The original NFS assumed both the client and server kernels are trustworthy and that the server UID's match with the client UID's.

How does NFS address the superuser? (obviously every root can't have root permissions)

- we make user 0 "nobody" and give them minimal privileges

This is simplistic: the naming scheme in reality is complicated and involves strings.

Soon after, we developed authentication (via Kerberos, etc) to encrypt packets. We need this authentication to prevent attacks against:

1. Privacy (unauthorized release of information)
2. Integrity (tempering with unauthorized data)
3. Service (Denial of Service)

Our goals are therefore:

- Allow authorized access (+)
- Disallow unauthorized access (-)
- Good performance (+)

Positive goals are easy because they get reported, but negative goals are more difficult

When attempting to build a secure system, we perform *Threat Modeling*; the threats to the security of a system include:

- insiders
- social engineering (generally non-coding)
- network attacks (virus, DOS)
- device attacks (USB) attacks

The last is so common that it is recommended people use a "usb condom".

Our security within a system is trivial; we simply chroot untrusted processes so they think they have access to the root. Even this is not perfect, however: for instance, processes P and P' can communicate with gettimeofday(), where P chews up computer time and P' checks the clock, transmitting data with only clock change. In this way, P can send messages using morse code. This is called a *covert channel*, and they are effectively unavoidable; we instead only focus on critical attacks

To prevent these, our system requires 4 major properties

1. Authentication (password, credentials)
2. Integrity (checksum)
3. Authorization (access control list)
4. Auditing (log)

and obviously correctness and performance.

0.1 Authentication

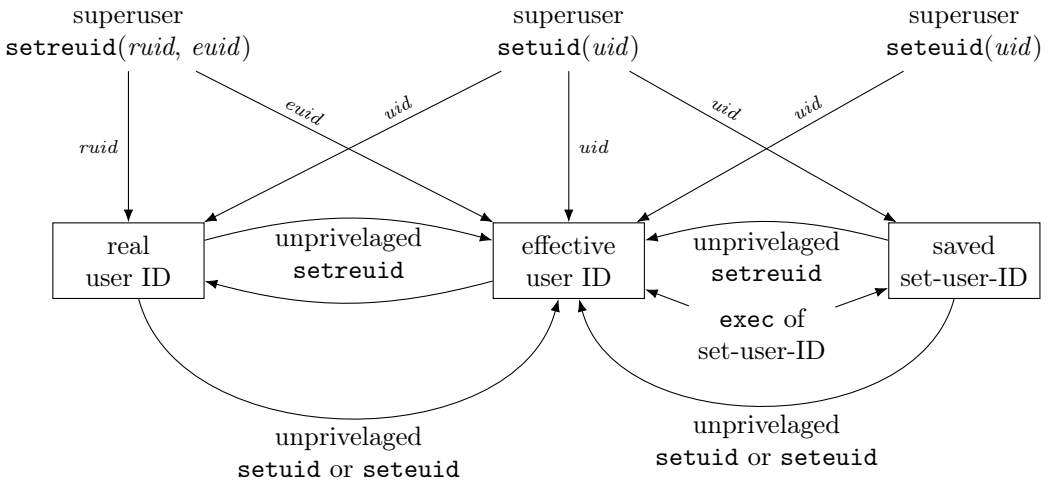
We utilize *authentication* to prevent masquerading. Authentication is based on 3 major properties:

1. who the principal is (ex. retinal scan)
2. something the principal has (ex. smart card)
3. something the principal knows (ex. password)

but these are often equivalent, as we can bootstrap from one to the other. For example, SEASNET is based on (1) but uses (3) for verification.

