Protection, identity, and trust
Illustrated with Unix “setuid”

Jeff Chase
Duke University
The need for access control

- Processes run programs on behalf of users. ("subjects")
- Processes create/read/write/delete files. ("objects")
- The OS kernel mediates these accesses.
- How should the kernel determine which subjects can access which objects?

This problem is called access control or authorization ("authz"). It also encompasses the question of who is authorized to make a given statement.

The concepts are general, but we can consider Unix as an initial example.
A reference monitor is a program that controls access to a set of objects by other programs. The reference monitor has a guard that checks all requests against an access control policy before permitting them to execute.
Requirements for a reference monitor

1. **Isolation**: the reference monitor is protected from tampering.
2. **Interposition**: the only way to access the objects is through the reference monitor: it can examine and/or reject each request.
3. **Authentication**: the reference monitor can identify the subject.
What is the nature of the isolation boundary?
If we’re going to post a guard, there should also be a wall. Otherwise somebody can just walk in past the guard, right?
The kernel is a reference monitor

- No access to kernel state by user programs, except through syscalls.
  - Syscalls transfer control to code chosen by the kernel, and not by the user program that invoked the system call.
- The kernel can inspect all arguments for each request.
- The kernel knows which process issues each request, and it knows everything about that process.
- User programs cannot tamper with the (correct) kernel.
  - The kernel determines everything about how the machine is set up on boot, before it ever gives user code a chance to run.
- **Later** we will see how the kernel applies access control checks, e.g., to file accesses.
What is the nature of the isolation boundary?
Clients can interact with the server only by sending messages through a socket channel. The server chooses the code that handles received messages.
Identity in an OS (Unix)

- Every process has a **security label** that governs access rights granted to it by kernel.
- Abstractly: the label is a list of named **attributes** and values. An OS defines a space of attributes and their interpretation.
- Some attributes and values represent an **identity** bound to the process.
- **Unix**: e.g., `userID`: uid

Every Unix user account has an associated **userID** (uid). (an integer)

There is a special **administrator** uid==0 called **root** or **superuser** or **su**. **Root** “can do anything”: normal access checks do not apply to root.
Labels and access control

Every system defines rules for assigning security labels to subjects (e.g., Bob’s process) and objects (e.g., file foo).

Every system defines rules to compare the security labels to authorize attempted accesses.

Should processes running with Bob’s userID be permitted to open file foo?
Unix: setuid and login

• A process with uid==root may change its userID with the `setuid` system call.
• This means that a root process can speak for any user or act as any user, if it tries.
• This mechanism enables a system login process to set up a shell environment for a user after the user logs in (authenticates). This is a refinement of privilege.

A privileged `login` program verifies a user password and execs a command interpreter (`shell`) and/or window manager for a logged-in user. A user may then interact with a shell to direct launch of other programs. They run as children of the shell, with the user's uid.
Init and Descendants

Kernel “handcrafts” initial root process to run “init” program.

Other processes descend from init, and also run as root, including user login guards.

Login invokes a `setuid` system call before `exec` of user shell, after user authenticates.

Children of user shell inherit the user’s identity (uid).
Labels and access control

Every file and every process is labeled/tagged with a user ID.

A privileged process may set its user ID.

A process inherits its userID from its parent process.

A file inherits its owner userID from its creating process.
Should processes running with Bob’s userID be permitted to open file foo?
Reference monitor and policy

How does the guard decide whether or not to allow access?
We need some way to represent **access control policy**.
Concept: access control matrix

We can imagine the set of all allowed accesses for all subjects over all objects as a huge matrix.

<table>
<thead>
<tr>
<th></th>
<th>obj1</th>
<th>obj2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>RW</td>
<td>---</td>
</tr>
<tr>
<td>Bob</td>
<td>R</td>
<td>RW</td>
</tr>
</tbody>
</table>

How is the matrix stored?
How is the matrix stored?

- **Capabilities**: each subject holds a list of its rights and presents them as proof of access rights. In many systems, a **capability** is an unforgeable reference/token that confers specific rights to access a specific object.

- **Access control list (ACL)**: each object stores a list of identifiers of subjects permitted to access it.
A **role** or **group** is a named set $S$ of subjects. Or, equivalently, it is a named boolean predicate that is true for subject $s$ iff $s \in S$.

