

CSIT 5740 Introduction to Software Security

Note set 2

Dr. Alex LAM



DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

***The Linux System
continued from
the last class...***

Users

- Linux (Unix) is a multiuser OS
 - `/etc/passwd`
 - Every user is identified by a number (user ID) and a name
 - lists all the users with their corresponding user ID (uid)
- Main security
 - Enables protection from other users
 - Enables protection of system code from users

Users

- System code runs as the super-user, also called **root** user
 - The root user has user ID equal to zero
- The system code is responsible (among other things) to ensure its own protection and the protections between users
 - System code is the kernel code plus code in user-space processes, run by the root user

Users

- Every process has an owner (the “user ID” of a process shows the owner)
 - The owner is one of the users in the system
 - Normally, the process owner is the user that started its execution
 - More details on this later...
- In general, what a process can do is determined by who its owner is and what its owner can do (again, more details later...)
 - You can see owners using `htop`

Users

- Typical setup:
 - In a desktop/laptop: one main user
 - In a server: one user for every person allowed to access it
 - Every user has a “home” folder
 - In addition, special users running specific processes and root
 - Android: typically one user per app

Users

- Every user belongs to one or multiple groups
- `/etc/group` contains all the available groups and the owners belonging to it
- The command `groups` lists all the groups which the current user belongs to
- Groups are useful if we want to allow multiple users to perform some operation, for instance:
 - The program `docker` can be used only by the root user or any user in the “docker” group
 - If we want to allow a user to use `docker`, we can just add it to the “docker” groups

- Some commands (some require root privileges):
 - `id`:
show the current user and its groups
 - `adduser`:
add a new user to the system
 - `useradd`:
add a new user and also create its home folder, ...
 - `usermod -a -G group user`:
add the user “user” to the group “group”
 - `sudo command`:
run command as root
 - `sudo -u user command`:
run a command as “user”
 - `sudo [-u user] -i`:
open a shell running commands as root [or “user”]

Users: Sudoers

- Normal users have an associated password
- Typically asked for “login”
 - System’s startup
 - Remote login (ssh)
- Users belonging to the group sudo can switch to another user using the sudo command
- By default, it requires inserting the user’s password

Permission Checking

- In general, to perform system-level operations a process invokes kernel code, by using system calls
- The called kernel code checks
 - Which process called it
 - The owner of the process
 - Is the owner of the process authorized to perform the requested operation?
 - Some operations may be restricted only to:
 - Users belonging to specific groups
 - The root user
 - ...

File Permissions

- Every file has an associated owner user and an associated owner group
 - **Use:** `ls -al`
to list files and their owners
 - `chown`: change owner user/group
- Every file has permissions associated to it
 - `chmod`: change file permissions

File Permissions

File Type # of Hard Links File size

Permissions Owners Last Modify Time

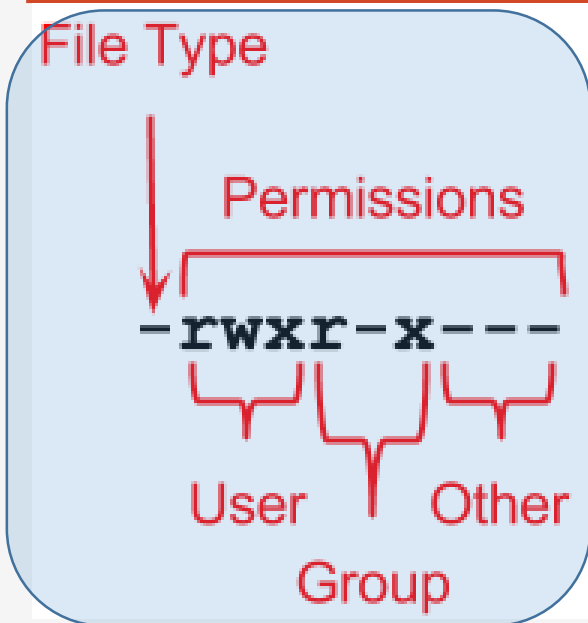
`-rwxr-x---` `1` `walbert support` `0` `Oct 31 11:06` `test`

