

Projet de cryptographie

*Kireche Brice
Laboudi Hocine*

Sommaire

- 1) *Introduction*
- 2) *Téléchargement des noms de domaines*
- 3) *Mise en place du téléchargement des Certificats X509*
- 4) *Extraction des clé RSA & Vérification des doublons*
- 5) *Batch GCD sur les clé*
- 6) *Conclusion*

Tous les liens qui ont pu être utilisés seront présentés en bas de page autant que possible.
Tous les dossiers que nous avons pu récupérer et obtenir dans le cadre de ce projet sont accessibles directement sur le Drive de l'école.

[Projet Crypto](#)

Vous y verrez notre code, le batch GCD utilisé, les certificats mais également les rapports de doublons et notre documentation la plus utile.

I. Introduction

L'idée du projet est de retracer une grande quantité de certificats et trouver une possible faille. En effet, tous les certificats(plusieurs centaines de millions) proviennent d'une fonction. Cependant, il arrive que cette fonction utilise un aérateur « Random » peu efficace de ce fait la fonction sera totalement aléatoire. Cela aura pour effet de trouver un multiple commun lorsque l'on retracera un certain nombre de possibilités.

Une fois ce multiple trouvé, il sera nécessaire de sauvegarder son nom de domaine auquel est rattaché sa clé puis son certificat comme nous le verrons dans les 2 prochains points de ce rapport.

C'est donc la raison pour laquelle, nous nous sommes dans un premier temps attelé à la recherche des noms de domaine directement lié aux certificats et ceux dont la quantité est la plus importante.

II. Téléchargement des noms de domaines

Après de sérieuses recherches nous sommes tombés sur l'outil Certstream open source, libre et facile d'accès.

Une fois les extensions installées nous avons pu télécharger de deux manière différentes, sur un format similaire à celui-ci, à l'aide de la commande Certstream dans le terminal.

[10/03/22 14:24:59] accountants-wales.co.uk (SAN:)

[10/03/22 14:24:59] *.prbuild.surakota.people.aws.dev (SAN:)

Mais il n'était pas très judicieux de le mettre dans ce sens. Toutefois nous nous sommes rendus compte que les dates n'étaient pas utiles et que nous aurions préférés avoir seulement les noms de domaines pour pouvoir les placer correctement et sans risque dans mon code.

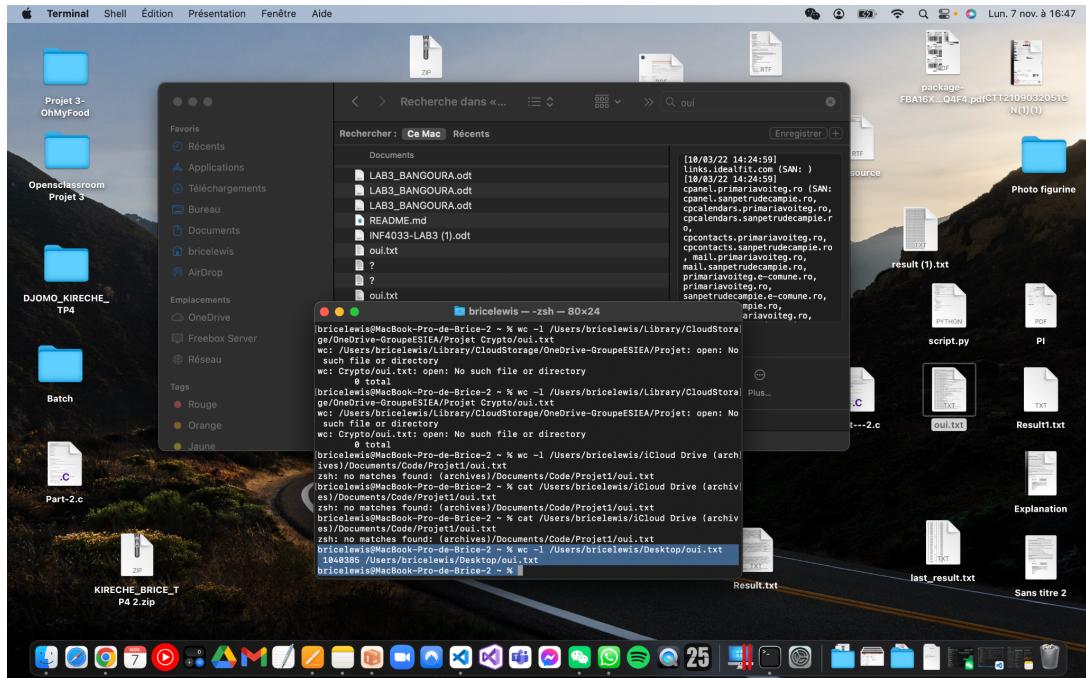
La très pratique commande « certstream --json | jq -r '.data.leaf_cert.all_domains[]' » directement écrite dans le terminal nous a permis de récupérer directement les noms de domaines au format Json. Cela nous a donné accès aux noms de domaines suivants (disposés dans un fichier texte) :

healthyfuel24.com

erinflorenceart.com

www.erinflorenceart.com

Ci dessous, un screenshots du nombre de ligne attestant donc du nombre de noms de domaines :



Pour expliquer un peu notre démarche, nous savions que le service de Certstream proposait de télécharger gratuitement 1 million de certificats en 6 heures approximativement (pour notre part). Cependant, plus le nombre de certificats est important, plus la chance de trouver une faille est grande. La base de données de Certstream contient 250 millions de certificats.

