Direction Générale de la Sécurité Extérieure
École Supérieure d'Ingénieurs en Électronique et Électrotechnique

Le 15 décembre 2020

Challenge Brigitte Friang : write-up

Les Consultants

ANTON RAVEENDRAN Joyston
joyston.antonraveendran@edu.esiee.fr

PRIOU Roger
roger.priou@edu.esiee.fr

Sommaire

1 Remerciements **3**

2 Recherche de vulnérabilités et exploitations **4**
2.1 Préambule 4
2.2 Code César 4
2.3 Service de messagerie 5
2.3.1 Code Morse 5
2.3.2 Code Binaire 6
2.3.3 Déchiffrement de l'archive par OpenSSL 7
2.3.4 Résolutions Mathématiques 8
2.3.5 Inscription sur la plateforme de Capture The Flag 9

3 Capture The Flag **10**
3.1 Web 10
3.1.1 ChatBot 10
3.2 Hardware 12
3.2.1 ASCII UART 12
3.2.2 Keypad Sniffer 15
3.3 Forensic 17
3.3.1 Sous l'océan 17
3.4 Stéganographie 19
3.4.1 Polyglotte 19
3.5 MISC 24
3.5.1 Définition 24

4 Bibliographie **25**

5 Annexes **26**
5.1 Script Sous l'océan 26
5.2 Script Keypad Sniffer 27

6 Index **28**
6.1 Liste des figures : 28
6.2 Liste des tableaux : 28
6.3 Liste des flags : 28
6.4 Liste des scripts : 28

Chapitre 1

Remerciements

Nous tenons à remercier vivement la DGSE et ESIEE Paris pour avoir organisé ce challenge de cybersécurité ! Ce challenge nous a été d'une aide précieuse pour pouvoir monter en compétences dans le domaine de la cybersécurité.

Enfin, nous tenons également à remercier toutes les personnes qui ont contribué d'une façon directe ou indirecte à l'organisation de ce challenge.

Chapitre 2

Recherche de vulnérabilités et exploitations

2.1 Préambule

Cette année la DGSE (Direction générale de la sécurité extérieure) s'est alliée avec une école d'ingénieurs ESIEE Paris afin de réaliser un challenge de cybersécurité. Ce challenge comporte des exercices à résoudre dans différents domaines (Web, Hardware, Forensic, Cryptographie, Stéganographie, Pwn et MISC) pour tout type de niveau du débutant à l'expert ! Attention, ces challenges doivent être résolus sans avoir recours à la technique de la bruteforce, sauf contre-indication.

Pour réaliser ce challenge, nous avons eu 19 jours entre le Samedi 24 Octobre 2020 et le Mercredi 11 Novembre 2020. Par conséquent, les organisateurs nous ont mis à disposition un site web [1] comportant uniquement une page d'accueil, nous souhaitant la bienvenue !

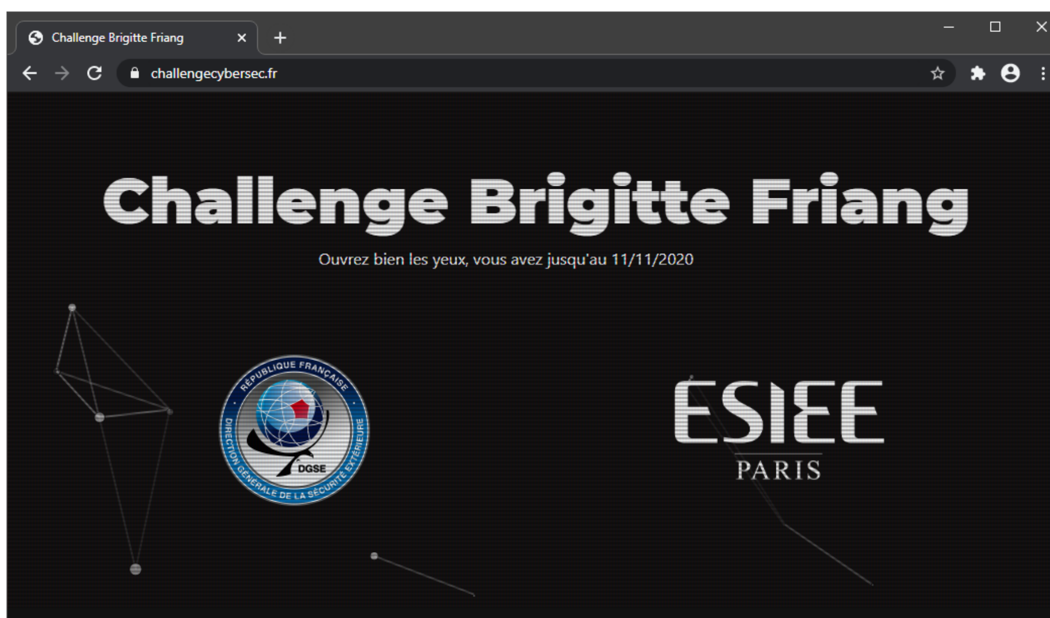


FIGURE 2.1 – Site internet du challenge Brigitte Friang

2.2 Code César

Tout d'abord, nous avons exploré le code source du site internet et nous avons remarqué qu'il existait un message caché qui a été mis en commentaire : `/static/message-secret.html` Il est important de rappeler qu'il ne faut en aucun cas laisser des informations prépondérantes en commentaire.