User Group User Group File name

Group

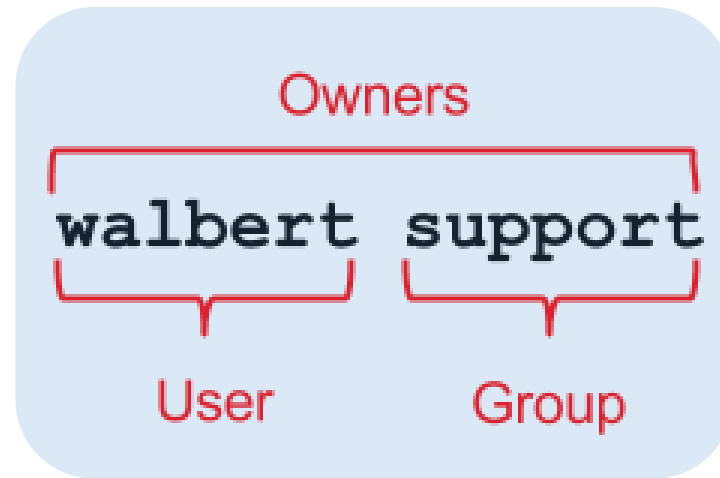
Owner	Group	Other
<code>rw</code>	<code>x</code>	<code>r-x</code>
$4+2+1$	$4+0+1$	$4+0+1$
7	5	5

File Permissions



Owner	Group	Other
<code>rwx</code>	<code>r-x</code>	<code>r-x</code>
$4+2+1$	$4+0+1$	$4+0+1$
7	5	5

File Permissions



File Permissions

Last Modify Time
Oct 31 11:06 test
File name

File Permissions

of Hard Links



1

File size



0

File Permissions

File Type # of Hard Links File size

Permissions Owners Last Modify Time

`-rwxr-x---` `1` `walbert support` `0` `Oct 31 11:06` `test`

User Group User Group File name

Group

Owner	Group	Other
<code>rw</code>	<code>r-x</code>	<code>r-x</code>
$4+2+1$	$4+0+1$	$4+0+1$
7	5	5

Directory Permissions

Read permission

- : The directory's contents cannot be shown.
- r**: The directory's contents can be shown (listed).

Write permission

- : The directory's contents (the list of files) cannot be modified.
- w**: The directory's contents can be modified (create new files or folders, rename or delete existing files, ...). It requires the execute permission, otherwise it has no effect

Directory Permissions

Execute permission

–: The directory cannot be accessed (cannot cd inside the directory)

x: The directory can be accessed using cd

(this is the only permission bit that in practice can be considered to be "inherited" from the ancestor directories, in fact if *any* folder in the path does not have the x bit set, the final file or folder cannot be accessed either)

→ You can create directories from which a user can read files, but you cannot list them → remove the “r” permission, but keep the “x” permission

Links: Security Concerns

Symbolic Links can be used to make exploitations easier

For instance (will illustrate using a real example):

- An application runs as root and takes a `<filename>` as the argument, this `<filename>` could be in a world-readable and/or world-writeable folder (`/tmp` or `/var/log` for example)
- An attacker could create a link from `<filename>` to a strategic file (`/etc/shadow`, `/root/.rhosts` etc), tricking the app to make the file world-readable or even world-writable
- Even if the application checks if `<filename>` is not a link, the attacker can create a link just after the check, but before the application changes `<filename>` file permissions (more on that)

Links: Security Concerns

Consider the program fragment being run by the superuser `root` at the `/tmp` directory

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1], O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1], S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1], O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1], S_IRWXU | S_IRWXG | S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```


Links: Security Concerns

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

This file is an innocent log file and in the /tmp directory should be accessible to everybody. S_IRWXU, S_IRWXG and S_IRWXO means we grant the **full set of RWX** rights on the file to the owner, group and other users!

Links: Security Concerns

- How we can make use of this buggy program run by the `root` to exploit the Linux system?
 - Scenario 1: a malicious user “alex” wants to read the password stored at `/etc/shadow` , but he does not have the permission
 - Scenario 2: a malicious user “alex” wants to remotely login the machine as the user “Kali”

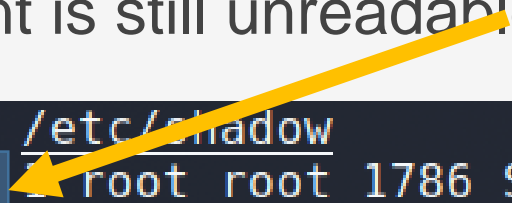
Links: Security Concerns

- For scenario 1, “alex” just need to create a symbolic link to point to `/etc/shadow`, then wait quietly for the program to start and change the permission of `/etc/shadow` , then he can read and launch a dictionary attack on the password
- Alex creates a symbolic link from a file called “log” to “`/etc/shadow`”:

```
(alex@kali) - [/tmp]  
$ ln -s /etc/shadow log
```

