

<b>Introduction.....</b>	<b>3</b>
<b>UID (User Identifier).....</b>	<b>4</b>
<b>RUID (Real User ID).....</b>	<b>5</b>
<b>EUID (Effective User ID).....</b>	<b>6</b>
<b>SUID (Saved User ID).....</b>	<b>7</b>
<b>GID (Group Identifier).....</b>	<b>8</b>
<b>EGID (Effective Group ID).....</b>	<b>9</b>
<b>RGID (Real Group ID).....</b>	<b>10</b>
<b>SGID (Saved Group ID).....</b>	<b>11</b>
<b>File Permissions.....</b>	<b>12</b>
<b>umask (Set File Mode Creation Mask).....</b>	<b>13</b>
<b>File Permissions are Not Cumulative.....</b>	<b>14</b>
<b>Sticky Bit.....</b>	<b>15</b>
<b>SUID Bit (Set User ID).....</b>	<b>16</b>
<b>SGID Bit (Set Group ID).....</b>	<b>17</b>
<b>ASLR (Address Space Layout Randomization).....</b>	<b>18</b>
<b>ASLR in Statically Linked ELF's.....</b>	<b>19</b>
<b>SAS (Secure Attention Sequence).....</b>	<b>20</b>
<b>Secure Computing Mode (seccomp).....</b>	<b>21</b>
<b>Linux Security Modules (LSM).....</b>	<b>22</b>
<b>LoadPin.....</b>	<b>23</b>
<b>SafeSetID.....</b>	<b>24</b>
<b>Yama.....</b>	<b>25</b>
<b>Keyrings.....</b>	<b>26</b>
<b>AppArmor (Application Armor).....</b>	<b>27</b>
<b>Primary Groups.....</b>	<b>28</b>
<b>Secondary Group.....</b>	<b>29</b>
<b>ACL (Access Control Lists).....</b>	<b>30</b>
<b>PAM (Pluggable Authentication Module).....</b>	<b>31</b>
<b>Capabilities.....</b>	<b>32</b>
<b>chroot (Change Root Directory).....</b>	<b>34</b>
<b>NetFilter.....</b>	<b>35</b>
<b>Chains (iptables).....</b>	<b>36</b>
<b>Table Types (iptables).....</b>	<b>37</b>
<b>nftables.....</b>	<b>38</b>
<b>Secure Execution Mode.....</b>	<b>39</b>
<b>su (Substitute User).....</b>	<b>40</b>

# UID (User Identifier)

UID stands for “User Identifier”, it is a number associated with each Linux user in order to represent it in the Linux kernel. We can think about it like SID<sup>1</sup> in Windows which uniquely identifies a security principal. We can see the UID for a specific user defined in “/etc/passwd”.

In most Linux distributions UIDs 1-500 are reserved for system users, while in distributions like Ubuntu/Fedora the UID for a new user starts from 1000 - by default, the UID of root is 0<sup>2</sup>. Moreover, we can also change the UID of a user with the “usermod”<sup>3</sup> command in the following manner: “sudo usermod -u [NEW\_UID] [USER\_NAME]” - as shown in the screenshot below. Also, all processes executed by the user will get its UID - more on that in a future writeup.

Lastly, it is important to understand that there are three different types of UID: EUID (Effective UID), RUID (Real UID) and SUID (Saved UID).

```
root@localhost:/tmp/troller# id troller
uid=1000(troller) gid=1000(troller) groups=1000(troller)
root@localhost:/tmp/troller# usermod -u 1337 troller
root@localhost:/tmp/troller# id troller
uid=1337(troller) gid=1000(troller) groups=1000(troller)
```

## RUID (Real User ID)

RUID stands for “Real UserID”, which is the user who initiated a specific operation<sup>4</sup>. Thus, we can say that it is basically the UID<sup>5</sup> of the user that started the specific task/process<sup>6</sup>. Overall, RUID is the “uid” field of the Linux’s “struct cred” data structure<sup>7</sup>. This information is also included as part of the “Auxiliary Vector”<sup>8</sup> in the “AT\_UID” entry<sup>9</sup>.

Moreover, there could be specific syscalls that use only the real uid/group id (and not the effective uid - which I am going to detail about in a future writeup), one example of that is access<sup>10</sup>.

Lastly, another example of usage is by the “passwd” command line utility<sup>11</sup>. When executing it gets the permissions of the root user. However, due to the fact the “real uid” is not changed by using a “suid bit”<sup>12</sup> we can’t change the password of a user which is not us (unless we are the “root” user) - as shown in the screenshot below.

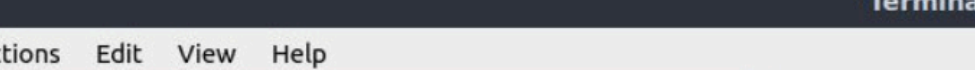
```
[test@localhost ~]$ id
uid=1000(test) gid=1000(test) groups=1000(test)
[test@localhost ~]$ passwd test2
passwd: You may not view or modify password information for test2.
[test@localhost ~]$ passwd test
Changing password for test.
Current password: _
```

## EUID (Effective User ID)

EUID (Effective User ID) is what is mostly used for determining the permissions of a certain task (process/thread) in a Linux system<sup>13</sup>. By default, the EUID is equal to the value of RUID, there are also use cases in which those two are different<sup>14</sup>.

Moreover, there are use cases in which the EUID is different than the RUID for enabling a non-privileged user to access files which are accessed only by privileged users like root<sup>15</sup>. For example the case of the utility “passwd” that needs to alter “/etc/shadow” when the user changes a password - as shown in the screenshot below.

Lastly, the EUID is stored in the “euid” field of “struct cred”<sup>16</sup> that is pointed from the PCB (Process Control Block)/TCB (Thread Control Block) data structure in Linux, which is “struct task\_struct”<sup>17</sup>.



The screenshot shows a terminal window titled "Terminal Session". The command prompt is "Troller \$". The command entered is "ps -a -o comm,euid,ruid,suid,fsuid,egid,rgid,sgid,fsuid". The output is as follows:

COMMAND	EUID	RUID	SUID	FSUID	EGID	RGID	SGID	FSGID
passwd	0	1000	0	0	1000	1000	1000	1000
ps	1000	1000	1000	1000	1000	1000	1000	1000

The "ps" process is highlighted with a red box, indicating that its EUID and RUID are 1000.

# SUID (Saved User ID)

In this context SUID stands for “Saved User ID” (and it is different from SUID bit<sup>18</sup>). It is used when we have a task (process/thread) executing with high privilege (such as root, but not limited to that) which needs to do something in an unprivileged manner. Due to the fact, we want to work in a “least privilege” principle<sup>19</sup>, we need to use the high privileges only when it is a must.

Thus, we use the SUID in order to save the EUID<sup>20</sup> and then do the change which causes the task to execute as an unprivileged user. After finishing the operation/s the EUID is taken back from the SUID<sup>21</sup>.

Lastly, we can use the “setresuid” syscall for setting a different value between EUID and SUID<sup>22</sup> - as shown in the screenshot below. We can see that we can set euid=0 if our suid=0 but we can't do that if suid!=0.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>

void main()
{
    printf("####SUID exmple####\n");
    uid_t euid=1000, ruid=1000, suid=0;
    int val=setresuid(ruid,euid,suid);
    printf("setresuid: ret value: %d\n*****\n",val);
    ruid=-1;euid=-1;suid=-1;
    val=getresuid(&ruid,&euid,&suid);
    printf("getresuid: retv value: %d\neuid=%d,ruid=%d,suid=%d\n*****\n",val,euid,ruid,suid);
    euid=0,ruid=0,suid=0;
    val=getresuid(&ruid,&euid,&suid);
    printf("setresuid: ret value: %d\n*****\n",val);
    euid=1000,ruid=1000;suid=1000;
    val=setresuid(ruid,euid,suid);
    printf("setresuid: ret value: %d\n*****\n",val);
    euid=0;ruid=0;suid=0;
    val=setresuid(ruid,euid,suid);
    printf("setresuid: ret value: %d\n*****\n",val);
}

Troller# ./code
####SUID exmple####
setresuid: ret value: 0
*****
getresuid: retv value: 0
euid=1000,ruid=1000,suid=0
*****
setresuid: ret value: 0
*****
setresuid: ret value: 0
*****
setresuid: ret value: -1
*****
Troller#
```

setresuid fails

# GID (Group Identifier)

GID stands for “Group Identifier”, it is a number associated with each Linux group in order to represent it in the Linux kernel. We can think about it like SID<sup>23</sup> in Windows which uniquely identifies a security principal. We can see the GID for a specific group defined in “/etc/group”.

Moreover, by using groups we can manage which resources users of a group can access. We can get the GID of a group using the “getent” command line utility<sup>24</sup> - as shown in the screenshot below. For creating a new group we can use the “groupadd”<sup>25</sup> command line utility - as also shown in the screenshot below.

Also, there are Linux distributions that reserve the GID range 0–99 for statically allocated groups, and either 100–499 or 100–999 for groups dynamically allocated by the system in post-installation scripts. These ranges are often specified in /etc/login.defs<sup>26</sup> - as shown in the screenshot below.

Lastly, it is important to understand that there are three different types of GID: EUID (Effective GID), RGID (Real GID) and SGID (Saved GID).

