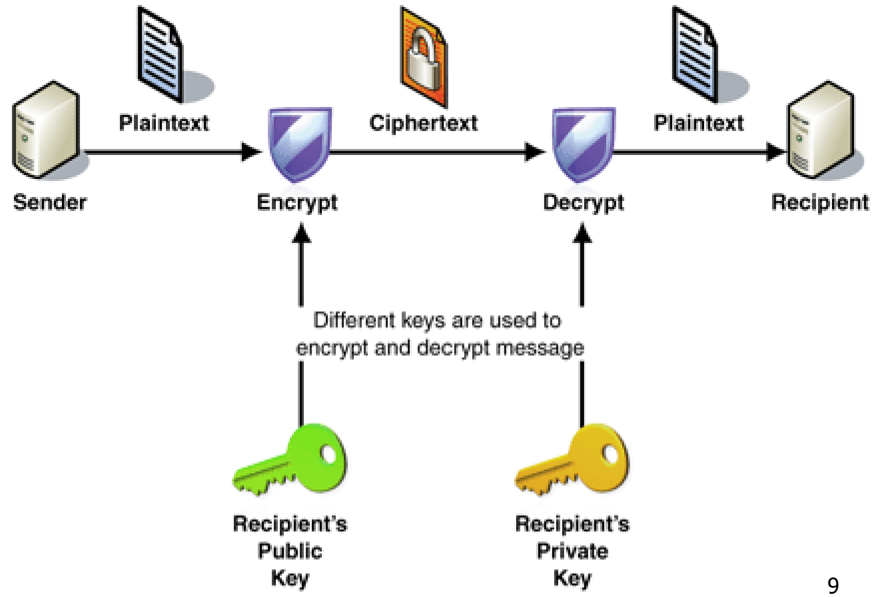
(Also called public-key encryption)

- Sender has:
  - Plaintext **p** (arbitrary size)
  - Recipient's public $\mathbf{k_{pub}}$ (fixed size)
    - Recipient makes this freely available (hence the name "public")
  - Encryption function **E**
  - Decryption function **D**
- Can produce ciphertext **c**:
  - $\mathbf{c = E(p,k_{pub})}$
- Can recover plaintext:
  - Need recipient private key $\mathbf{k_{priv}}$
    - Recipient keeps this hidden at all costs (hence the name "private")
  - $\mathbf{p = D(c,k_{priv})}$
- Also works if you reverse the keys:
  - $\mathbf{D(E(p,k_{priv}),k_{pub}) == p}$



Cryptography basics: Asymmetric encryption

- Public and private keys mathematically related, but one cannot be determined from the other
- Far slower than symmetric encryption
  - Common trick: Use asymmetric to send a secret key, then use symmetric with that key
- Common algorithms: RSA, Diffie-Hellman key exchange
  - If you're developing something with asymmetric encryption and you're using these slides as your reference, **stop**. You're doing it wrong.



Cryptography basics: Hashing

- You're already familiar with hashing (right?)
- Usual hash function properties:
  - Produces fixed size output for variable size input quickly (O(n))
  - Statistically, any output is as likely as any other

  ^ Good enough to make a hash table
- Additional requirements for cryptography:
  - **Irreversibility:** hash reveals absolutely <u>nothing</u> about input content
  - **Avalanche effect:** small input change will completely alter hash
  - **No collisions:** Big enough hash that collision probability is near-zero

  ^ Result: can't determine input from hash except by brute force
- Given message **p** and hash function **H**, get hash value **h**:
  - **h = H(p)**
- Common choices: ~~SHA-1~~, SHA-2, SHA-3, RIPEMD-160
  - Most lists include MD5, too, but MD5 was slightly broken in 1996 and badly broken in 2005! There's more detail than that, but to keep it simple: Don't use it!

Cryptography basics: Hashing to verify integrity

- Simple integrity check: send message **p** with **h=H(p)**
  - Recipient verifies that $\mathbf{H(p_{received}) = h}$
- Password verification: instead of password **p**, send **h=H(p)**
  - Receiver verifies that $\mathbf{h_{received}=h_{stored}}$
  - Advantage: Server doesn't store actual passwords, only hashes
  - *HEY YOU: never store passwords in plaintext! NEVER!*
    - *Best solution: use a key-derivation function like PBKDF2 that does it right for you!*
- Encryption by itself doesn't verify that the encrypted message isn't tampered with, so let's add hash verification:
  - Given message p, send **c=E(p,k)** and **h=H(p)**
  - Recipient verifies that **H(D(c,k)) = h**
- Can also combine with asymmetric encryption...

Cryptography basics: Electronic signatures

- Integrity verification mixed with asymmetric encryption

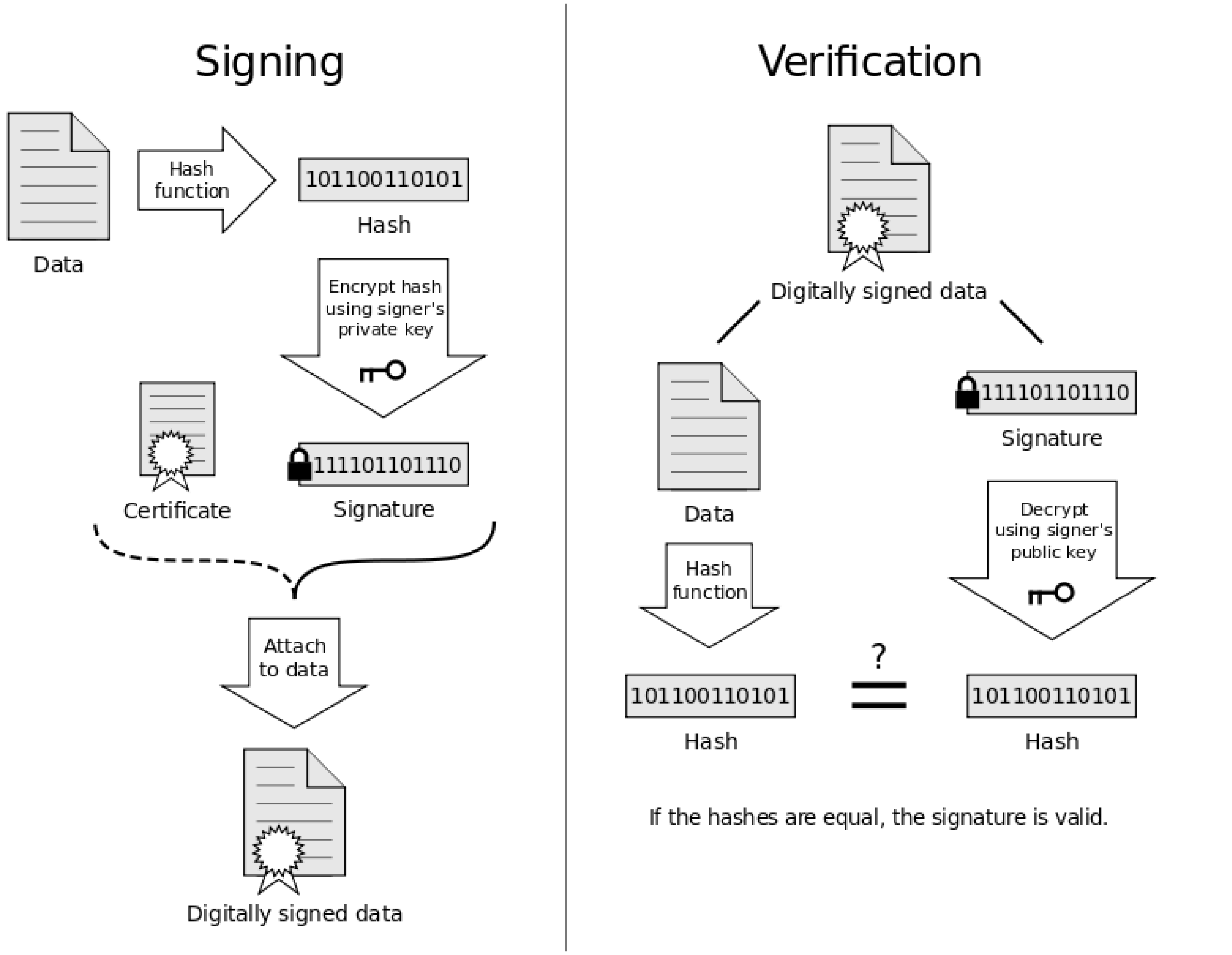
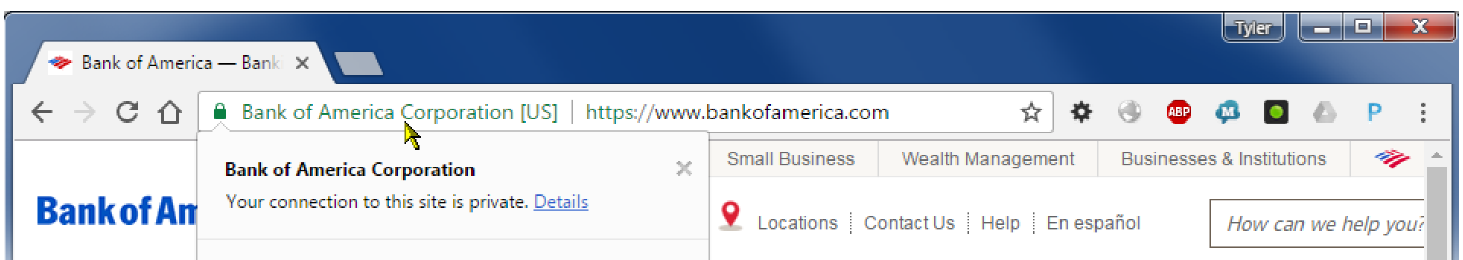