Roles/groups provide a level of indirection that simplifies the access control matrix, and makes it easier to store and manage.
More generally, each subject/object is labeled with a list of named attributes with typed values (e.g., boolean).

Generally, an access policy for a requested access is a boolean (logic) function over the subject and object attributes.
Access control: general model

- We use this simple model for identity and authorization.
- Real-world access control schemes use many variants of and restrictions of the model, with various terminology.
  - **Role-Based Access Control** (RBAC)
  - **Attribute-Based Access Control** (ABAC)
- A **guard** is the code that checks access for an object, or (alternatively) the access policy that the code enforces.
- There are many extensions. E.g., some roles are nested (a partial order or lattice). The guard must reason about them.
  - Security clearance level $\text{TopSecret} \subseteq \text{Secret}$ [subset]
  - Equivalently: $\text{TopSecret}(s) \rightarrow \text{Secret}(s)$ [logical implication]
File permissions in Unix (vanilla)

- The **owner** of a Unix file may tag it with a “**mode**” value specifying access rights for subjects. (Don’t confuse with kernel/user mode.)
  - Unix “mode bits” are a simple/compressed form of an **access control list (ACL)**. Later systems like AFS and AWS have richer ACLs.
  - Subject types = {owner, group, other/anyone}  [3 subject types]
  - Access types = {read, write, execute}  [3 bits for each of 3 subject types]
  - If the file is executed, should the system setuid the process to the userID of the file’s owner.  [1 bit, the **setuid bit**]
  - **10 mode bits total**: (3x3)+1. Usually given in octal: e.g., “777” means all 9 bits are set: anyone can r/w/x the file, but no setuid.

- Unix provides a syscall for owner to set the owner, group, and mode on each file (inode). A command utility of the same name calls it.
  - **chmod, chown, chgrp** on file or directory

- “Group” was added later and is a little more complicated: a user may belong to multiple groups.
Unix
file
access
mode
bits
Using the Unix access mode bits
A simple example

• Bob wants to read file foo.
  – Owner=Alice, group=students, mode 640

• The owner of file foo is Alice, and the file is associated with the group students.

• Is Bob Alice? No. So the owner bits don’t apply.

• Is Bob in the group students?

• Yes: group members have access “4”: the read is permitted.

• No: Bob is “other”. “Other” has access “0”: the read is rejected.
Protection, access control, security

DAY 2
First a quick interlude to talk about p2 and exam
A silly difference among machine architectures creates a need for byte swapping when unlike machines exchange data over a network.
Using the heap

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char* cb = (char*) malloc(14);
    cb[0] = 'h';
    cb[1] = 'i';
    cb[2] = '!';
    cb[3] = '\0';
    printf("%s\n", cb);
    int *ip = (int*)cb;
    printf("0x%x\n", *ip);
    free(cb);
}
```

chase$ cc -o heap heap.c
chase$ ./heap
hi!
0x216968
chase$

Try:
- http://wikipedia.org/wiki/ASCII
- http://wikipedia.org/wiki/Endianness
x86 is little-endian

**Little-endian**: the lowest-numbered byte of a word (or longword or quadword) is the least significant.

```
chase$ cc -o heap heap.c
chase$ ./heap
hi!
0x216968
```

(high)

(low)