- Alex patiently waits for the root to start the program and run it on the “log” file, at that point is still unreadable for Alex:

```
# ls -al /etc/shadow  
-rw-r----- 1 root root 1786 Sep  3 04:42 /etc/shadow
```




Links: Security Concerns

- For scenario 1, “alex” just need to create a symbolic link to point to `/etc/shadow`, then wait quietly for the program to start and change the permission of `/etc/shadow` , then he can read and launch a dictionary attack on the password
- The `root` executes the program from the `/tmp` directory, as a regular daily task

```
(root👤kali) - [/tmp]  
# ./symlink_vuln log
```

- Look at the permissions for “other users” (Alex included)!

```
# ls -al /etc/shadow  
-rwxrwxrwx 1 root root 1786 Sep  3 04:42 /etc/shadow
```



Links: Security Concerns

- For scenario 2, “alex” just need to create a symbolic link to point to `/home/kali/.rhosts`. “.rhosts” is the file containing the list of remote hosts that are allowed to connect to the current host without the need of supplying a password! The symbolic link can be created even if the target is not existing!!
- Alex creates a symbolic link to “.rhosts”,

```
(alex@kali) - [/tmp]  
$ ln -s /home/kali/.rhosts log
```

- The file doesn't even exist!

```
# ls -al /home/kali/.rhosts  
ls: cannot access '/home/kali/.rhosts': No such file or directory
```

Links: Security Concerns

- For scenario 2, “alex” just need to create a symbolic link to point to `/home/kali/.rhosts`. “.rhosts” is the file containing the list of remote hosts that are allowed to connect to the current host without the need of supplying a password! The symbolic link can be created even if the target is not existing!!
- Again Alex waits patiently for the unalarmed `root` to run the buggy program

```
(root👁️kali)-[/tmp]  
# ./symlink_vuln log
```

- The file is created, granting a full set of rights to everybody! Can you imagine how happy Alex is? He can add his host to login without the need of password.

```
(root👁️kali)-[/tmp]  
# ls -al /home/kali/.rhosts  
-rwxrwxrwx 1 root root 0 Sep 10 03:52 /home/kali/.rhosts
```



Links: Security Concerns

- What we can do to mitigate that? Do the following as the root:

```
(root👤kali) - [/tmp]  
# echo 1 > /proc/sys/fs/protected_symlinks
```

- When `protected_symlinks` is 1, if the sticky bit is set on a world-writable directory (`/tmp`), Linux will allow the program to follow a symbolic link to the target if:
 - the process `uid` is the same as the symlink `uid`
 - the symlink and the `/tmp` directory have the same owner (i.e. usually the `root`)

Links: Solutions

- When `protected_symlinks` is 1, if the sticky bit is set on a world-writable directory (`/tmp` , “world-writable” so that Alex can put a symlink file there), Linux will allow the program to follow a symbolic link to the target if :
 - **either** the process `uid` is the same as the symlink `uid`
 - **Or** the symlink and the directory (i.e. `/tmp` here) have the same owner (i.e. usually the `root`)
- That will solve scenario 1 , and Alex will not be able to change the permissions of “`/etc/shadow`” with his symlink,
 - because the process `uid` is `root`, symlink `uid` is `alex`
 - the symlink owner is `alex`, but the `/tmp` directory belongs to `root`
- That will also solve scenario 2 , and Alex will not be able to create the `.rhosts` file

Links: Solutions

- What we can do to even better protect the system? Replace the line:

```
// when we open the file with "O_CREAT" flag,  
// open will follow the link (if it is a symbolic link)  
// and create the file in the target directory!  
int file = open(argv[1], O_CREAT);
```

- with this:

```
// when we open the file with "O_EXCL" and "O_CREAT" flag,  
// The file must not be existing, and it will be created atomically  
// if the file exists, open() will return an error message  
int file = open(argv[1], O_CREAT | O_EXCL);
```

