DATA SECURITY AND PRIVACY

Week 2: Weakness of DAC

PURDUE
UNIVERSITY®

Readings for this lecture

- Seeley: “A Tour of the Worm”. In Proc. Winter Usenix Conf., February 1989.
  - https://collections.lib.utah.edu/details?id=702918
- Hardy: “Confused Deputy.” ACM SIGOPS Operating Systems Review. Oct. 1988
  - https://dl.acm.org/doi/10.1145/54289.871709
- Miller et al. “Capability Myths Demolished”
  - https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf
- Mao et al. “Combining Discretionary Policy with Mandatory Information Flow in Operating Systems” ACM TISSEC, November 2011.
  - https://dl.acm.org/doi/10.1145/2043621.2043624
  - Reading the introduction is sufficient

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Analysis of DAC Weaknesses
  - Confused deputy
  - DAC’s implicit trust in programs being benign and correct
- Sandboxing/virtualization/isolation approaches
- Create access control policies depend on programs

What is a Worm?

- What is a worm?
  - Self-propagating malware

- Three steps
  - Find targets
  - Compromise target
  - Copy itself and execute

Morris Worm (November 1988)

- First major internet worm
- Written by Robert Morris Jr.
  - Son of former chief scientist of NSA's National Computer Security Center

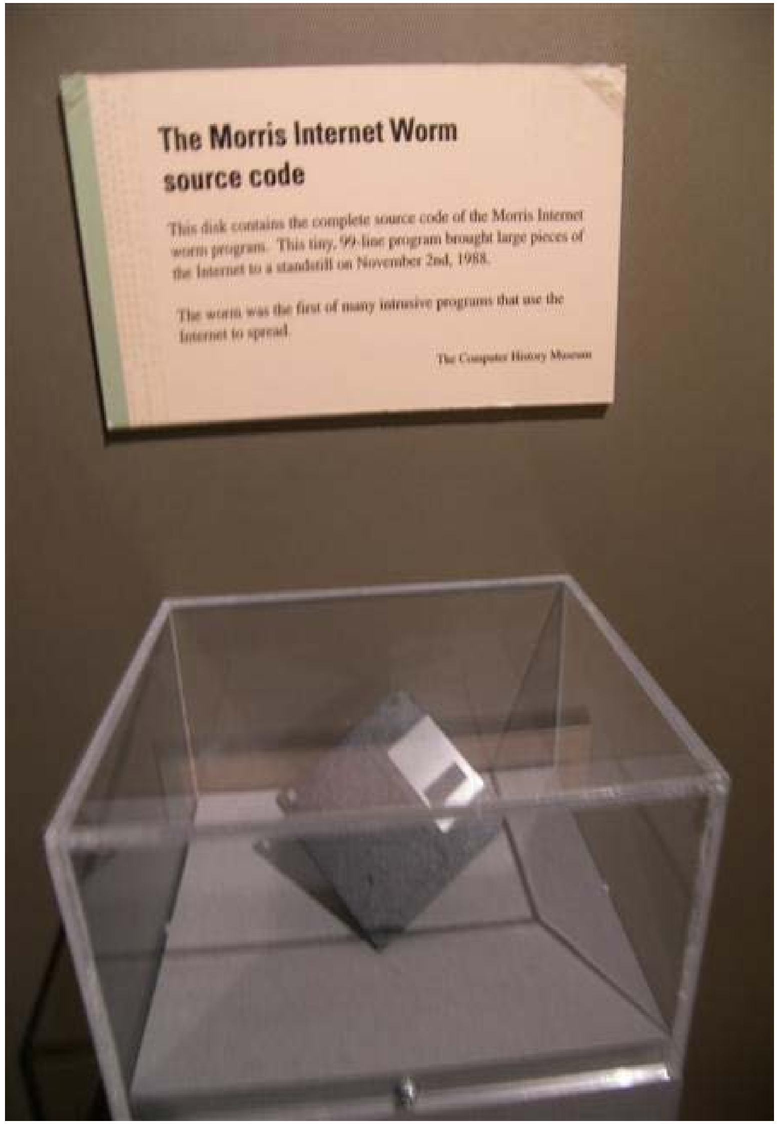

The Morris Internet Worm
source code
This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.
The worm was the first of many intrusive programs that use the Internet to spread.
The Computer History Museum

Image from wiki   Under CC BY_SA 2.0

Morris Worm Description

- Two parts
  - Main program to spread worm
    - look for other machines that could be infected
    - try to find ways of infiltrating these machines
  - Vector program (99 lines of C)
    - compiled and run on the infected machines
    - transferred main program to continue attack

Vector 1: Debug feature of sendmail

- Sendmail
  - Listens on port 25 (SMTP port)
  - Some systems back then compiled it with DEBUG option on
- Debug feature gives
  - The ability to send a shell script and execute on the host

Vector 2: Exploiting fingerd

- What does finger do?
- Finger output
  - arthur.cs.purdue.edu% finger ninghui
  - Login name: ninghui   In real life: Ninghui Li
  - Directory: /homes/ninghui   Shell: /bin/csh
  - Since Jan 18 09:50:47 on pts/2 from pal-10-184-63-172.itap (4 seconds idle)
  - No unread mail.
  - No Plan.

