Data Security and Privacy

Topic 3: Operating System Access Control Enhancement

Readings for this lecture

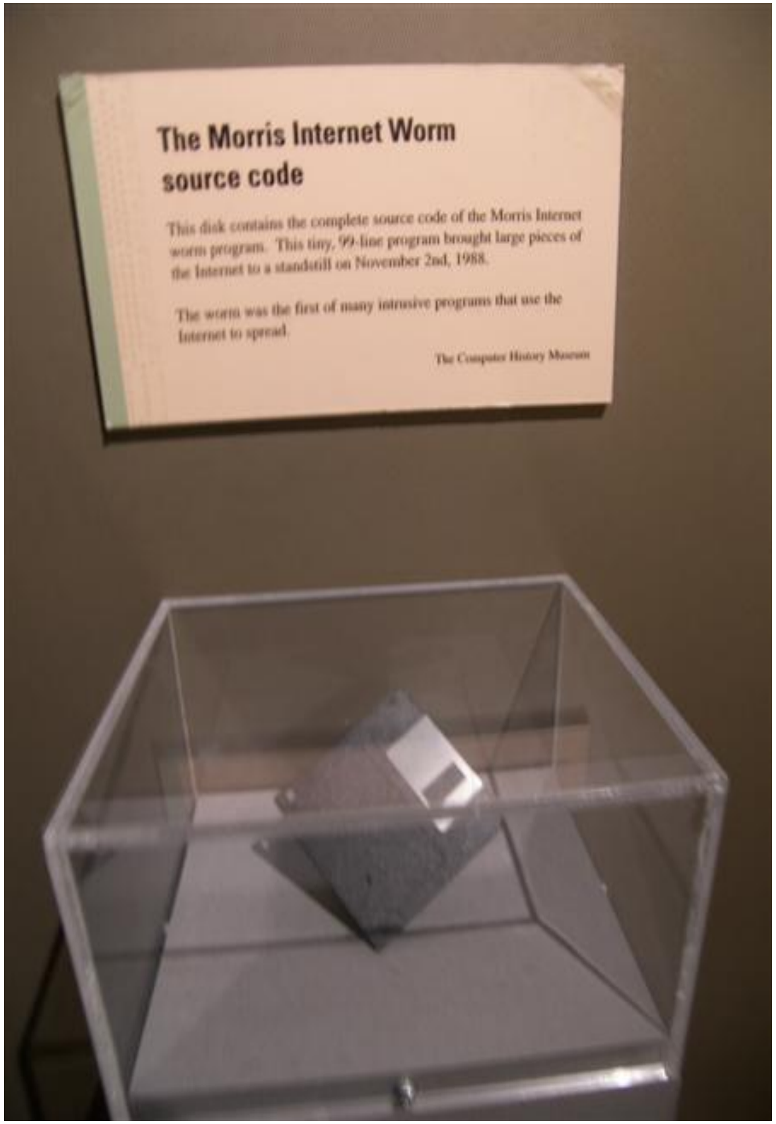
- Readings
  - On Trusting Trust
  - wikipedia topics: Operating system-level virtualization, Paravirtualization, Full virtualization

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Virtualization/isolation approaches
- Create access control policies depend on programs

Morris Worm (November 1988)

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



What comes next: *1 11 21 1211 111221?*

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 Sep 28 14:36:12 on pts/15 from csdhcp-120-173 (9 seconds idle)

New mail received Tue Sep 28 14:36:04 2010;

unread since Tue Sep 28 14:36:05 2010

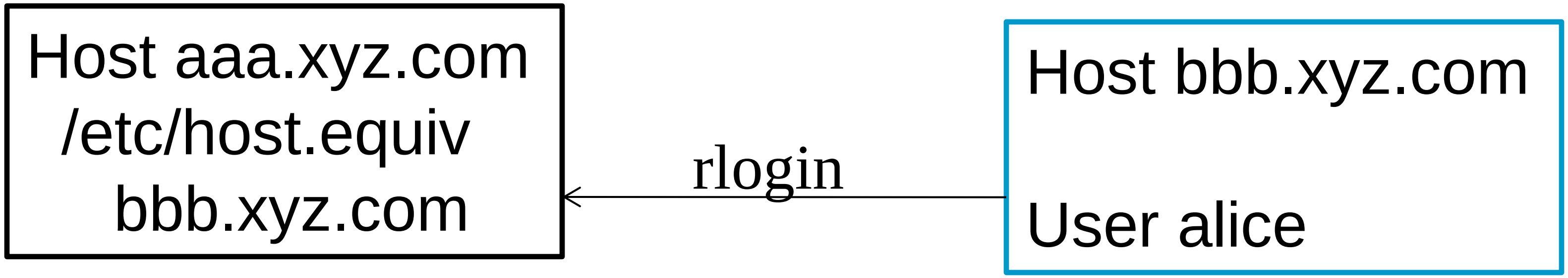
No Plan.

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



Vector 3: Exploiting Trust in Remote Login

- Worm exploited trust information
  - Examining trusted hosts files
  - Assume reciprocal trust
    - If X trusts Y, then maybe 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 made it more damaging.
  - Asks a host whether it is compromised; however, even if it answers yes, still compromise it with probability 1/8.