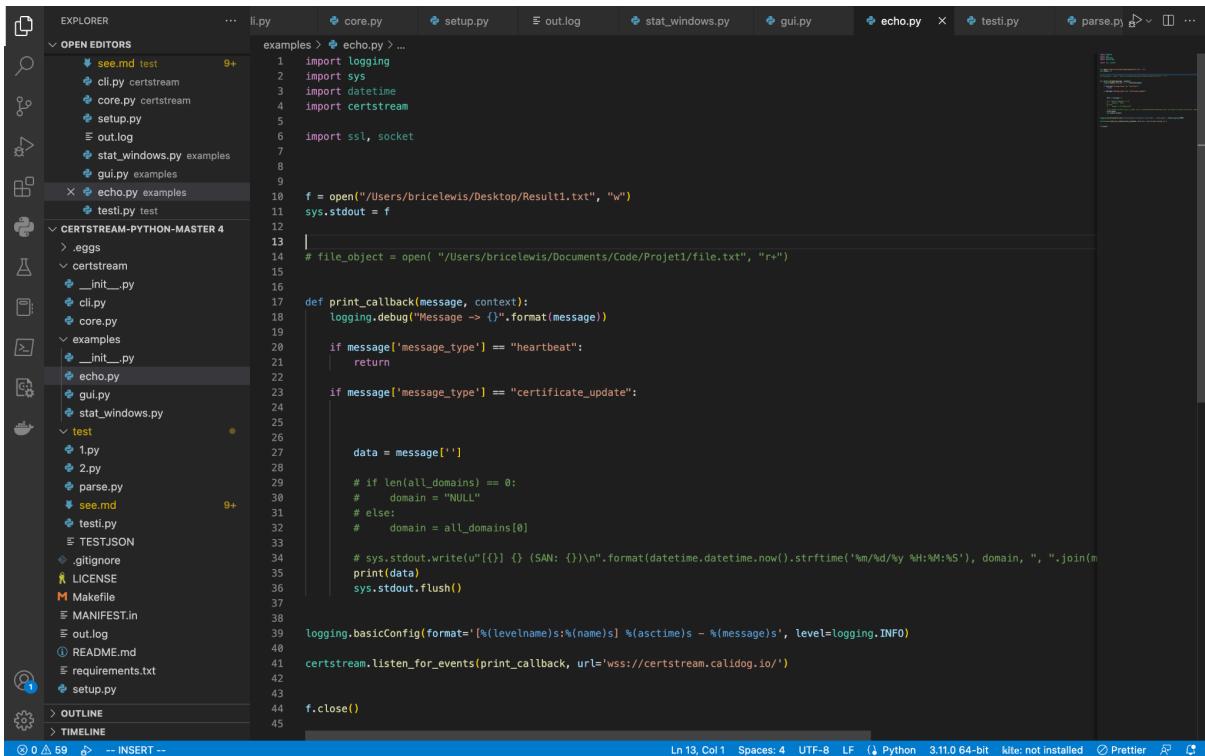
Nous avons donc, après coûт, trouver un site proposant directement les noms de domaines. Plus de 10 millions noms de domaines en quelques secondes (lien en bas de page) pouvant même aller jusqu'à 30 millions (non testé)

Mais après réflexion, sur un grand nombre de données, prendre le risque d'avoir des données inexactes peut être compliqué. En effet, avec Certstream, nous sommes sur d'avoir des données à jour et exact, provenant directement des serveurs de Google.

III. Mise en place du téléchargement des Certificats X509

Afin de faire fonctionner les codes suivants, nous avons installé les extensions comme le module « Cryptographie » ainsi que « Request » qui nous ont été très utiles, et bien d'autres encore.