```
h=0x68
i=0x69
!=0x21
0
```
X86 is little-endian

(wikipedia)
Stepping by instruction in gdb

stei
stei arg
si

Execute one machine instruction, then stop and return to the debugger.

It is often useful to do ‘display/i $pc’ when stepping by machine instructions. This makes gdb automatically display the next instruction to be executed, each time your program stops. See Automatic Display.

An argument is a repeat count, as in step.
P3. Trace that schedule (50 points)

Consider the following pseudocode for a soda machine with N=1; the machine has buffer space for a single soda (initially empty). This problem asks you to trace execution of this code in detail for the given main program. Assume that the code runs on a uniprocessor (single core) with no involuntary preemptions; thus the schedule is deterministic. Trace the schedule as an ordered list containing every thread state change. The thread primitives behave as in lab 1; all queues are FIFO, as required for p1. The threads are named P0 (main), P1, C0, and C1. It may help to list queue contents at various points.

main_thread_starts_here

thread_create(consume, ...);
thread_create(consume, ...);
thread_create(produce, ...);
thread_yield();
produce();

void produce(...)
{
  thread_lock(1);
  while (full) {
    thread_yield();
    thread_wait(1, 2);
    add_soda;
    thread_signal(1, 2);
    thread_yield();
    thread_unlock(1);
  }
  thread_lock(1);
  while (empty) {
    thread_yield();
    thread_wait(1, 2);
    take_soda;
    thread_signal(1, 2);
    thread_yield();
    thread_unlock(1);
  }
}

P0 runs ... <continue below>
A trusted program: sudo

Example: sudo program runs as root, checks user authorization to act as superuser, and (if allowed) executes requested command as root.
The setuid bit: another way to setuid

- The mode bits on a program file may be tagged to setuid to owner’s uid on `exec*`.
- This is called **setuid bit**. Some call it the most important innovation of Unix.
- It enables users to request protected ops by running **secure programs** that execute with the userID of the program owner, and not the caller of exec*.
- The user cannot choose the code: only the program owner can choose the code to run with the program owner’s uid.
- Parent cannot subvert/control a child after program launch: a property of Unix exec*.

**Example:** `sudo` program runs as root, checks user authorization to act as superuser, and (if allowed) executes requested command as root.
The secret of setuid

- The **setuid bit** can be seen as a mechanism for a trusted program to function as a **reference monitor**.

- E.g., a **trusted program** can govern access to files.
  - Protect the files with the program owner’s uid.
  - Make them inaccessible to other users.
  - Other users can access them via the trusted program.
  - But only in the manner permitted by the trusted program.
  - Example: “moo accounting problem” in 1974 paper (cryptic)

- What is the reference monitor’s “isolation boundary”? What protects it from its callers?
MOO accounting problem

• From a cryptic reference in an early Unix paper…
• Multi-player game called Moo
  • Players run the game as a process
  • Want to maintain high score in a file
• Here’s the problem:
  • What happens when a player earns a qualifying score?
  • We want to update the scores file.
  • But the game process runs with the user’s UID.
• We want players to be able to modify the file, but only in a manner prescribed by the Moo program.
MOO accounting problem

- Multi-player game called Moo
  - Want to maintain high score in a file
- Could have a trusted process update scores

- Is this good enough?
MOO accounting problem

- Multi-player game called Moo
  - Want to maintain high score in a file
- Could have a trusted process update scores

- Is this good enough?
  - Can’t be sure that reported score is genuine
  - Need to ensure score was computed correctly
Access control

• **Insight:** sometimes simple inheritance of uids is insufficient
  • Tasks involving management of “user id” state
  • Logging in (login)
  • Changing passwords (passwd)

• **Why isn’t this code just inside the kernel?**
  • This functionality doesn’t really require interaction w/ hardware
  • Would like to keep kernel as small as possible

• **How are “trusted” user-space processes identified?**
  • Run as **super user** or **root** (uid 0)
  • Like a software kernel mode
  • If a process runs under uid 0, then it has more privileges
Access control

• Why does login need to run as root?
  • Needs to check username/password correctness
  • Needs to fork/exec process under another uid

• Why does passwd need to run as root?
  • Needs to modify password database (file)
  • Database is shared by all users

• What makes passwd particularly tricky?
  • Easy to allow process to shed privileges (e.g., login)
  • passwd requires an escalation of privileges

• How does UNIX handle this?
  • Executable files can have their setuid bit set
  • If setuid bit is set, process inherits uid of image file’s owner on exec
MOO accounting problem

- Multi-player game called Moo
  - Want to maintain high score in a file
- How does setuid solve our problem?
  - Game executable is owned by trusted entity
  - Game cannot be modified by normal users