There are two phases of authentication:

1. *external authentication* := protection from the outside world.
   - the system verifies that the user is a part of the system
   - examples include a trusted login agent (OS), key, password, biometrics, MFA
   - attack types
     - theft of hardware tokens
     - fraudulent servers
     - chain breaking
     - buggy login agents
     - well known default passwords
     - SIN attacks (SIM card backup)
     - password recovery attacks
     - bad routers
   - From an OS standpoint, many of these are beyond our purview.
2. *internal authentification* := protection from inside the system
   - the system verifies the user is a specific member of a system
   - by design, internal authentification is much more efficient
   - common type of attack is a PATH injection:
     - Internal authentication in Linux comes in the form of UIDs
     - The UID is the current user, and the EUID is the program's creator
     - There are select programs (called setuid programs) which this is not true for; these are marked with a UID execution bit of "s"
     - These programs are automatically trusted by the root (ex. /usr/bin/passwd)
     - a PATH injection is a bad user setting the path such that a system() call behaves poorly
     - For that reason, we avoid usages such as system("ls")



Shells are at risk for PATH injection, since they implicitly call system(), but our c programs must be reliable (especially in terms of buffer overflow). Sometimes, however, there isn't much we can do (take setuid()). At least setuid() requires root access

Once we verify identity, we move on to:

0.2 Authorization

There are two schemes for access to resources:

1. *Direct Access*

```
char *p = mmap(fd, offset, etc...);
```

The system checks authority once and maps a pointer onto the process address space. Thus is done with page table manipulation. Properties:

+ high performance
- easy resource corruption (by bad actors and race conditions)
- necessitates hardware support

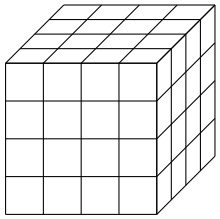
2. *Indirect Access*

```
int fd = open(file, ...);
read(fd, ...);
```

The system issues a request to handler or system call to check on every attempted resource access.

- low performance
+ avoid corruption
+ easier to revoke permission

We can think of an operating system as a set of principals and resources:



- We require 3 parameters to check whether a given user can access a given resource.
- This array seems massive, but the data is not random, so there must be a more efficient way to represent the information.

It turns out we can represent it in two dimensions!

We do this with an *Access Control List*

```
$ getfacl .
user: rwx
group: r-x
other: r-x
$ setfacl -m g:tas:rwx .
$ getfacl .
user: rwx
group: r-x
other: r-x
tas: rwx
# we have appended the group "tas" to the end of the directory's list
```

Associated with each file is a list of access rights under the control of the file's owner. Using this, we can form our own ad-hoc groups outside of the built-in ones.

This is not the only way to represent authorization, and isn't perfect for every purpose; we may need to capturing rights accurately or letting ordinary users specify rights. ACLs in general are not flexible enough for large organizations. Instead we can use role-based access control (RBAC); this model more complex but secure than ACLs.

The model: keep a distinction between users and their roles.

For example, the user "eggert" could have multiple roles assigned to him: ie eggert as cs-faculty and as sys-admin

- cs-faculty allows him to change grades.
- sys-admin allows him to make changes to other users' permissions.

These permissions are usually mutually exclusive, and roles can have sub-roles, which may share some permissions.

In both models, each file lists permissions for each user; these are *resource-centric*. We can instead use a principal-centric model called a *Capability Model*, where users hold an object with an unforgeable unique ID representing a resource.

Linux utilizes both approaches

- Each descriptor has its own separate permissions from the file (capability)
- Each file is resource-centric and holds its own permissions

```
fd = open ("abc", O_RDONLY)              <--- code
r--              rwx                     <--- permissions
read-only        read, write, execute    <--- description of permissions
// the file "abc" has the permissions rwx
// the fd is the capability with permissions r--
// any action done with fd can only read, even though the file is rwx
```

Note that the flags could include O_PATH, which gives no permissions, only showing the existence of the file. We could then duplicate the file descriptor using dup2, but then you'd have 2 useless fd's.

We can therefore run:

```
int stat(const char *path, struct stat *buf);
// return information about the file that fd points to
```

We have addressed *auditing* with logs and *integrity* with checksums already.