Par conséquent, nous avons décidé de nous rendre sur cette page afin de récupérer le message caché. Ce message a été codé en César et nous devons le déchiffrer. Pour cela, nous avons utilisé les outils en ligne [2] à notre disposition. Il s'agissait d'un décalage de 7. Voici l'alphabet utilisé dans le cadre du chiffrement du message :

```
HIJKLMNOPQRSTUVWXYZABCDEFGABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Suite au déchiffrement du message, nous avons compris qu'il y avait un second message dans le message déchiffré. Il nous était indiqué qu'il faudra utilisé ce second message dans l'adresse URL. Par conséquent, grâce à la relecture de façon minutieuse du message déchiffré, nous avons pu remarquer la présence de caractères en gras. Nous les avons tous récupérés, et lorsque nous les mettons l'un après l'autre, cela nous donne `/chat`. Ainsi, il s'agit du mot à compléter dans l'adresse URL [3].

1. Lien permanent : https://www.challengecybersec.fr
2. Lien permanent : https://www.dcode.fr/chiffre-cesar
3. Lien permanent : https://www.challengecybersec.fr/chat

2.3 Service de messagerie

Le site précédemment énuméré nous permet de se connecter sur un service de messagerie factice, où il y a cinq grands scientifiques avec lesquels nous pouvons discuter. Le directeur Armand Richelieu nous accueille en nous indiquant qu'il s'agit d'une mission classée secret-défense, et qu'il faut résoudre des énigmes le plus vite possible afin de limiter les actions d'Evil Gouv contre sa population. Pour cela, nous devons remplir notre mission en résolvant l'une des quatre énigmes mise à notre disposition (Cryptographie, Web, Algorithme et Forensic). Cette mission aura pour nom de code Brigitte Friang, et nous incarnerons l'Agent 042

Nous avons décidé de résoudre l'énigme proposée par Antoine Rossignol du service cryptographie. Pour cela, Antoine Rossignol nous a fourni les documents suivants :

1. Les échanges avec notre prédécesseur
2. Un compte-rendu
3. Un archive chiffrée
4. Une photographie au microscope électronique à balayage des fusibles

Les échanges entre Antoine Rossignol et notre prédécesseur (Agent 040) nous permettent de comprendre le problème initial. Il nous indique qu'un des agents a intercepté du matériel de chiffrement et un message chiffré qui doit contenir des informations prépondérantes sur la livraison de produits chimiques. Nous pouvons penser que ces dernières seront utilisées par Evil Gouv contre sa population. Il devient important d'explorer le matériel de chiffrement afin d'avoir des informations sur les différentes communications d'Evil Gouv et de limiter les actions de ce dernier contre sa population. Malheureusement, notre prédécesseur n'a pas réussi à l'exploiter car le processeur n'échange uniquement des données avec un circuit intégré dédié (ASIC). Par conséquent, il a été envoyé à Eve Descartes d'ESIEE Paris pour une rétro-conception matérielle. Ainsi, suite à son expertise, elle nous a fait un compte rendu expliquant différentes informations sur la méthode de chiffrement utilisée. De plus, en fin de mail, elle nous indique qu'elle reste joignable par mail ou par téléphone pour de plus amples informations. En ce qui concerne l'archive et la photographie, elles sont protégées par un clé de chiffrement et nous devons les déchiffrer pour connaître le contenu !

2.3.1 Code Morse

Suite à l'expertise des fichiers mises à disposition, nous n'avons pas trouvé de porte d'entrée afin d'intercepter les algorithmes de chiffrement utilisés dans le cadre de déchiffrer les fichiers. De plus, le compte rendu nous explicite que la photographie au microscope électronique à balayage des fusibles comporte sensiblement l'algorithme de chiffrement et la clé utilisée, or cette dernière est chiffrée par un mot de passe.

Par conséquent, nous avons décidé d'envoyer un mail à Eve Descartes en lui demandant de l'aide, pensant qu'il y aurait un message automatique avec un indice. Cela est bien le cas, car cette dernière nous invite à l'appeler pour lui poser toutes éventuelles questions.

DESCARTES, Eve
À moi

Je suis actuellement indisponible.
Pour tout problème même minuscule, n'hésitez pas à me contacter par téléphone.

--
Eve Descartes
Ingénieur de Recherche Salle Blanche
Service pour la Microélectronique et les Microsystèmes (S.M.M.)
Tél. : +33 (0)1 45 92 60 96
2 boulevard Blaise Pascal - BP 99
93162 Noisy-le-Grand CEDEX • FRANCE
www.esiee.fr • www.univ-gustave-eiffel.fr

FIGURE 2.2 – Mail envoyé à Eve Descartes

Ainsi, nous avons composé le numéro de téléphone d'Eve Descartes, et bingo, il y avait un message pré-enregistré en morse [4].

Nous avons enregistré ce message sur Audacity et étudié les différents sons émis afin de retrouver le message suivant : `esistance`

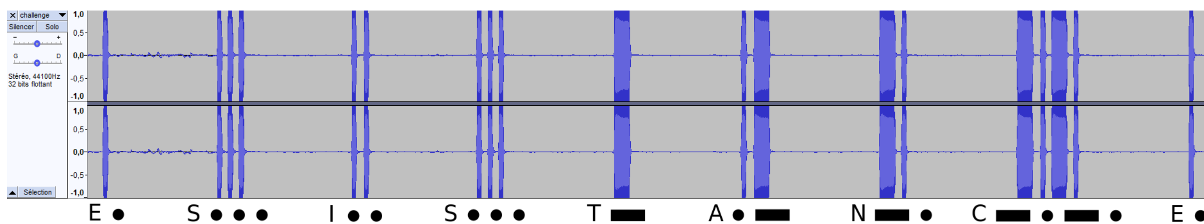


FIGURE 2.3 – Code Morse capturé par Audacity

Nous avons déduit grâce au contexte la valeur exacte du code Morse :

```
resistance
```

2.3.2 Code Binaire

Ce message est en réalité le mot de passe permettant de déchiffrer la photographie au microscope électronique à balayage des fusibles. Suite au déchiffrage de cette dernière, nous avons pu identifier les 256 micro-fusibles. Sur cette photographie, nous avons pu remarquer des micro-fusibles où le courant électrique circulait (état ON) et d'autres où le courant ne circulait pas (état OFF). Par conséquent, il nous ait venu à l'esprit de le transcrire en 0 pour l'état OFF et en 1 pour l'état ON :