A l'aide de Certstream nous sommes parvenus à obtenir les certificats au format Json.



```
examples > echo.py > ...
1 import logging
2 import sys
3 import datetime
4 import certstream
5
6 import ssl, socket
7
8
9
10 f = open("//Users/bricelewis/Desktop/Result1.txt", "w")
11 sys.stdout = f
12
13
14 # file_object = open( "/Users/bricelewis/Documents/Code/Projet1/file.txt", "r+" )
15
16
17 def print_callback(message, context):
18     logging.debug("Message -> {}".format(message))
19
20     if message['message_type'] == "heartbeat":
21         return
22
23     if message['message_type'] == "certificate_update":
24
25         data = message['']
26
27         # if len(all_domains) == 0:
28         #     domain = "NULL"
29         # else:
30         #     domain = all_domains[0]
31
32         # sys.stdout.write(u"{} {} (SAN: {})\n".format(datetime.datetime.now().strftime('%m/%d/%Y %H:%M:%S'), domain, ".join(
33         #     print(data)
34         #     sys.stdout.flush()
35
36
37
38 logging.basicConfig(format='%(levelname)s:(%name)s) %(asctime)s - %(message)s', level=logging.INFO)
39
40 certstream.listen_for_events(print_callback, url='wss://certstream.calidog.io/')
41
42
43
44
45 f.close()
```

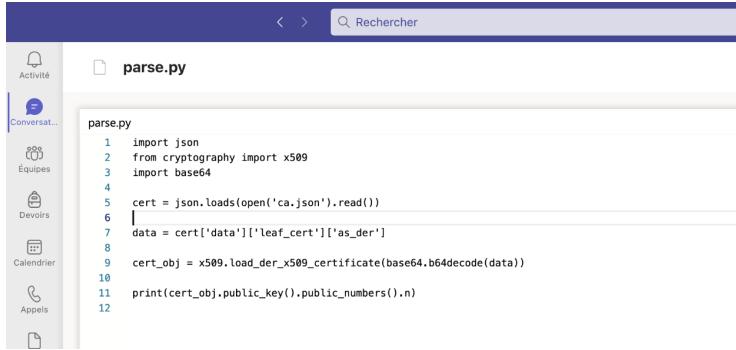
Données sortant de Certstream

```
{
  "message_type": "certificate_update",
  "data": {
    "update_type": "X509LogEntry",
    "leaf_cert": {
      "subject": {
        "aggregated": "/CN=app.theaffairsite.com",
        "C": null,
        "ST": null,
        "L": null,
        "O": null,
        "OU": null,
        "CN": "app.theaffairsite.com"
      },
      "extensions": {
        "keyUsage": "Digital Signature, Key Encipherment",
        "extendedKeyUsage": "TLS Web Server Authentication, TLS Web Client Authentication",
        "basicConstraints": "CA:FALSE",
        "subjectKeyIdentifier": "01:BE:17:27:B8:D8:26:EF:E1:5C:7A:F6:14:A7:EA:B5:D0:D8:B5",
        "authorityKeyIdentifier": "keyid:A8:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF",
        "authorityInfoAccess": "OCSP - URI:http://ocsp.int-x3.letsencrypt.org\nCA Issuers",
        "subjectAltName": "DNS:app.theaffairsite.com",
        "certificatePolicies": "Policy: 2.23.140.1.2.1\\nPolicy: 1.3.6.1.4.1.44947.1.1.1\\n",
        "not_before": 1509908649.0,
        "not_after": 1517684649.0,
        "serial_number": "33980d1bef9b6a76cf708e3139f55f33c5",
        "fingerprint": "95:CA:86:6B:B4:98:59:D2:EC:C7:CA:E8:42:70:80:0B:18:03:C7:75",
        "as_der": "MIIFDTCCA/WgAwIBAgISAzmA0b75tqds/HCOMTn1XzPFMA0GCSqGSIb3DQEBCwJAMEoxCzAJBgI",
        "all_domains": [
          "app.theaffairsite.com"
        ]
      },
      "cert_index": 27910635,
      "seen": 1509912803.959279,
      "source": {
        "name": "Certstream"
      }
    }
  }
}
```

Le certificat surligné nous avait donné du mal à récupérer et nous avions sollicité votre aide M.Larinier afin de la récupérer. Vous nous aviez aidé sur un bout de code et nous avons finalement eu accès aux informations les plus importantes :

- Le nom de domaine
- Le certificat X509 (As_Der)
- Et la clé RSA (authorityKeyIdentifier normalement)

En s'appuyant sur un code exemple directement sur le Github Certstream, nous avons construit l'appel de la fonction. (Plus ou moins comme pour les noms de domaines) et avons ensuite appliqué votre code pour récupérer les informations.



```
parse.py
1 import json
2 from cryptography import x509
3 import base64
4
5 cert = json.loads(open('ca.json').read())
6
7 data = cert['data']['leaf_cert']['as_der']
8
9 cert_obj = x509.load_der_x509_certificate(base64.b64decode(data))
10
11 print(cert_obj.public_key().public_numbers().n)
12
```

Cependant, nous trouvions cette méthode fort efficace mais un peu trop consommatrice sur le plan hardware. Nous avions en parallèle sous les yeux depuis le début le code de Certificat Transparency, qui peut certainement être une très bonne alternative.

Cependant, nous retiendrons une solution efficace, nommée "Core" disponible sur le sharepoint.

