System Security

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

alessandro.barenghi - at - polimi.it

May 13, 2014

Lesson contents

Overview

- The *nix user control and file permission model
- Linux Capabilites based permission control
- Secure programming practices

How to train your users

Overview

- Linux[a] are natively conceived as a multi-user operating system
- Multiple users will naturally perform different task, and have different needs on the machine
- To avoid abuses, the user access to the hardware should be somehow restricted
- It should also be possible to cluster users in groups, in order to ease the issuing of permissions

[a] and more in general all the Unices

Access to the system

Login and credential storage

- The basic Linux authentication and user control method relies on assigning a unique numeric User IDentifier to every user
- The UIDs, together with some information about the users are stored in `/etc/passwd`
- For safety reasons, the actual user passwords, processed through a one way function[a] are preserved in `/etc/shadow`
- At login, the system checks if the user exists, hashes the passwords and matches it against the correct hash and executes the preferred command interpreter for the user
- For large system it is possible to use a database backend to store the credentials together with a centralized login

[a] as noone, in his sane mind should store cleartext passwords

Processes

Overview

- Every process in modern Unices has three UID identifying on behalf of whom it acts:
  - Real UID: the UID of the user who launched the program
  - Effective UID: the UID which is used by the kernel for access control
  - Saved UID: an extra stored UID which can be swapped with the current Effective UID
- Every process in Linux also has an FSUID, which acts as a separate UID for file access
- Usually the FSUID follows the EUID, unless explicitly set
- Analogously to UIDs, 4 group IDs are attached to each process

File permissions

Overview

- As we recall from the basic system administration lessons, under Unix everything is a file
- It is thus natural to have the access permissions set on every file
- Every file has both an owner and a group to which it belongs
- The basic permissions to act on a file are stored for both the owner and the group

File permissions

the User-Group-Other model

- Three access check are made under Unix : the permission to read (`r`), write (`w`) and execute a file (`x`)
- The permission bits are usually represented visually by 9 characters `rwxrwxrwx`[a]
- The first 3 (`rwx`rwxrwx) are the permissions for the owner
- The second 3 (rwx`rwx`rwx) are the permissions for the members of the same group which owns the file
- The last 3 (rwxrwx`rwx`) are the permissions for the others
- There are 3 extra permission bits, which are needed to mark the set-user ID, set-group ID and stickyness

[a] The extra character at the beginning of the line `ls` provides is a marker for the file type

File permissions

Changing permissions and ownerships

- File Permissions can be changed via the `chmod` command
- `chmod` accepts either a 4-digit octal representation of the permissions or a list of the permissions to be added-revoked
- Revoking the execute permission on a directory forbids its traversal
- The `chown` command allows to change the user/group owning a file: syntax `chmod <user>:<group> filename`
- A special permission, the *sticky bit* allows only the owner of a file to remove it (even if others have full access)

Saved IDs

Set user ID and set group ID

- Executable programs are started with RUID and EUID equal to the user who launched them
- If the set user ID / set group ID permission bits are set on the executable file, the EUID/EGID will be equal to the owner of the file and not the caller
- This allows to execute programs as someone else, even without logging in as him/her
- Suid and sgid bits can be changed via chmod: they are the first two bits of an extra digit in octal
- You can find all of them via

```
find / -perm +6000 -type f -user <username>
```

The administrator

Got `root`?

- Among all the users, one is special: the system administrator
- The system administrator, by default named root, has UID 0
- *Any* permission check will be skipped if the issuer of an action has UID 0: security checks are enclosed in an `if(UID){...}`
- This implies that any process with EUID 0 can read/write/run anything
- Having an EUID of 0 allows a process to modify any access credentials without knowing the previous ones

A programmer point of view

Linux API for UNIX permissions

- The `getuid` and `geteuid` function return respectively the RUID and EUID
- `setuid` tries to set all the UIDs `seteuid` only the EUID
- The `chmod` and `chown` functions act exactly as the commands
- For chmod to work, either the EUID matches the owner of the file, or the program runs as `root` (i.e., EUID=0)
- The same concept applies to the `chown` function