```
Troller # getent group root
root:x:0:
Troller # groupadd trollers
Troller # getent group trollers
trollers:x:1001:
Troller # cat /etc/login.defs | grep ^GID
GID_MIN          1000
GID_MAX          60000
```

# EGID (Effective Group ID)

As with EUID<sup>27</sup> we also have EGID (Effective Group ID). It is what is mostly used for determining the permissions of a certain task (process/thread) in a Linux system in the sense of group membership.

Moreover, there are use cases in which the EGID is different from the RGID (Real Group ID) for enabling a non-privileged user to access files which are accessed only by privileged users like root. We can access the EGID using the inside the kernel using the “current\_egid” macro<sup>28</sup>.

Lastly, we can use the “id”<sup>29</sup> command line utility for printing the effective group ID. Also, from user mode we can use the “getegid()” system call to retrieve the effective group id of the calling process/task<sup>30</sup>. There is also a library call with the same name<sup>31</sup> - as shown in the screenshot below taken using copy.sh<sup>32</sup>.

```
000d0240 <getegid@GLIBC_2.0>:
d0240: f3 0f 1e fb      endbr32
d0244: b8 ca 00 00 00   mov     $0xca,%eax
d0249: 65 ff 15 10 00 00 call    */%gs:0x10
d0250: c3              ret
d0251: 66 90          xchg    %ax,%ax
d0253: 66 90          xchg    %ax,%ax
d0255: 66 90          xchg    %ax,%ax
d0257: 66 90          xchg    %ax,%ax
d0259: 66 90          xchg    %ax,%ax
d025b: 66 90          xchg    %ax,%ax
d025d: 66 90          xchg    %ax,%ax
d025f: 90             nop
```

## RGID (Real Group ID)

RGID stands for “Real Group ID”, which is the main group of the user who initiated a specific operation. Thus, we can say that it is basically the GID<sup>33</sup> of the user that started the specific task/process<sup>34</sup>. Overall, RUID is the “uid” field of the Linux’s “struct cred” data structure<sup>35</sup>. This information is also included as part of the “Auxiliary Vector”<sup>36</sup> in the “AT\_UID” entry<sup>37</sup>.

Moreover, there could be specific syscalls that use only the real uid/group id (and not the effective uid - which I am going to detail about in a future writeup), one example of that is access<sup>38</sup>.

Lastly, another example of usage is by the “passwd” command line utility<sup>39</sup>. When executing it gets the permissions of the root user. However, due to the fact the “real uid” is not changed by using a “suid bit”<sup>40</sup> we can’t change the password of a user which is not us (unless we are the “root” user) - as shown in the screenshot below.

```
[test@localhost ~]$ id
uid=1000(test) gid=1000(test) groups=1000(test)
[test@localhost ~]$ passwd test2
passwd: You may not view or modify password information for test2.
[test@localhost ~]$ passwd test
Changing password for test.
Current password: _
```



# SGID (Saved Group ID)

In this context SGID stands for “Saved Group ID” (and it is different from SGID bit). It is used when we have a task (process/thread) executing with high privilege (such as root, but not limited to that) which needs to do something in an unprivileged manner. Due to the fact, we want to work in a “least privilege” principle<sup>41</sup>, we need to use the high privileges only when it is a must.

Thus, we use the SGID in order to save the EGID<sup>42</sup> and then do the change which causes the task to execute as an unprivileged user. After finishing the operation/s the EGID is taken back from the SGID.

Lastly, we can use the “setresgid” syscall for setting a different value between EGID and SGID<sup>43</sup>. We can `egid=0` if our `sgid=0` but we can’t do that if `sgid!=0`. Thus, SGID uses the same concepts as SGID<sup>44</sup>. SGID is also an attribute saved as part of the “struct cred”<sup>45</sup> - as shown in the screenshot below.

```
111  ✓ struct cred {
112      atomic_long_t  usage;
113      kuid_t         uid;          /* real UID of the task */
114      kgid_t         gid;          /* real GID of the task */
115      kuid_t         suid;        /* saved UID of the task */
116      kgid_t         sgid;        /* saved GID of the task */
117      kuid_t         euid;        /* effective UID of the task */
118      kgid_t         egid;        /* effective GID of the task */
119      kuid_t         fsuid;       /* UID for VFS ops */
120      kgid_t         fsgid;       /* GID for VFS ops */
121      unsigned       securebits;   /* SUID-less security management */
122      kernel_cap_t    cap_inheritable; /* caps our children can inherit */
123      kernel_cap_t    cap_permitted; /* caps we're permitted */
124      kernel_cap_t    cap_effective; /* caps we can actually use */
125      kernel_cap_t    cap_bset;    /* capability bounding set */
126      kernel_cap_t    cap_ambient; /* Ambient capability set */
127  #ifdef CONFIG_KEYS
128      unsigned char   jit_keyring; /* default keyring to attach requested
129                                  * keys to */

```

# File Permissions

As you probably know there is a basic saying in Linux: “everything is a file”, so it's no surprise that file permissions are core to the security model used by Linux systems. By using them we can define who can access files/directories. In general, we have three levels of permissions: read, write and execute<sup>46</sup>.

Overall, every file/directory in Linux (in case the filesystem supports permissions) has as part of its metadata three categories of ownership: user, group and other. User, is the owner of the file which by default is the one that created it. Group, it is the primary group<sup>47</sup> of the user when the file/directory was created. Other, which applies to all other users in the system<sup>48</sup>.

Moreover, we can view those permissions using the “-l” switch of the “ls”<sup>49</sup> command - as shown in the screenshot below. There are also special permissions SUID<sup>50</sup>, SGID<sup>51</sup> and sticky bit<sup>52</sup> - demonstrated below.

Lastly, we can change the permissions of files (aka file mode bits) using the “chmod” command, this can be done using the form “[ugoa]\*([-+=])([rwxXst]\*[ugo]))+”<sup>53</sup> - as shown in the example below. By the way, we can also use numeric values for setting the permissions where “read=4”, “write=2” and execute=1” - example is also shown in the screenshot below. Also, SUID/GUID/Sticky can be defined by adding a fourth number where “SUID=4”, “SGID=2” and “Sticky bit=1” - also shown in the example below.

```
root@localhost:~# ls -l
total 2
-rw-r--r-- 1 root root 0 file1
-rw-r--r-- 1 root root 0 file2
-rw-r--r-- 1 root root 0 file3
root@localhost:~# chmod u-w file1
root@localhost:~# chmod g+x file2
root@localhost:~# chmod o-r file3
root@localhost:~# ls -l
total 2
-r--r--r-- 1 root root 0 file1
-rw-r-xr-- 1 root root 0 file2
-rw-r----- 1 root root 0 file3
root@localhost:~# chmod 777 file3
root@localhost:~# chmod 421 file2
root@localhost:~# chmod 600 file1
root@localhost:~# ls -l
total 2
-rw-r--r-- 1 root root 0 file1
-r--w---x 1 root root 0 file2
-rwxr-xr-x 1 root root 0 file3
root@localhost:~# chmod 4777 file1
root@localhost:~# chmod 2777 file2
root@localhost:~# chmod 1777 file3
root@localhost:~# ls -lah
total 2.5K
drwxr-xr-x 2 root root 0
drwxr-xr-x 5 root root 51
-rwxr-xr-x 1 root root 0 file1
-rwxr-xr-x 1 root root 0 file2
-rwxr-xr-x 1 root root 0 file3
```

## umask (Set File Mode Creation Mask)

When creating a new file/directory the default file mode permissions<sup>54</sup> are 666 (rw-rw-rw), however those permissions are masked/filtered by the umask (Set File Mode Creation Mask) value. Thus, if we have “umask=0022” the permissions of a newly created file is set to 644 (rw-r--r--). In case of “umask=0077” the permissions of a newly created file is set to 600 (rw-----) and for “umask=0000” we get 666 (rw-rw-rw-) - as shown in the screenshot below.

Overall, “umask” is a system call used for setting the file mode creation mask. This system call always succeeds and the previous value of the mask is returned. “umask” is used by in conjunction with syscalls like “open”<sup>55</sup> and “mkdir”<sup>56</sup>.

Moreover, as opposed to the “chmod”<sup>57</sup> syscall which affects the permissions of a specific file/directory “umask” affects every file/directory created by the user. In most Linux distributions the “umask” value are configured in system wide configuration files like: “/etc/profile” or “/etc/bash.bashrc”<sup>58</sup>.

Lastly, based on the shell environment used, “umask” can be a dedicated binary or a builtin command of the shell. There are case in which “umask” binary is used we can just read the value and not change it, because it will change it for a different process session, so for altering the umask value in those cases the builtin shell command is need<sup>59</sup>.