```
1011111010111010
1010110011011111
1100110111001010
1100100111011111
1011101010111100
1011110111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
1101111111011111
```

TABLE 2.1 – Transcription des micro-fusibles à l'état OFF et ON en bits

Cependant, le compte rendu d'Eve Descartes nous explicite que l'ordre des bits et la position du MSB n'est pas définis. De ce fait, il est nécessaire de tester les différentes positions possibles afin de retrouver dans un premier temps l'algorithme de chiffrement puis la clé. Il est vrai que l'ordre des bits récupéré ne nous permettait pas d'aller plus loin, c'est-à-dire retrouver un message spécifique lorsque nous affichons le binaire en ASCII.

4. Lien permanent : https://upload.wikimedia.org/wikipedia/commons/b/b5/International_Morse_Code.svg

Par conséquent, nous avons dans un premier temps inversé les bits 0 et 1, afin d'avoir le bit 0 pour l'état ON et le bit 1 pour l'état OFF :

0100000101000101
0101001100100000
0011001000110101
0011011000100000
0100010101000011
0100001000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000
0010000000100000

Table 2.2 – Transcription inversé des micro-fusibles à l'état ON et OFF en bit

Ici, contrairement à la première tentative, nous avons pu récupérer un message très intéressant nous indiquant l'algorithme de chiffrement lorsque nous affichons le binaire en ASCII :

`AES 256 ECB`

Ainsi, cela nous incite à utiliser l'algorithme de chiffrement symétrique AES en 256 bits avec le mode ECB afin de déchiffrer l'archive chiffrée. Néanmoins, il nous manque la clé de déchiffrement, et cette dernière doit avoir une taille de 256 bits compte tenu de l'algorithme de chiffrement trouvé. Du coup, le compte rendu d'Eve Descartes nous indiquait que la photographie au microscope électronique à balayage des fusibles contenait la clé, et donc nous avons récupérer notre transcription inversée et nous l'avons écrite en hexadécimal, et en supposant qu'il s'agirait de la clé :

`4145532032353620454342202020202020202020202020202020202020202020`

Il nous reste plus qu'à vérifier !

2.3.3 Déchiffrement de l'archive par OpenSSL

Dans la précédente sous-partie, nous avions récupéré l'ensemble des paramètres nécessaires pour tenter le déchiffrage de l'archive, à savoir le type d'algorithme utilisé et la clé de chiffrement.

En conséquence, nous avons utilisé l'outil OpenSSL disponible sur notre machine Kali afin d'effectuer ce déchiffrement :

Terminal

```
kali@kali:~$ openssl enc -aes-256-ecb -d -K 41455320323536204543422020202020202020
20202020202020202020202020 -in archive_chiffree -out archive_dechiffre.zip