Remarque : Tous ces projets ou du tout moins celui ci est capable de se connecter au logs listes de google car il fait appel ou aux librairies Certstream ou Certificates transparency (qui appartient à google). Et ici, le code allait vite, voir très très vite.

```

OPEN EDITORS
Get Started
setup.py
requirements.txt
README.md
MANIFEST.in
LICENSE
certlib.pyc axeman
demo.gif img
logo.png img
core.py

AXEMAN-MASTER
> axeman
> _pycache__
> __init__.py
certlib.py
core.py
> img
.gitignore
cert.md
LICENSE
MANIFEST.in
README.md
requirements.txt
setup.py

OUTLINE
TIMELINE
briclewis@MacBook-Pro-de-Brice-2 Axeman-master % 
Ln 127, Col 1  Spaces: 4  UTF-8  LF  Python 3.11.0 64-bit  kite: not installed  Prettier

```

Nous avons donc opté pour cette solution, nous nous sommes posé la question au début du nom de domaine. En effet, via certificate transparency, nous ne sommes pas arrivé à trouver le moyen en python de générer aussi les noms de domaines. Quel est alors l'intérêt de dénicher une faille dans le certificat si le site est inconnu. Nous avons finalement trouvé un site web (également en description) qui permet de retrouver le site en question à partir du nom de domaine.

Cet ouvrage est un super outil. Très varié et divers.

Nous avons donc obtenu les dossiers directement sur le bureau plutôt que dans le dossier temporaire. 1,3 millions de certificats ont été téléchargés en 30 minutes le tout répertorié dans 40,000 fichiers excel, fusionnés en un seul. Il se présente avec en fichier « csv » avec la clé RSA puis le certificat et le nom de domaine après modification de bon nombre de lignes de code.

Nous voilà en possession de plus d'un million de clés, toutes fournies par Google.

IV. Extraction des clé RSA & Vérification des doublons

Une fois les clés RSA extraites du fichier csv, nous les avons toutes analysées.

Nous avons pris la liberté de faire un code pour les comparer mais forcés de voir qu'EXCEL est un bien meilleur outil, nous avons vérifié les doublons avec ce dernier. Comme nous nous attendions, le dossier était beaucoup trop lourd pour excel à ouvrir en 1 seule fois, nous avons donc découpé le fichier en 4. Ce qui nous donne plus ou moins 400,000 échantillons de clés par fichier.

Nous avons alors vérifié les certificats x509, étaient tous différents sans exception, et nous avons eu la confirmation logiciel (excel) qu'aucune similarité ne fut remarquée sur les certificats. Cependant, de nombreux doublons ont apparus dans le code.

nous avons d'ailleurs demandé à M. Erra la provenance de ces derniers.

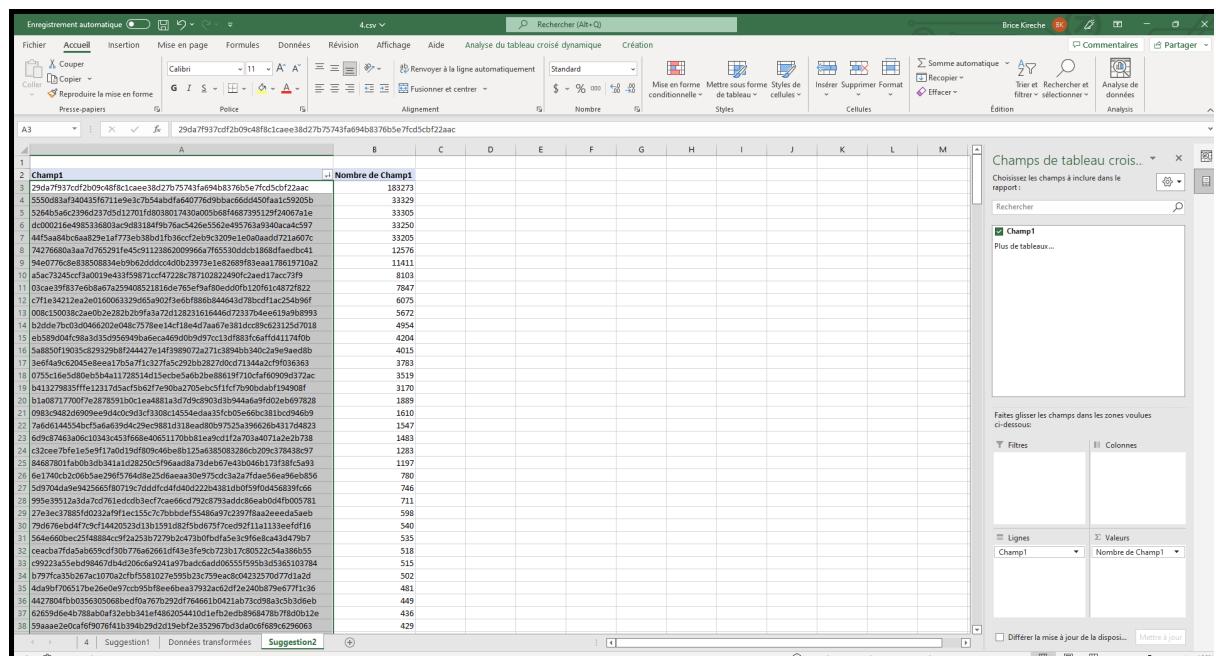
Environ 10% de ses clé sont des doublons.