Real-world symlink attack examples

- Scenario 1: Look at CVE-2020-8019 (CVE: Common Vulnerabilities and Exposures)
- Scenario 2 : Look at CVE-1999-1187
- Scenario 3 (not covered): Look at CVE-2022-35631 (we can use this to launch a simple denial of service attack, this CVE is in particular embarrassing, because the bug is present in the Velociraptor software, which is a forensics and incident response tool many cybersecurity professionals used to monitor and check systems that may have been hacked)



Linux: setuid and effective user

Linux shell and bash

- When you interact with a command line interface (typically called, a “shell”) in a Linux machine you are actually interacting with a dedicated program
 - Typically this program `sh` or `bash`
 - Alternatives are possible
 - When connecting with a machine using `ssh`, `ssh` eventually opens `bash` as one of its child processes
 - When opening a `shell` in a machine using a graphical interface, typically a graphical program (e.g., `gnome-terminal`) opens `bash` as one of its child processes

Linux shell and bash

- `bash` then creates child opens processes to run the program you specify on the command line
- Note the parent/child relationship between the created processes
 - For instance:
 - `init → ... → ssh → bash → ls`
 - You can use `htop` to visualize these relationships as a tree (to do that, you can press “t” after starting `htop`)

setuid

- Normally, when a program is executed (`./executableName`), it becomes a process. The owner of the process is the current user (displayed by `uid`)
- However if a program is stored as a **setuid** file, the owner of the process will be the owner of the file
- This allows users to perform privileged operations, by opening setuid programs
- setuid is a file attribute
 - set using: `chmod u+s executableName`
 - checked using: `ls -la`

```
(root@kali) - [~alex]  
# chmod u+s symlink_vuln
```

```
-rwsr-xr-x 1 root root 16064 Sep 10 03:18 symlink_vuln
```

- More precisely, every process has two properties:
 - **real user ID**
 - the user who started a process
 - **effective user ID**
 - the user used by the OS to determine what a process can/cannot do

- Normally, when a process creates a new process (i.e., a child process) the child process **keeps the same real user ID and effective user ID** of its parent process
 - New processes are normally created with the `fork/execv` system calls. Don't worry too much about "system calls" if you don't know about them, we will discuss them in 2-3 weeks.
 - This is what normally happens when you use a shell (e.g., `bash`) to run a command
 - `bash` runs with real/effective user ID of the logged in user
 - Processes created using the shell have the same real/effective user ID of the user using the shell

- However
 - if a program is stored with the **setuid bit set**
 - when it runs, the effective user ID of the executed program is equal to the owner of the file
 - the real user ID is unchanged

setuid

- This mechanism allows users to perform privileged operations by using setuid programs, since the code of a setuid program runs with:
 - Effective user ID = Owner of the file
(typically root)
 - Instead of:

Effective user ID = Parent process effective user ID
(typically the current user)

setuid

- setuid programs must check that the calling user is supposed to execute the requested operation
- setuid programs should be written very carefully not to allow privilege escalation:
 - a non privilege process asks to a setuid program to execute a privileged operation

setuid

- For instance, `sudo` is a setuid program
 - the file containing `sudo` is own by `root` and is setuid
 - Therefore the code of `sudo` runs with effective user ID equal to zero (`root`)
 - `sudo` allows to execute any command as the `root` user
 - but only if the correct user's password is inserted, and the user is authorized to do so (i.e the `sudo` group)

More details: bash, system, ptrace

- The setuid bit may be ignored in some folders if the filesystem is mounted with the nosuid option (e.g., sometimes the /tmp folder)

- i.e. `mount /dev/foo /dir nosuid`

You can check using: `mount | grep nosuid`

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```

- In Kali the default shell (bash) and “sh” “drops privileges”
 - they set their effective user ID to its real user ID
 - Execute (the syscall): `setreuid(getuid(), getuid());`
 - `getuid()` → returns the real user ID
 - `setreuid(real_user_id, real_user_id)`
 - unless we open bash or sh with the `-p` option

More details: *bash*, *system*, *ptrace*

- The libc call `system(<command>)` executes:
`execv("/bin/sh", ["/bin/sh", "-c", <command>])`
 - Therefore, it also “drops privileges”, why?