kali@kali:~$ mkdir archive_challenge_dgse_esiee

kali@kali:~$ unzip archive_dechiffre.zip -d archive_challenge_dgse_esiee/

kali@kali:~$ cd archive_challenge_dgse_esiee/archive && ls -al
-rwxrwxrwx 1 kali kali 51636 sept. 20 15:06 code_acces.pdf
-rwxrwxrwx 1 kali kali 64910 sept. 20 15:08 message.pdf
```

Ainsi, nous avons remarqué qu'il y avait deux fichiers *PDF* au sein de notre archive chiffrée. Cependant, l'un d'eux est un message explicitant la méthode à utiliser pour déchiffrer le deuxième fichier *PDF* qui comporte le code d'accès.

2.3.4 Résolutions Mathématiques

La précédente sous-partie nous a montré l'existence de deux fichiers dans l'archive. De ce fait, après avoir exploré ces fichiers, nous avons remarqué que nous devions résoudre une équation mathématique afin de déchiffrer le fichier **code_acces.pdf**.

De ce fait, résolvons l'équation modulaire suivante :

Le mot de passe est x tel que :

$$\begin{cases} x^3 \equiv 573[8387] \\ x^3 \equiv 98[9613] \\ x^3 \equiv 2726[7927] \end{cases}$$

Pour cela, nous avons utilisé le système de résolution de dCode[5] :

$$x^3 \equiv 573[8387] \Downarrow x = 5622 \qquad x^3 \equiv 98[9613] \Downarrow x = 5622 \qquad x^3 \equiv 2726[7927] \Downarrow x = 5622$$

Par conséquent, nous avons réussi à calculer le mot de passe du fichier chiffré **code_acces.pdf** :

5622

Dès à présent, nous pouvons ouvrir ce fichier et voir ce qu'il contient ! En ouvrant ce dernier, nous avons pu voir que le mot de passe était encore chiffré et qu'il est nécessaire de le déchiffrer afin de résoudre notre mission et de parvenir à la dernière étape. Pour cela, il faut encore une fois résoudre un problème mathématique, un polynôme dans un corps de Galois cette fois-ci :

$$GF(256) = \frac{Z_2[X]}{(X^8 + X^4 + X^3 + X + 1)}$$

Pour résoudre ce dernier, nous avons eu une indication sur le chiffrement : Le mot de passe a été chiffré par substitution mono-alphabétique, c'est-à-dire que chaque caractère ASCII est remplacé par son inverse dans le corps.

Voici le mot de passe chiffré : `0xAF3A5E20A63AD0`

De ce fait, d'un point de vue technique, nous avons converti les caractères chiffrés hexadécimaux en valeur décimale (ASCII) puis calculer l'inverse dans le corps de Galois grâce au système de résolution de l'Université du Nouveau-Brunswick au Canada[6] avec le polynôme précédemment décrit et enfin retranscrit la valeur décimale obtenue en ASCII selon la table de correspondance :

Caractère hexadécimal	Valeur décimale (ASCII)	Valeur de l'inverse dans le corps de Galois	Caractère ASCII
AF	175	98	b
3A	58	32	[[espace]]
5E	94	97	a
20	32	58	:
A6	166	101	e
3A	58	32	[[espace]]
D0	208	122	z

TABLE 2.3 – Résolution de l'inverse dans le corps de Galois

Par conséquent, nous avons réussi à trouver le mot de passe de la mission :

`b a:e z`

Ainsi, ce message doit être envoyé à Antoine Rossignol afin de recevoir les nouvelles directives concernant la suite de notre mission.

5. Lien permanent : https://www.dcode.fr/solveur-equation-modulaire
6. Lien permanent : http://www.ee.unb.ca/cgi-bin/tervo/calc2.pl?num=1&den=inconnue&f=d&p=36&d=1&y=1

2.3.5 Inscription sur la plateforme de Capture The Flag

A la suite l'envoi du message précédemment trouvé à Antoine Rossignol, nous avons reçu le message suivant :

Message

C'est ça ! Tiens, ça me fait penser à Enigma. Notez-le ça pourrait vous servir ultérieurement... On sait maintenant qu'Evil Country va utiliser du VX contre sa propre population, connectez-vous sur cette plateforme top secrète pour continuer l'enquête : /7a144cdc500b28e80cf760d60aca 2ed3, on ne peut pas prendre le risque qu'un agent double ne s'en rende compte.

Nous remarquons ici, que nous sommes invités à rejoindre le site internet suivant, qui est en réalité la plateforme de Capture The Flag de ce challenge :

`https://ctf.challengecybersec.fr/7a144cdc500b28e80cf760d60aca2ed3`

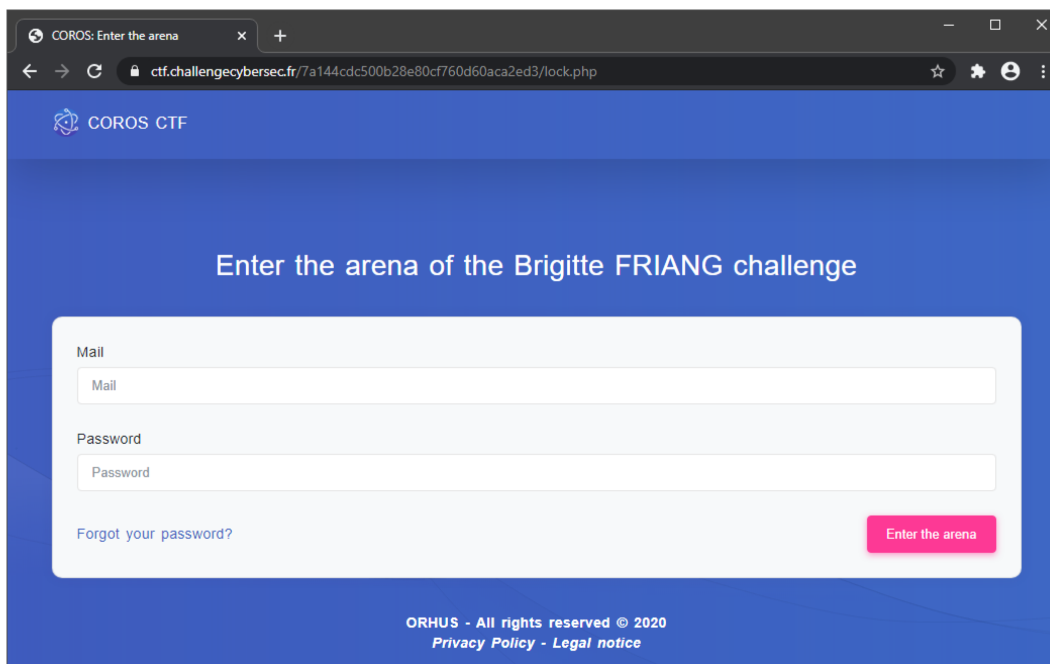


FIGURE 2.4 – Plateforme de Capture The Flag

Ainsi, nous nous sommes inscrit et constitué notre équipe de choc pour pouvoir débuter la partie de Capture The Flag !

Chapitre 3

Capture The Flag

3.1 Web

3.1.1 ChatBot

Chatbot, est l'unique challenge de la section *Web*. Son nom et la section nous donne une indice sur le contenue du challenge : l'étude d'une page web simulant un assistant virtuel. D'après l'énoncé du challenge, il existerait une faille de sécurité qui pourrait nous permettre d'accéder aux machines internes du système d'information, c'est-à-dire aux machines du même réseau local que celui qui permet d'héberger le chatbot, et notamment à l'intranet.

Pour cela, dans un premier temps, nous nous sommes rendu sur le chabot et nous avons exploré le code source. Aux premiers abords, le code source ne compte aucune faille de sécurité compte tenu qu'il comporte uniquement des feuilles de styles *CSS* et des scritps *JavaScript* non vulnérables.

Dans un second temps, nous avons étudié les requêtes envoyées par les différents boutons présents sur le chatbot via l'utilitaire BurpSuite, un logiciel permettant d'effectuer des tests de pénétration sur les applications web.

Le premier bouton effectue une requête GET avec le paramètre suivant : `bot?message=Bonjour`

Ce paramètre ne présent pas de failles car il ne sert uniquement à envoyer un message au chatbot et ce dernier répond par :

1. *« Bonjour ! Vous pouvez me poser n'importe quelle question sur les emplois que nous proposons* » pour avoir entré *"Bonjour"*
2. *« Nous proposons tous types d'emplois, administratifs ou sur le terrain. Nous recherchons principalement des agents, pour le moins secrets"* pour avoir entré *"Liste des emplois »*
3. *« Je ne comprends pas »* pour avoir entré tout autre chose !

En revanche, le dernier bouton nous permet d'envoyer une requête GET avec une demande d'accès à un site externe via le proxy : `proxy?url=https://www.qwant.com/`

Nous remarquons que la demande d'accès à un site peut être externe ou bien interne car le proxy est directement lié avec les machines du système d'information. Par conséquent, nous pouvons avec un moyen quelconque accéder aux machines connectés sur le même réseau que le proxy.

En ce sens, nous avons lancé la commande `dirb` depuis une machine Kali Linux à l'adresse suivante afin d'énumérer tous les dossiers énumérables et disponibles depuis le proxy :

Terminal

```
kali@kali:~$ dirb https://challengecybersec.fr/[...]/proxy?url=http://

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Mon Nov  9 15:53:31 2020
URL_BASE: https://challengecybersec.fr/[...]/proxy?url=http://
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612

---- Scanning URL: https://challengecybersec.fr/[...]/proxy?url=http:// ----
+ https://challengecybersec.fr/[...]/proxy?url=http://admin.pl (CODE:200|SIZE:7999)
[...]
+ https://challengecybersec.fr/[...]/proxy?url=http://intra (CODE:200|SIZE:1052)
+ https://challengecybersec.fr/[...]/proxy?url=http://intranet (CODE:200|SIZE:698)
```

L'énumération faite via `dirb` nous permet de voir qu'il existe de nombreux dossiers voire des machines accessibles depuis le proxy, dont ceux qui réfèrent à l'intranet. Par conséquent, il suffit de se rendre sur les liens suivantes URL[1] [2] pour voir si nous avons accès à ces derniers et si nous pouvons trouver le flag demandé.

L'accès au site `http://intra/`, nous permet d'accéder à l'intranet via le proxy. Et ce dernier, comporte des informations intéressantes dont le flag, comme nous pouvons le voir ci-dessous :