```
root@localhost:/tmp/files# umask
0022
root@localhost:/tmp/files# touch troller1
root@localhost:/tmp/files# umask 077
root@localhost:/tmp/files# touch troller2
root@localhost:/tmp/files# umask 000
root@localhost:/tmp/files# touch troller3
root@localhost:/tmp/files# ls -lah
total 2.5K
drwxr-xr-x 2 root root 0
drwxrwxrwt 5 root root 51
-rw-r--r-- 1 root root 0 2024 troller1
-rw----- 1 root root 0 2024 troller2
-rw-rw-rw- 1 root root 0 2024 troller3
```

# File Permissions are Not Cumulative

As opposed to what we might think, file permissions<sup>60</sup> under Linux are not cumulative. For example if we have a file that our user only has “read” permission and our primary group<sup>61</sup> has “read and write” permissions we will still just get the “read” permission. Even if any other user has full permissions to the file we won’t be able to alter it - as shown in the screenshot below.

Thus, we can say that the permissions are checked in the following order: user, group and other. In case there are any permissions given to the subject accessing the object (file/directory) the check is stopped and by that file permissions are not cumulative - as demonstrated below.

Lastly, this fact is not relevant for the “root” user due to the fact it has the capability “CAP\_DAC\_OVERRIDE”<sup>62</sup>, which overrides the read/write/execute file permission check - as also shown below. This is of course relevant for any user holding that capability.

```
[test@localhost test]$ id
uid=1000(test) gid=1000(test) groups=1000(test)
[test@localhost test]$ ls -l ./file
-r--rw-rwx 1 test test 8 Mar 12 2024 ./file
[test@localhost test]$ cat ./file
Tr0ll3R
[test@localhost test]$ echo BlA > ./file
bash: ./file: Permission denied
[test@localhost test]$

root@localhost:/tmp/test# id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:/tmp/test# ls -la file
-r--r----- 1 root root 8 Mar 12 2024 file
root@localhost:/tmp/test# cat ./file
Tr0ll3R
root@localhost:/tmp/test# echo BlA > ./file
root@localhost:/tmp/test# cat file
BlA
root@localhost:/tmp/test# chmod 0000 ./file
root@localhost:/tmp/test# chown test ./file
root@localhost:/tmp/test# chgrp test ./file
root@localhost:/tmp/test# ls -lah ./file
----- 1 test test 4 Mar 12 2024 ./file
root@localhost:/tmp/test# cat ./file
BlA
root@localhost:/tmp/test# echo TeST > ./file
root@localhost:/tmp/test# cat ./file
TeST
root@localhost:/tmp/test# _
```

# Sticky Bit

Beside the ordinary permissions that a file/directory can have in Linux (read, write & execute) we can also assign specific permissions bits that have a special meaning: suid, sgid and sticky bit.

Have you ever asked yourself what the “t” in the output of “ls -l” stands for? (as you can see in the screenshot below taken from copy.sh). As you can see everyone can read and write to “/tmp”, but in the place of “execute” there is a “t” (and not an “x”) - it means “sticky bit”.

The goal of “sticky bit” when setting it on a directory is to allow the removal of files in the directory only by their owner. You can see a full demonstration of that in the image below. As shown even if the file (/tmp/test1\_file) has full permissions for everyone it still can’t be deleted by the user test2 (by the way the permissions of the file are not relevant as we will show in a different writeup).

```
root@localhost:~# ls -ld /tmp
drwxrwxrwt 4 root root 117 Feb 20 20:02 /tmp
root@localhost:~# su test2
[test2@localhost root]$ cd /tmp
[test2@localhost tmp]$ ls -lh
total 512
-rwxrwxrwx 1 test1 test1 2 Sep 22 2022 test1_file
[test2@localhost tmp]$ rm -f ./test1_file
rm: cannot remove './test1_file': Operation not permitted
[test2@localhost tmp]$ _
```

## SUID Bit (Set User ID)

Beside the ordinary permissions that a file/directory can have in Linux (read, write & execute) we can also assign specific permissions bits that have a special meaning: suid, sgid and sticky bit.

Have you ever asked yourself what the “s” in the output of “ls -l” stands for? As you can see in the screenshot below taken from an Ubuntu 22.04 VM regarding the “passwd” executable.

If we have a file with a “suid bit” it will be executed using the permissions of the owner of the file. It is important to understand that it sets the “euid” but not “ruid” - As you can see in the screenshot below. Information about “euid”, “ruid”, “fsuid” and more are going to be covered as part of the description about “struct cred”<sup>63</sup> in the future.

For now all you need to know is that “euid” (Effective UID) is used for most of the permissions checks and “ruid” (Real UID) is the uid (User Identifier) of the user that started the executable. Due to that even though “passwd” runs using root it does not allow changing a password that is not the one that started it (despite root that can change any user's password).

```
Troller $ ls -lah `which passwd`  
-rwsr-xr-x 1 root root 59K Mar 14 2022 /usr/bin/passwd  
Troller $ id | cut -d'"' -f1  
uid=1000  
Troller $ passwd  
Changing password for root:  
Current password:   
New password:   
Retype new password:   
passwd: password updated successfully  
Troller $ ps -a -o comm,euid,ruid,suid,fsuid,egid,rgid,sgid,fsgid  
COMMAND      EUID  RUID  SUID  FSUID  EGID  RGID  SGID  FSGID  
passwd        0    1000    0     0    1000  1000  1000  1000  
ps            1000  1000  1000  1000  1000  1000  1000  1000  
Troller $
```

## SGID Bit (Set Group ID)

SGID is a special permission, its meaning is based if it is set on a file or a directory. If it is set on a file it allows the file to be executed with the permissions of the group that owns the file - it is similar to SUID which does the same with the user that owns the file<sup>64</sup>. In case of a directory, if we set the SGID bit, any files created in the directory will have their group ownership set to that of the directory owner<sup>65</sup>.

Thus, this permission is marked with “s” in the location that specifies the “execute” permission. In order to set SGID we can use the “chmod”<sup>66</sup> command, after doing so the execute indication “x” in the group portion is going to change to “s”.

This can be verified using the “-l” switch of the “ls”<sup>67</sup> command - as shown in the screenshot below. By the way, if we remove the execute permission the indication is changed from “s” to “S” - as also shown in the screenshot below.

Lastly, as with the normal permissions in which we have the numeric system for setting/removing them by using a 3 numbers, by adding another one we can also specify the other special permissions<sup>68</sup>. For setting the SGID bit we can use a number in the pattern of 2xyz, where x/y/z are 1 or 2 or 4<sup>69</sup>.

