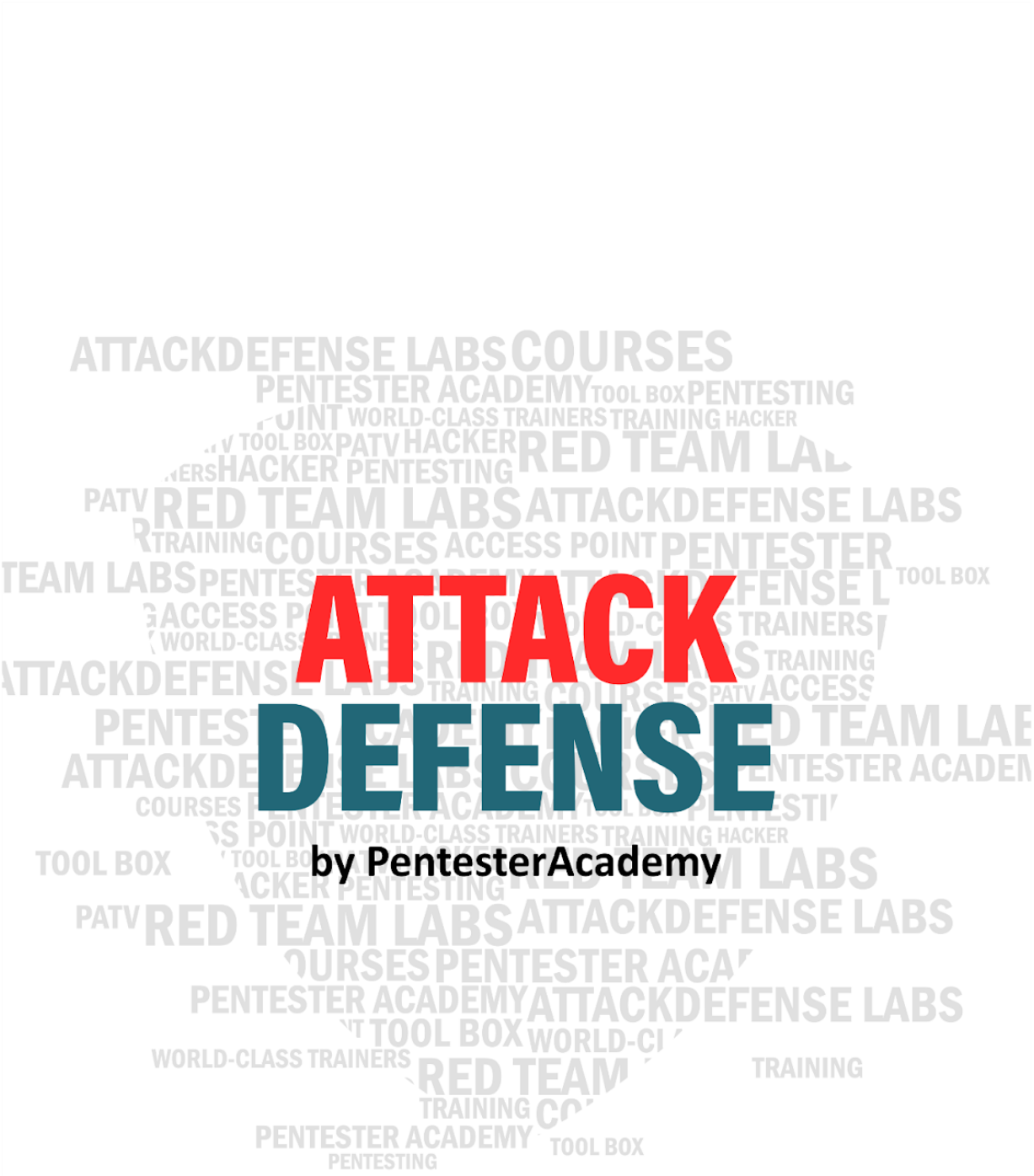
ATTACK
DEFENSE
by PentesterAcademy

Name	Introduction to Seccomp
URL	https://attackdefense.com/challengedetails?cid=1826
Type	Privilege Escalation : Seccomp

Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

Objective: Learn how to use Seccomp profiles with Docker for blocking syscalls.

Seccomp or Secure Computing mode is a feature of Linux kernel which can act as syscall filter and not a sandbox. However, it is often used to augment sandboxes to block syscalls.