```
DGSESIEE{2cf1655ac88a52d3fe96cb60c371a838}
```

1. Lien permanent : https://challengecybersec.fr/[...]/proxy?url=http://intra
2. Lien permanent : https://challengecybersec.fr/[...]/proxy?url=http://intranet

3.2 Hardware

3.2.1 ASCII UART

ASCII UART est l'un des deux challenges de la section *Hardware*. De part son nom, nous avons déjà une bonne indication de ce qui nous attend ici : l'**UART**. Nous le connaissions de nom mais le protocole pour pouvoir déchiffrer nous était inconnu. Ainsi, nous avons effectué des recherches et nous sommes tombés sur une description très complète sur ce site [3]. Nous avons notamment la structure d'un paquet qui permettra par la suite d'extraire les données une par une :

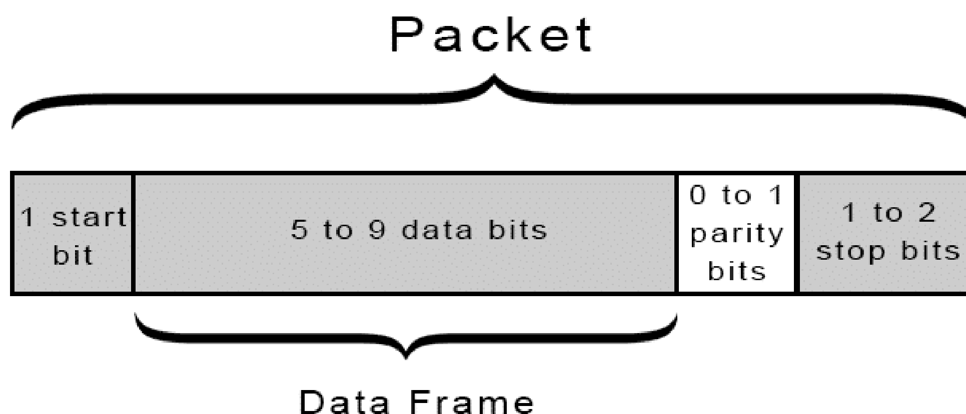


Figure 3.1 – Structure d'un paquet de données en UART

Maintenant, il ne nous reste plus qu'à savoir quel est le signal récupéré par l'informateur. Pour cela, nous savons que les données sont en 8 bits signés autrement dit que les valeurs peuvent être positives comme négatives. Un bon moyen est d'utiliser le logiciel *Audacity* pour visualiser le signal. Pour ce faire une fois le logiciel ouvert, il faut effectuer : *File>Import>Raw Data...* puis configurer les options comme ci-dessous :

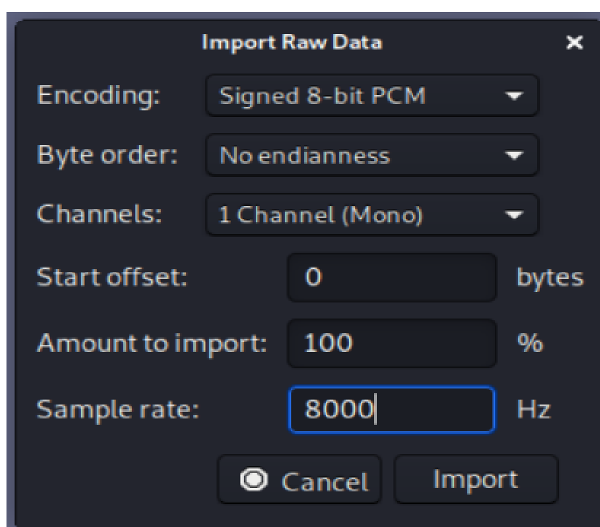


Figure 3.2 – Configuration d'importation du fichier

3. Lien permanent : `https://www.circuitbasics.com/basics-uart-communication/`

Une fois l'importation effectuée, nous voyons bien un signal duquel nous pouvons extraire un flux binaire et se référer au protocole pour déchiffrer le message. La première chose à faire est de trouver la plus petite unité d'information : c'est-à-dire 1 bit, qui va nous servir par la suite de valeur étalon pour établir combien de '**1**' ou '**0**' à la suite il peut y avoir lorsque c'est supérieur à 1 bit. Nous allons expliciter l'extraction de la donnée du premier paquet de données qui intervient après le premier '**bit de start**' autrement dit juste après le premier passage à '**0**'.

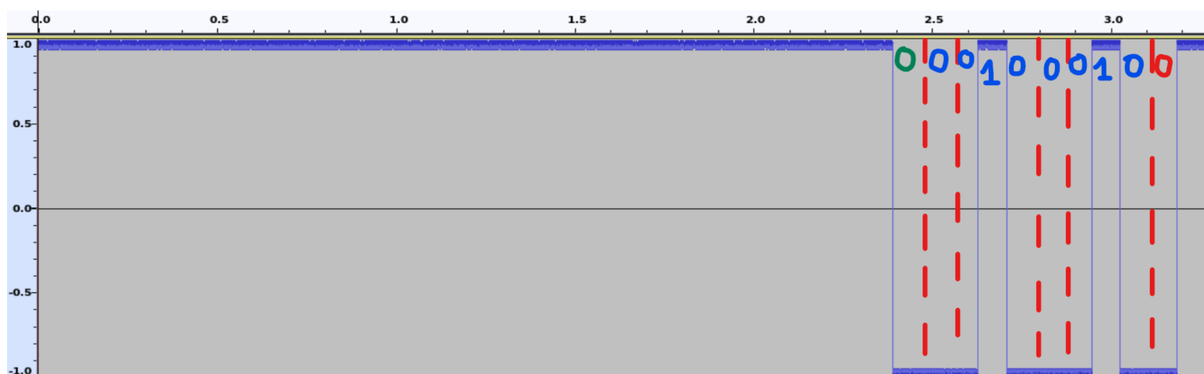


FIGURE 3.3 – Extraction des données du 1er paquet

Nous avons extrait ainsi du premier paquet les données suivantes :

$$001000100$$

Or d'après le protocole, c'est le bit de point faible qui est envoyé en premier, il faut donc inverser l'ordre d'arrivée des bit pour obtenir la valeur souhaitée qu'il ne reste plus qu'à transcrire en valeur ASCII via un tableau de correspondance[4] pour avoir :

$$\overline{001000100}_2 = \overline{44}_{16} = \text{'D'}$$

Nous obtenons ainsi la première valeur, néanmoins nous n'avons pas mentionné le '**bit de parité**' qui permet de contrôler l'intégrité des données dans une moindre mesure. Effectivement, il peut prendre deux valeurs qui vont permettre de contrôler les données précédemment réceptionnées tel que :

— **BP=0** : quand le nombre de '**1**' dans les 8 bits (dans notre cas) de données est pair

— **BP=1** : quand le nombre de '**1**' dans les 8 bits (dans notre cas) de données est impair

Ainsi pour la première valeur, nous trouvons deux '**1**' et le '**BP**' vaut bien '**0**'. Voilà, il ne reste plus qu'à appliquer cela sur l'ensemble du signal et faisant bien attention à l'intégrité des données qui sont refusées si incorrectes. Dans le tableau ci-dessous les données incorrectes verront leur ligne colorée en rouge. Nous obtenons donc le flag suivant :