Figure from Wikipedia: Electronic signature

Cryptography basics: Web of trust

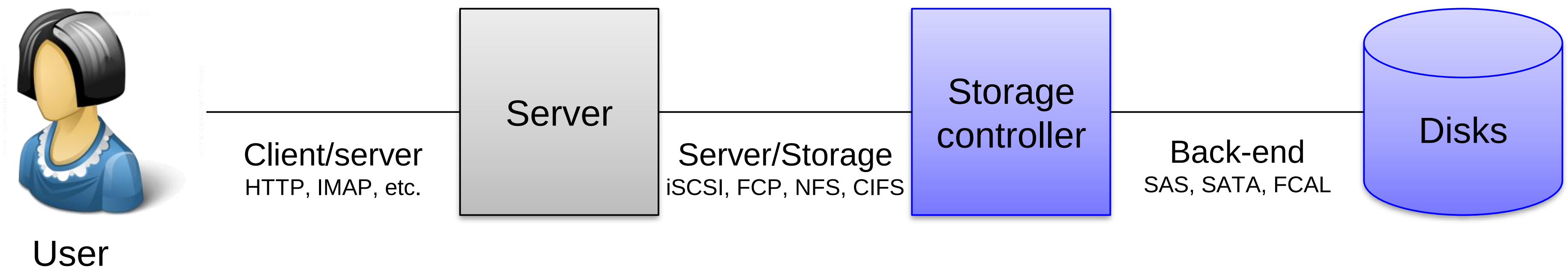
- "Web of trust" is a complex thing, here's the short version
- Using electronic signatures, you can "prove" you are the holder of a given private key
- We assume that a few certain keyholders are "trusted" enough to verify the identity of other keyholders
- The electronic signature that identifies someone in this manner is called a **certificate**.
- Example:
  - I go to Verisign and say (1) I'm Tyler Bletsch and (2) I own tylerbletsch.com.
  - They require documentation to prove this, then they electronically sign a certificate attesting to it.
  - Any browser that connects to tylerbletsch.com will automatically download and verify the certificate.



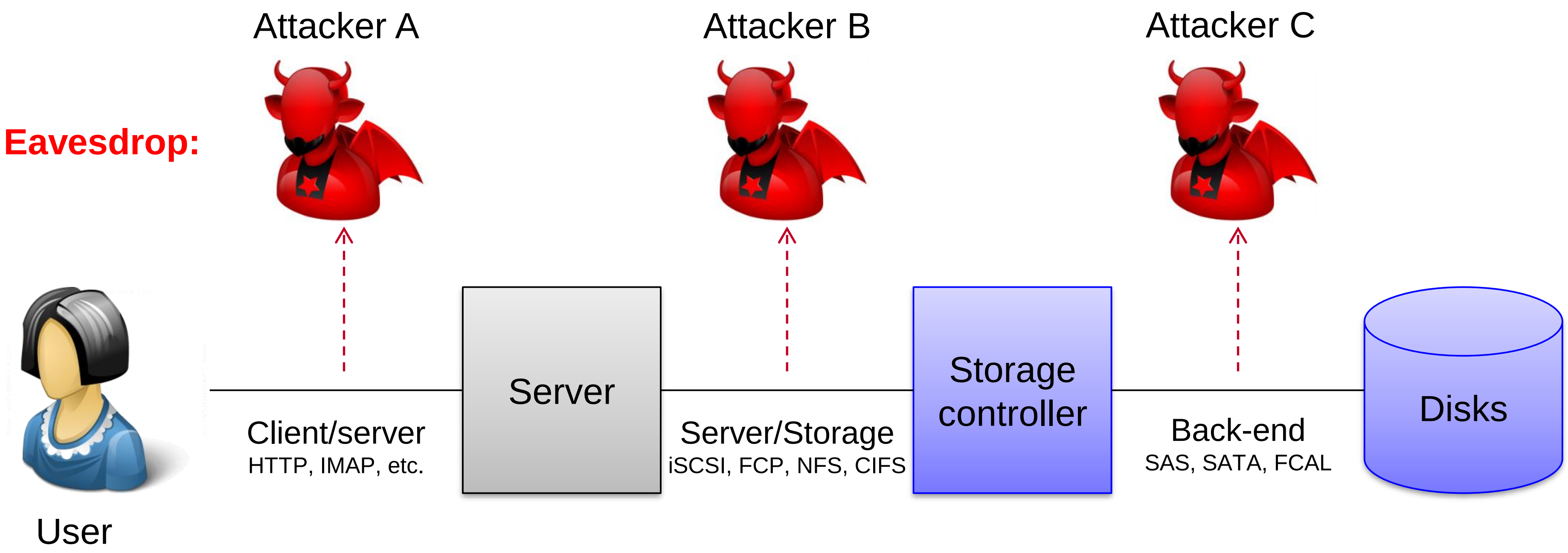
Applying cryptography to storage

Common threat models in storage

- A basic enterprise storage deployment.