  - Users can run executable though
  - High-score is also owned by trusted entity
- This is a form of trustworthy computing
  - Only trusted code can update score
  - Root ownership ensures code integrity
  - Untrusted users can invoke trusted code
Unix setuid: recap

Refine the privilege of this process (using setuid syscall).

Root (admin/superuser)

Alice

Root (admin/superuser)

login

setuid("alice")

exec

shell

setuid("alice")

exec("power")

shell

exec("power")

uid = "root"

setuid("alice")

shell

power

power

shell

read, write...

open("secret")

root!

setuid bit = true

owner = "root"

755 (exec perm)

creat("power")

write(...)

chmod(+x,setuid=true)

setuid bit = true

owner = "root"

600

Refine the privilege of processes running this trusted program (using setuid bit).

Amplify/escalate the privilege of processes running this trusted program (using setuid bit).
Server access control

A server can act as a reference monitor, and apply an access control policy to incoming requests.
Concept: reference monitor

Reference monitor
Example: Unix kernel

Principles of Computer System Design ♥ Saltzer & Kaashoek 2009
More terminology

• A reference monitor monitors references to its objects…
• …by a subject (one or more), also called a principal.
  – I generally reserve the term “principal” for an entity with responsibility and accountability in the real world (e.g.: you).
  – A subject identity may be a program speaking for a user, which is distinct from the user herself.
• The reference monitor must determine the true and correct identity of the subject.
  – This is called authentication. Example: system login
  – We’ll return to this later. In Unix, once a user is logged in, the kernel maintains the identity across chained program executions.
  – Things are different on a network, where there is no single trusted kernel to anchor all interactions.
Reference monitors: three examples

1. The Kernel

2. Setuid bit: a mode bit on an executable program file indicating that the program always runs as its owner.
   - A process running that program always has the owner’s userID.
   - Exec* system call implementation retrieves the setuid bit and owner’s userID from the file’s inode, and sets the process userID if the setuid bit is set.

3. Server process
   - Listens on a named port, accepts request from clients over sockets, and decides whether to allow or deny each request.

• For each example, what is the nature of the isolation boundary?
• For each example, how does it identify the subject?
Protection: abstract concept

• Running **code** can access some set of **data**.
  – **Code** == program, module, component, instance,
  – **Data** == objects, state, files, VM segments…

• We call that set a **domain** or **protection domain**.
  – Defines the “powers” that the running code has available to it.
  – Determined by **context** and/or **identity** (label/attributes).
  – Protection domains may overlap (data sharing).

**Domain**: a set of accessible names bound to objects. Accesses are checked by a reference monitor.

Yes, this is verrry abstract.
Protection

- Programs may **invoke** other programs.
  - E.g., call a procedure/subprogram/module.
  - E.g., launch a program with **exec**.
- Often/typically/generally the invoked program runs in the same protection domain as the caller.
- (Or i.e., it has the same privilege over some set of objects. It may also have some private state. We’re abstracting.)

Examples?
Protection

• Real systems always use protection.
• They have various means to invoke a program with more/less/different privilege than the caller.
• In the reference monitor pattern, B has powers that A does not have, and that A wants. B executes operations on behalf of A, with safety checks.

Why?
Breaking the lockbox

• If any part of a domain is tainted…
• Then **all** of the domain is tainted.
  – Attacks propagate: if an attacker has a foothold, it can spread.
  – E.g., modify a program or take over an identity that another victim trusts, based on a password or crypto key stored in an accessible file.
• If an attacker can choose code that runs in a domain, then it controls the domain.
Any program you install or run can be a Trojan Horse vector for a malware payload.
Malware

[Source: somewhere on the web.]

![Malware warning message](Image)

Warning!

You are about to install some malware. Malware is bad. By reading this warning through to the end and still clicking yes you're failing the Windows Darwin Test. Don't be that guy, if you're reading this message still then wise up and for the love of your family photos on your hard drive click the 'No' button.

Yes  No

[Source: Google Chrome comics.]
Tusting Programs

• In Unix
  – Programs you run use your identity (process UID).
  – Maybe you even saved them with setuid so others who trust you can run them with your UID.
  – The programs that run your system run as root.

• You trust these programs.
  – They can access your files
  – send mail, etc.
  – Or take over your system…

• Where did you get them?
Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX\(^1\) swept into popularity with an industry-wide change from central main-

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE 1