```
DGSESIEE{d[-_-]b  \_(''/)_/  (^_-)  @}-;---  (*^_^*)  \o/ }
```

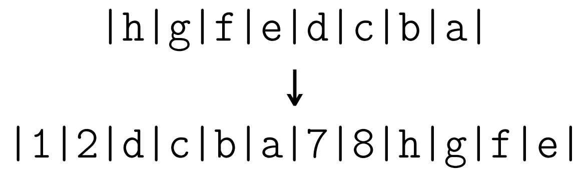
4. Lien permanent : http://www.asciitable.com/

Binaire (LSB)	Binaire (MSB)	Bit de parité	Valeur décimale	ASCII
00100010	01000100	0	68	D
11100010	01000111	0	71	G
11001010	01010011	0	83	S
10100010	01000101	1	69	E
11001010	01010011	0	83	S
10011010	01011001	1	89	Y
10010010	01001001	1	73	I
10100010	01000101	1	69	E
10100010	01000101	1	69	E
11011110	01111011	0	123	{
00000100	00100000	1	32	
00100110	01100100	1	100	d
11011010	01011011	1	91	[
10110100	00101101	0	45	-
11111010	01011111	0	95	_
10110100	00101101	0	45	-
10111010	01011101	1	93	]
01000110	01100010	1	98	b
00000100	00100000	1	32	
00000100	00100000	1	32	
00111010	01011100	0	92	\
11111010	01011111	0	95	_
00010100	00101000	0	40	(
11100100	00100111	0	39	'
11100100	00100111	0	39	'
11110100	00101111	1	47	/
10010100	00101001	1	41	)
11111010	01011111	0	95	_
11110100	00101111	1	47	/
00000100	00100000	1	32	
00000100	00100000	1	32	
00010100	00101000	0	40	(
01111010	01011110	1	94	^
11111010	01011111	0	95	_
10110100	00101101	0	45	-
10010100	00101001	1	41	)
00000100	00100000	1	32	
00000100	00100000	1	32	
00000100	00100000	1	32	
00000010	01000000	1	64	@
10111110	01111101	0	125	}
10110100	00101101	0	45	-
11011100	00111011	1	59	;
10110100	00101101	0	45	-
10111100	00111101	0	61	=
10110100	00101101	0	45	-
10110100	00101101	0	45	-
00000100	00100000	1	32	
00000100	00100000	1	32	
00000100	00100000	0	32	
00000100	00100000	1	32	
00000100	00100000	1	32	
00010100	00101000	0	40	(
01010100	00101010	1	42	*
01111010	01011110	1	94	^
11111010	01011111	0	95	_
01111010	01011110	1	94	^
01010100	00101010	1	42	*
10010100	00101001	1	41	)
00000100	00100000	1	32	
00000100	00100000	1	32	
00111010	01011100	0	92	\
11110110	01101111	0	111	o
11110100	00101111	1	47	/
00000100	00100000	1	32	
10111110	01111101	0	125	}

Table 3.1 – Tableau récapitulatif de la communication UART

3.2.2 Keypad Sniffer

Nous arrivons au deuxième challenge du domaine *Hardware*, après avoir regardé l'ensemble des fichiers mis à notre disposition, nous avons établi un plan d'action pour récupérer le code d'accès tapé via ce keypad. Dans un premier temps, il faut établir un lien entre les 8 pinouts du keypad et les 12 pinouts de la carte portant l'ensemble du dispositif. Pour ce faire, nous nous sommes appuyés sur les deux images (de face et de dos) du système global et nous avons obtenu cela (en se plaçant face au keypad) :



Cela permet d'identifier les bits qui vont nous intéresser pour la suite du challenge, c'est-à-dire tout ceux qui sont des lettres dans l'exemple ci-dessus. Dans un deuxième temps, il faut trouver comment est transcrite chaque touche au format 8 bits, autrement dit en sortie du keypad. Pour cela, nous nous sommes renseigné et nous avons trouvé un site[5] expliquant le fonctionnement de la matrice et notamment le tableau des connexions suivant :

Pmod Connector	Pmod Pin Number	Pmod Port	Keypad Pmod Controller Port
J1	1	COL4	columns(4)
J1	2	COL3	columns(3)
J1	3	COL2	columns(2)
J1	4	COL1	columns(1)
J1	5	GND	-
J1	6	VCC	-
J1	7	ROW4	rows(4)
J1	8	ROW3	rows(3)
J1	9	ROW2	rows(2)
J1	10	ROW1	rows(1)
J1	11	GND	-
J1	12	VCC	-

FIGURE 3.4 – Tableau des connexions de la matrice 4x4 du keypad

Ainsi, nous pouvons établir le lien entre les bits et la matrice qui sont résumés dans le tableau suivant :

	d	c	b	a
h	1	2	3	F
g	4	5	6	E
f	7	8	9	D
e	A	0	B	C

TABLE 3.2 – Matrice du keypad

De plus, le fonctionnement est très simple grâce aux 4 premiers bits, nous déterminons la colonne sur laquelle se trouve la touche qui est marquée par un '**0**' et les 4 derniers bits permettent de trouver la ligne. Grâce à cela, par croisement nous avons la valeur de la touche.

5. Lien permanent : https ://www.digikey.com/eewiki/pages/viewpage.action ?pageId=86278246

Enfin, il ne nous reste plus qu'un dernier problème à régler, qui est de déterminer quand une touche est relâchée et quand nous appuyons dessus. Pour cela, nous avons analysé les données et nous avons remarqué le schéma suivant :

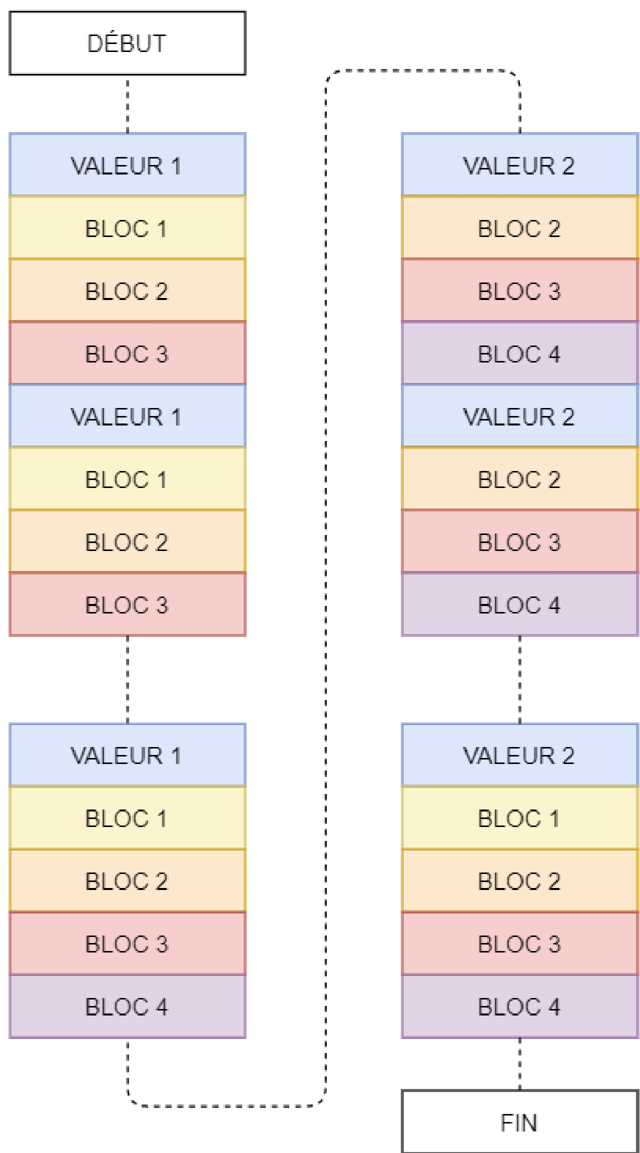


FIGURE 3.5 – Vue macro des données

Avant d'aller plus loin, nous allons explicité certains éléments du schéma :

— **VALEUR 1** : correspond à une suite de valeurs binaires identiques consécutivement et qui par correspondance représentent une touche du keypad

— **BLOC n** : correspond à une suite de valeurs binaires identiques consécutivement et qui ne correspondent pas à une valeur du keypad

Nous voyons tout de suite dans les données tapées, les blocs **VALEUR** sont séparées par trois **BLOC** et encadrées par des suites de quatre **BLOC**. Nous pouvons donc ainsi supposer que lorsque nous appuyons sur une touche les **VALEUR** se répètent et sont séparées par trois **BLOC** jusqu'à ce que la touche soit relâchée et ainsi de suite. Il nous faut donc dorénavant extraire la séquence de valeurs binaire en prenant en compte l'appuie et le relâchement de la touche qui vient d'être expliqué. Pour ce faire, nous avons mis en place un script en *Python3* présent en *Annexe*. En exécutant le script nous obtenons le flag suivant :

`DGSESIEE{AE78F55C666B23011924}`

3.3 Forensic

3.3.1 Sous l'océan

Nous arrivons à la première épreuve de *Forensic* dans laquelle nous faisons face à un dump mémoire d'un téléphone Android. Ce qui nous est demandé, est de retrouver la dernière position du téléphone. La première démarche que nous avons eu est naïve, nous avons tout simplement parsé le fichier à la recherche de mots-clé comme **GPS**. Malheureusement, dans le dump mémoire il y a beaucoup d'informations concernant les processus, les applications et beaucoup d'entre elles font appel au gps de manière générale et donc cette recherche donne beaucoup trop de résultat pour qu'elle soit facilement exploitable. Pour obtenir des résultats plus intéressants, nous allons affiner la recherche en ignorant la case, autrement dit nous ne prenons pas en compte le fait que cela soit une majuscule ou une minuscule, avec la commande suivante :

Terminal