```
root@localhost:/tmp/troller# ls -l
total 1
-rwxr-xr-x 1 root root [REDACTED] 2023 troller.log
root@localhost:/tmp/troller# chmod g+s ./troller.log
root@localhost:/tmp/troller# ls -l
total 1
-rwxr-sr-x 1 root root [REDACTED] 2023 troller.log
root@localhost:/tmp/troller# chmod g-x ./troller.log
root@localhost:/tmp/troller# ls -l
total 1
-rwxr-Sr-x 1 root root [REDACTED] 2023 troller.log
```

# ASLR (Address Space Layout Randomization)

ASLR is a mitigation techniques used to increase the difficulty of running arbitrary code in case of an exploitation of a memory corruption vulnerability such as a buffer overflow (We will go over it in a different writeup but you can read the first article about it from phrack<sup>70</sup> magazine).

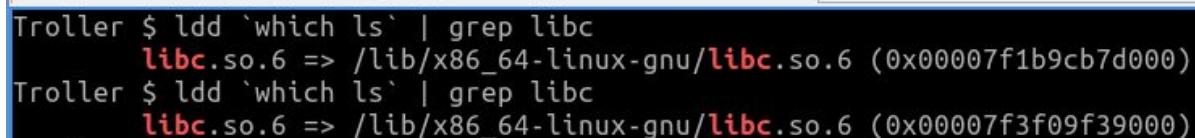
Basically what ASLR does is to randomly select the base address of the executable, it also randomizes the heap, stack and loaded libraries. Thus, in case an attacker can control the flow of execution (such as controlling the instruction pointer), the location of the arbitrary code to execute is unknown.

In order for ASLR to work we need support in the OS (mostly the loader of executables) and in the executable itself, it should be compiled as PIE (position independent code) and have support relocations (in case of absolute addressing used). More on PIE and relocations in future writeups.

There are several limitations in case of ASLR, some of them are general and some of them are relevant for only specific implementations. In that sense different operating systems support ASLR in different manners. For example, in Linux the base address is randomized every call to a syscall from the family of `execve()` ('man 2 `execve`'). Thus, if we just use `fork()` the base address won't change. The screenshot below shows the difference between the address of `libc` between two executions of the command "ls". We will dive more into specific OS implementations in the future. Also, ASLR is not relevant to other attack types such as "Data Only" (more on them in the future).

You have to remember that there are several ways to bypass ASLR (depending on the OS and other factors). You can search online for those techniques or wait for the next writeup. Also, although there are bypasses, ASLR is a must in my opinion.

Lastly, there is also an ASLR version for the kernel itself, it is usually called KSLR (Kernel Space Layout Randomization).



```
Troller $ ldd `which ls` | grep libc
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1b9cb7d000)
Troller $ ldd `which ls` | grep libc
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3f09f39000)
```



# ASLR in Statically Linked ELFs

When compiling code to a statically linked ELF we bake all the code our binary needs from shared libraries inside our own executable<sup>71</sup>. The question which arises is how and if it effects the ASLR<sup>72</sup> posture of the process executing the statically linked binary?

Thus, as we can see in the screenshot below when linking the binary statically (using “-static”) any time we execute it the addresses of the stack/heap/vdso/vvar memory regions are randomized. However, the memory regions mapped from the binary are not randomized.

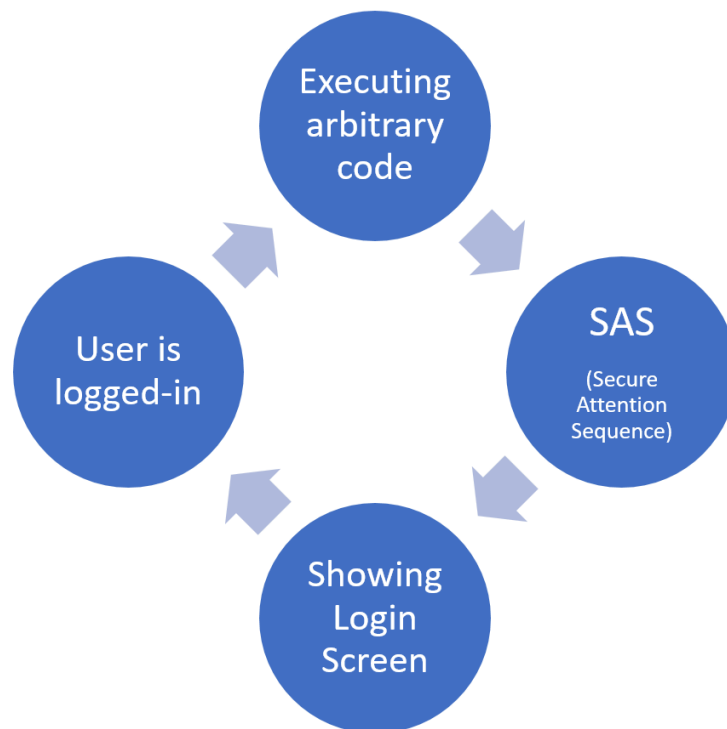
In order to fix this we can use “-static-pie” which can load the memory regions mapped for the statically linked binary to randomized addresses without the need of the dynamic linker<sup>73</sup>. We can also see that in the screenshot below.

```
root@localhost:~# gcc -static ./troller.c -o ./troller
root@localhost:~# file ./troller
./troller: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, BuildID[sha1]=ad45a04b2b1505fc27b218c19a1b6c2722b0f21f, for GNU/Linux 4.4.0, with debug_info, not stripped
root@localhost:~# ./troller &
[1] 1350
root@localhost:~# cat /proc/$(pidof troller)/maps
00049000-00049000 r--p 00000000 00:17 98081 /tmp/troller/troller
00049000-0004d000 r-xp 00001000 00:17 98081 /tmp/troller/troller
0004d000-0004c000 r--p 00005000 00:17 98081 /tmp/troller/troller
0004c000-0004f000 r--p 00003000 00:17 98081 /tmp/troller/troller
0004f000-0004e000 rw-p 00006000 00:17 98081 /tmp/troller/troller
0004e000-00040000 rw-p 00000000 00:00 0
00902000-00924000 rw-p 00000000 00:00 0 [heap]
b7ef0000-b7f01000 r--p 00000000 00:00 0 [uvar]
b7f01000-b7f03000 r-xp 00000000 00:00 0 [vdso]
bf497000-bf4b8000 rw-p 00000000 00:00 0 [stack]
root@localhost:~# kill -9 $(pidof troller)
root@localhost:~# ./troller &
[2] 1354
[1] Killed
root@localhost:~# cat /proc/$(pidof troller)/maps
00049000-00049000 r--p 00000000 00:17 98081 /tmp/troller/troller
00049000-0004d000 r-xp 00001000 00:17 98081 /tmp/troller/troller
0004d000-0004c000 r--p 00005000 00:17 98081 /tmp/troller/troller
0004c000-0004f000 r--p 00003000 00:17 98081 /tmp/troller/troller
0004f000-0004e000 rw-p 00006000 00:17 98081 /tmp/troller/troller
0004e000-00040000 rw-p 00000000 00:00 0
00a8a000-00aac000 rw-p 00000000 00:00 0 [heap]
b7f25000-b7f29000 r--p 00000000 00:00 0 [uvar]
b7f29000-b7f2b000 r-xp 00000000 00:00 0 [vdso]
bf8f4000-bf91e000 rw-p 00000000 00:00 0 [stack]
root@localhost:~# gcc -static-pie ./troller.c -o ./troller
root@localhost:~# kill -9 $(pidof troller)
root@localhost:~# ./troller &
[3] 1363
[2] Killed
root@localhost:~# cat /proc/$(pidof troller)/maps | grep troller
b7ea4000-b7ea7000 r--p 00000000 00:17 98087 /tmp/troller/troller
b7ea7000-b7f0c000 r-xp 00003000 00:17 98087 /tmp/troller/troller
b7f0c000-b7f3d000 r--p 00006000 00:17 98087 /tmp/troller/troller
b7f3d000-b7f40000 r--p 00008000 00:17 98087 /tmp/troller/troller
b7f40000-b7f42000 rw-p 0000b000 00:17 98087 /tmp/troller/troller
root@localhost:~#
```

# SAS (Secure Attention Sequence)

SAS (sometimes called SAK - “Secure Attention Key”) is a special key sequence/combination that triggers the opening of the login screen. The goal of SAS is to make sure that the login screen is not spoofed. Due to the fact that the OS interacts with the hardware it can detect the SAS (think about keyboard interrupts) and then suspend any application and run the login screen<sup>74</sup> - As shown in the diagram below.

If you want to read about the support of Linux (kernel 2.4.2) for SAS<sup>75</sup> (SAK in the documentation). This concept is also relevant for other operating systems like Windows<sup>76</sup>.



# Secure Computing Mode (seccomp)

“Secure Computing Mode” (seccomp) is a Linux kernel feature that allows restricting system calls that applications can use, by doing that it reduces their attack surface. With seccomp a process can perform a one-way transition to a “secure mode”, in that mode the process can just run the following syscall: read, write, sigreturn and exit. It was merged into the Linux kernel in version 2.6.12 (released in March 2005).

We can configure seccomp by the libseccomp<sup>77</sup>, the prctl<sup>78</sup> system call and/or the seccomp<sup>79</sup> syscall and/or other CLI tools<sup>80</sup>. Due to its capabilities seccomp is commonly used in sandboxes like in Docker, LXC, systemd’s sandboxing, kubernetes and even within internet browsers (like Google Chrome and Mozilla Firefox) as an additional layer of security. By that, seccomp can help prevent malicious applications from exploiting vulnerabilities or gaining unauthorized access to resources. Also, it can be used to limit an application’s ability to access the network or access the file system<sup>81</sup>.

Since kernel 4.14 there is also a “/proc” interface for seccomp located at “/proc/sys/kernel/seccomp”. There we can find “actions\_avail” (a read-only list of seccomp filter return actions supported by the kernel) and “actions\_logged” (a read-write list of filter actions that are allowed to be logged)<sup>82</sup>. Also, since kernel 4.14 we can log actions returned by seccomp in the audit log.

In the example shown in the screenshot below we can see how we can block a syscall (mkdir) by passing the relevant parameters to docker which uses seccomp<sup>83</sup>.