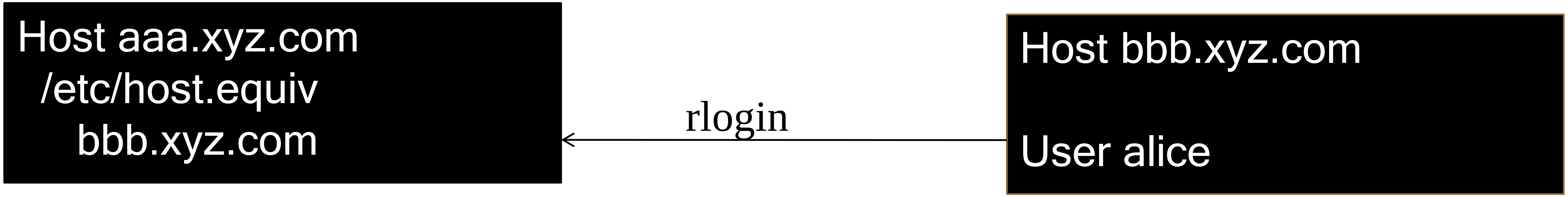
Vector 2: Exploiting fingerd

- Fingerd
  - Listen on port 79
- It uses the function char* gets(char *)
  - Fingerd expects an input string
  - Worm writes long string to internal 512-byte buffer
- Overrides return address to jump to shell code

Vector 3: Exploiting Trust in Remote Login

- Remote login on UNIX
  - rlogin, rsh

- Trusting mechanism
  - Trusted machines have the same user accounts
  - Users from trusted machines
  - /etc/host.equiv – system wide trusted hosts file
  - /.rhosts and ~/.rhosts – users' trusted hosts file



Host aaa.xyz.com
/etc/host.equiv
bbb.xyz.com

rlogin

Host bbb.xyz.com

User alice

Vector 3: Exploiting Trust in Remote Login

- Worm exploited trust information
  - Examining trusted hosts files
  - Assume reciprocal trust
    - If X trusts Y, then most likely Y trusts X
- Password cracking
  - Worm coming in through fingerd was running as daemon (not root) so needed to break into accounts to use .rhosts feature
  - Read /etc/passwd, used ~400 common password strings & local dictionary to do a dictionary attack

Other Features of The Worm

- Self-hiding
  - Program is shown as 'sh' when ps
  - Files didn’t show up in ls
- Find targets using several mechanisms:
  - 'netstat -r -n‘, /etc/hosts, …
- Compromise multiple hosts in parallel
  - When worm successfully connects, forks a child to continue the infection while the parent keeps trying new hosts
- Worm has no malicious payload
- **Where does the damage come from?**

Damage

- One host may be repeatedly compromised
- Supposedly designed to gauge the size of the Internet
- The following bug/feature made it more damaging.
    - Asks a host whether it is already running the Morris Worm; however, even if it answers yes, still compromise it with probability 1/8.

Review: How does a computer get compromised?

- Buggy programs accept malicious input
  - daemon programs that receive network traffic
  - client programs (e.g., web browser, mail client) that receive input data from network
  - buggy programs (e.g., pdf readers) read malicious files saved from the network
- Configuration errors (e.g., weak passwords, guest accounts, DEBUG options, etc)
- Human errors (e.g., leaking passwords due to social engineering attacks, executing malicious code such as email attachment, or downloading and executing trojan horses)
- Giving attacker physical access to computer

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Analysis of DAC Weaknesses
  - Confused deputy and capability system
  - DAC's implicit trust in programs being benign and correct
- Sandboxing/virtualization/isolation approaches
- Create access control policies depend on programs

Could Better Access Control Help Stop Morris Worm?

- Vector 1: Exploiting buffer overflow vulnerability in fingerd, and then take over the fingerd process to execute a malicious shell script
  - In UNIX access control, fingerd runs as a daemon user which can run shell and many other programs
  - If fingerd is prevented from running shell, then this attack would fail.
- Vector 2: Exploit DEBUG option
  - Cannot be stopped by access control.
- Vector 3: Exploit mutual trust
  - Cannot be stopped by access control, if the convenience is desired. This is an issue only when a host on a local network is compromised.