Sur les memes doublons, les certificats étais différent.

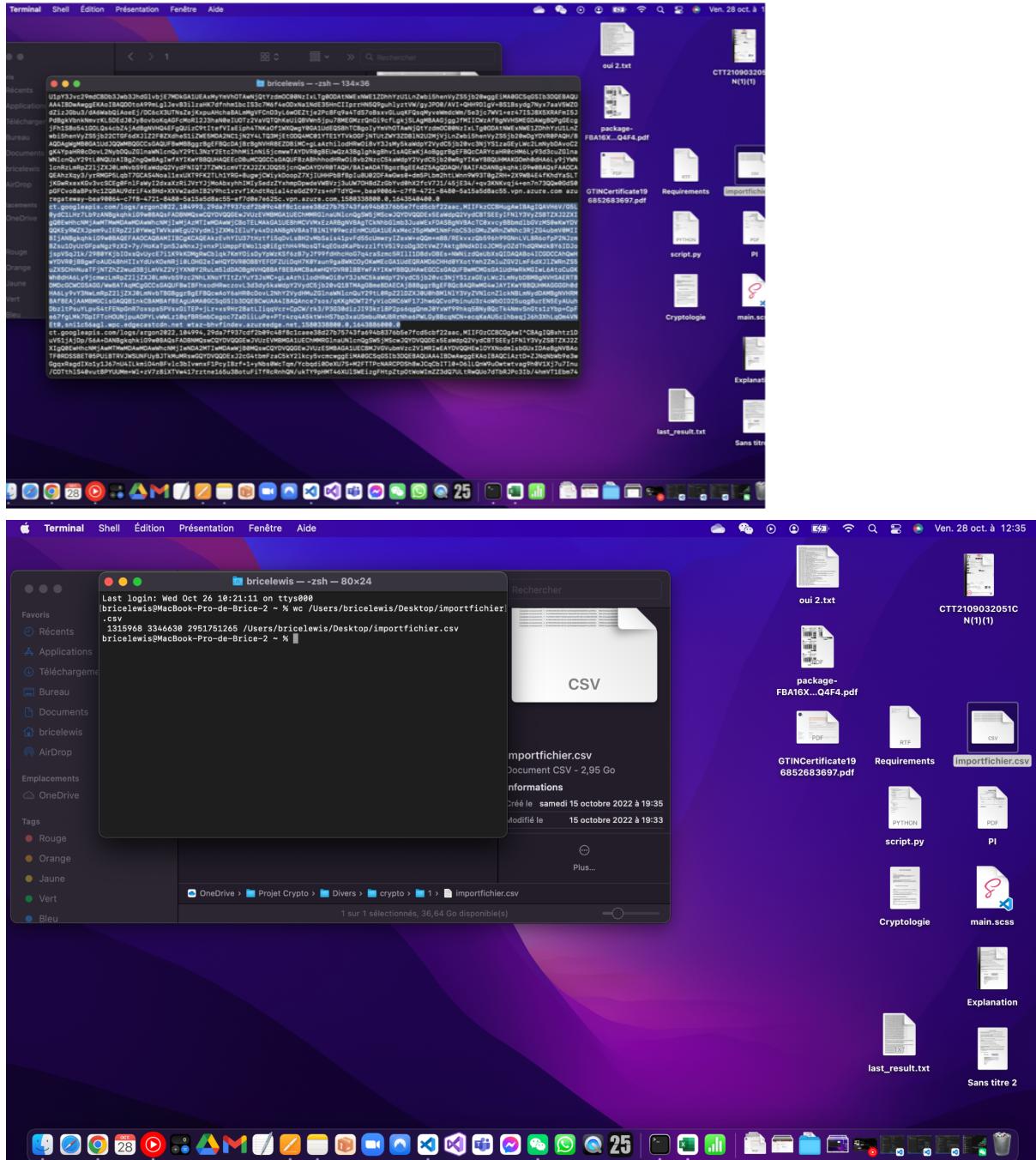
Selon nous laisse avec 1.3 Millions de certificats et 1.1/1.2 millions de clé.

Nous sommes arrivés à la conclusion que c'est donc bien la même clé RSA pour bon nombre de ces sites. (Sauf erreur de notre part, le certificat étant différent)

De ce fait, une faille importante serait accessible si nous arrivions à trouver la clé associée (un N ou un Q)



Il est vrai que nous avons plus d'un million de certificats et allons passer le Batch GCD sur tous les certificats différents de notre liste.