```
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:~#
```

# Linux Security Modules (LSM)

“Linux Security Modules” (LSM) is a framework which allows the kernel to support various security modules. It was mainly designed to allow implementation of MAC (Mandatory Access Control) with minimal changes to the Linux kernel. For now, you should know that MAC is an organizational-wide security policy that users can’t override (I am going to post about MAC and DAC in more detail separately).

Despite the name containing “Modules” it is not implemented as loadable kernel modules (“.ko” files). The LSM framework is of course optional and needs to be enabled by the CONFIG\_SECURITY variable.

If we want to get a list of the running LSMs we just need to read “/sys/kernel/security/lsm” - see the screenshot below (taken from Ubuntu 22.04.01 LTS). It is a comma separated list, at minimum it includes the “capabilities system”. The reason for seeing “capabilities” is due to the fact it was implemented as a “security module”. You can also see the source code for capabilities including “lsm\_hooks.h”<sup>84</sup> and thus using different LSM’s enums, macros and functions.

One of the biggest design goals of LSM is to avoid manipulation of the syscall table in order to implement the “security modules”. It is done in order to avoid issues of race conditions and scale problems. Having said that, LSM was not created in order to provide a generic instrumentation/tracing/hooks mechanism for the Linux kernel<sup>85</sup>.

There are a couple of security features which are implemented as “security modules” like: AppArmor, SELinux, TOMOYO, LoadPin, LandLock and Smack - part of them appear in the screenshot below. A detailed explanation about them will be posted separately.

```
Troll # cat /sys/kernel/security/lsm  
lockdown,capability,landlock,yama,apparmorTroll #
```

# LoadPin

After talking about LSM<sup>86</sup> (“Linux Security Modules”) it is time to show different technologies which leverage them. The goal of LoadPin is to ensure that all the files the kernel can load/execute reside on the same filesystem. The idea is that the file system would be based on a read-only device (for example think about a DVD/CD-ROM/any other RO hardware device, it could be also software based (but it has its own drawbacks). You can go over the code of LoadPin<sup>87</sup>.

Among the kernel files artifacts are: kernel modules, firmware, security policies and kexec images (kexec is a syscall which enables loading and booting to a different kernel from the current running one - more on that in a future writeup).

The way it is done is by trapping the kernel’s file reading interface and basically pinning (it for kernel code loading) to the first file system which is used. Thus, any file that is part of a different file system will be rejected - As you can see the screenshot below<sup>88</sup>.

One of the use-cases for using LoadPin is to ensure the integrity of kernel code (such as \*.ko file) without signing them (I will detail kernel module signing in a different writeup). It is important to note that I am not recommending using/not using LoadPin, my goal is to explain about it (and leave to the reader what to do with it ;-).

In order to enable LoadPin (which is supported since kernel 4.7) we need to set CONFIG\_SECURITY\_LOADPIN before building the kernel. If it is enabled we can also toggle it using the kernel command line (“loadpin.enforce=0” or “loadpin.enforce=1”).

```
root@esomimx6s:~# insmod /media/sda1/esom_verify_sys.ko
```

```
[ 889.996504] LoadPin: kernel-module denied obj="/media/sda1/esom_verify_sys.ko"  
pid=734 cmdline="insmod /media/sda1/esom_verify_sys.ko"
```

```
insmod: ERROR: could not insert module /media/sda1/esom_verify_sys.ko: Operation not  
permitted
```

# SafeSetID

“SafeSetID” is an LSM that was merged in kernel 5.1. The goal of “SafeSetID” is to restrict the transitions to target UID/GID from current UID/GID based on a system wide whitelist. Thus, it governs the family of syscalls which allows those types of transitions (like seteuid and setegid. In general set\*uid/set\*gid).

One of the most used use-cases for “SafeSetID” is to enable a non-root application to transition to other untrusted UIDs/GIDs without the need of giving it an uncontrolled CAP\_SET{U/G}ID capabilities<sup>89</sup>.

It is important to understand that we still grant CAP\_SET{U/G}ID capabilities to the non-root application, however “SafeSetID” allows us to restrict its actions. Thus, we can limit the application from entering/creating a new user namespace or setting the uid to 0 (root).

The configuration of “SafeSetID” is done using performing manipulation on files residing on a securityfs mounting point. The relevant files are “safesetid/uid\_allowlist\_policy” and “safesetid/gid\_allowlist\_policy” (the format of adding a policy is ‘<UID>:<UID>’ or ‘<GID>:<GID>’). By the way, on my Ubuntu 22.04 VM securityfs is mounted on “/sys/kernel/security”.

We can go over the code of “SafeSetID” as part of the source code of the Linux kernel<sup>90</sup>. Moreover, you can see on the image below the source code for creating of the configuration files entries in securityfs<sup>91</sup>.

```
uid_policy_file = securityfs_create_file("uid_allowlist_policy", 0600,
                                         policy_dir, NULL, &safesetid_uid_file_fops);
if (IS_ERR(uid_policy_file)) {
    ret = PTR_ERR(uid_policy_file);
    goto error;
}

gid_policy_file = securityfs_create_file("gid_allowlist_policy", 0600,
                                         policy_dir, NULL, &safesetid_gid_file_fops);
if (IS_ERR(gid_policy_file)) {
    ret = PTR_ERR(gid_policy_file);
    goto error;
}
```

# Yama

“Yama” is an LSM that was merged in kernel 3.4. The goal of “Yama” is to restrict the usage of the “ptrace” syscall (“man 2 ptrace”). Limiting the usage of “ptrace” is one way to isolate between running applications, thus even in case one application is breached it won’t be able to attach to other running applications (like browsers, crypto wallets, encryption apps, remote connection session and more) and read/write to sensitive data or change the flow. In order to include “Yama” the CONFIG\_SECURITY\_YAMA should be enabled while building the kernel. The configuration of “Yama” could be configured at runtime using “/proc/sys/kernel/yama”, which includes “ptrace\_scope” (as shown in the screenshot below).

Based on the kernel documentation<sup>92</sup> “ptrace\_scope” can hold 4 different values (0-3). “0” (“classic ptrace permissions”) allows attaching to any running application with the same uid unless it was marked as undumpable. “1” (“restricted ptrace”) allows attaching to running applications based on their relationship, by default only descendants applications could be attached to (you can also change the behavior of the relationship using “prctl” syscall with the option “PR\_SET\_TRACER”). “2” (“admin-only attach”) allows attaching only from applications holding the “CAP\_SYS\_PTRACE” capability. “3” (“no attach”) blocks all running applications from attaching to any other running application.

A demonstration of the different values described earlier and their relevant impact on “ptrace” is shown in the screenshot below (just as a reminder “strace” leverages the “ptrace” syscall for attaching running applications). You can go over the code of “Yama” as part of the Linux kernel source code<sup>93</sup>.

```
Troller $ cat /proc/sys/kernel/yama/ptrace_scope
1
Troller $ strace -o /tmp/log `which pwd`
/tmp
Troller $ cat /proc/sys/kernel/yama/ptrace_scope
2
Troller $ strace -o /tmp/log `which pwd`
strace: test_ptrace_get_syscall_info: PTRACE_TRACEME: Operation not permitted
strace: ptrace(PTRACE_TRACEME, ...): Operation not permitted
Troller $ sudo strace -o /tmp/log2 `which pwd`
/tmp
```

# Keyrings

When writing an application sometimes a need for storing sensitive data elements (like tokens, passwords and cryptographic keys) arises. For that Linux provides “keyrings” which is a data store that allows applications to access data securely without exposing it to other applications/processes/users. Based on the man page “keyrings” is an in-kernel key management and retention facility<sup>94</sup>. Overall, “keyrings” are used by different types of applications such as authentication servers, web servers and database servers. Examples for those types are: MySQL<sup>95</sup>.

In order to use “keyrings” we can leverage on of the following syscalls: “add\_key()”<sup>96</sup>, “request\_key()”<sup>97</sup> or “keyctl()”<sup>98</sup>. Each key has several attributes as follows: serial number (ID), type, description (name), payload (data), access rights, expression time and reference count. The types of keys which are supported are: “keyring”, “user”, “logon” and “big\_key”<sup>99</sup>. They are different libraries/modules in a variety of programming languages that enable programmers to read/write data into/from keyring. An example in Python is shown in the screen below.

Moreover, there are different entries in proc that give us information about the keyrings, we are going to focus only on two. “/proc/keys” which is relevant since kernel 2.6.10, it displays all the keys the reading thread has view permissions. “/proc/key-users” which is also relevant since kernel 2.6.10, that shows various information for each uid that has at least one key on the system<sup>100</sup>. Lastly, we can also go over the kernel code that handles keyring<sup>101</sup>. Also there is “keyutils” which is a library and a set of utilities that allows access to the in-kernel keyrings facility<sup>102</sup>.