More details: *bash*, *system*, *ptrace*

- Debugging tools (e.g., `gdb`) use the syscall `ptrace`
 - `ptrace` allows an external process (a debugger) to read/modify code and data of another process (i.e. the **debuggee** which is the program being debugged)
 - If a program is started using `ptrace` → the `setuid` bit is ignored
 - `ptrace` also allows to attach to an **existing process** (i.e. running program):

More details: id

The utility `id` prints out the current user `id` (and the groups it belongs to). If the current effective user id is different from the current user id, it prints both of them:

```
(alex@kali)-[~]  
$ id  
uid=1001(alex) gid=1001(alex) groups=1001(alex),27(sudo)
```

Then I copied `/usr/bin/id` to `/var/tmp/id`, change it to be executed like a root (by adding the `setuid` flag)

```
(alex@kali)-[~]  
$ sudo cp /usr/bin/id /var/tmp/id
```

```
(alex@kali)-[~]  
$ sudo chmod u+s /var/tmp/id
```

```
(alex@kali)-[~]  
$ ls -al /var/tmp/id  
-rwsr-xr-x 1 root root 48064 Aug 14 05:49 /var/tmp/id
```

This is what I get, note the effective user id is the owner (which is the `root`!)

```
(alex@kali)-[~]  
$ /var/tmp/id  
uid=1001(alex) gid=1001(alex) euid=0(root) groups=1001(alex),27(sudo)
```


setuid: sample code

```
#define _GNU_SOURCE // access to nonstandard GNU/Linux extension function
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char* get_user_name(uid_t uid)
{
    struct passwd *pws;
    pws = getpwuid(uid);
    return strdup(pws->pw_name);
}
void main(){
    uid_t uid, euid;
    uid = getuid();
    euid = geteuid();
    printf("real user id: %s (%d)\n", get_user_name(uid), uid);
    printf("effective user id: %s (%d)\n", get_user_name(euid), euid);
}
//rm -f userids; gcc -o userids userids.c ; ./userids;
//sudo chown root:root userids; sudo chmod +s userids; ./userids
```

Other Security Mechanisms

- Enforcing the **principle of least privilege**:
 - A process should only be allowed to do what it is strictly necessary
 - In this case, if a process is exploited, we limit the possible damage to the system

Privilege escalation: command injection

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char command[1000+1];
    snprintf(command, 1000, "ping %s", argv[1]);
    //run command as if it were executed in a shell
    char* args[5] = {"/bin/bash", "-p", "-c", command, NULL};
    execv("/bin/bash", args);
    return (0);
}
```

./inj '8.8.8.8'

./inj 'asd; cat /etc/shadow'

Attacking Linux Systems: Examples

Remote vs. Local Attacks

- **Local attacks**

- Require a previously-established presence on the host
 - an account
 - another application under the control of the attacker
 - ...
- Manipulate the behavior of an application through local interaction
- If the attack is successful, allow one to execute operations with privileges that are different (usually superior) from the ones that the attacker would otherwise have
- In general, these attacks are easier to perform, because
 - The attacker has a better knowledge of the environment
 - The attacker has better control over the system

Remote vs. Local Attacks

- **Remote attacks**

- Do not require a previously-established presence, but they still require some way to interact with the victim's system
 - Typically, a network connection (e.g., Internet)
- Allow one to manipulate the behavior of an application
- Allow one to execute operations with the privileges of the vulnerable application
- In general, these attacks are more difficult to perform but they do not require prior access to the system
 - Therefore, it may be possible to scan “the entire Internet”, find vulnerable hosts, and attack them

Attacking UNIX Systems

- Remote attacks against a network service
- Remote attacks against the operating system
- Remote attacks against a browser

- Local attacks against SUID applications
- Local attacks against the operating system

Attack Classes

- File access attacks
 - Path attacks
 - TOCTTOU
- Command injection
- Memory Corruption
 - Stack corruption
 - Heap corruption
 - Format string exploitation
 - ...

File Access Attacks

- Access to files in the file system is performed by using a **path** strings
- An attacker controlling how or when a privileged application builds a path string can lure the application into performing unwanted operations

Path Traversal Attack

- An application builds a path by concatenating a path prefix with values provided by the user (the attacker)