Voici le total de lignes

Le but étant de repérer les similarités de ces clés en les divisant par un échantillon prédéterminé et aléatoire.

En effet, le logiciel qui crée ces clés possède une fonction Random peu diverse ou qui peut se répéter dès lors que l'on compare un très grand échantillon de produit donc plus l'échantillon est grand, mieux c'est.

Nous avons donc compté soigneusement le nombre de clés différentes de notre liste, et il en ressort 120,000. Nous pouvons donc passer au Batch GCD au plus vite.

Pour ce projet, nous nous sommes grandement appuyés sur divers documents présents sur internet, ainsi que sur des projets GitHub, dont un Batch GCD codé en C++ d'une qualité, pour moi quasiment parfaite.

Plus de 10 batch ont été évalué et finalement le batch en Python (De fiona) excellent également à été retenu.

Ce sont les sources les plus notables, en description.

V. Batch GCD sur les clés

Après de nombreux projets et idées analysé, nous avons finalement jeté notre dévolu sur le Batch GCD de « Fiona » en Python.

En général, les clés RSA sont très difficiles à factoriser : le plus grand module RSA factorisé par les algorithmes standards compte 768 bits (232 chiffres décimaux __ Source wikipédia), et les normes actuelles pour la plupart des services de chiffrement de haut niveau utilisent des modules RSA de 1024 ou 2048 bits. **Cependant, si une grande collection de clés RSA est construite à l'aide de nombres premiers donnés par un générateur de nombres aléatoires défectueux, un nombre significatif de ces clés peut partager des facteurs premiers.**

L'algorithme batch GCD exploite cette faiblesse dans la génération de clés RSA.

Supposons que deux clés RSA distinctes N1 et N2 partagent un facteur, c'est-à-dire que $d = \text{pgcd}(N_1, N_2) > 1$; alors, puisque $N_i = p_i * q_i$ pour primepiandqi, il faut que soit $d = N_i$ ou $d = q_1$.

Ainsi, si nous constatons que deux clés partagent des facteurs, nous pouvons rapidement factoriser les deux clés en utilisant leur PGCD. Le processus GCD par lots

Étant donné une liste $\{N_1, N_2, \dots, N_m\}$ d'entiers que nous avons soigneusement fabriqué (cf code)

Avant de passer la liste, nous avons donc du créer une liste d'entier, comprise entre 0 et 100 ici, modifiables en fonction du besoin. Cette dernière comprend environ une quinzaine d'éléments. **La seul condition à cette liste est que ces nombres soient premiers bien évidemment**

Remarque:

Au delà d'un certain seuil, la probabilité de trouver un PGCD est plus faible selon [wikipedia](#).

[En effet, la proportion des nombres premiers diminue](#) avec au delà de 1,000,000

Cependant, prendre les 1000 premiers nombres semble correct pour une approche.

Remarque:

Lors d'une attaque, plus ce nombre est élevé, plus la manipulation sera efficace.

Remarque:

On aurait plus facilement calculer le coût d'un programme [avec la commande User Time par exemple](#)

Remarque :

En cas de nombre très important et de recherche poussé, adapté son code en conséquence aurait probablement été une bonne idée. Changer de language également.

Le python est exactement 86 fois plus lent que le C.

Le javascript est exactement 26 fois plus lent que le C.

La fonction retourne un G ou PGCD

Remarquez que $\{Ni|Gi>1\}$ est l'ensemble de clés qui partagent un facteur avec une autre clé de la liste. La plupart du temps, Gi est premier et donc un facteur non trivial de Ni . En reconnaissant ces premiers Gi , on peut factoriser les clés Ni correspondantes.

Nous voilà au cœur du projet, et nous allons vous donner notre code pour la Batch GCD qui nous a donné beaucoup de mal !

Ce code vient totalement de nous, et a pour finalité de créer un dossier, que nous pourrons analyser (à également retrouver en annexe)

```
from math import prod, floor, gcd
from collections.abc import Sequence

def products(*integers: int) -> list[list[int]]:
    """Tree with the root as the product, input as leaves and intermediate
    states as intermediate nodes"""
    xs = list(integers)
    result = [xs]
    while len(xs) > 1:
        xs = [prod(xs[i * 2: (i + 1) * 2]) for i in range((len(xs) + 1) // 2)]
    //la division en elle même
    result.append(xs)
    return result

def batch_gcd(*integers: int) -> list[int]:
    xs = list(integers)
    tree = products(*xs)
    node = tree.pop()
    while tree:
        xs = tree.pop()
        node = [node[floor(i / 2)] % xs[i] ** 2 for i in range(len(xs))]
    //la division en elle même

    res = []
    for r, n in zip(node, xs):
        res.append("GCD= " + str(gcd(r // n, n)) + ", r = " + str(r) + ", n = " + str(n))
    //le résultat

    return res

counter = 0
```

result = []

```
with open("RSAkey.txt") as f:
```

for line in f:

```
key = int(line, base=16)
```

```
result.append(key)
```

```
counter += 1
```

counter

```
result = batch_gcd(*result)
```

```
#after batch = batch_gcd(*result) # send the result for batch gcd
```

```
with open("result.txt", "w") as f:
```

for item in result:

```
f.write("%s\n" % item)
```

```
# print(after_batch)
```