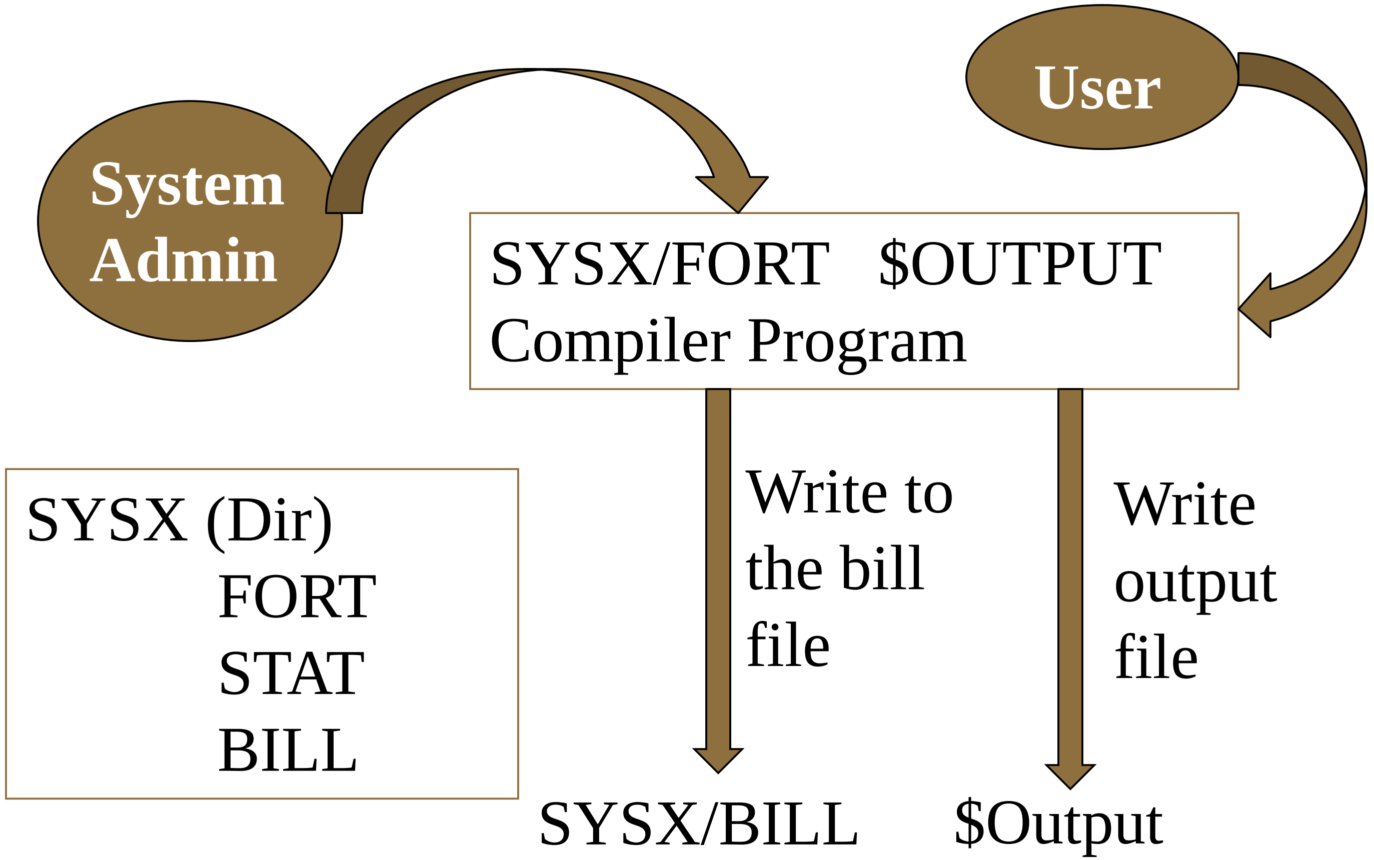
Discretionary Access Control

- No precise definition. Basically, DAC allows access rights to be propagated at subject’s discretion
  - often has the notion of owner of an object
  - used in UNIX, Windows, etc.
- According to TCSEC (Trusted Computer System Evaluation Criteria)
  - "A means of restricting access to objects based on the identity and need-to-know of users and/or groups to which they belong. Controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (directly or indirectly) to any other subject."
- Often compared to Mandatory Access Control

Analysis why DAC is not Good enough

- DAC causes the Confused Deputy problem
  - Solution: use capability-based systems
- DAC does not preserve confidentiality when facing Trojan horses
  - Solution: use Mandatory Access Control (BLP)
- DAC implementation fails to keep track of for which principals a subject (process) is acting on behalf of
  - Solution: fixing the DAC implementation to better keep track of principals
  - Solution: adding additional access control mechanism