It is voluntary i.e. the application moves to a restricted state on its wish and it is not enforced by other systems like AppArmor/SELinux. .

There are two types of seccomp: mode 1 (strict) and mode 2 (filter)

Mode 1 (Strict)

This mode is original seccomp that is extremely restrictive and only allows four syscalls:

1. read()
2. write()
3. exit()
4. rt_sigreturn

If any other syscall is made, the process is killed using SIGKILL. A single specific syscall is made to one of the following:

- seccomp(SECCOMP_SET_MODE_STRICT)
- prctl (PR_SET_SECCOMP, SECCOMP_MODE_STRICT).

Once this call is made, mode 1 will be active for the lifetime of the process.

Example

Seccomp Mode 1 is used in the folowing code.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>


int main(int argc, char **argv)
{
	int output = open("output.txt", O_WRONLY);
	const char *val = "test";

	printf("Calling prctl() to set seccomp strict mode...\n");
	prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

	printf("Writing to an already open file...\n");
	write(output, val, strlen(val)+1);

	printf("Trying to open file for reading...\n");
	int input = open("output.txt", O_RDONLY);

	printf("You will not see this message--the process will be killed first\n");
}
```

Code Source: https://sysdig.com/blog/selinux-seccomp-falco-technical-discussion/

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>


int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp strict mode...\n");
        prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("You will not see this message--the process will be killed first\n");
}
```

Example explanation

Line 15: The program enables strict seccomp mode. After this it can only use four syscalls (mentioned above).

Line 18: The handle output is already opened so the program can write to it.

Line 21: Program tries to open another file which is NOT allowed. So, it will be killed by the kernel.

Line 23: Will never be printed as the program is already killed.

Trying it out

Step 1: Save the code provided above as seccomp_mode1.c and compile it with gcc.

Command: gcc seccomp_mode1.c -o seccomp_mode1

Step 2: Execute the compiled binary.

Command: ./seccomp_mode1

```
root@localhost:~# ./seccomp_mode1
Calling prctl() to set seccomp strict mode...
Writing to an already open file...
Trying to open file for reading...
Killed
root@localhost:~#
```

As expected, the program is killed before reaching the end of the program.

Many Linux based sandboxes use this mode.

Mode 2 (Seccomp-bpf)

This mode is the newer one that involves a userspace-created policy being sent to the kernel. This policy defines the permitted syscalls and arguments along with the action to take in the case of a violation.

The filter comes in the form of eBPF bytecode (a special type instruction set that is interpreted in the kernel and used to implement filters).

Example

Seccomp Mode 2 is used in the following code.

program that prints the current PID using seccomp-bpf:

```
#include <seccomp.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

void main(void)
```

```
{
    /* initialize the libseccomp context */
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    /* allow exiting */
    printf("Adding rule : Allow exit_group\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

    /* allow getting the current pid */
    printf("Adding rule : Allow getpid\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);
    //printf("Adding rule : Deny getpid\n");
    //seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(getpid), 0);

    /* allow changing data segment size, as required by glibc */
    printf("Adding rule : Allow brk\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);

    /* allow writing up to 512 bytes to fd 1 */
    printf("Adding rule : Allow write upto 512 bytes to FD 1\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 2,
       SCMP_A0(SCMP_CMP_EQ, 1),
       SCMP_A2(SCMP_CMP_LE, 512));

    /* if writing to any other fd, return -EBADF */
    printf("Adding rule : Deny write to any FD except 1 \n");
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(write), 1,
       SCMP_A0(SCMP_CMP_NE, 1));

    /* load and enforce the filters */
    printf("Load rules and enforce \n");
    seccomp_load(ctx);
    seccomp_release(ctx);

    printf("this process is %d\n", getpid());
}
```

Code Source:
https://security.stackexchange.com/questions/168452/how-is-sandboxing-implemented/175373

```c
#include <seccomp.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

void main(void)
{
    /* initialize the libseccomp context */
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    /* allow exiting */
    printf("Adding rule : Allow exit_group\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

    /* allow getting the current pid */
    printf("Adding rule : Allow getpid\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);
    // printf("Adding rule : Deny getpid\n");
    // seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(getpid), 0);

    /* allow changing data segment size, as required by glibc */
    printf("Adding rule : Allow brk\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);

    /* allow writing up to 512 bytes to fd 1 */
    printf("Adding rule : Allow write upto 512 bytes to FD 1\n");
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 2,
        SCMP_A0(SCMP_CMP_EQ, 1),
        SCMP_A2(SCMP_CMP_LE, 512));

    /* if writing to any other fd, return -EBADF */
    printf("Adding rule : Deny write to any FD except 1 \n");
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(write), 1,
        SCMP_A0(SCMP_CMP_NE, 1));

    /* load and enforce the filters */
    printf("Load rules and enforce \n");
    seccomp_load(ctx);
    seccomp_release(ctx);

    printf("this process is %d\n", getpid());
}
```