— 1 —

Et voici son résultat :

Nous avons finalement réussi à créer un fichier txt en écrivant les données de la division entre notre liste et chaque clé. nous avons ensuite comparé les résultats

Nous avons finalement trouvé deux diviseurs communs et donc la faille associée
Pour la repérer nous avons évidemment utilisé excel.

Pour générer un module RSAN, nous générerons une paire d'étendues premières aléatoires et q satisfaisant quelques conditions techniques (comme $\text{pgcd}(p-1, q-1)=2$), puis nous les multiplions ensemble pour obtenir N.

Ici nous voyons que le même Q a été retrouvé

Tous les N ont été générés en utilisant la méthode ci-dessous

A	B	C	D	E	F	G	H	I	J	K	L	M
1	GCD	1,89E+76 r	o n		1,89E+76							
2	GCD	6,73E+76 r	o n		6,73E+76							
3	GCD	8,25E+76 r	o n		8,25E+76							
4	GCD	4,09E+76 r	o n		4,09E+76							
5	GCD	7,72E+75 r	o n		1,72E+75							
6	GCD	9,04E+76 r	o n		9,04E+76							
7	GCD	2,48E+74 r	o n		2,48E+74							
8	GCD	8,09E+76 r	o n		8,09E+76							
9	GCD	8,15E+76 r	o n		8,15E+76							
10	GCD	1,75E+76 r	o n		1,75E+76							
11	GCD	4,30E+75 r	o n		4,30E+75							
12	GCD	4,23E+76 r	o n		4,23E+76							
13	GCD	5,54E+76 r	o n		5,54E+76							
14	GCD	8,83E+76 r	o n		8,83E+76							
15	GCD	7,72E+76 r	o n		7,72E+76							
16	GCD	8,03E+76 r	o n		8,03E+76							
17	GCD	4,98E+76 r	o n		4,98E+76							
18	GCD	3,57E+76 r	o n		3,57E+76							
19	GCD	1,75E+76 r	o n		1,75E+76							
20	GCD	2,67E+76 r	o n		2,67E+76							
21	GCD	4,45E+76 r	o n		4,45E+76							
22	GCD	1,80E+76 r	o n		1,80E+76							
23	GCD	5,40E+76 r	o n		5,40E+76							
24	GCD	1,08E+77 r	o n		1,08E+77							
25	GCD	1,09E+77 r	o n		1,09E+77							
26	GCD	2,99E+76 r	o n		2,99E+76							
27	GCD	66 r	2,13E+152 n			8,85E+76						
28	GCD	5,23E+76 r	o n			5,23E+76						
29	GCD	3,32E+76 r	o n			3,32E+76						
30	GCD	7,35E+75 r	o n			7,35E+75						
31	GCD	1,63E+152 r	1,63E+152 n			7,09E+76						
32	GCD	1,11E+76 r	o n			1,11E+76						
33	GCD	5,75E+76 r	o n			5,75E+76						
34	GCD	7,81E+76 r	o n			7,81E+76						
35	GCD	1,18E+76 r	o n			1,18E+76						
36	GCD	7,49E+76 r	o n			7,49E+76						
37	GCD	1,08E+77 r	o n			8,00E+76						
38	GCD	5,25E+76 r	o n			1,08E+77						
39	GCD											

Nous avons dans un premier temps comparé le PGCD et regarder la similarité.

Remarque: Certains Batch avaient une fonction pour retrouver un n et q commun

Également, ici, nous ne connaissons que le modulus des nombres trouvés et non les nombres en eux-même.

Cependant, sur une multiplication simple de deux nombres, retrouver le nombre initial à partir du modulo est possible assez facilement

Alors, ce ce fait, on à dans chaque clé l'équation la plus importante

Remarque : Nous n'avons pas eu l'occasion de savoir lequel P ou Q nous aurions pu trouver.