Trusting Trust

• Perhaps you wrote them yourself.
  – Or at least you looked at the source code…

• You built them with tools you trust.

• But where did you get those tools?
  • Perhaps you wrote them yourself.
    – Or at least you looked at the source code…
  • You built them with tools you trust.
  • But where did you get those tools?
Where did you get those tools?

- Thompson’s observation: compiler hacks cover tracks of Trojan Horse attacks.
Login backdoor: the Thompson Way

- **Step 1: modify login.c**
  - (code A) if (name == “ken”) login as root
  - This is obvious so how do we hide it?

- **Step 2: modify C compiler**
  - (code B) if (compiling login.c) compile A into binary
  - Remove code A from login.c, keep backdoor
  - This is now obvious in the compiler, how do we hide it?

- **Step 3: distribute a buggy C compiler binary**
  - (code C) if (compiling C compiler) compile code B into binary
  - No trace of attack in any (surviving) source code
Reflections on trust

• What is “trust”?  
  – **Trust** is a belief that some program or entity will be faithful to its expected behavior.

• Thompson’s parable shows that trust in programs is based on a chain of reasoning about trust.  
  – We often take the chain for granted. And some link may be fragile.  
  – **Corollary**: successful attacks can propagate down the chain.

• The **trust chain** concept also applies to executions.  
  – We trust whoever launched the program to select it, configure it, and protect it. **But who booted your kernel?**
How much damage can malware do?

- If it compromises a process with your uid?
  - Read and write your files?
  - Overwrite other programs?
  - Install backdoor entry points for later attacks?
  - Attack your account on other machines?
  - Attack your friends?
  - Attack other users?
  - Subvert the kernel?

What if an attacker compromises a process running as **root**?
Rootkit

• If an attacker obtains root or subverts/compromises the kernel (TCB), then **all bets are off.**

• The machine is “**rooted**”: the attacker has full control.

• Attacker may install a **rootkit**: software that maintains continuous and/or undetectable control.

• A rootkit can:
  - Log keystrokes
  - Hook system APIs
  - Open attacker backdoor
  - Subvert (re)boot
  - Etc….
Subverting services

• There are lots of security issues/threats here.
• **TBD Q**: Is networking secure? How can the client and server authenticate over a network? How can they know the messages aren’t tampered? How to keep them private? A: crypto.
• **Q**: Can a malicious client inject code that runs on the server or in another client’s browser? What are the isolation defenses?
• **Q**: Can a malicious server inject code that runs on a client?
• **Q for now**: Can an attacker penetrate the server, e.g., to choose the code that runs in the server?

![Diagram](image)

*Inside job*

Install or control code inside the boundary.
Executive Summary

This security update resolves a privately reported vulnerability in the Server service. The vulnerability could allow remote code execution if an affected system received a specially crafted RPC request. On Microsoft Windows 2000, Windows XP, and Windows Server 2003 systems, an attacker could exploit this vulnerability without authentication to run arbitrary code. It is possible that this vulnerability could be used in the crafting of a wormable exploit. Firewall best practices and standard default firewall configurations can help protect network resources from attacks that originate outside the enterprise perimeter.

This security update is rated Critical for all supported editions of Microsoft Windows 2000, Windows XP, Windows Server 2003, and rated Important for all supported editions of Windows Vista and Windows Server 2008. For more information, see the subsection, Affected and Non–Affected Software, in this section.

The security update addresses the vulnerability by correcting the way that the Server service handles RPC requests. For more information about the vulnerability, see the Frequently Asked Questions (FAQ) subsection for the specific vulnerability entry under the next section, Vulnerability Information.

Recommendation. Microsoft recommends that customers apply the update immediately.

“confused deputy”

→ principle of least privilege
Code Analysis and Review

I want to start by analyzing the code to understand why we did not find this bug through manual code review nor through the use of our static analysis tools. First, the code in question is reasonably complex code to canonicalize path names; for example, strip out `..` characters and such to arrive at the simplest possible directory name. The bug is a stack-based buffer overflow inside a loop; finding buffer overruns in loops, especially complex loops, is difficult to detect with a high degree of probability without producing many false positives. At a later date I will publish more of the source code for the function.

The loop inside the function walks along an incoming string to determine if a character in the path might be a dot, dot-dot, slash or backslash and if it is then applies canonicalization algorithms.

The irony of the bug is it occurs while calling a bounded function call:

```
_tcscpy_s(previousLastSlash, pBufferEnd - previousLastSlash, ptr + 2);
```

This function is a macro that expands to `wcscpy_s(dest, len, source);` technically, the bug is not in the call to `wcscpy_s`, but it’s in the way the arguments are calculated. As I alluded to, all three arguments are highly dynamic and constantly updated within the while() loop. There is a great deal of pointer arithmetic in this loop. Without going into all the gory attack details, given a specific path, and after the while() loop has been passed through a few times, the pointer, previousLastSlash, gets clobbered.

In my opinion, hand reviewing this code and successfully finding this bug would require a great deal of skill and luck. So what about tools? It’s very difficult to design an algorithm which can analyze C or C++ code for
The Worm That Ate the Web

The latest version of Conficker isn’t the first bot to plague the Internet, but it may be the smartest and most sophisticated. And it starts phoning home Wednesday.

By Farhad Manjoo
Posted Monday, March 30, 2009, at 5:20 PM ET

Last week, I pulled out my Internet cable, unplugged my USB drives, and searched my Windows machine for Conficker, the astounding computer worm that threatens to wreak global havoc once its latest version begins to phone home for further instructions on April 1. Well, maybe: While security researchers warn that the worm’s creators may be planning on conducting fraud or even “information warfare” aimed at disrupting the Internet, nobody knows what terrible deed Conficker will ultimately pull off.

What we do know is that Conficker is devilishly smart, terrifyingly contagious, and evolving. Each time experts discover a way to constrain its spread, its creators release new, more sophisticated versions that can push even further. The latest version, Conficker C, hit the Internet early in March. Estimates aren’t precise, but researchers say the worm—in all its variants—has so far infected more than 10 million machines around the world.
I've got a bunch of virtual windows machines networked together, hooked up to an incoming pipe from the net. They execute email attachments, share files, and have no security patches.

Between them they have practically every virus.

There are mail trojans, Warhol worms, and all sorts of exotic polymorphics. A monitoring system adds and wipes machines at random. The display shows the viruses as they move through the network, growing and struggling.

You know, normal people just have aquariums.

Pretty, isn't it?

What is it?

Good morning, Blaster. Are you and W32.Welchia getting along?

Who's a good virus? You are! Yes, you are!
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

User Meg wants these 6 letters: POTATO.
Server, are you still there? If so, reply "HAT" (500 letters).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHReSct". User Bobber requests pages.
EXTRA SLIDES,
ILLUSTRATION ONLY
Example: Amazon Public Cloud (AWS)
AWS Identity and Access Management (IAM)

- AWS users are associated with the organizational account of a customer of AWS.
- AWS objects are the various resources defined by AWS:
  - Virtual machine instances, virtual files (S3 Simple Storage Service buckets, ...), virtual networks, databases, ...
- Each service or resource defines an API with a list of named actions.
  - Create a VM instance or S3 object
  - Read/write an S3 bucket
- When a subject requests an action on a service or object, the AWS service checks to determine if the subject has a named permission required for the requested action.
- Permission statements (policies) are given in JSON documents specifying permissions for specified subjects and objects.
AWS Free Tier

Launch new applications, test existing applications in the cloud, or simply gain hands-on experience with AWS.

Get started with AWS for Free »

Get Started for Free » Launch virtual machines and apps in minutes.

WHAT IS CLOUD COMPUTING?
Learn the benefits of Cloud Computing with AWS

WHAT IS AMAZON WEB SERVICES?
Learn about the AWS platform, products and services

GET STARTED WITH AWS
Start using AWS in under 15 minutes
An AWS customer creates an account with a payment method.

The account holder/administrator for an organization may define many users, groups, and/or roles linked to the account.

Policies may be attached to users, groups, roles, or resources: ACLs (“resource-based policies”) or capabilities (“user-based policies”).

Users, groups, roles, and resources may have pathnames in a name hierarchy, like the Unix FS: Amazon Resource Names (ARNs).
IAM: a few interesting details

- IAM roles are distinct from IAM groups. To take on the powers of a role, a user (or its VM instances) must explicitly request to assume the role. If successful, assume returns new temporary security credentials (keys) to use when acting in the role.

- The API to assume a role checks access in the usual fashion. There are interesting special cases.
  - A user may have permission to launch an instance that assumes a role.
  - A user from another account may have permission to assume a role.

- Policy statements use ARNs to name users, groups, roles, objects, and permissions. Policy docs may use wildcarding in ARNs to name collections of subjects, objects, or permissions.