Secure Programming; Sandboxing

Becoming Root

- Sometimes, we need `root` privileges to do things
- However, doing things as `root` is potentially dangerous
- Our goal: being `root` *safely*, while still getting things done
- Scenarios: shell access for the administrator; `setuid` programs

Running Commands as Root

Shell Access as Root

Three possible approaches:

- Traditional: log in directly as `root`
- Traditional temporary access: the `su` (superuser) command
- More modern: `sudo` or `sudo -i`

The difference: accountability

Logging in as Root

- Traditional answer: it's a bad idea, because all sysadmins have to share the `root` password—who is using it?
- Also traditional: users are dialing in via phone lines (or equivalent)—not readily traceable
- Dial-up? What's that?
- Today, we connect over the network—are the risks the same?
- Not quite. . .

Login via ssh

On the sysadmin's computer:

```
ssh root@w4181.cs.columbia.edu
```

On the server:

```
# tail /var/log/auth.log
Nov  4 21:18:58 w4181 sshd[212391]: Accepted publickey for root
    from 128.59.13.23 port 35316 ssh2: ED25519 SHA256:09UamV21PlHO18xa
Nov  4 21:18:58 w4181 sshd[212391]: pam_unix(sshd:session): session op
    for user root by (uid=0)
Nov  4 21:18:58 w4181 systemd-logind[790]: New session 279 of user roo
```

We see the client's IP address (128.59.13.23) and which SSH key was used for authentication. Problem solved?

Login via ssh

On the sysadmin's computer:

```
ssh root@w4181.cs.columbia.edu
```

On the server:

```
# tail /var/log/auth.log
Nov  4 21:18:58 w4181 sshd[212391]: Accepted publickey for root
    from 128.59.13.23 port 35316 ssh2: ED25519 SHA256:09UamV21PlHO18xa
Nov  4 21:18:58 w4181 sshd[212391]: pam_unix(sshd:session): session op
    for user root by (uid=0)
Nov  4 21:18:58 w4181 systemd-logind[790]: New session 279 of user roo
```

We see the client's IP address (128.59.13.23) and which SSH key was used for authentication. Problem solved? Not quite. . .

Issues

- What if a non-attributable IP address is used? 128.59.13.23 is in fact a VPN exit address
- What if passwords are still accepted? That rules out the other form of authentication
- And there are often still console logins via the virtual machine hypervisor—does it keep good logs?
- But we often need the ability:

```
$ ssh root@w4181.cs.columbia.edu apt update
```

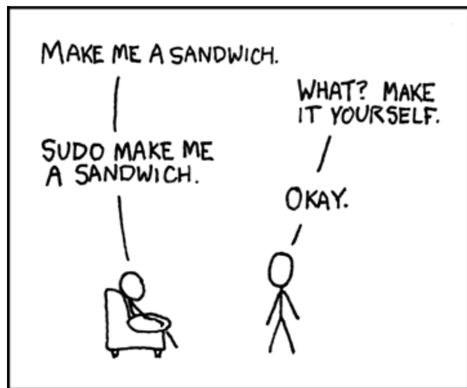
- Best solution: allow `ssh` as root, but restrict it (stay tuned)

Using su

- Login as yourself and then use su
- More accountable: login records show who was logged in to the session and then executed su
- You can use su to issue a single command, but it's more geared towards interactive shells
- But: there's still a shared root password, though you generally need extra authorization to be able to use su
- And there's no way to restrict what this user can do as root
- In other words: a decent solution but not a perfect one

Root via sudo

- A command to grant temporary privileged status to specific users, after they authenticate
- Easily usable for single commands (“`sudo cmd`”) or for interactive sessions (“`sudo -i`”)
- Can be restricted to specific users, specific commands, etc.
- Password authentication lasts for several minutes, for multiple invocations
- More accountability, more usability, *and* more authorization
- The normal approach on Linux



https://xkcd.com/149/

Programming with `setuid`

Programming with setuid

- We've spent a lot of time talking about writing secure code
- The same, of course, applies to `setuid` code
- But there are special techniques that are of particular use here
- Two primary goals:
  - Don't make mistakes peculiar to `setuid` code
  - Use the principle of least privilege to limit or eliminate the damage
- Important definitions:

  Real UID The UID of the process that invoked the `setuid` program

  Effective UID The UID that the program is `setuid` to, i.e., the privileged UID
- Note: Most of what I'll say is true of `setgid` programs as well; in the interests of brevity, I won't mention that again

Opening Files

- Many privileged programs will open user-specified files
- Should the files be opened?
- Example: to print a file, it's handed to the print spooler—but that's usually `setuid` to user `lp` or some such
- How should such files be opened?
  - Just open them
  - Use the `access()` system call
  - Shed privilege

Just Open Them

- Nope!

Just Open Them

- Nope!
- `lp` may not be able to open user files if they're read-protected
- The user may try to print files that are read-protected by `lp`

The `access()` System Call

- From the man page: `access()` checks whether the calling process can access the file *pathname*...
  The check is done using the calling process's real UID and GID.
- It seems to do what we want...

The `access()` System Call

- From the man page: `access() checks whether the calling process can access the file` *pathname*`...`
  `The check is done using the calling process's real UID and GID.`
- It seems to do what we want. . .
- This is a classic situation for a race condition attack. From the Linux man page:

  > *Warning: Using these calls ...creates a security hole...For this reason, the use of this system call should be avoided.*

Shedding Privilege

- The only safe way to open a user-specified file is to do it as that user
- Temporarily shed privilege:

```
seteuid(getuid());
fd = open(filename, O_RDONLY);
```

Shedding Privilege

- The only safe way to open a user-specified file is to do it as that user
- Temporarily shed privilege:

```
seteuid(getuid());
fd = open(filename, O_RDONLY);
```

- Set the *effective* UID to the *real* UID, i.e., the UID of the invoking user

Shedding Privilege

- The only safe way to open a user-specified file is to do it as that user
- Temporarily shed privilege:

```
seteuid(getuid());
fd = open(filename, O_RDONLY);
```

- Set the *effective* UID to the *real* UID, i.e., the UID of the invoking user
- When the file is open, resume privileges using the *saved UID*

Shedding Privilege

- The only safe way to open a user-specified file is to do it as that user
- Temporarily shed privilege:

```
saveuid = geteuid();
seteuid(getuid());
fd = open(filename, O_RDONLY);
seteuid(saveuid);
```

- Set the *effective* UID to the *real* UID, i.e., the UID of the invoking user
- When the file is open, resume privileges using the *saved UID*
- The saved UID is what the program was originally `setuid()` to—non-root programs can only `setuid()` to the real UID or the saved UID

Shedding Privilege

- The only safe way to open a user-specified file is to do it as that user
- Temporarily shed privilege:

```
saveuid = geteuid();
seteuid(getuid());
fd = open(filename, O_RDONLY);
seteuid(saveuid);
```

- Set the *effective* UID to the *real* UID, i.e., the UID of the invoking user
- When the file is open, resume privileges using the *saved UID*
- The saved UID is what the program was originally `setuid()` to—non-root programs can only `setuid()` to the real UID or the saved UID
- Note: permissions are checked at `open()` time, not at `read()` or `write()` time

Opening Files with Message-Passing

- In message-passing systems, the privileged program can't temporarily shed and then regain privileges—it always has them
- Instead, an unprivileged program has to open the file and *pass the file descriptor to the privileged program*
- Linux can do that with Unix-domain sockets, using `sendmsg()/recvmsg()` and `SCM_RIGHTS`

Using ACLs

- Assert that all printable files must be readable by, e.g., group `SysDaemon`
- Use initial ACLs to make that the default for files
- The print daemon is then `setgid()` to group `SysDaemon` and hence can open such files
- Its own files would need to be readable/writeable only by `SysDaemon`

Least Privilege

Least Privilege

- Much of the time, even privileged programs are doing unprivileged things
- Why do them with privilege?
- Policy: shed privileges at the beginning; resume them temporarily only when needed

```
saveuid = geteuid();
seteuid(getuid());
...
seteuid(saveuid);
<do privileged stuff>
seteuid(getuid());
```

- Are we safe?