The Confused Deputy Problem



System Admin

User

SYSX/FORT $OUTPUT
Compiler Program

SYSX (Dir)
FORT
STAT
BILL

Write to the bill file

Write output file

SYSX/BILL

$Output

The Confused Deputy by *Norm Hardy*

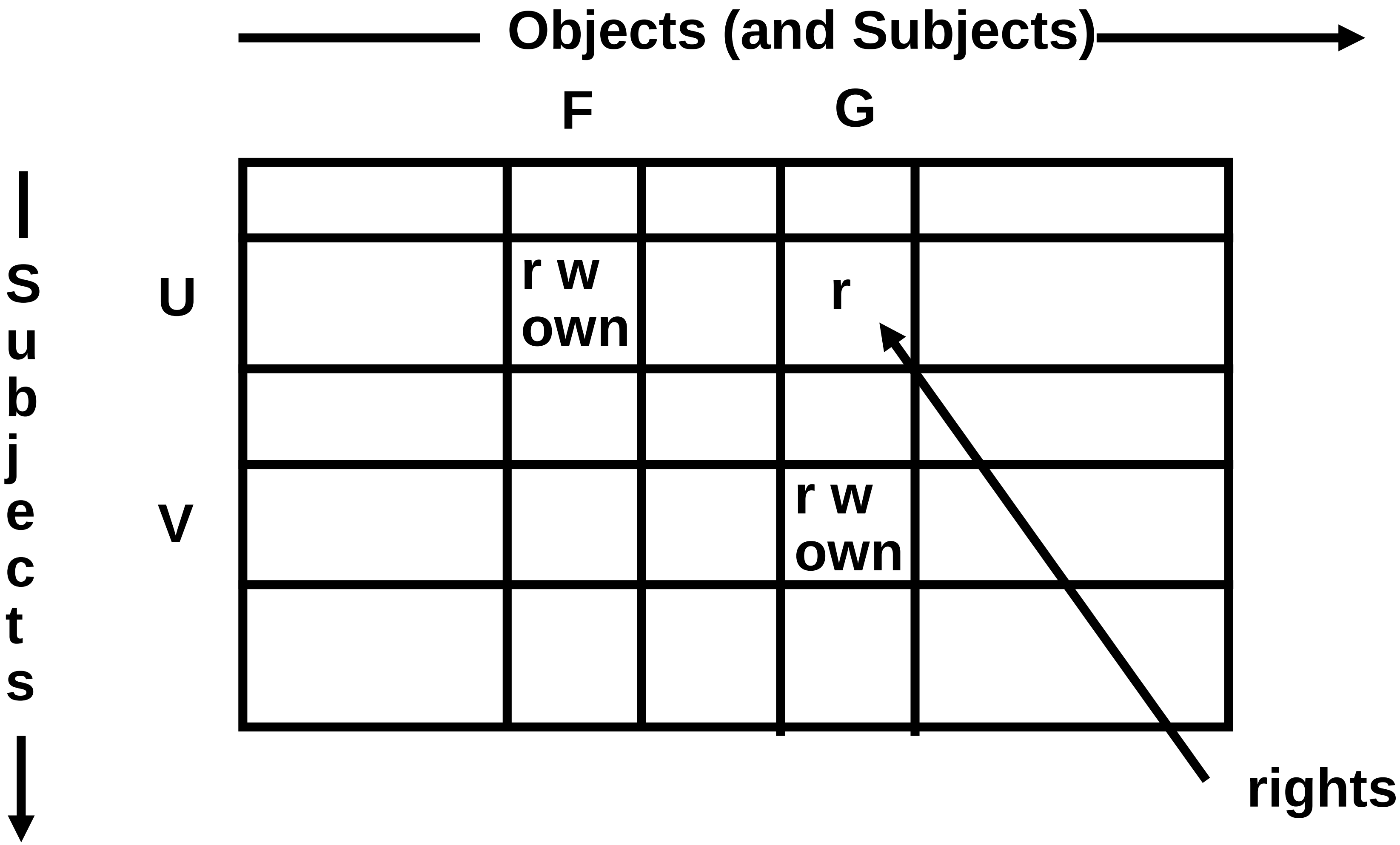
Analysis of The Confused Deputy Problem

- The compiler runs with authority from two sources
  - the invoker (i.e., the programmer)
  - the system admin (who installed the compiler and controls billing and other info)
- It is the deputy of two masters
- There is no way to tell which master the deputy is serving when performing a write
- Solution: Use capability

Different Notions of Capabilities

- Capabilities used in POSIX/Linux as a way to divide the root power into multiple pieces that can be given out separately
- Capabilities as a row representation of Access Matrices
- Capabilities as a way of implementing the whole access control systems
- We will examine the second and third notion next in this lecture

ACCESS MATRIX MODEL



Objects (and Subjects)

Subjects

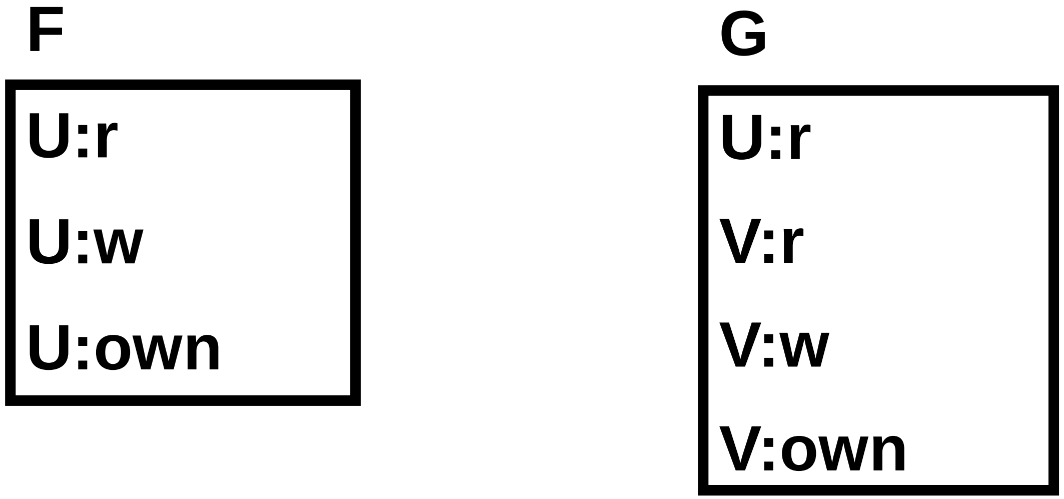
rights

		F		G	
U		r w own		r	
V				r w own	

IMPLEMENTATION OF AN ACCESS MATRIX

- Access Control Lists
  - Encode columns
- Capabilities
  - Encode rows
- Access control triples
  - Encode cells