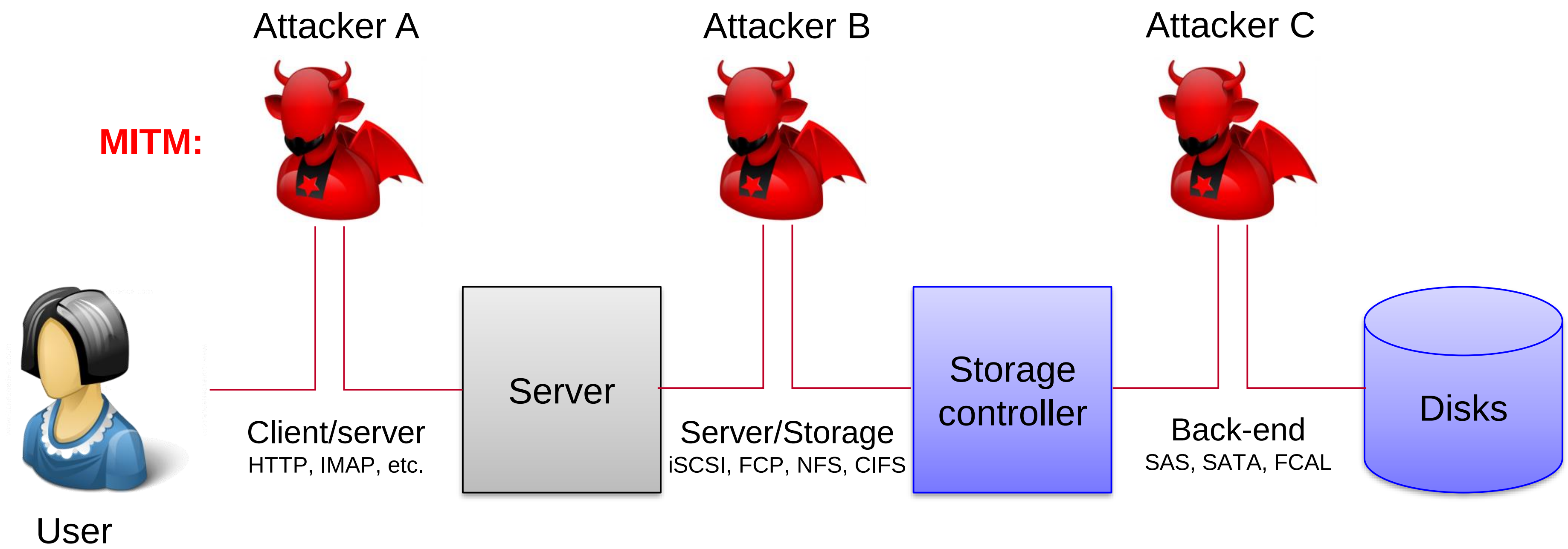


Common threat models in storage: Eavesdropping



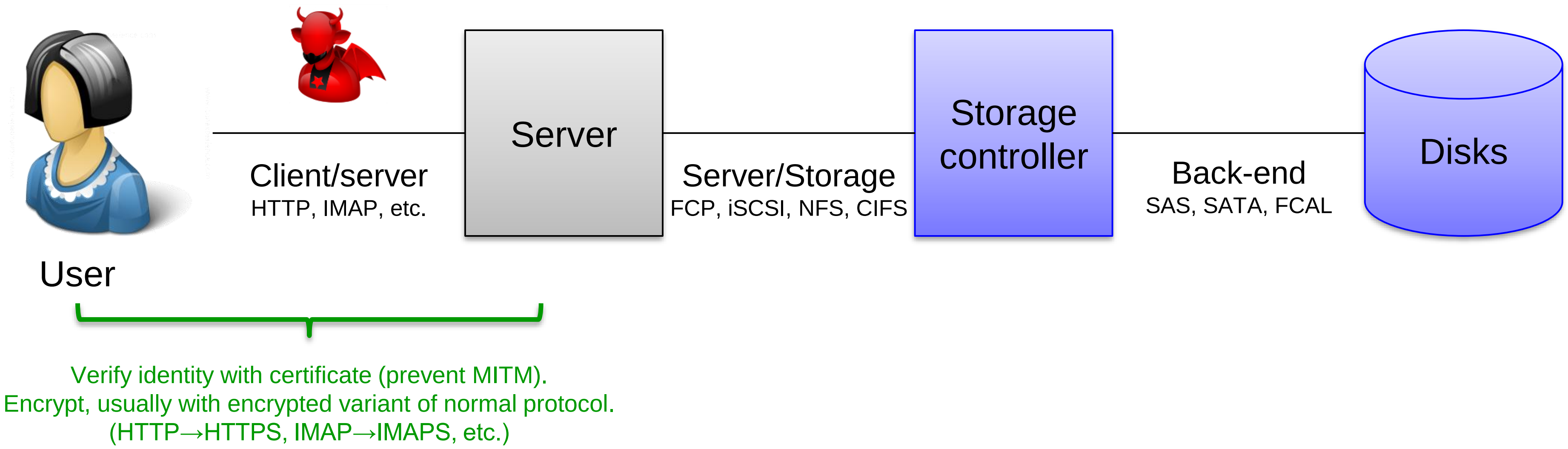
- **Eavesdrop**: attacker has a read-only tap on the wire. E.g.:
  - Physical access
  - Compromised user machine or maybe even server
    (in the case of compromised storage controller, we're dead no matter what, so we omit consideration of this case)
  - Network spoofing or compromised switch; configured to forward traffic

Common threat models in storage: Man-in-the-middle



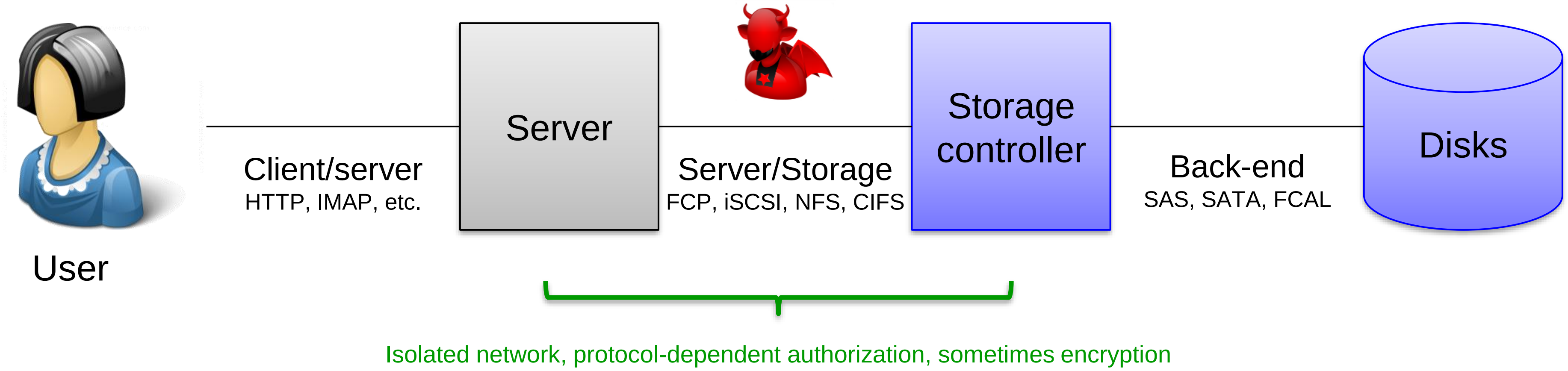
- **Man-in-the-middle**: attacker intercepts, can drop and spoof packets.
  - Similar attacks to gain this access; more visible to detection schemes