How does a computer get infected with malware or being intruded?

- Executes malicious code via user actions (email attachment, download and execute trojan horses)
- 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
  - Programs Read malicious files with buggy file reader program
- Configuration errors (e.g., weak passwords, guest accounts, DEBUG options, etc)
- Physical access to computer

Why is UNIX DAC insufficient?

- UNIX DAC is based on users.
- When attacker exploits the bug in a program and takes over a program, it gets the privileges of the user on whose behalf the program executes.
- UNIX DAC cannot different between benign and malicious processes.

Defense

- Remove bugs from software
- Make bugs not exploitable
  - reactive, many mechanisms, none perfect
- Make sure users do not make mistakes
- Make system withstand exploitable buggy software and malicious software by additional access control
  - Confinement by virtualization
  - Add access control policies that are based on programs

Outline

- Morris Worm as an example to illustrate the limitation of UNIX DAC protection
- Virtualization/isolation approaches
- Create access control policies depend on programs

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: best performance, easy to set up/administer
  - Cons: all servers are same OS, some confinement can be broken

chroot

- The chroot system call **ch**anges the **root** directory of the current and all child processes to the given path.
- Using chroot
  - creates a temporary root directory for a running process,
  - takes a limited hierarchy of a filesystem (say, /chroot/named) and making this the top of the directory tree as seen by the application.
  - 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?
- Examples of using chroot
  – ftp for anonymous user
- How to set up chroot?
  – need to set up the necessary library files, system utilities, etc., in the new environment

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., VMWare, Microsoft Virtual PC

  - Pros: can run other guest OS without modification to the OS
  - Cons: worst performance

Confinement by Virtualization (Option 3)

- Paravirtualization
  - No host OS, a small Virtual Machine Monitor runs on hardware, guest OSes need to be modified to run
  - Requires operating systems to be ported to run
  - e.g., Xen

  - Pros: better performance compared with (2), supports more OSes compared with (1)
  - Cons: each guest OS must be modified to run on it, (each new version of the OS needs to be patched)

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
- 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

Example systems of Program-Based Policies Access Control

- Systrace
  - Create system call policies for programs
  - http://www.citi.umich.edu/u/provos/systrace/
- Security Enhanced Linux (SELinux)
  - initially developed by people in NSA
  - shipped with Fedora and some other Linux distributions
  - Also part of Android as Security Enhanced Android
- AppArmor
  - shipped with SUSE Linux distributions

Systrace Overview

- Sandbox an application that could potentially be controlled by an attacker
  - E.g., a web server, an ftp server,
- Implemented by system call interposition
- Systrace constrains an application's access to the system by specifying and enforcing system call policies for programs
  - One can create one or more policies for each program,
  - When using exec, one can specify which policy to apply.

Syscall: An Example Policy

```
Policy: /bin/ls, Emulation: native
   native-munmap: permit
[...]
   native-stat: permit
   native-fsread: filename match "/usr/*" then permit
   native-fsread: filename eq "/tmp" then permit
   native-fsread: filename eq "/etc" then deny[enotdir]
   native-fchdir: permit
   native-fstat: permit
   native-fcntl: permit
[...]
   native-close: permit
   native-write: permit
   native-exit: permit
```

Systrace Policy Generation

- Systrace notifies the user about all system calls that an application tries to execute. The user configures a policy for the specific system call that caused the warning. After a few minutes, a policy is generated that allows the application to run without any warnings. However, events that are not covered still generate a warning. Normally, that is an indication of a security problem.

SELinux

- Developed by National Security Agency (NSA) and Secure Computing Corporation (SCC) to promote MAC technologies
- MAC functionality is provided through the FLASK architecture
- Can be applied to Unix-like operating systems, such as Linux and BSD
- Available as a patch for 2.4 kernels
- Integrated into 2.6 kernels

FLASK

- **Fl**ux **A**dvanced **S**ecurity **K**ernel
- General MAC architecture
- Supports flexible security policies, "user friendly" security language (syntax)
- Separates policies from enforcement
- Contains a Security Server and Object Managers

- Idea
  - Consider more information when making access control decisions
  - Give fine-grain control
  - Should an apache server load a kernel module?

Policy: Domain-type Enforcement

- Each object is labeled by a type
  - Object semantics
  - 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
  - httpd_t
  - sshd_t
  - sendmail_t

Policy: Domain-type Enforcement

- Access control decision
  - When a process wants to access an object
  - Process domain, object type, object security class, operation
- 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 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

SubDomain (AppArmor)

- Provide a sufficiently fine-grained mechanism
- Try to achieve least privilege for programs

- Administrators specify the *domain* of 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,
   }
}
```

Sub-process confinement

- Scriptable servers, Loadable modules, Plug-ins
- Provide a system call: change_hat()
- Like sandboxing
- The developer should make appropriate calls

Compatibility

- Who write the profile?
  - Vendors
  - Administrators
- Which programs need to be confined?
  - Policy
  - All programs
  - All listed user-ids
  - All root programs
  - Only specified programs
  - All network programs
- How to generate the profile?
  - Run, log, grant
  - Tool: dep, strace

Next Topic

- Limitation of DAC: Theoretical Analysis