Example explanation

The code is thoroughly commented so just to summarise the code, the structure for storing seccomp-bpf profile is initialized and 4 allow and 1 deny rules are added to it. Then the filter is loaded and enforced.

The line 41 calls the previously whitelisted (allowed) getpid syscall successfully.

Trying it out

Step 1: Save the code provided above as seccomp_mode2.c and compile it with gcc.

Command: gcc seccomp_mode2.c -o seccomp_mode2 -lseccomp

Step 2: Execute the compiled binary

Command: ./seccomp_mode2

```
root@localhost:~# ./seccomp_mode2
Adding rule : Allow exit_group
Adding rule : Allow getpid
Adding rule : Allow brk
Adding rule : Allow write upto 512 bytes to FD 1
Adding rule : Deny write to any FD except 1
Load rules and enforce
this process is 3392
root@localhost:~#
```

As expected, the program is able to print the PID.

To see the other aspect, comment the lines 16,17 and uncomment lines 18,19

```
14
15      /* allow getting the current pid */
16      //printf("Adding rule : Allow getpid\n");
17      //seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);
18      printf("Adding rule : Deny getpid\n");
19      seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(getpid), 0);
20
```

So, now the getpid syscall is blacklisted.

Compile and run it

```
root@localhost:~# ./seccomp_mode2
Adding rule : Allow exit_group
Adding rule : Deny getpid
Adding rule : Allow brk
Adding rule : Allow write upto 512 bytes to FD 1
Adding rule : Deny write to any FD except 1
Load rules and enforce
this process is -9
root@localhost:~#
```

As expected, the getpid syscall failed and the program is showing some garbage value.

Seccomp and Docker

Seccomp-eBPF (mode 2) is supported by Docker to restrict the syscalls from the containers effectively decreasing the surface area. Seccomp profile can be defined in the form of a JSON file.

Docker's default seccomp profile is a whitelist which blocks 44 syscalls. The default seccomp profile can be found here:
https://github.com/moby/moby/blob/master/profiles/seccomp/default.json

At the time of writing, syscalls that were effectively blocked, can be found here:
https://docs.docker.com/engine/security/seccomp/

Example profile

To block mkdir syscall (http://man7.org/linux/man-pages/man2/mkdir.2.html):

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
```

```
            "name": "mkdir",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        }
    ]
}
```

Trying it out

Step 1: Save the profile content as block_mkdir.json

Command: cat block_mkdir.json

The default action of the profile is to allow the syscall so all syscalls which are not explicitly denied (like mkdir in this case) will be allowed.

Step 2: Run ubuntu:18.04 docker image with this profile and try to create a directory inside the container using mkdir command.

Commands:
docker run -it --security-opt seccomp=block_mkdir.json ubuntu:18.04 bash
cd ~
mkdir test

```
root@localhost:~# docker run -it --security-opt seccomp=block_mkdir.json ubuntu:18.04 bash
root@ebb5f6b095de:/#
root@ebb5f6b095de:/# cd ~
root@ebb5f6b095de:~#
root@ebb5f6b095de:~# mkdir test
mkdir: cannot create directory 'test': Operation not permitted
root@ebb5f6b095de:~#
```

The root user of the container is not able to create a directory in his own home directory. The result will be the same for any other location.

Learning:

- Seccomp has two main modes. Mode 1 is the original mode which is highly restricted and can't be used for a wide variety of needs.
- Mode 2 is flexible due to its support for JSON profile files. This mode is the mostly used one.

References:

- Seccomp (http://man7.org/linux/man-pages/man2/seccomp.2.html)
- Difference between seccomp and linux capabilities (https://security.stackexchange.com/questions/210400/difference-between-linux-capabilities-and-seccomp)
- How Sandboxing is implemented (https://security.stackexchange.com/questions/168452/how-is-sandboxing-implemented/175373#175373)