```
Troller$ cat keyring_demo.py
import keyring
#Creating a keyring password item
keyring.set_password('Test', 'troller', 'Tr0LeR')
#Retrieving the keyring password item
password = keyring.get_password('Test', 'troller')
print("The password read from the keyring store is : " + str(password))
Troller$ python3 keyring_demo.py
The password read from the keyring store is : Tr0LeR
```



# AppArmor (Application Armor)

“AppArmor” (Application Armor) is an LSM<sup>103</sup> which allows an administrator to create a per-program profile which restricts what an application can do. It basically provides an extension for the DAC (Discretionary Access Control) under Linux by providing MAC (Mandatory Access Control), which constrains a user (subject) can perform on a operating system object<sup>104</sup>.

Moreover, one of the differences between “AppArmor” and other MAC systems is that it is path-based. AppArmor’s security model is bound to access control attributes granted to specific programs and not users. It was first seen in Immunix and later included in Linux distributions like Ubuntu<sup>105</sup>. Thus, profiles contain the lists of access control rules which are loaded and used by “AppArmor”. By default, in Ubuntu those profiles are stored in “/etc/apparmor.d”<sup>106</sup>. We can use the “apparmor\_status” command to know what profiles are loaded and the current status<sup>107</sup>.

In every profile there are two main types of rules: “path entries” and “capability entries”. “Path Entries” define which files an application can access in the file system. “Capability Entries” define what privileges a confined process is allowed to use. An example of a profile for “/bin/ping” is shown in the screenshot below<sup>108</sup>.

Also, “AppArmor” core functionality is part of the Linux mainline since kernel 2.6.3.6<sup>109</sup>. You can also go over the source code of “AppArmor” as part of the Linux source code in “/security/apparmor”<sup>110</sup>. In order to enable “AppArmor” we need set “CONFIG\_SECURITY\_APPARMOR=y”, we will still need to install also the usermode tools<sup>111</sup>. By the way, “AppArmor” is seen as an alternative to “SELinux”<sup>112</sup>.

```
#include <tunables/global>
/bin/ping flags=(complain) {
    #include <abstractions/base>
    #include <abstractions/consoles>
    #include <abstractions/nameservice>

    capability net_raw,
    capability setuid,
    network inet raw,

    /bin/ping mixr,
    /etc/modules.conf r,
}
```

# Primary Groups

Overall, a group is a convenient way to combine users/other groups as one entity in order to manage them as a single unit (such as with permissions). The goal of a primary group is that the operating system can assign it to files/directories that the user is creating<sup>113</sup>.

Overall, GID (group identifier) is used in order to uniquely identify the primary group ID that the user belongs to. By the way, we can see it using the “id”<sup>114</sup> command (it is the data which follows “gid=”), or by using the “-gn” switch - as shown in the screenshot below<sup>115</sup>.

Moreover, we can change it using the “usermod” tool<sup>116</sup>, it is important to know that for the change to be visible we need to login again - as shown in the screenshot below. We can also see it as the first group in the output of the “groups”<sup>117</sup> command - as also shown in the screenshot below. The information about the primary groups is saved as part of “/etc/passwd”<sup>118</sup>. Lastly, a user can be part of only one primary group at a time. In parallel the information about the secondary groups is saved in “/etc/group”.

```
Troller $ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin)
Troller $ id -gn
user
Troller $ groups
user adm cdrom sudo dip plugdev lpadmin
Troller $ cat /etc/passwd | grep "user:"
user:x:1000:1000:user:/home/user:/bin/bash
Troller $ sudo usermod -g cdrom user
Troller $ id -gn
user
Troller $ sudo login user
Password:
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-76-generic x86_64)
last login: [redacted] Oct [redacted] 2023 on pts/3
Troller $ id -gn
cdrom
Troller $
```

## Secondary Group

In general, we can divide the groups in Linux to two main types: primary<sup>119</sup> and secondary. A secondary group is one/more groups which a user is also part of in parallel to the primary group<sup>120</sup>.

Thus, when creating a new user with the “useradd”<sup>121</sup> command the user is added to a new primary group which has the same name as the user. In order to create new groups we can use the “groupadd”<sup>122</sup> command - as shown in the screenshot below. When adding users to groups we can use the “gpasswd”<sup>123</sup>, those are added as secondary groups- as also shown in the screenshot below.

Lastly, the configuration of secondary groups is stored in “/etc/group”<sup>124</sup>. We can also say that secondary groups are those groups which already created users are added<sup>125</sup>.

