This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Sicurezza

Nell'ambito dei sistemi Unix-like

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

Introduzione

- Il s.o. costituisce le fondamenta di ogni applicativo: non possiamo costruire sistemi sicuri se quello che c'è sotto non lo è
  - Non possiamo realizzare s.o. sicuri se l'HW non lo è. . . etc etc
- Cosa vuol dire "sicuro"? Ne parlerete a *Computer Security* ma, ad altissimo livello, si parla di tre obiettivi:

Confidentiality	Confidenzialità/segretezza
Integrity	Integrità
Availability	Disponibilità

  in generale, *condivisione controllata*; per esempio, potremmo voler fare leggere un nostro file ad alcuni utenti ma non ad altri

Questa prima parte di slide è basata su
`http://pages.cs.wisc.edu/~remzi/OSTEP/security-intro.pdf` e
`http://pages.cs.wisc.edu/~remzi/OSTEP/security-authentication.pdf`

Contesto

Prima di eseguire una syscall il kernel deve valutare se la richiesta

1. è "sensata" (#-syscall e parametri validi)
2. rispetta la politica di sicurezza; terminologia:
   - l'entità che fa la richiesta è chiamato principal o subject
   - le richieste sono relative a una risorsa, l'object
   - e, tipicamente, una modalità di accesso; per esempio, lettura o scrittura

   nel valutare se eseguire, o meno, bisogna considerare il *contesto*; per esempio in alcuni casi tutti i processi avviati da un utente girano con gli stessi permessi, ma non sempre
   - Linux vs Android, per esempio

Un utente non parla *direttamente* con il kernel; servono:

- un'associazione fra utenti e processi
- un modo per verificare l'identità degli utenti
- ...

AAA

Authentication	Autenticazione: verifica dell'identità
Authorization	Autorizzazione: applicazione di una politica di sicurezza; decidere se accettare/rifiutare le richieste
Accounting	Contabilità: *logging* e gestione del consumo di risorse

Outline

Identificazione e autenticazione

Argomento molto vasto; caso più comune, nei sistemi Unix-like:

- gli utenti si identificano con uno username, e si autenticano con una password
  - file `/etc/passwd` e `/etc/shadow`
- il kernel identifica ogni utente con un numero intero: UID
  - analogamente, i gruppi con un GID; vedere `/etc/group`
  - da riga di comando: `id`
- ci concentreremo sugli UID, il discorso per i GID è analogo
- l'UID zero corrisponde a root, l'amministratore di sistema
  - attenzione: non confondete la radice del FS con l'utente amministratore
- tradizionalmente, i processi si dividono in

  privilegiati con UID=0, e
  non-privilegiati con UID≠0

  le *Linux capabilities* permettono una maggiore granularità, si veda `capabilities(7)`; ai fini di SETI, le ignoriamo per semplicità

Outline

Autorizzazione

Ci sono due approcci generali all'autorizzazione:

- access control list
  - per ogni *object*: lista delle coppie *subject/access*
  - Unix: versione "ottimizzata" (=ridotta) di ACL in 9 bit
    - r, w, x per proprietario, gruppo, altri
- *capabilities* (attenzione: non c'entrano nulla con le Linux capabilities)
  - per ogni object/access ci sono delle "chiavi", che ne permettono l'uso

L'idea generale è semplice, ma poi vanno considerati i dettagli:

- dove memorizzare queste informazioni/quanto spazio serve
- cosa bisogna aggiornare quando aggiungiamo/eliminiamo utenti
- ...

Questa parte di slide è basata su

`http://pages.cs.wisc.edu/~remzi/OSTEP/security-access.pdf`

MAC vs DAC

Indipendentemente dall'uso di ACL/capabilities, chi decide chi può/non-può accedere a una risorsa?

- Il proprietario, si parla di DAC: discretionary access control
- Una qualche autorità impone le regole (e.s. ambito militare), si parla di MAC: mandatory access control

I sistemi più comuni seguono il DAC, anche se non al 100%

- Per esempio, tradizionalmente root può leggere qualsiasi file
- Esistono varianti MAC di Linux; per esempio, `https://github.com/SELinuxProject`

Principio del minimo privilegio

Principio del minimo privilegio

In ogni momento, ciascuna entità dovrebbe avere il *minimo privilegio*, che gli permetta di eseguire i suoi compiti legittimi. Detto diversamente, nessun processo dovrebbe mai poter accedere a più risorse del minimo indispensabile per il suo corretto funzionamento.

In generale, è importante considerare il ruolo (in quel momento): lo stesso utente può avere diversi ruoli

In Unix:

- pensate a `login(1)`
- per aumentare i privilegi
  - storicamente: `su(1)`
  - oggi: `sudo(1)`

Come funzionano?

UIDs

- Ogni processo ha tre UID:
  - real UID — il proprietario del processo, chi può usare `kill`
  - effective UID — identità usata per determinare i permessi di accesso a risorse condivise; per e.s., ai file
  - saved UID
  - (solo Linux: FS UID, lo ignoriamo; si veda `credentials(7)`)

  al login, tutti e tre coincidono
- tramite `setuid(2)` un processo può modificare l'effective UID, facendolo diventare come il real o il saved
  - se il chiamante è privilegiato, può modificare tutti e tre come vuole
- Un nuovo processo "eredita" gli id del parent
- Di solito, `execve` non cambia gli id del processo chiamante
- Se il file eseguibile ha il bit set-userid abilitato, allora:
  - Effective UID e Saved UID diventano quelli del proprietario del file

Chroot jails e namespace

- `chroot(2)` è una syscall che permette di modificare il significato di "/" nella risoluzione dei percorsi assoluti
  - la modifica è per il processo e tutti i (futuri) figli
  - è necessario che il processo sia privilegiato
  - nota: i FD aperti non vengono toccati
- permette di creare le cosiddette chroot jail
  - limitando la visibilità del file-system a una directory (e sotto-directory)
  - è un'applicazione del principio del minimo-privilegio *ma...*
  - se il processo rimane privilegiato e/o la syscall non è correttamente associata a una `chdir(2)`, è possibile "uscire di prigione"
- per esempio, anche in caso di bug, in incApache un client non può "sbirciare" fuori dalla *www-root*

Meccanismo "antenato" dei namespace, limitato al file-system

Container

Il meccanismo moderno per creare isolamento sono i container (usati, per esempio, dal famoso *Docker*); NON fanno parte del programma di SETI, ma per chi vuole approfondire, consiglio i seguenti video:

- breve panoramica: *How Docker Works - Intro to Namespaces* by LiveOverflow: `https://youtu.be/-YnMr1lj4Z8`
- *Introduction to Docker and containers - PyCon 2016* by J. Petazzoni `https://youtu.be/ZVaRK10HBjo`
- *Containers unplugged: Linux namespaces* by Michael Kerrisk: `https://www.youtube.com/watch?v=0kJPa-1FuoI`
- *Containers unplugged: understanding user namespaces* by Michael Kerrisk: `https://www.youtube.com/watch?v=73nB9-HYbAI`
- *Understanding and Working with the Cgroups Interface* by Michael Anderson: `https://www.youtube.com/watch?v=z7mgaWqiV90`

Outline

Nebula

Per fare un po' di pratica con debolezze/vulnerabilità comuni potete provare Nebula: `https://exploit.education/nebula/`

Vediamo qualche esempio assieme (da Nebula e non solo)