Securing the stack: client/server



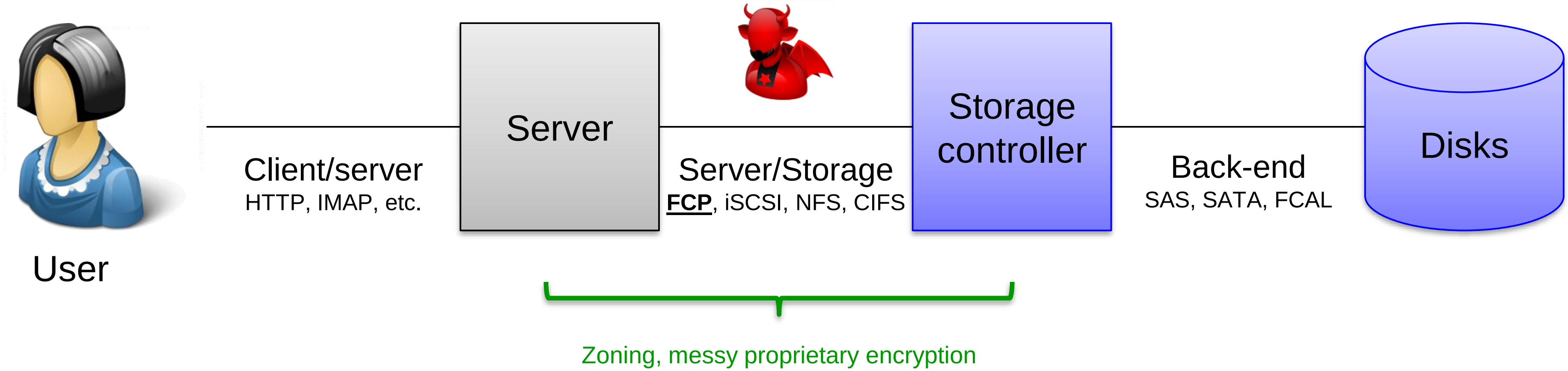
- Client/server security
  - A bit out of scope of this class
  - Basically, it's web-of-trust to verify identity, asymmetric key exchange to get a shared key, then symmetric crypto on the payload

Securing the stack: storage controller



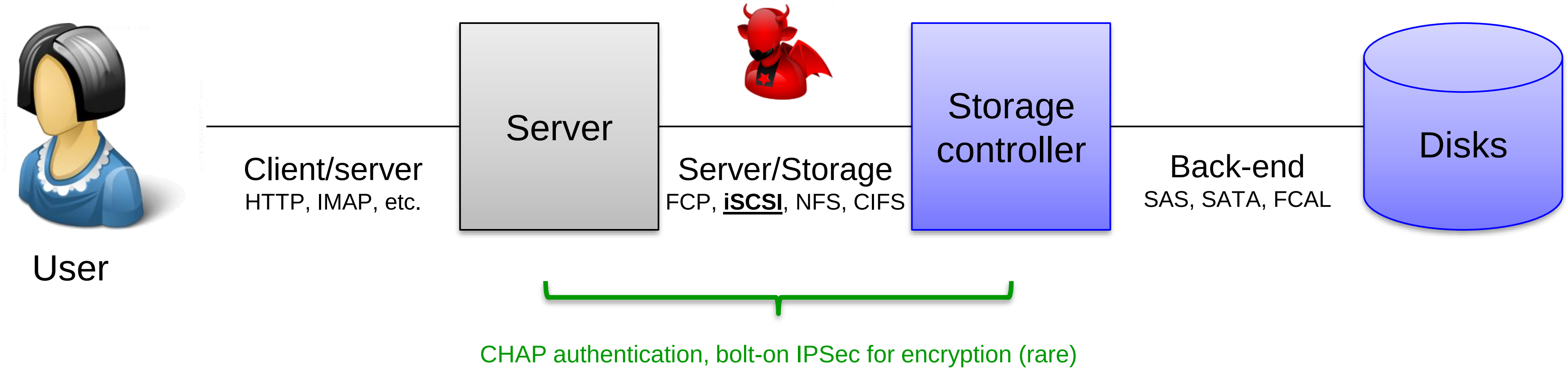
- Storage controller security <u>in general</u>
  - Sadly, it's kind of worse than the client/server link...
  - Primary defense: **isolated network**
    - Physical isolation (separate switches, "air gap") – expensive
    - Virtual isolation (VLANs) – cheaper, but configuration mistakes can break isolation
  - Other defenses are protocol-specific and...not...really.....good........

Securing the stack: storage controller FCP



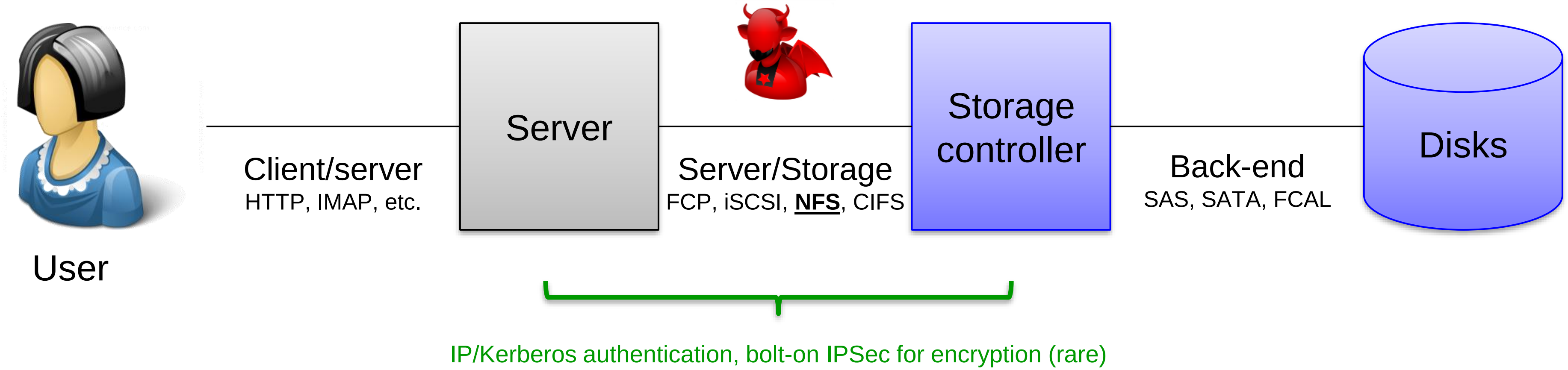
- Storage controller security: FCP
  - Identity verification: **Zoning and world-wide names**
    - Switch limits access based on names (no actual secrets)
    - If switch is secure and configured correctly, okay
    - If not, well, there are no secrets, so no security… (bad)
  - Encryption: **hahahahaha what a mess, good lord**
    - Lots of proprietary bolt-on products that claim FCP encryption
    - All are black-box mystery machines, leave a gap between the box and your controller

Securing the stack: storage controller iSCSI



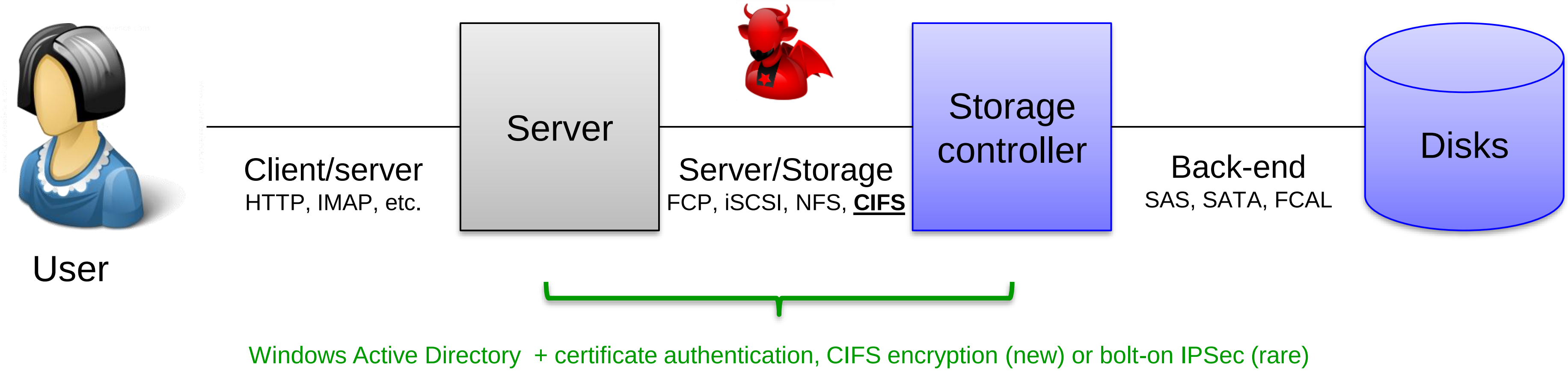
- Storage controller security: iSCSI
  - Identity verification: **CHAP protocol**
    - Basically it's hash-based password checking; fairly weak
  - Encryption (and also enhanced identity verification): **IPSec**
    - IPSec is a generic encryption layer on IP
    - Storage controller may do IPSec directly, or could add a tunnel device
      - (But if you have to add a tunnel, what about network between tunnel and storage controller...)