Selon nous, nous aurions pu très bien trouver un P ou Q, deux P ou deux Q.

$$N = P * Q$$

Ici, nous avons trouvé le même PGCD pour 2 éléments (Fichier Last Résult)

Le PGCD en question :

25680532640

Dès lors, nous l'avons divisé par un grand N est trouvé les équations ci dessous.

```
n =
11929413484016950905552721133125564964460656966152763801206748195494305685115033
38063159570377156202973050001186287708466899691128922122454571180605749959895170
80042105263427376322274266393116193517839570773505632231596681121927337473973220
312512599061231322250945506260066557538238517575390621262940383913963

p =
10933766183632575817611517034730668287155799984632223454138745671121273456287670
008290843302875521274970245314593222946129064538358581018615539828479146469

q =
10910616967349110231723734078614922645337060882141748968209834225138976011179993
394299810159736904468554021708289824396553412180514827996444845438176099727
```

et

```
p =
57581761151703473066828715575758176115170347306682871557066828715579998463222345413874567112127345628
7670
008290843302875521274970245314593222946129064538358581018615539828479146469

q =
10910616967349110231723734078614922645337060882141748968209834225138976011179993
394299810159736904468554021708289824396553412180514827996444845438176099727

n =
21927337473973220950905552721133125564964460656966152763801206748195494305685115033
38063159570377156202973050001186287708466899691128922122454571180605749959895170
800421052634273763222742663931161935178395707735056322315966811219231251259906123
1322250945506260066557538238517575390621262940383913963
```

L'ensemble est un ensemble de modules RSA.

Dans le monde réel, il y a de fortes chances que certaines personnes aient utilisé des sources aléatoires faibles, mal configurées ou fatalement compromises.

En particulier, certains nombres premiers peuvent apparaître plus d'une fois !

C'est une très mauvaise nouvelle pour la sécurité Internet, mais une bonne nouvelle pour nous, car nous pouvons calculer efficacement ces nombres premiers partagés et casser les clés correspondantes. La raison pour laquelle nous pouvons faire cela est que s'il est difficile de factoriser un nombre, il est facile de détecter des facteurs communs entre n'importe quelle paire de nombres.

VI. Conclusion

Grâce à toutes ces manipulations nous pouvons affirmer qu'une attaque est possible. Cependant, même si possible, la cible serait aléatoire, parmi les noms de domaines existants. Trouver un n ou q d'une de ses clés serait très utile dans le cas d'une clé multiple..

Faire cela à plus grande échelle serait également une solution. 250 millions de certificats sont présents sur la database dont Certstream à accès. Cependant, cette attaque ne sera résolu que lorsque les fonctions random seront efficaces, produisant une entropie maximum. On pourrait également envisager, si jamais la sécurité venait à manquer, le scindement d'un n ou q de manière à avoir 3 inconnues et multiplier ainsi la difficulté de calcul exponentiellement. $N = p * Q^Z$ (Z étant le scindement).

Sur un plan moins technique, je pense que nous avons compris que l'apprentissage en action était plus efficace qu'en suivant seulement un TD. Le cours suivi a toutefois été essentiel dans la compréhension des bases qui nous ont permis de mener à bien ce projet. Nous en retirons des compétences techniques mais aussi une capacité accrue à travailler en équipe et cela nous servira évidemment dans le monde professionnel. Pour finir nous tenions à remercier M.Larinier pour ses cours et sa présence en TP.

Annexe:

Sources:

Projet Crypto

<https://github.com/google/certificate-transparency-go>

<https://certstream.calidog.io/>

<https://github.com/CaliDog/Axeman>

https://www.di-mgt.com.au/rsa_alg.html

Batch GCD :

1. En c++ <https://github.com/hfiuza/Euclid-vs-RSA-Cryptanalysis-with-batch-GCDs>
2. <https://github.com/fionn/batch-gcd>
3. <https://github.com/ForeverAnApple/Tetanus>
4. <https://github.com/dieggoluis/rsa-attack>
5. <https://github.com/therealmik/batchgcd>

```
from math import prod, floor, gcd
from collections.abc import Sequence

def products(*integers: int) -> list[list[int]]:
    """Tree with the root as the product, input as leaves and intermediate
    states as intermediate nodes"""
    xs = list(integers)
    result = [xs]
    while len(xs) > 1:
        xs = [prod(xs[i * 2: (i + 1) * 2]) for i in range((len(xs) + 1) // 2)]
        result.append(xs)
    return result

def batch_gcd(*integers: int) -> list[int]:
    xs = list(integers)
    tree = products(*xs)
    node = tree.pop()
    while tree:
        xs = tree.pop()
        node = [node[floor(i / 2)] % xs[i] ** 2 for i in range(len(xs))]
    res = []
    for r, n in zip(node, xs):
        res.append("GCD=" + str(gcd(r // n, n)) + ", r = " + str(r) + ", n = " + str(n))
    return res

counter = 0
result = []

with open("RSAkey.txt") as f:
    for line in f:
        key = int(line, base=16)
        result.append(key)
        counter += 1
        if counter == 100:
            break

result = batch_gcd(*result)

#after_batch = batch_gcd(*result) # send the result for batch gcd

with open("result.txt", "w") as f:
    for item in result:
        f.write("%s\n" % item)

# print(after_batch)

for item in result:
    print(item)
```

Ln: 53 Col: 0