ACCESS CONTROL LISTS (ACLs)



F

- U:r
- U:w
- U:own

G

- U:r
- V:r
- V:w
- V:own

each column of the access matrix is stored with the object corresponding to that column

CAPABILITY LISTS

U	F/r, F/w, F/own, G/r
V	G/r, G/w, G/own

each row of the access matrix is stored with the subject corresponding to that row

ACCESS CONTROL TRIPLES

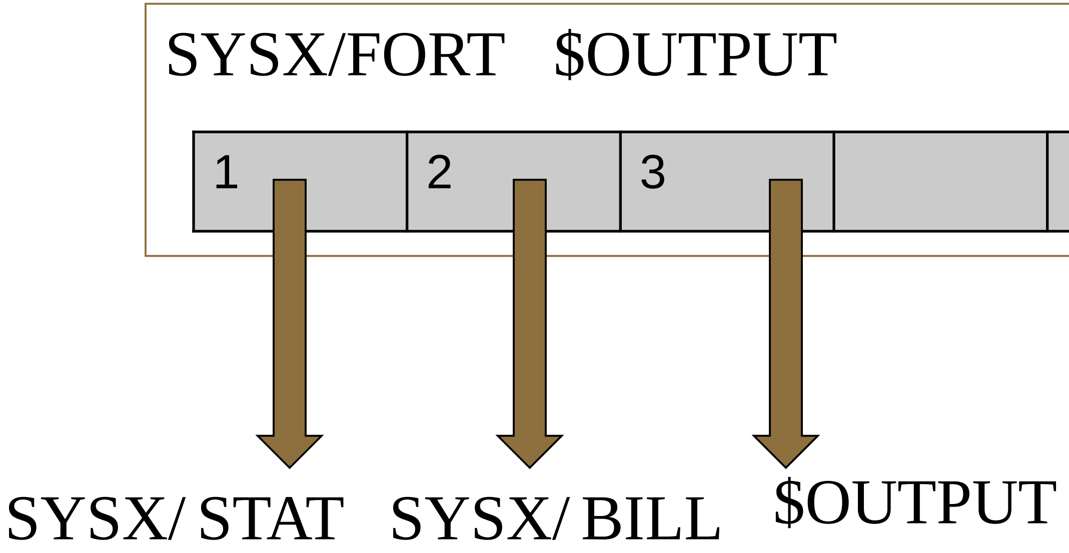
Subject	Access	Object
U	r	F
U	w	F
U	own	F
U	r	G
V	r	G
V	w	G
V	own	G

commonly used in relational DBMS

Capability Based Access Control

- Subjects have capabilities, which
  - Give them accesses to resources (similar to keys that can open doors)
  - Can be transferred to other subjects
  - Are unforgeable tokens of authority
- Example: a UNIX system where only owner of a file can open the file, and file sharing is done by passing opened file descriptors around
- Why capabilities may solve the confused deputy problems?
  - When access a resource, must select a capability, which also selects a master

How Do Capabilities Solve the Confused Deputy Problem



- Invoker must pass in a capability for $OUTPUT, which is stored in slot 3.
- Writing to output uses the capability in slot 3.
- Invoker cannot pass a capability it doesn’t have.

Capability vs. ACL

- Consider two security mechanisms for bank accounts.
- One is identity-based. Each account has multiple authorized owners. You go into the bank and shows your ID, then you can access all accounts you are authorized.
  - Once you show ID, you can access all accounts.
  - You have to tell the bank which account to take money from.
- The other is token-based. When opening an account, you get a passport to that account and a PIN, whoever has the passport and the PIN can access

Capabilities vs. ACL: Ambient Authority

- Ambient authority means that a user's authority is automatically exercised, without the need of being selected.
  - Causes the confused deputy problem
  - Violates the least privilege principle
- No Ambient Authority in capability systems

Capability vs. ACL: Naming

- ACL systems need a namespace for objects
- In capability systems, a capability can serve both to designate a resource and to provide authority.
- ACLs also need a namespace for subjects or principals
  - as they need to refer to subjects or principals
- Implications
  - the set of subjects cannot be too many or too dynamic
  - most ACL systems grant rights to user accounts principals, and do not support fine-grained subject rights management

Conjectures on Why Capability-based AC is Rarely Used

- Capability is more suitable for process level sharing, but not user-level sharing
  - user-level sharing is what is really needed
- Processes are more tightly coupled in capability-based systems because the need to pass capabilities around
  - programming may be more difficult