Securing the stack: storage controller NFS



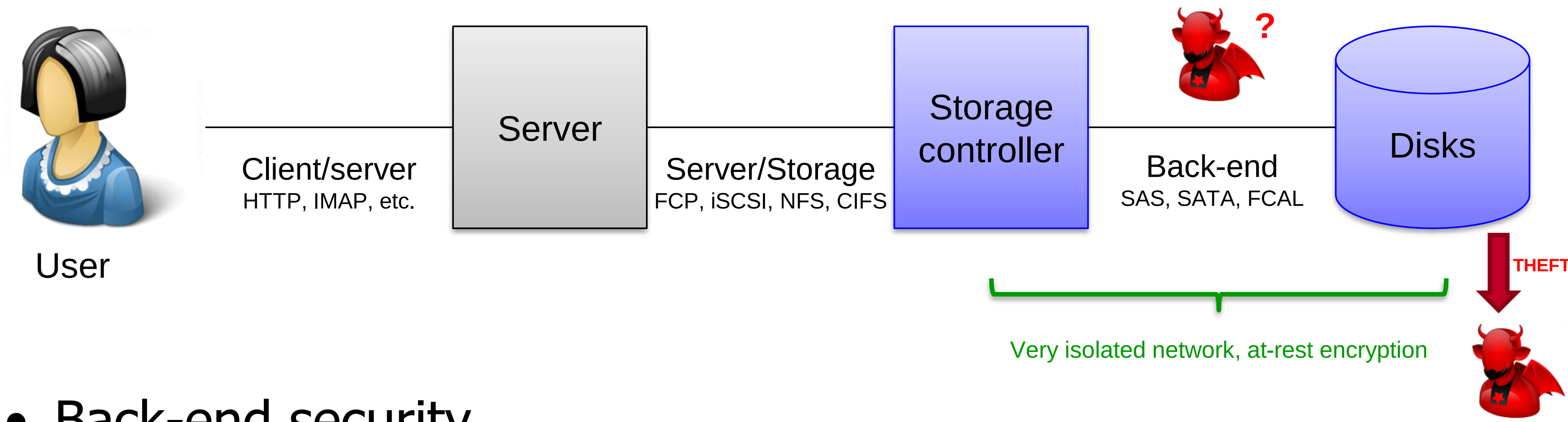
- Storage controller security: NFS
  - Identity verification: **IP-based check** or **Kerberos**
    - IP-based check: garbage
    - Kerberos: server authenticates with central login authority; basically equivalent to hash-based password verification
  - Encryption: **IPSec**
    - No built-in encryption standard (or even cert verification)
    - Instead we use generic IPSec again; similar tradeoffs as with iSCSI

Securing the stack: storage controller CIFS



- Storage controller security: CIFS
  - Identity verification: **Windows certificates**
    - Similar certificate system to the client/server side, nice
  - Encryption: **CIFS encryption** (new) or **IPSec**
    - Historically had to do IPSec (similar to iSCSI/NFS)
    - Windows server 2012+ and Windows 8+ can do CIFS-level encryption

Securing the stack: at-rest encryption



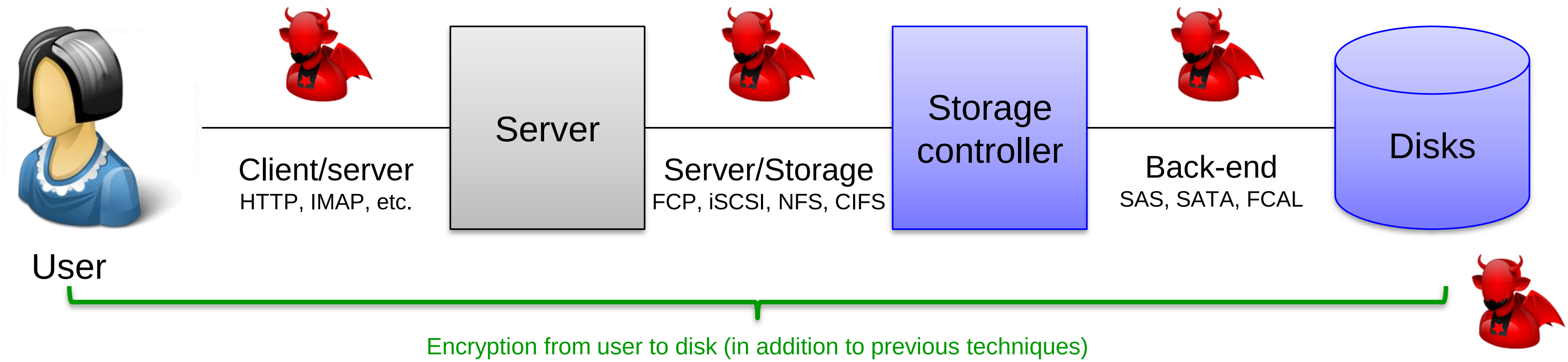
- Back-end security
  - Not usually concerned with data "in-flight" from controller to disk
    - If attacker has attached a wire to your SAS bus, game over
  - More common concern: disk theft or inspection
  - **"At-rest" encryption**: controller encrypts on way to physical media
  - Typically symmetric encryption
  - Question: Where does the key live???

Key management

- Fundamental problem with at-rest encryption: Where does the key live?
  - In RAM?
    - How did it get there?
    - How do I get it back after an outage?
  - One solution: boot-time key storage (admin must insert cart to provide key, key copied to RAM, admin takes card out and secures it)

- The "LOL DRM" issue:
  - Systems that store key with encrypted data

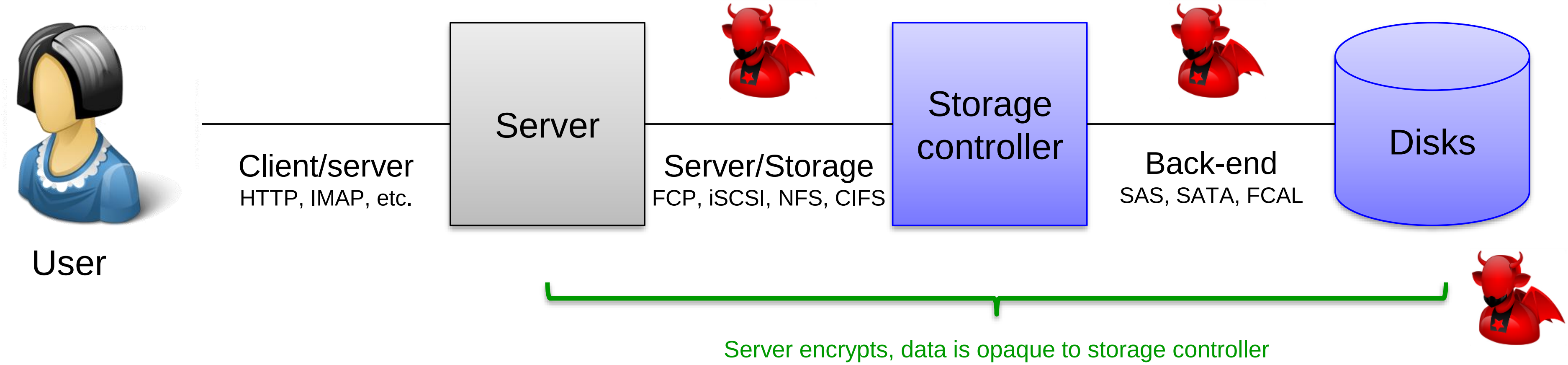
Securing the stack: end-to-end encryption