```
kali@kali:~$ cat memdump.txt | grep -i 'last known location'
```

Avec cette commande, nous obtenons un résultat intéressant ! Pour voir, un peu plus le contexte, nous allons effectuer la commande suivante (certaines parties non intéressantes pour le problème ont été tronquées) :

Terminal

```
kali@kali:~$ cat memdump.txt | grep -i -B10 -A10 'last known location'
[...]
  Last Known Locations:
    gps: Location[gps 37.421998,-122.084000...]
    passive: Location[gps 37.421998,-122.084000...]
  Last Known Locations Coarse Intervals:
    gps: Location[gps 37.421998,-122.084000...]
    passive: Location[gps 37.421998,-122.084000...]
  Custom Location History :
  Custom Location 1
        gps: Location[gps -47,1462046   30,9018186 hAcc=20...]
        gps: Location[gps -47,1963297   30,9012294 hAcc=20...]
        gps: Location[gps -47,1970164   30,8641039 hAcc=20...]
        gps: Location[gps -47,1438013   30,8652827 hAcc=20...]
        gps: Location[gps -47,1448313   30,9642508 hAcc=20...]
  Custom Location 2
        [...]
```

Nous voyons bien ici que des coordonnées GPS sont présentes, et nous voyons également qu'elles ont un motif commun : **gps : Location[gps ...]**. Grâce à cela, nous allons extraire toutes les données GPS pour les mettre dans un fichier comme ceci :

Terminal