```
root@localhost:~# useradd test
root@localhost:~# id test
uid=1000(test) gid=1000(test) groups=1000(test)
root@localhost:~# groupadd test2
root@localhost:~# groupadd test3
root@localhost:~# gpasswd -a test test2
Adding user test to group test2
root@localhost:~# gpasswd -a test test3
Adding user test to group test3
root@localhost:~# id test
uid=1000(test) gid=1000(test) groups=1000(test) 1001(test2),1002(test3)
root@localhost:~# cat /etc/group | grep test2
test2:x:1001:test
root@localhost:~# cat /etc/group | grep test3
test3:x:1002:test
```

# ACL (Access Control Lists)

In general, ACL (Access Control Lists) provides the ability to set permissions to a file/directory in a more granular way than the normal Linux file permissions<sup>126</sup>. This can be done without changing the ownership or the granular Linux permissions<sup>127</sup>.

Overall, in order to set or retrieve a file access control list we can use the “getfacl”<sup>128</sup> and the “setfacl”<sup>129</sup> command line utilities - as shown in the screenshot below. In case the filesystem does not support ACL or the feature is not enabled an error of “operation not supported” is returned.

Lastly, to enable ACL we need to mount the filesystem with the “acl” option (for example this can be done using fstab entries)<sup>130</sup>. We can also use “tune2fs” for listing the default mounting options of a filesystem, by default in case of btrfs and ext2/3/4 the acl option is enabled<sup>131</sup>. In case acl is configured of a file a “+” character appears in the output of “ls -l”<sup>132</sup> - as shown in the screenshot below.

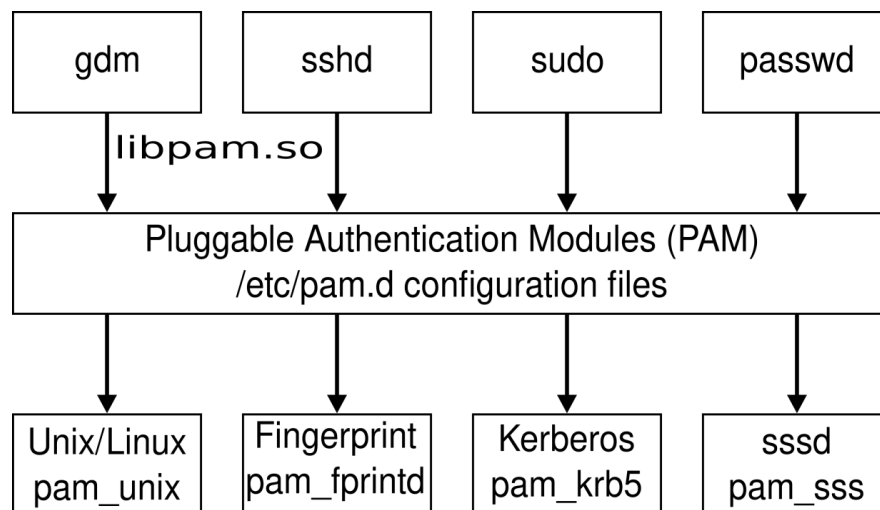
```
Troller $ ls -l troller.txt
---rwx---+ 1 root root 7 Jun 15 02:41 troller.txt
Troller $ id
uid=1001(troller) gid=1001(troller) groups=1001(troller),100(users)
Troller $ cat ./troller.txt
Tr0LeR
Troller $ getfacl ./troller.txt
# file: troller.txt
# owner: root
# group: root
user::---
user:troller:rwx
group::---
mask::rwx
other::---
```

# PAM (Pluggable Authentication Module)

The goal of PAM (Pluggable Authentication Module) is to separate the task of authentication for applications (for example login, sshd, ftpd and gdm). This is to avoid the need for every developer to code his own authentication checks. PAM supports local user authentication and also authentication of users defined in a centralized location (for example leveraging the kerberos protocol). A user can enter a username and password/certificate/fingerprint and this information is authenticated using the correct method<sup>133</sup>.

Overall, the different applications are linked with the “libpam.so” library - as shown in the screenshot below. The functions in this library provide for PAM<sup>134</sup>. An example function is the “pam\_authenticate” which is used for authenticating users. In case of successful completion, the function returns PAM\_SUCCESS<sup>135</sup>.

Lastly, the configuration of PAM is stored by default at “/etc/pam.conf”. Also, the configuration of PAM can be stored as the content of the “ /etc/pam.d/” directory. In case the directory exists Linux-PAM ignores the “/etc/pam.conf”<sup>136</sup>. We can also go over the code of PAM for more information<sup>137</sup>.



# Capabilities

Historically, Linux has had two levels of permissions relevant from process: low privilege (effective uid is not 0) and high privilege (effective uid is 0, aka root). Since kernel 2.2 Linux has broken up the high privileges of the root user into smaller distinct units called capabilities. We can assign only selected capabilities for a process/executable without the need to give the full root access. Thus, limiting the risk due to the reduction in the set of privileges granted.

Overall, there are 5 capabilities sets for a task (process or thread): CapEff (Effective Capabilities), CapPrm (Permitted Capabilities), CapInh (Inherited Capabilities), CapBnd (Bounding Set) and CapAmb (Ambient Capabilities Set). Let us go over each of these.

CapEff, this set represents all the capabilities the process is using at a specific moment in time. Those are the privileges which the kernel performs permission checks on.

CapPrm, this set is a superset which acts as a limiting boundary for the effective set. Those capabilities which are not set cannot be enabled in the effective set (they are some edge cases that we are going to talk about in the following write-up).

CapInh, specifies all the capabilities allowed to be inherited from parent to child, after a call to `execve` syscall (for privilege process/threads).

CapBnd, by using this we can block capabilities that we don't want a process to receive, only those that are in the set will be allowed in the permitted and inherited sets.

CapAmb, applies to non-suid (non privileged) executables that don't have file capabilities (more about it in the next write-up). The goal is to keep capabilities in case of calling the `execve` syscall family. It is important to know that not all the capabilities in the set can be kept like those which are dropped if they are not in the inheritable/permitted capability set.

In order to see the different sets for a task we can read the file `"/proc/[pid]/status"` (for a process) or `"/proc/[pid]/task/[tid]/status"` (for a specific thread) — as shown in the screenshot below (taken from a `copy.sh`).

There are different capabilities such as: `CAP_AUDIT_LOG` (gives the ability to write data to the kernel audit log), `CAP_CHOWN` (gives the ability to change the GIDs and UIDs of files), `CAP_DAC_OVERRIDE` (bypasses the permissions of files [r/w/x]) and more. To see the list of capabilities and the kernel version which they are relevant for please checkout `"man capabilities"`.

```
root@localhost:~# cat /proc/$$/status | grep Cap
CapInh: 0000000000000000
CapPrm: 000001fffffffffff
CapEff: 000001fffffffffff
CapBnd: 000001fffffffffff
CapAmb: 0000000000000000
root@localhost:~# cat /proc/$$/task/1250/status | grep Cap
CapInh: 0000000000000000
CapPrm: 000001fffffffffff
CapEff: 000001fffffffffff
CapBnd: 000001fffffffffff
CapAmb: 0000000000000000
root@localhost:~# _
```

## chroot (Change Root Directory)

chroot is a Linux system call which allows changing the root directory of a calling process to a specific path. After doing so the directory will be used for the path names beginning with “/”. The changed root directory is inherited to all children of the calling process. By the way, only privileged processes can call “chroot” - root or with “CAP\_SYS\_CHROOT” in its user namespace<sup>138</sup>.

Moreover, there are different use case (which are not joust security related) for using chroot like: rebuilding initramfs image, reinstalling a bootloader, upgrading/downgrading a package and more<sup>139</sup>. By the way, we can use the “chroot” CLI tool (and not the system call) for preventing access outside the new root directory<sup>140</sup> - as shown in the screenshot below. It is recommended to go over the implementation of the “chroot” syscall<sup>141</sup>.

Lastly, we can think about “chroot” as a mitigation/hardening feature (and not a security feature) due to the fact there are specific ways to bypass it<sup>142</sup>. We can find it in use when creating sandboxed environments<sup>143</sup> - they are better solutions than just using “chroot” as described in future writeups (namespaces and seccomp as an example). An example for that is “wu-ftp” which can run in a chrooted environment for anonymous users<sup>144</sup>.

```
Troller $ ls /tmp/troller/
busybox sh
Troller $ sudo chroot /tmp/troller/ ./sh

BusyBox ( ) built-in shell (ash)
Enter 'help' for a list of built-in commands.

Troller $ ls
busybox sh
Troller $ pwd
/
```



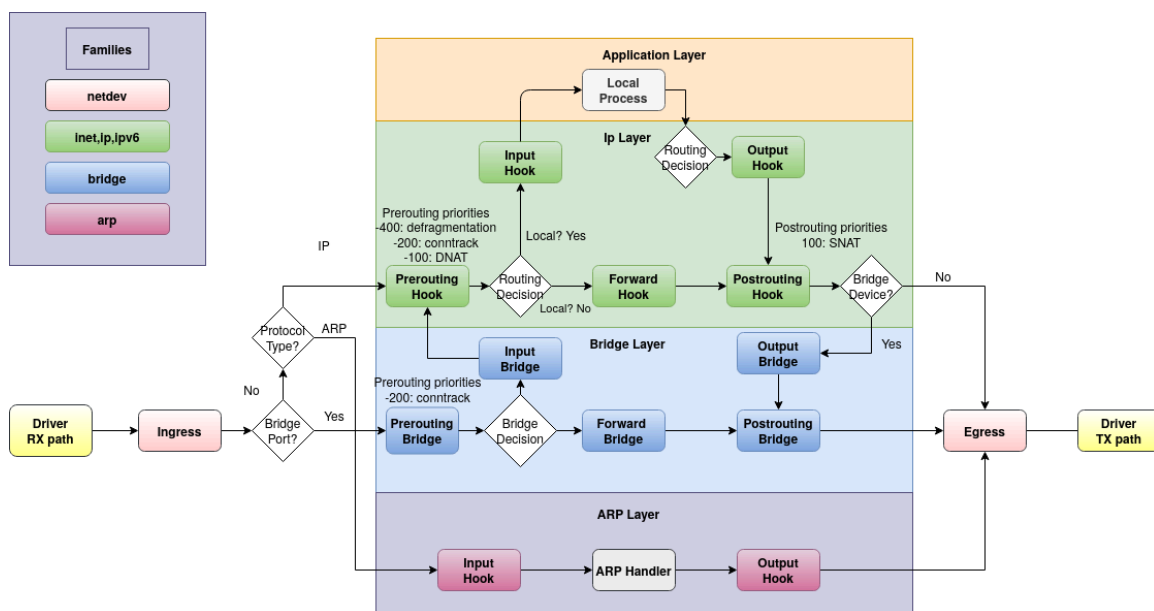
# NetFilter

NetFilter is a “Free and Open Source” (FOSS) project that provides packet filtering software for Linux (kernel 2.4 version and later). The main features provided by NetFilter are: stateless packet filtering (IPv4/IPv6), stateful packet filtering (IPv4/IPv6), different kinds of network/port address translations (NAT/PAT), packet logging, userspace packet queuing and other packet mangaling<sup>145</sup>. Thus, NetFilter is used for creating Firewalls (stateless/stateful), NAT based transparent proxies and other packet manipulation technologies.

One of the most important features of NetFilter is “Connection Tracking”. It allows the kernel to keep track of all the sessions/network connections in order to relate all the packets that make up a connection<sup>146</sup>. We can interface with the connection tracking feature using the “conntrack” CLI tool<sup>147</sup>.

Moreover, NetFilter provides “netfilter hooks” which enables using callbacks to provide filtering inside the Linux kernel. There are five different types of “netfilter hooks”: “Pre-Routing”, “Input”, “Forward”, “Output” and “Post-Routing” - as shown in the diagram below<sup>148</sup>.

Lastly, there are different tools that leverage NetFilter like: iptables, arptables, ebtables and nftables (more on them in future writeups). We can also go over the source code of “NetFilter” as part of the Linux kernel<sup>149</sup>.



# Chains (iptables)

In general “iptables” is an administration tool used IPv4/6 packet filtering and NAT<sup>150</sup>. “iptables” uses a series of rules that are organized into chains, in order to handle network traffic. Overall there are 5 built-in chains: PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING. Those chains are based on the NetFilter’s hooks callbacks<sup>151</sup>. We can also see that in the source code both for IPv4<sup>152</sup> and IPv6<sup>153</sup>.

Moreover, we can also create user defined chains using the following command “sudo iptables -N CHAIN\_NAME”. After we created the chain we can add new rules (more on rules in a future writeup) for it by specifying the chain name with the “-A” switch in “iptables” - as shown in the screenshot below. In order to move to another chain we need to use a “Jump Target” , which causes the evaluation to be done on a different chain for additional processing<sup>154</sup>. More on the different targets which are available in a future writeup. Lastly, we also have chains in other networking tools like “ebtables”