- Special case: end-to-end encryption
  - Client encrypts data in app-specific manner
  - Application on server understands this, doesn't decrypt it (and can't!)
    - Some meta-data is visible
  - Lands on disk with encryption intact
  - Not generalizable – only applicable with app can ignore user content

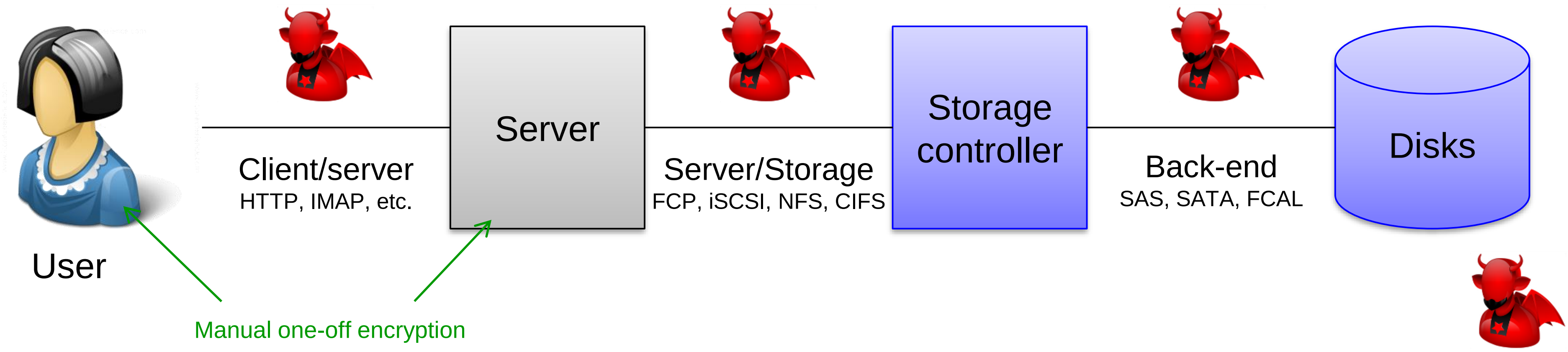
  - Example: secure email systems, cloud backup

Securing the stack: server encryption



- Special case: server encryption
  - Server runs encryption wrapper over storage controller's NAS/SAN volume
  - Encrypted data is opaque to storage controller
    - Simple to implement
    - Negates storage efficiency features

Securing the stack: "one-off" encryption



- Special case: manual file encryption
  - Can use a simple app to encrypt one or more files
  - Encrypted files are otherwise stored normally
  - With automation, a cheap "bolt on" solution

Encryption side-effects

- Encrypted content cannot be compressed or deduplicated
  - Storage efficiency features have to be applied first

- What about metadata?
  - Filenames, sizes, dates can be valuable information
  - If you're encrypting SAN traffic, you encrypt metadata for free
  - If NAS, though...how to organize file system of encrypted metadata?
    - Would have to add key semantics to file IO, break things, etc.
    - Applying file system encryption above block device is not common

- Encryption makes backup harder
  - Backup the plaintext? Security failure.
  - Backup the ciphertext? Need to back up the key, too...

Access control

Includes content from Computer Security: Principles and Practices by William Stallings and Lawrie Brown (the slate blue slides)

Access control topics

- Core concepts
- Access control policies:
  - Discretionary Access Control (DAC)
    - UNIX file system
    - Access Control Lists (ACLs)
  - Mandatory Access Control (MAC)
  - Role-based Access Control (RBAC)

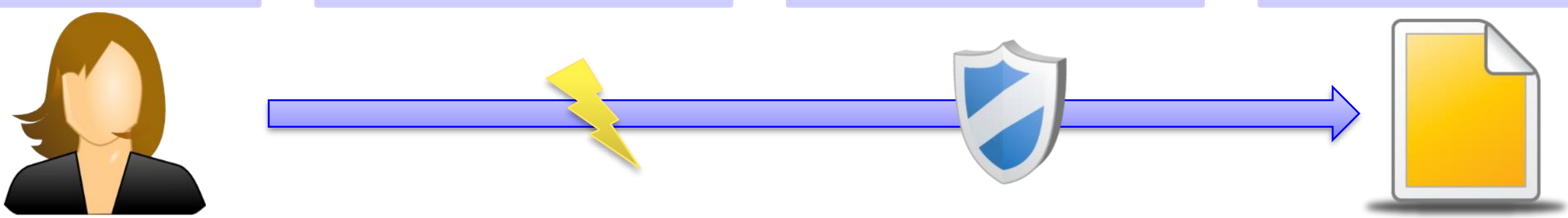
Subjects, Objects, Actions, and Rights

Subject (initiator)
- The thing making the request (e.g. the user)

Verb (request)
- The operation to perform (e.g., read, delete, etc.)

Right (permission)
- A specific ability for the subject to do the action to the object.

Object (target)
- The thing that's being hit by the request (e.g., a file).



Access Control (AC) Policies

- **Discretionary AC (DAC)**: There's a list of permissions attached to the subject or object (or possibly a giant heap of global rules).
- **Mandatory AC (MAC)**: Objects have classifications, subjects have clearances, subjects cannot give additional permissions.
  - An overused/abused term
- **Role-based AC (RBAC)**: Subjects belong to roles, and roles have all the permissions.
  - The current Enterprise IT buzzword meaning "good" security

Access control topics

- Core concepts
- Access control policies:
  - Discretionary Access Control (DAC)
    - UNIX file system
    - Access Control Lists (ACLs)
  - Mandatory Access Control (MAC)
  - Role-based Access Control (RBAC)

DAC model

```
bool IsActionAllowed(subject, object, action) {
  if (action ∈ get_permissions(subject,object))
    return true
}
```

- Can use various data structures, none of which should surprise you

Matrix

		OBJECTS			
		File 1	File 2	File 3	File 4
SUBJECTS	User A	Own Read Write		Own Read Write	
	User B	Read	Own Read Write	Write	Read
	User C	Read Write	Read		Own Read Write

(a) Access matrix

Flat list

Subject	Access Mode	Object
A	Own	File 1
A	Read	File 1
A	Write	File 1
A	Own	File 3
A	Read	File 3
A	Write	File 3
B	Read	File 1
B	Own	File 2
B	Read	File 2
B	Write	File 2
B	Write	File 3
B	Read	File 4
C	Read	File 1
C	Write	File 1
C	Read	File 2
C	Own	File 4
C	Read	File 4
C	Write	File 4

Linked list

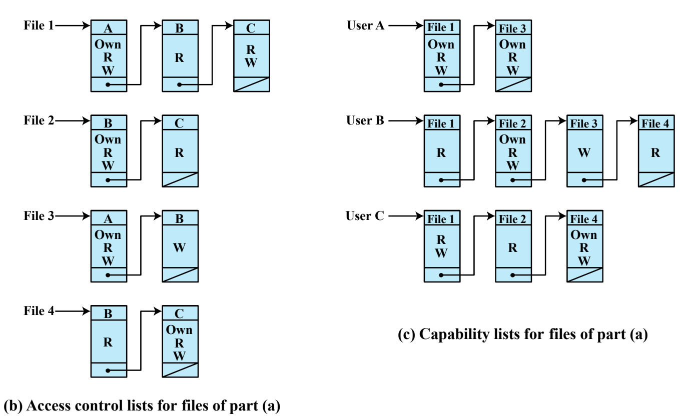


Figure 4.2 Example of Access Control Structures

UNIX File Access Control

UNIX files are administered using inodes (index nodes)