Least Privilege

- Much of the time, even privileged programs are doing unprivileged things
- Why do them with privilege?
- Policy: shed privileges at the beginning; resume them temporarily only when needed

```
saveuid = geteuid();
seteuid(getuid());
...
seteuid(saveuid);
<do privileged stuff>
seteuid(getuid());
```

- Are we safe?
- Well, safer. . .

Residual Risk

- One of the big risks for privileged programs is attackers running code: code injection attacks or ROP
- If legitimate code can regain privilege via `seteuid(saveuid)`, so can attacker code
- There are other risks in asynchronous situations, e.g., catching signals
- (Threads have complex interactions with `setuid()`; I'm not even going to try to cover that here...)
- Shedding privilege permanently is safer

The Apache Web Server Redux

- It must protect its private key
- It must write to log files, but no one else should be able to overwrite them
- It's very exposed to attack

Shedding Privileges

- Apache starts as `root`
- Note: it must be invoked by `root`; it is not `setuid()`
- It opens its private key file
- It opens the socket and some log files, then forks and sheds privileges: it sets its real and effective UIDs to `www`
- When serving web pages, it runs as that non-privileged user
- (Why isn't it `setuid()`?)

Shedding Privileges

- Apache starts as `root`
- Note: it must be invoked by `root`; it is not `setuid()`
- It opens its private key file
- It opens the socket and some log files, then forks and sheds privileges: it sets its real and effective UIDs to `www`
- When serving web pages, it runs as that non-privileged user
- (Why isn't it `setuid()`?)
- Prevent local machine attacks—it has no privileges if invoked by some other user

Some Other Useful Routines

getpwduid() Map a numeric UID to a username

getpwdnam() Map a username to a numeric UID

mlock() Lock some memory into RAM; keep sensitive data from being written to the page file

FD_CLOEXEC Use with `fcntl()` to force certain file descriptors to be closed on exec() of another program

*at() Linux has system calls, e.g., `openat()`, that operate relative to an open file descriptor rather than the current directory; this helps avoid race conditions

Sandboxing

Sandboxing

- We want to *really* strip some applications of privileges
- More precisely, we don't want them to have even ordinary privileges
- Limit files, network access, and more
- Why? They're high-risk—we don't think they can be made adequately secure
- We want to protect almost all of the system from them
- These are called *sandboxes*—in one sense, an area where an application can make a mess without it mattering, but some make analogies to a cat's sandbox. . .

Couldn't We Use ACLs?

- ACL usually do not have permissions that say "don't allow access to anything else"
- We'd have to find and change the protections of every file on the system that was writable/readable/searchable by `other`
- We'd have to ensure that no other such files were created
- This is all possible but difficult
- More seriously, it is not high *assurance*

Other Resources

- What other resources need to be protected?
- CPU time
- Memory, real and virtual
- Disk space
- Network identity
- Network access rights
- More. . .

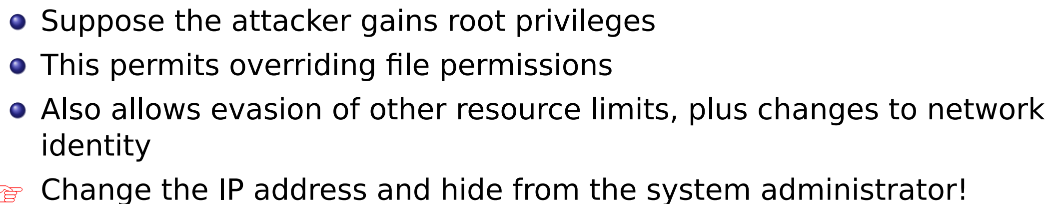
Some Are Easy

- Operating systems already regulate access to some resources
- Unix examples: `setrlimit()`, file system quotas

Network Identity and Access Rights

- A machine has an IP address
- A compromised application can use this address to exploit address-based access control
- If nothing else, it can confuse intrusion detection systems

Bypassing File Permissions



Goals

- Security
- High assurance
- Simple setup
- General-purpose mechanism
- Available to all applications
- We can't get them all...

Questions?

(Black-and-white warbler, Central Park, October 19, 2019)