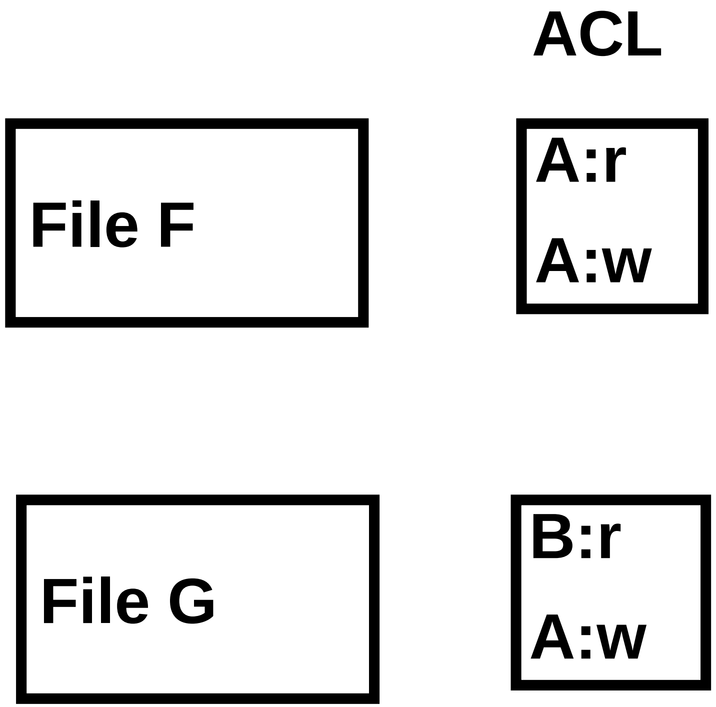
Analysis why DAC is not Good enough

- DAC causes the Confused Deputy problem
  - Solution: use capability-based systems
- DAC does not preserve confidentiality when facing Trojan horses
  - Solution: use Mandatory Access Control (BLP)
- DAC implementation fails to keep track of for which principals a subject (process) is acting on behalf of
  - Solution: fixing the DAC implementation to better keep track of principals
  - Solution: adding additional access control mechanism

Weakness OF DAC in Information Flow Control

- Unrestricted DAC allows information flows from an object which can be read to any other object which can be written by a subject
  - Suppose A is allowed to read some information and B is not, A can reads and tells B
- Suppose that users are trusted not to do this deliberately. It is still possible for Trojan Horses to copy information from one object to another.

TROJAN HORSE EXAMPLE



ACL

File F

A:r

A:w

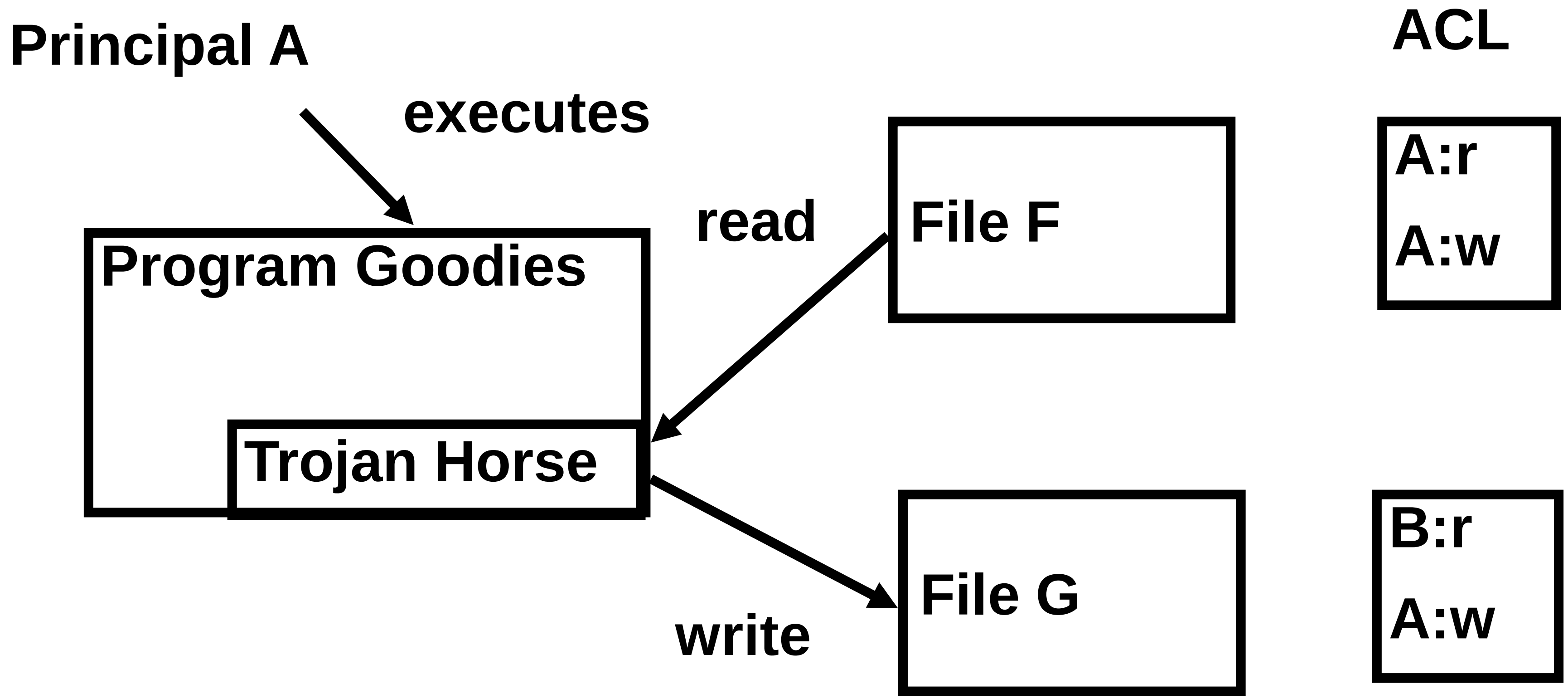
File G

B:r

A:w

Principal B cannot read file F

TROJAN HORSE EXAMPLE



Principal B can read contents of file F copied to file G

Buggy Software Can Become Trojan Horse

- When a buggy software is exploited, it execute the code/intention of the attacker, while using the privileges of the user who started it.
- This means that computers with only DAC cannot be trusted to process information classified at different levels
  - Mandatory Access Control is developed to address this problem
  - We will cover this in the next topic