- Control structures with key information needed for a particular file
- Several file names may be associated with a single inode
- An active inode is associated with exactly one file
- File attributes, permissions and control information are sorted in the inode
- On the disk there is an inode table, or inode list, that contains the inodes of all the files in the file system
- When a file is opened its inode is brought into main memory and stored in a memory resident inode table

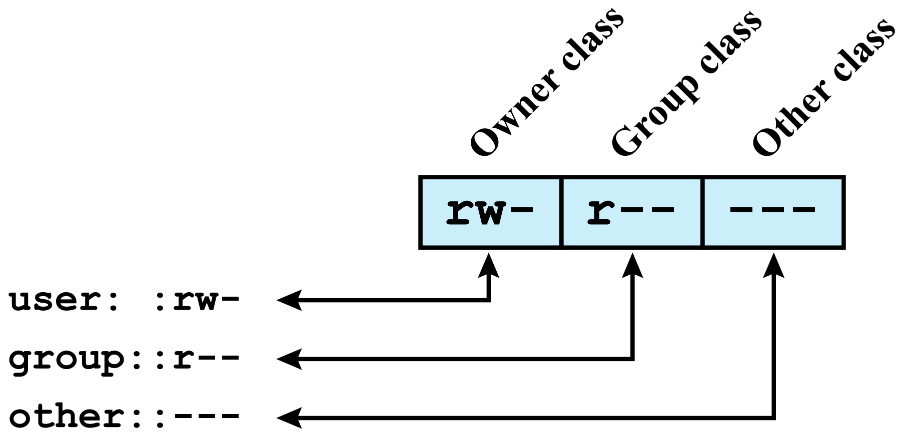
Directories are structured in a hierarchical tree

- May contain files and/or other directories
- Contains file names plus pointers to associated inodes

UNIX

File Access Control

- Unique user identification number (user ID)
- Member of a primary group identified by a group ID
- Belongs to a specific group
- 12 protection bits
  - Specify read, write, and execute permission for the owner of the file, members of the group and all other users
- The owner ID, group ID, and protection bits are part of the file's inode



(a) Traditional UNIX approach (minimal access control list)

Relevant UNIX commands

`chmod`: Change these bits
`chown`: Change owner
`chgrp`: Change group

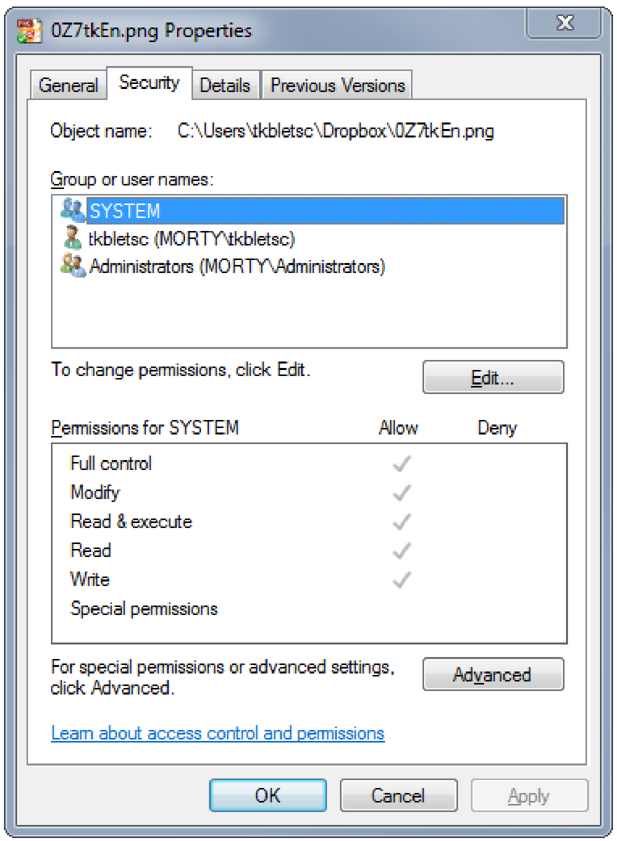
Traditional UNIX File Access Control

- "Set user ID"(SetUID)
- "Set group ID"(SetGID)
  - System temporarily uses rights of the file owner/group in addition to the real user's rights when making access control decisions
  - Enables privileged programs to access files/resources not generally accessible
- Sticky bit
  - When applied to a directory it specifies that only the owner of any file in the directory can rename, move, or delete that file
- Superuser
  - Is exempt from usual access control restrictions
  - Has system-wide access

File system access control lists (ACLs)

- Arbitrary list of rules governing access per-file/directory
- More flexible than classic UNIX permissions, but more metadata to store/check

Windows ACL UI



Examples of Linux ACL commands

Set all permissions for user johny to file named "abc":

```
# setfacl -m "u:johny:rwx" abc
```

Check permissions

```
# getfacl abc

# file: abc
# owner: someone
# group: someone
user::rw-
user:johny:rwx
group::r--
mask::rwx
other::r--
```

Change permissions for user johny:

```
# setfacl -m "u:johny:r-x" abc
```

Check permissions

```
# getfacl abc

# file: abc
# owner: someone
# group: someone
user::rw-
user:johny:r-x
group::r--
mask::r-x
other::r--
```

Remove all extended ACL entries:

```
# setfacl -b abc
```

From Arch Wiki

Access control topics

- Core concepts
- Access control policies:
  - Discretionary Access Control (DAC)
    - UNIX file system
    - Access Control Lists (ACLs)
  - Mandatory Access Control (MAC)
  - Role-based Access Control (RBAC)

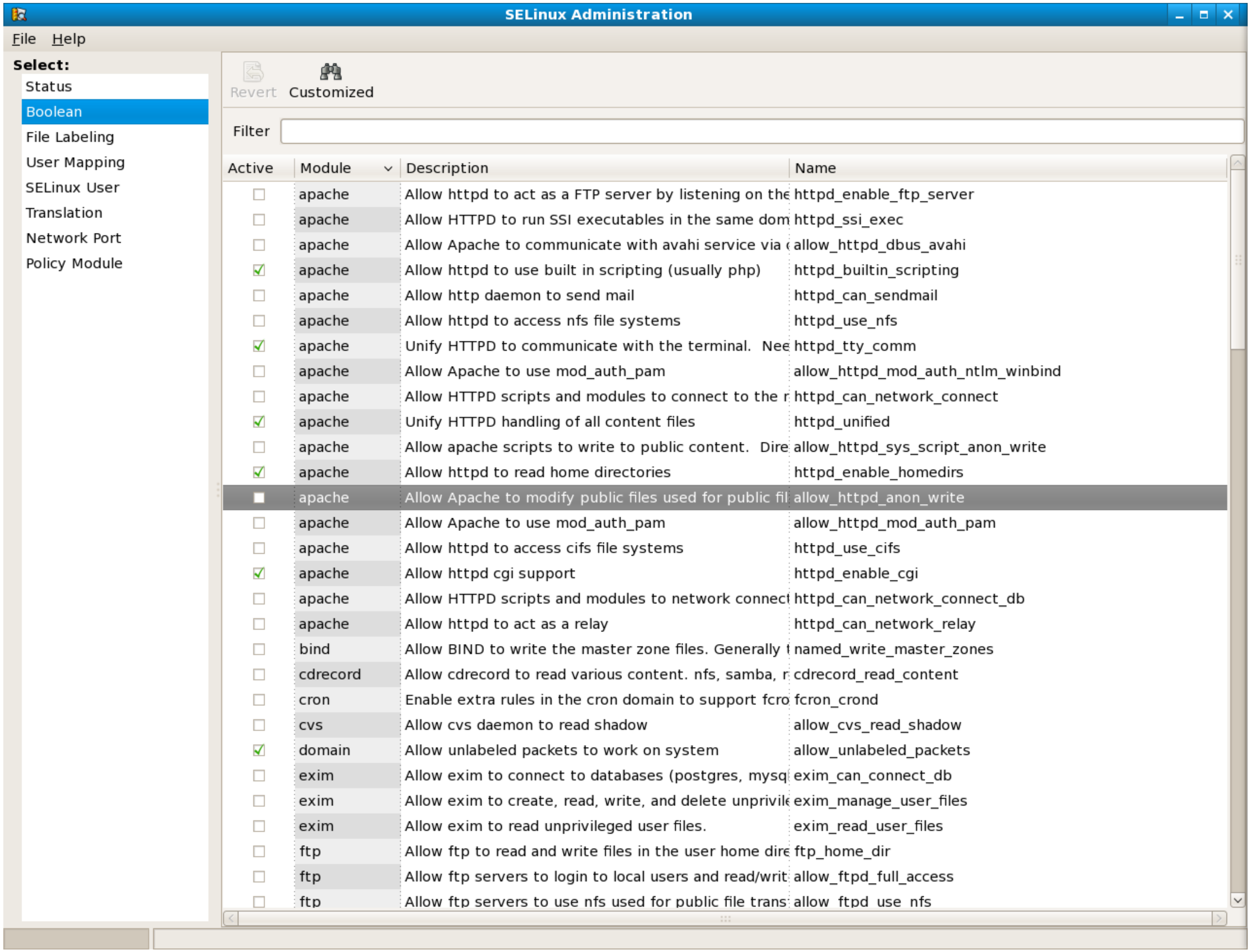
MAC model

```
bool IsActionAllowed(subject, object, action) {
  for each rule in rules:
    if rule allows (subject,object,action) return true
  return false
}
```