Dropping privileges

Typical suid privilege dropping

The correct practice if you have a suid `root` program to be run is to drop the privileges as soon as you do not need them anymore:

State	Action	EUID	Real UID	Saved UID
Startup		0	user	0
temp. drop	seteuid(getuid())	user	user	0
restore	seteuid(0)	0	user	0
Perm. drop	setuid(getuid())	user	user	user
restore	seteuid(0)	user	user	user

Linux Capabilities

When `root` is too much

- The power held by the administrator in UNIX is absolute, yielding it in a single block may be too much
- Typical example : why must my web server run as `root` when it just needs to open a privileged port?
- Solution : POSIX Capabilities : partition the administrative rights in a set of capabilities
- Affix some of the capabilites to an executable file, instead of setting it suid `root`

Linux Capabilities

Common capabilities

- The list of all system capabilities is available with `man capability`
- Useful capabilities are :
  - `CAP_NET_RAW` : allows to use raw sockets
  - `CAP_NET_ADMIN` : allows to change routing tables
  - `CAP_KILL` : unlocks signal sending to everyone
  - `CAP_SYS_NICE` : allows to renice with negative values
- You can retrieve the list of capabilities of a file with `getcap <filename>`
- Symmetrically, you can set the capabilities `setcap <capability>[+|-|=]ep <filename>`

Dropping privileges

Typical cap privilege dropping

- Active capabilities can be checked via the `cap_get_flag` function (returning true or false)
- Capabilities can be dropped easily via the `cap_set_flag` function
- Capability dropping is permanent for the current run of the program (no mechanism as saved caps)
- the whole capability set can be retrieved via a call to the `cap_get_proc` function

Key Ideas

Do not take anything for granted

- Secure programming implies checking that the inputs and the environment of a program are safe
- Capabilities or extra privileges granted should be discarded when not needed
- Error messages should be explanatory, but not too much
- Memory management errors are a possible source of endless issues
- The `printf` case

Hostile Environments

Mind user input

- User input may not just be "rotten", but actually intentionally poisoned
- It is possible to deviate the common behaviour of a program if it does not take any care in verifying that the input matches a sane format
- We will see two examples :
  - the `system` call which runs a command in a new instance of the command interpreter
  - the `exec` call family, which runs a command directly, replacing the code of the running process
- Remember that, when a call to `fork` or `exec` is issued, the value of EUID and EGID is preserved

Hostile Environments

Error notifications

- Our good nature pushes us towards using meaningful error messages to help the user
- However, the error messages have historically been a source of information leakage
- Every time an error message is displayed, some information about the state of the program leaks
- Sample issue: what if I help the user outputting the wrong line of a config file?
- Best practice : employ verbose error messages only enclosed in `DEBUG` macros and remove them afterwards

Hostile Environments

What's in a SEGFAULT? Wouldn't it be as dangerous as...

- Bad memory management and boundary checking looks as just a safety issue
- It turns out that performing memory copy operations without checking boundaries may allow the attacker to write anywhere in the memory segment of the program
- Since the stack is a commonly writeable area, it may happen that a part of the stack is overwritten during a `strcpy` operation
- ... but on the stack there's the return pointer of the actual call!

Hostile Environments

Proper memory management

- Arbitrary memory rewriting allows control flow hijacking: it's feasible and it's being done since 1996
- Always allocate enough space when you are copying strings (#char+1)
- Use only boundary checking copy functions (`strncpy`)
- Always check if memory allocations succeed, copying into a NULL pointer is also dangerous[a]

[a] NULL is defined as (void*) 0 , so if someone mmaps things to 0...

Hostile Environments

Format string issues

- At a first glance, the `printf` function looks *mostly* harmless
- However, the number of arguments it treats is implicitly indicated in the format string
- What if a `printf` does not have any parameters , and the user can specify a part of the string being printed?
- Reminder: among the format options allowed by C there is `%n`, which writes the number of printed character in a variable...