```
path = strncat("/var/log/app/", user_file, free_size);  
file = open(path, O_RDWR);
```

- The user (attacker) provides a filename containing a number of: `"../"` that allow for escaping from the directory and access any file on the file system

- `user_file == "../.../etc/shadow"`
⇒ `path = "/var/log/app/.../.../.../etc/shadow"` ⇒ opens `"/etc/shadow"`

Lessons Learned

- Input provided by the user should be sanitized before being used in creating a path

Some useful functions for sanitizing the path inputs

<i>Linux</i>	<i>Windows</i>	<i>Java</i>	<i>Python</i>
<code>realpath()</code> <code>canonicalize_file_name()</code>	<code>PathCanonicalize()</code>	<code>getAbsolutePath()</code> <code>getCanonicalPath()</code>	<code>os.path.abspath()</code> <code>os.path.realpath()</code>

PATH Attack

- The `PATH` environment variable determines how the shell searches for commands
 - It contains a list of paths, separate by colons
 - You can print it using “`echo $PATH`” and modify it using “`export`”, for instance “`export PATH=$PATH:<additional_path>`”
 - `Export` only changes `PATH` used by the current shell
- The shell searches for a binary having the same name of the typed command in all the paths listed in the `PATH` environment variable

PATH Attack

- Some library calls, `execvp`, `execvp`, and `system`, also search for the provided filename in the paths listed in `PATH`
- Calling `system(<command>)` is equivalent to running `sh -c <command>`, which is equivalent to typing `<command>` in a shell and pressing enter
 - Remember, however, `sh` drops privileges unless there is `-p` option

HOME Attack

- The HOME environment variable determines how the home directory path is expanded by the shell, when the character ~ is used
- If an application uses using a home-relative path (e.g., ~/myfile.txt), an attacker can modify the HOME environment variable to control which files are accessed

Lessons Learned

- Absolute paths should always be used when executing external commands
- Home-relative paths should never be used

Command Injection

- Applications invoke external commands to carry out specific tasks
- The `system(<string>)` libc function executes a command specified in a string by calling:
`/bin/sh -c <string>`
- The `popen()` libc function opens a process by creating a pipe, forking, and invoking the shell as in `system()`
- If the user can control the string passed to these functions, the string can be employed to inject additional commands

Link Attacks

- An application may check the path to a file (e.g., to verify that the file is under a certain directory) but not the nature of the file
- By creating symbolic links an attacker can force an application to access files outside the intended path
- When an application creates a temporary file it might not check for its properties in the assumption that the file has been created with the correct privileges
- We have seen this in details earlier

TOCTTOU Attacks

- Attacker may race against the application by exploiting the gap **between testing and accessing** an property
 - **Time-Of-Check-To-Time-Of-Use**
 - e.g., testing a file property and then accessing it
- Time-Of-Check (t1): validity of assumption A on entity E is checked
- Time-Of-Use (t2): E is used, assuming A is still valid
- Time-Of-Attack (t3): assumption A is invalidated by an attacker, but the attacker can still access E, as long as:
 - **$t1 < t3 < t2$**
- Data race condition

TOCTTOU Example

- A SETUID program may want to avoid accessing a file if its real user (real UID) is not allowed to access it
- The `access()` system call returns an estimation of the access rights of the user specified by the **real UID**
- The `open()` system call checks permissions using the **effective UID**

```
if(access(file, W_OK) == 0) { //time of check t1
```

```
    //an attacker replaces file with a symlink to the file /etc/shadow t3  
    // i.e.  symlink("etc/shadow",file);  
    // the line below , it opens /etc/shadow using the root privilege
```

```
    if ((fd = open(filename, O_WRONLY)) < 0){ //time of use t2  
        return -1;  
    }
```

```
    write(fd, buf, count);
```

```
}
```

TOCTTOU Example: same code as previous slide

- The `access()` system call returns an estimation of the access rights of the user specified by the **real UID**
- The `open()` system call checks permissions using the **effective UID**

Victim

```
if(access(file, W_OK) == 0) {  
  
    if ((fd = open(filename, O_WRONLY)) < 0){  
        :  
        :  
        write(fd, buf, count);  
    }  
}
```

Attacker

`symlink("/etc/shadow", "foo");`

t1

t3

t2

time

Lessons Learned

- Use system calls that check and use something at the same time (atomically)
 - To create a temporary file, use `mkstemp()`, which creates a file and opens it atomically

A Real Example: Shellshock

- On September 2014, a new bug in how bash processes its environment variables was disclosed
 - The bug was present since version 1.03, released in 1989

A Real Example: Shellshock

- 1) The bash program can pass its environment to other instances of bash, including one or more function definitions
 - This is accomplished by setting environment variables whose value contains function definitions
 - The shell automatically executes the content of environment variables that contain function definitions.
 - The entire content of the environment variable is executed, not just the function definition, therefore:

```
() { :; }; <MALICIOUS_CODE> executes <MALICIOUS_CODE>
```

A Real Example: Shellshock

- 2) If a user has access to a restricted shell, and a command that is not allowed is requested, the original not-allowed command is put in the variable `SSH_ORIGINAL_COMMAND`

Therefore, by passing as a command the string:

```
() { :; }; cat /etc/shadow
```

the environment variable `SSH_ORIGINAL_COMMAND` becomes:

```
() { :; }; cat /etc/shadow
```

Since the content of this environment variables starts with `()`, it is executed.

- Detailed explanation: <https://fedoramagazine.org/shellshock-how-does-it-actually-work/>

Lessons Learned

- Invoking commands with `system()` and `popen()` is dangerous
 - They should never be called with strings that can be influenced by the user
 - Alternative: input from the user should always be sanitized
 - Sanitizing input is hard: use library functions
 - e.g., removing semicolon is not enough
- Complex designs are error prone

Application Security: The Life of an Application

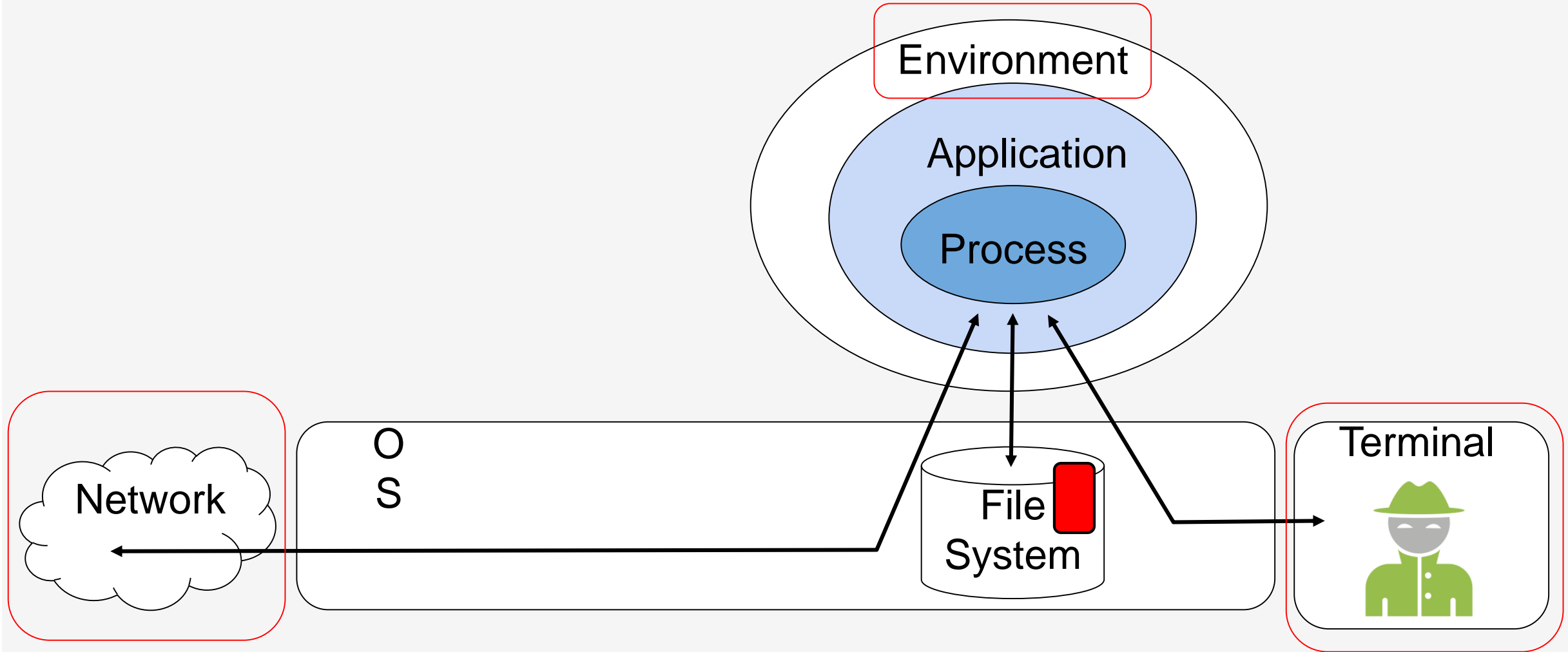
Application Security

- Applications provide services
 - Locally (e.g., word processing, file management)
 - Remotely (e.g., remote file transfer, remote web server)
- The behavior of an application is determined by
 - the code being executed
 - the data being processed
 - the environment in which the application is run

Application Security

- Attacks against applications aim at bringing applications to perform operations that violate the security of the system
- Using the CIA triad
 - Violation of confidentiality
 - Violation of integrity
 - Violation of availability

Application Security



Application Vulnerability Analysis

- Identifying vulnerabilities in applications
 - As deployed in a specific operational environment
 - Assuming a specific threat model
- Design vulnerabilities
- Implementation vulnerabilities
- Deployment vulnerabilities

Design Vulnerabilities

- Also called: “logic bugs”
- These vulnerabilities are flaws in the overall logic of the application
 - Lack of authentication and/or authorization checks
 - Erroneous trust assumptions
 - ...
- These vulnerabilities are the most difficult to identify (especially automatically)
 - they require a clear understanding of the functionality implemented by the application

Confused Deputy

- In this attack pattern, an application is confused into performing a malicious action *on behalf of an attacker*
 - The code of the “victim’s application” is not modified
 - The interaction leads the application into an unforeseen state

Confused Deputy

- Example
 - An Android app saves the the user's contact list to a publicly accessible file, when it receives a specific Intent (a message from another app)
 - A malicious app, without the permission of accessing the contact list, can send a specific Intent to the victim's app and make it save the contact list in a public location
 - The malicious app can then read the contact list from the public location

Implementation Vulnerabilities

- The application is not able to correctly handle unexpected events or inputs
 - Unexpected input
 - The input is too long
 - The input does not follow the intended format