MAC example: SELinux

- Developed by U.S. Dept of Defense
- General deployment starting 2003
- Can apply rules to virtually every user/process/hardware pair
- Rules are governed by system administrator only
  - No such thing as “selinux_chmod” for users

MAC example: SELinux



Access control topics

- Core concepts
- Access control policies:
  - Discretionary Access Control (DAC)
    - UNIX file system
    - Access Control Lists (ACLs)
  - Mandatory Access Control (MAC)
  - Role-based Access Control (RBAC)

RBAC: The thing you invent if you spend enough time doing access control

- Scenario:
  - Frank: "Bob just got hired, please given him access."
  - Admin: "What permissions does he need?"
  - Frank : "Same as me."
- Later, a new system is added
  - Bob: "Why can't I access the new system?!"
  - Admin: "Oh, I didn't know you needed it too…"
  - Bob: "I need everything Frank has!"
- Later, Frank is promoted to CTO
  - Admin: "Welp, looks like Bob also needs access to our private earnings, since this post-it says he gets everything Frank has…"
- The admin is later fired amidst allegations of conspiracy to commit insider trading with Bob. He dies in prison. ☹

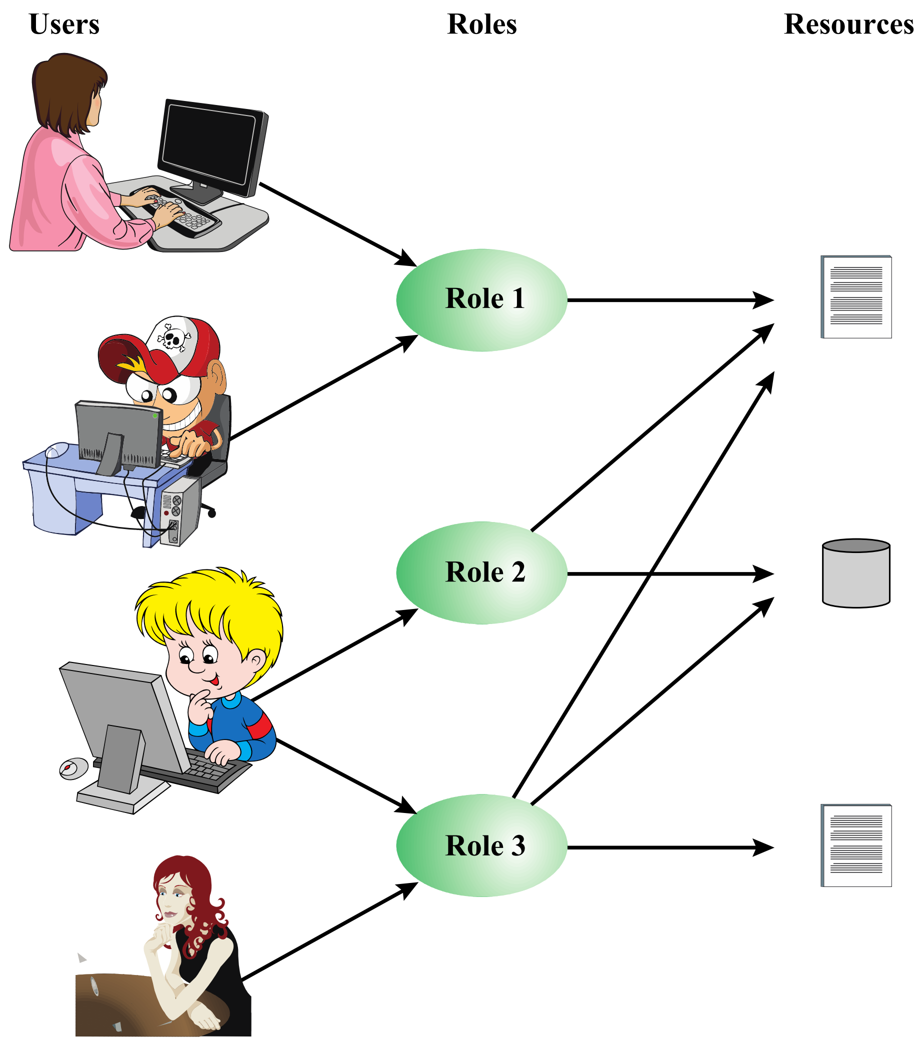


Figure 4.6 Users, Roles, and Resources

RBAC

- Decide what KINDS of users you have (**roles**)
- Assign **permission** to **roles**.
- Assign **users** to **roles**.

- When a role changes, everyone gets the change.
- When a user's role changes, that user gets a whole new set of permissions.
- No more special unique snowflakes.

- Roles may be partially ordered, e.g. "Production developer" inherits from "Developer" and adds access to the production servers

```
bool IsActionAllowed(subject, object, action) {
  if (action ∈ get_permissions(subject.role,object))
    return true
}
```

Secure deletion

Secure deletion

- Must destroy data when we need to (e.g. decommissioning a storage system)
- Destroying is easy, right?
  - When you spend all this effort preventing data loss, intentionally losing data can get surprisingly hard.
- Things preventing data destruction:
  - **'Delete' doesn't destroy**: it just updates metadata and marks blocks freed
  - **Journaling**: we keep scraps of written data separate from the actual data blocks; these aren't affected by simple deletion
  - **Failed drives**: If the drive dies enough to replace, we may not be able to tell the drive to overwrite data, but it's still there...
  - **Hardware redundancy**: SSDs redirect blocks internally for wear leveling; disks redirect blocks for bad sector compensation
  - **Snapshots**: their whole purpose was to recover from accidental deletion
  - **Backups**: We've replicated this data across the country...

How to overcome: technical/procedural

- **Block-level IO**: Overwrite raw disk below file system level
  - Traditional: "`dd if=/dev/zero of=/dev/sda`"
    (basically that means "`cat /dev/zero > /dev/sda`")
  - Gets around file system, snapshots, journaling.

- **ATA security erasure:** erase command built into drive

- **Procedural**: Documented, automated processes for snapshot deletion, destruction of backups, etc.

- **"Crypto-shredding"**: Do at-rest encryption all along. Then, to destroy data, simply lose the key.

How to overcome: physical

- **Destroy!!!!!!**