Analysis why DAC is not Good enough

- DAC causes the Confused Deputy problem
  - Solution: use capability-based systems
- DAC does not preserve confidentiality when facing Trojan horses
  - Solution: use Mandatory Access Control (BLP)
- DAC implementation fails to keep track of for which principals a subject (process) is acting on behalf of
  - Solution: fixing the DAC implementation to better keep track of principals
  - Solution: adding additional access control mechanism

DAC's Weaknesses Caused by The Gap

- A request: a subject wants to perform an action
  - E.g., processes in OS
- The policy: each principal has a set of privileges
  - E.g., user accounts in OS
- Challenging to fill the gap between the subjects and the principals
  - relate the subject to the principals

Unix DAC Revisited (1)

Action	Process	Effective UID	Real Principals
User A Logs In	shell	User A	User A
Load Binary “Goodie” Controlled by user B	Goodie	User A	? ?

- When the Goodie process issues a request, what principal(s) is/are responsible for the request?
- Under what assumption, it is correct to say that User A is responsible for the request?

- Assumption: Programs are benign, i.e., they only do what they are told to do.

UNIX DAC Revisited (2)

Action	Process	Effective UID	Real Principals
	shell	User A	User A
Load AcroBat Reader Binary	AcroBat	User A	User A
Read File Downloaded from Network	AcroBat	User A	? ?

- When the AcroBat process (after reading the file) issues a request, which principal(s) is/are responsible for the request?
- Under what assumption, it is correct to say that User A is responsible for the request?

- **Assumption: Programs are correct, i.e., they handle inputs correctly.**

Why DAC is vulnerable?

- Implicit assumptions
  - Software are benign, i.e., behave as intended
  - Software are correct, i.e., bug-free
- The reality
  - Malware are popular
  - Software are vulnerable
- Arguably the problem is not caused by the discretionary nature of policy specification!
  - i.e., owners can set policies for files

Why DAC is Vulnerable? (cont')

- A limitation in the enforcement mechanism
  - UNIX DAC maintains a single principal (euid) for a subject/process; this is not enough to capture on whose behalf the process is acting
- When the program is a Trojan
  - The program-provider should also be responsible for the requests
- When the program is vulnerable
  - It may be exploited by input-providers
  - The requests may be issued by injected code from input-providers
- Solution: accept that a subject may be acting on behalf of multiple principals, and that we are uncertain.

Proposals to Radically Change DAC

- DAC causes the Confused Deputy problem
  - Solution: use capability-based systems
- DAC does not preserve confidentiality when facing Trojan horses
  - Solution: use Mandatory Access Control, e.g., BLP
- DAC implementation fails to keep track of for which principals a subject (process) is acting on behalf of
  - Solution: UMIP and IFEDAC
- None of these is widely used in commercial systems

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Analysis of DAC Weaknesses
  - Confused deputy
  - DAC’s implicit trust in programs being benign and correct
- Sandboxing/virtualization/isolation approaches
- Create access control policies depend on programs

Goal of Sandxboing/virtualization/Isolation

- Sandboxing: Separate running programs, to mitigate system failures and/or software vulnerabilities
- Ensure that a program, even if compromised, causes only limited damage.

Confinement by Virtualization (Option 1)

- Runs a single kernel, virtualizes servers on one operating system using built-in mechanism
  - e.g., chroot, FreeBSD jail, …
  - used by service providers who want to provide low-cost hosting services to customers.
  - Pros: little performance overhead, easy to set up/administer
  - Cons: some confinement can be broken, some servers cannot be easily confined

chroot

- The chroot system call **ch**anges the **root** directory of the current and all child processes to the given path.
- To use chroot,
  - One first creates a temporary root directory for a running process,
  - Then takes a limited hierarchy of a filesystem (say, /chroot/named) and making this the top of the directory tree as seen by the application.
  - Make the chroot system call: a network daemon program can call chroot itself, or a script can call chroot and then start the daemon

Using chroot

- What are the security benefits?
  - under the new root, many system utilities and resources do not exist, even if the attacker compromises the process, damage can be limited
  - **consider the Morris worm, how would using chroot for fingerd affect its propagation?**

Limitations of chroot

- Only the root user can perform a chroot.
  - intended to prevent users from putting a setuid program inside a specially-crafted chroot jail (for example, with a fake /etc/passwd file) that would fool it into giving out privileges.
- chroot is not entirely secure on all systems.
  - With root privilege inside chroot environment, it is sometimes possible to break out
- process inside chroot environment can still see/affect all other processes and networking spaces
- chroot does not restrict the use of resources like I/O, bandwidth, disk space or CPU time.