 - Unexpected errors/exceptions
 - Unexpected interleaving of events (race conditions)
 - ...
- An important consequence is: **memory corruption**

Deployment Vulnerabilities

- These vulnerabilities are introduced by an incorrect/faulty deployment/configuration of the application
 - An application is installed with more privileges than the ones it should have
 - An application is installed on a system that has a faulty security policy and/or mechanism
 - e.g., a file that should be read-only is actually writeable
 - An application is configured with easy-to-guess default credentials
 - Or even worst, hardcoded credentials
 - An application is deployed in “debug” mode
- If correctly deployed, the application would be (more) secure

The Life of an Application

- programmer writes code in high-level language
- The application is translated in some executable form and saved to a file
 - Interpretation vs. compilation
- The application is loaded in memory
- The application is executed
- The application terminates
- Knowing these steps is crucial to determine exploitability of a vulnerability

Interpretation

- The program is passed to an interpreter
 - The program might be translated into an intermediate representation
 - Python byte-code
- Each instruction is parsed and executed
- In most interpreted languages, it is possible to generate and execute code dynamically
 - Bash: `eval "..."`
 - Python: `eval("...")`
 - JavaScript: `eval("...")`

Interpretation

- Complex programs may contain interpreters
- For instance, the browser contains a Javascript interpreter
 - For performance reasons the interpreters typically convert the interpreted code into executable machine code
 - Just-in-Time compilation (JIT)
 - JIT engines are one of the targets of modern exploitation

Compilation

- The preprocessor expands the code to include definitions, expand macros
 - GNU/Linux: The C preprocessor is `cpp`
- The compiler turns the code into architecture-specific assembly
 - GNU/Linux: default C compiler is `gcc` (`llvm` is an alternative)
 - `gcc -S prog.c` will generate the assembly
 - Use `gcc`'s `-m32` option to generate 32-bit assembly

Compilation

- The assembler turns the assembly into a binary object
 - GNU/Linux: The assembler is as
- A binary object contains the binary code and additional metadata
 - Relocation information about things that need to be fixed once the code and the data are loaded into memory
 - Information about the symbols defined by the object file and the symbols that are imported from different objects
 - Debugging information
 - optionally, depending on compilation options
 - Function names
 - Original source code
 - ...
 - help significantly with debugging

Compilation

- The linker combines the binary object with libraries
 - resolving references that the code has to external object (e.g., functions) and creates the final executable
 - GNU/Linux: The linker is ld
 - Static linking is performed at compile-time
 - The code of all the used libraries is included in the final executable file
 - Dynamic linking is performed at run-time
- Most common executable formats:
 - GNU/Linux: ELF
 - Windows: PE

Libraries and Syscalls