```
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target    prot opt source                destination

Chain FORWARD (policy ACCEPT)
target    prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target    prot opt source                destination
root@localhost:~# iptables -N TROLLER
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target    prot opt source                destination

Chain FORWARD (policy ACCEPT)
target    prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target    prot opt source                destination

Chain TROLLER (0 references)
target    prot opt source                destination
root@localhost:~# iptables -A TROLLER -p tcp --dport 2222 -j DROP
root@localhost:~# iptables -L
Chain INPUT (policy ACCEPT)
target    prot opt source                destination

Chain FORWARD (policy ACCEPT)
target    prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target    prot opt source                destination

Chain TROLLER (0 references)
target    prot opt source                destination
DROP     tcp  --  anywhere             anywhere             tcp dpt:EtherNet-IP-1
root@localhost:~#
```

## Table Types (iptables)

In general “iptables” is an administration tool used IPv4/6 packet filtering and NAT<sup>155</sup>. “iptables” is based on different types of tables: FILTER, RAW, NAT and MANGLE. By the way, there is also the SECURITY table used to set internal SELinux security context on packets. The goal of the tables is to hold rules based on the area of concern we want to evaluate packets<sup>156</sup>.

Overall, the rules on a specific table are organized into “chains”<sup>157</sup>. We can say that rules are clustered in “tables” based on their goal and “chains” represent the netfilter hooks which are going to trigger the rules. It is time to elaborate on each of the tables.

Filter table is used for controlling if a packet can get to its destination or not. It is the most used type for creating firewalls (which filters packets). A type of a NAT table is used for network address translation rules (think about deciding how to modify the destination/source IP address of a packet). We can use a table of type MANGLE for altering the IP header of a packet in the sense of changing the TTL/Type of service/internal mark for further processing<sup>158</sup>.

Lastly, the RAW type can be used for avoiding the use of the connection tracking capability of “iptables”. For better understanding we can see the relationship between “tables” and “chains” in the illustration below<sup>159</sup>.

Chains → Tables ↓	PREROUTING	INPUT	FORWARD	OUTPUT	POSTROUTING
filter		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
nat	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
mangle	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
raw	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
security		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

# nftables

nftables is the replacement of the legacy “\*tables” tools (iptables/ip6tables/arptables/ebtables). It is the modern Linux kernel packet classification framework. nftables has been available since version 3.13 of the Linux kernel which was released on 2014<sup>160</sup>.

Overall, the creation of nftables is due to different limitations both at the functional and code design level. The core design of nftables is based on a pseudo-virtual machine inspired by BPF - an example is shown in the screenshot below<sup>161</sup>. There is a backward compatibility regarding iptables, thus we can use the specific versions of iptables/iptables utilities that are able to convert iptables rules to nftables bytecode<sup>162</sup>.

Moreover, among the differences between iptables and nftables we can include that: nftables users a new syntax, a single nftables rule can take multiple actions, support for new protocols without the need for a kernel update, no built-in counter per chain/rule, Support for concatenations (since kernel 4.1) tables/chains a fully configurable, simplified dual stack for IPv4/6 administration and better support for dynamic rule set updates<sup>163</sup>.

Lastly, new Linux distributions use nftables as the recommended/default firewalling framework<sup>164</sup>. The user-mode command line tool for managing nftables is “nft”. We can summarize that nftables is a project of netfilter providing firewalling/NAT/packet mangling capabilities for Linux<sup>165</sup>.

```
nft --debug=netlink add rule inet t c ip daddr 10.1.2.3 counter accept
inet t c
[ meta load nfproto => reg 1 ]
[ cmp eq reg 1 0x00000002 ]
[ payload load 4b @ network header + 16 => reg 1 ]
[ cmp eq reg 1 0x0302010a ]
[ counter pkts 0 bytes 0 ]
[ immediate reg 0 accept ]
```

# Secure Execution Mode

In general, a binary is executed in “Secure Execution Mode” in case the “AT\_SECURE” entry of the auxiliary vector<sup>166</sup> contains a non-zero value. There are different cases that cause this value to be non-zero such as: a LSM<sup>167</sup> has set the value, the “real uid”<sup>168</sup> and the “effective uid”<sup>169</sup> of the task/process differ (and the same for the groups values), a non-root user executed a binary which conferred capabilities to the process<sup>170</sup>.

Overall, secure execution mode is a feature of the dynamic linker/loader. In case it is enabled specific environment variables are ignored when executing a binary. Examples of such variables are: “LD\_LIBRARY\_PATH”, “LD\_DEBUG” (unless /etc/suid-debug exists), “LD\_DEBUG\_OUTPUT”, “LD\_DEBUG\_WEAK” (since glibc 2.3.4), “LD\_ORIGIN\_PATH”, “LD\_PROFILE” (since glibc 2.2.5), “LD\_SHOW\_AUXV” (since glibc 2.3.4) and “LD\_AUDIT”<sup>171</sup> - as shown in the screenshot below.

Lastly, the goal of the secure execution mode is to block the ability of causing a binary which can be executed with “setuid”/setgid” to load/execute arbitrary code and thus perform a privilege escalation (due to the fact it is executed by one user but executed with the permissions of another user which case also be root).

```
[troller@localhost troller]$ ls -la --color
total 92
drwxr-xr-x 2 root root    0 2024 .
drwxrwxrwt 5 root root   51 18:35 ..
-rwxr-xr-x 1 root root 46556 02:52 normal_id
-rwsr-sr-x 1 root root 46556 2024 suid_id
[troller@localhost troller]$ export LD_DEBUG=statistics
[troller@localhost troller]$ ./normal_id

1353:
1353:  runtime linker statistics:
1353:    total startup time in dynamic loader: 870000 cycles
1353:    time needed for relocation: 183999 cycles (21.1%)
1353:    number of relocations: 165
1353:    number of relocations from cache: 9
1353:    number of relative relocations: 28
1353:    time needed to load objects: 368000 cycles (42.2%)
uid=1000(troller) gid=1000(troller) groups=1000(troller)
[troller@localhost troller]$ ./suid_id
uid=1000(troller) gid=1000(troller) euid=0(root) egid=0(root) groups=0(root),1000(troller)
[troller@localhost troller]$
```

## su (Substitute User)

“su” (Substitute User) is a utility as part of the Linux ecosystem which is used for running commands with the privileges of another user (by default the root user). This command allows us to switch to a specific account that we want in the current login session even if the user is not allowed to login using SSH/using GUI display manager<sup>172</sup>.

Overall, when switching to another user (su [USERNAME]) we need to know the password of that target user. However, if we have root privileges we can switch to whatever user we want without the need of knowing the target user’s password - as shown in the screenshot below. The substitution is done by setting the user id with the “setuid”\”setuid32” syscall<sup>173</sup>.

Lastly, we can also execute a specific command as a different user using the following pattern “su - [USERNAME] -c [COMMAND]”. For enhanced security we should restrict su access, add additional settings with PAM (Pluggable Authentication Modules) and configure logging and monitoring both locally and remotely<sup>174</sup>.

```
root@localhost:~# whoami
root
root@localhost:~# useradd troller
root@localhost:~# su troller
[troller@localhost root]$ whoami
troller
[troller@localhost root]$ exit
exit
root@localhost:~# whoami
root
root@localhost:~# su troller
[troller@localhost root]$ su root
Password:
```