Confinement by Virtualization (Option 2)

- Virtual machines: emulate hardware in a user-space process
  - the emulation software runs on a host OS; guest OSes run in the emulation software
  - needs to do binary analysis/change on the fly
  - e.g., Oracle VirtualBox, VMWare,
  - Pros: can run other guest OS without modification to the OS
  - Cons: significant performance overhead

Limitation of Confinement by Virtualization

- Pro. Policy is simple: just isolate each instance

- Con. Things within one virtual machine can still affect each other.

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Analysis of DAC Weaknesses
  - Confused deputy
  - DAC's implicit trust in programs being benign and correct
- Sandboxing/virtualization/isolation approaches
- Create access control policies depend on programs

Program-Based Access Control

- For each process, there is an additional policy limiting what it can do, which is based on the binary file
  - E.g., what system call it can make, what files it can access, et.c
  - This is in addition to the DAC restriction based on the user ids

- The key challenge
  - how to specify the policy

Examples of Program-Based Policies Access Control

- Security Enhanced Linux (SELinux)
  - Developed by National Security Agency (NSA) and Secure Computing Corporation (SCC) to promote MAC technologies
  - Shipped with Fedora and some other Linux distributions
  - Also part of Android as Security Enhanced Android
- AppArmor
  - Shipped in Debian, Ubuntu, OpenSUSE Linux distributions

Main Idea of SElinux

- Consider more information (especially which program is running) when making access control decisions
- Enable fine-grain control
- Support flexible security policies, “user friendly” security language (syntax)
  - Overall policy is extremely complex

Policy: Domain-type Enforcement

- The access matrix consisting of subjects and objects is too large and impractical.
- To reduce the size of the access matrix, subjects are grouped into domains, objects are grouped into types.
- A smaller (but still big) access matrix with domains and types can then be specified.

Policy: Domain-type Enforcement

- Each object is labeled by a type
  - Example:
  - /etc/shadow   etc_t
  - /etc/rc.d/init.d/httpd   httpd_script_exec_t
- Objects are grouped by object security classes
  - Files, sockets, IPC channels, capabilities
  - Operations are defined upon each security class
- Each subject (process) is associated with a domain
  - E.g., httpd_t, sshd_t, sendmail_t

Policy: Domain-type Enforcement

- Access control decision
  - When a process wants to access an object, the decision is based on process domain, object type, object security class, type of operation
- Example access vector rules
  - allow sshd_t sshd_exec_t: file { read execute entrypoint }
  - allow sshd_t sshd_tmp_t: file { create read write getattr setattr link unlink rename }

Policy: Domain-type Enforcement

- How the domain if a new process is determined?
  - The domain for a new process is based on the domain of the parent process and the label for the executable binary
- How the type of a new file is determined?
  - Based on the domain of the creating process and the parent directory
- TE transition rules
  - type_transition initrc_t sshd_exec_t: process sshd_t
  - type_transition sshd_t tmp_t: notdevfile_class_set sshd_tmp_t

SELinux in Practice

- Strict policy
  - A system where everything is denied by default.
  - Minimal privilege's for every daemon
  - Separate user domains for programs like GPG,X, ssh, etc
  - Difficult to enforce in general purpose operating systems
  - Default in Fedora Core 2
  - #1 Question: How do I turn off SELinux

- Targeted policy
  - System where everything is allowed. use deny rules.
  - Only restrict certain daemon programs
  - Default in Fedora Core 3
  - No protection for client programs

AppArmor

- Provide a sufficiently fine-grained mechanism
- Try to achieve least privilege for programs
- For each program one wants to confine, one provides a profile, which specifies the activities the program can perform
  - Files, Operations

Example Profile

```
#include <tunables/global>

# a comment naming the
application to confine
/usr/bin/foo
{
   #include <abstractions/base>

   capability setgid,
   network inet tcp,

   /bin/mount          ux,
   /dev/{,u}random     r,
   /etc/ld.so.cache    r,
   /etc/foo.conf       r,
   /etc/foo/*          r,
   /lib/ld-*.so*       mr,
 /lib/lib*.so*       mr,
   /proc/[0-9]**       r,
   /usr/lib/**         mr,

/tmp/                r,
   /tmp/foo.pid        wr,
   /tmp/foo.*          lrw,
   /@{HOME}/.foo_file  rw,
   /@{HOME}/.foo_lock  kw,

  # a comment about foo's subprofile,
     bar.
   ^bar {
    /lib/ld-*.so*      mr,
    /usr/bin/bar       px,
    /var/spool/*       rwl,
   }
}
```

Summary

- Buggy programs can be exploited
- Existing DAC mechanisms allow exploited programs to control a whole system
- Existing DAC has some fundamental weaknesses
  - Attempts to fix them have their own limitations and are not widely deployed
- Additional access control can help at the cost of the need to specify additional policies

Next Topic

- Multi-level Security (MLS) and Bell-La Padula Model
- Biba Integrity Model, Clark-Wilson Model, and Chinese Wall Policy