```
kali@kali:~$ cat memdump.txt | grep 'gps: Location\[gps' > gps_location
```

Ensuite, avant de se lancer dans le traitement des données GPS, nous remarquons qu'en plus de la '**Last Known Locations**' qui est la dernière position connue du téléphone, il y a plus d'une dizaine de '**Custom Location**'. Nous avons recherché sur Internet cette position, et cela nous a mené jusqu'à l'Océan Indien ! Étrange, puisque le titre du challenge est *Sous l'océan*, nous nous penchons davantage sur ces mesures, nous voyons qu'elles sont très proches mais qu'il n'y a rien à cet emplacement (aucune ville, île...). Il nous reste une possibilité qui consiste en placer les points (tout ceux "*sous l'océan*") et de les relier sur un graphique, via le script en *Python3* en *Annexe*, nous obtenons le graphique suivant : Ainsi, nous remarquons qu'il s'agit bien de la réponse au challenge

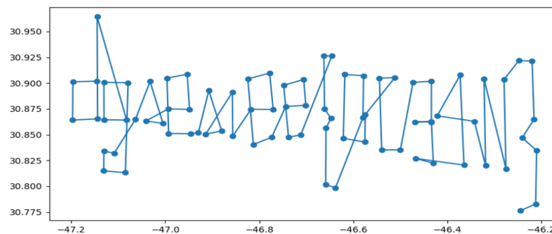


Figure 3.6 – Graphique de réponse au challenge

et qu'il s'agit d'un message à retranscrire. Nous obtenons le flag suivant :

`DGSESIEE{OC34N}`

3.4 Stéganographie

3.4.1 Polyglotte

Voici le premier challenge de stéganographie, ce dernier nous met à disposition deux fichiers : un fichier *PDF* et une archive. La première chose à faire est d'analyser ces fichiers et nous nous rendons compte que l'archive est protégée par un mot de passe. Ainsi, nous allons analyser le fichier *PDF* pour en extraire un mot de passe. Pour ce faire, nous allons utilisé des commandes de bases pour afficher les chaînes de caractères présentes dans le documents *PDF* comme ci-dessous :

Terminal

```
kali@kali:~$ strings message.pdf
<%PDF-1.2
<html>
<!DOCTYPE html>
<html>
        <head>
                <title>Flag</title>
                <meta charset="utf-8">
        </head>
        <body>
        <script>var flag = [91,48,93,97,97,57,51,56,97,49,54];</script>
<!--
[...]
 <5b 31 5d 34 64 38 36 32 64 35 61> Tj
[...]
 <43 65 20 64 6f 63 75 6d 65 6e 74 20 63 6f 6e 63 65 72 6e 65 20 6c 20 6f 70 65 72 61
  74 69 6f 6e 20 73 6f 6c 65 69 6c 20 61 74 6f 6d 69 71 75 65 2e 0a 43 65 74 74 65 20
  6f 70 65 72 61 74 69 6f 6e 20 65 73 74 20 73 74 72 69 63 74 65 6d 65 6e 74 20 63 6f
  6e 66 69 64 65 6e 74 69 65 6c 6c 65 20 65 74 20 6e 65 20 64 6f 69 74 20 65 6e 20 61
  75 63 75 6e 20 63 61 73 20 ea 74 72 65 20 64 65 76 6f 69 6c 65 65 2e 20 0a 4c 65 73
  20 69 6e 66 6f 72 6d 61 74 69 6f 6e 73 20 73 75 72 20 6c 20 6f 70 65 72 61 74 69 6f
  6e 20 73 6f 6e 74 20 64 69 73 73 65 6d 69 6e e9 65 73 20 64 61 6e 73 20 63 65 20 66
  69 63 68 69 65 72 2e 0a 43 68 61 71 75 65 20 70 61 72 74 69 65 20 64 65 20 6c 20 69
  6e 66 6f 72 6d 61 74 69 6f 6e 20 65 73 74 20 69 64 65 6e 74 69 66 69 65 65 20 70 61
  72 20 75 6e 20 6e 6f 6d 62 72 65 20 70 61 72 20 65 78 20 3a 20 0a 5b 30 5d 61 65 37
  62 63 61 38 65 20 63 6f 72 72 65 73 70 6f 6e 64 20 61 20 6c 61 20 70 72 65 6d 69 e8
  72 65 20 70 61 72 74 69 65 20 64 65 20 6c 20 69 6e 66 6f 72 6d 61 74 69 6f 6e 20 71
  75 20 69 6c 20 66 61 75 74 20 63 6f 6e 63 61 74 65 6e 65 72 20 61 75 20 72 65 73 74
  65 2e> Tj
[...]
--!>
                <script>
                    for(i=0;i<flag.length;i++){
                        flag[i] = flag[i]+4
                    }
                    alert(String.fromCharCode.apply(String, flag));
                </script>
        <body>
</html>
```

Nous pouvons voir tout de suite plusieurs choses intéressantes au début du fichier. Tout d'abord, la présence de code *HTML* avec du script *Javascript* (qui a été mis en forme plus haut pour une meilleure visibilité). D'autre part, entre les balises de commentaires (**<!—-!>**), nous retrouvons deux flux de données en hexadécimale regroupées par paquet de 2 octets, ce qui peut correspondre à du texte en transcrivant cela à l'aide d'une table ASCII.

Avant de recopier de le code *HTML* pour interpréter le code *Javascript*, nous allons vérifier si les flux de données correspondent à quelque chose. Il s'avère que nous trouvons des informations intéressante pour les deux, qui en transcrivant en ASCII nous donne :

5b 31 5d 34 64 38 36 32 64 35 61

[1]4d862d5a

43 65 20[...]74 65 2e

```
Ce document concerne l operation soleil atomique.
Cette operation est strictement confidentielle et ne doit en aucun
cas être devoilee.
Les informations sur l operation sont disseminées dans ce fichier.
Chaque partie de l information est identifiee par un nombre par ex :
[0]ae7bca8e correspond a la première partie de l information qu il
faut concatener au reste.
```

Nous avons là quelque chose de très intéressant puisqu'un des flux de données correspond à la trame à suivre pour constituer le flag final (avec la première partie de ce dernier) tandis que le deuxième flux nous donne des informations sur la typologie du flag final et nous permet de trouver une deuxième partie du flag. De plus, si nous le regardons plus en détail le code *HTML* et en particulier la variable **flag** qui est une suite de nombres décimaux et que nous le retranscrivons en ASCII, nous obtenons :

91 48 93 97 97 57 51 56 97 49 54

[0]aa938a16

Ensuite, nous avons cherché à trouver d'autres informations dans ce fichier, sans succès. Cependant, le fichier comporte énormément de sections **Comments**, qui paraît à premier abord inexploitable. Nous essayerons de voir par la suite que cela n'est pas anodin. Pour voir cela, nous avons utilisé un parser de fichiers *PDF* : **pdf-parser** ; en utilisant simplement l'option **a** permettant d'afficher l'ensemble des statistiques du fichier.

Terminal

```
kali@kali:~$ pdf-parser -a message.pdf
Comment: 212
XREF: 0
Trailer: 0
StartXref: 0
Indirect object: 9
  5: 2, 7, 8, 9, 11
 /Page 2: 5, 6
 /Pages 2: 3, 4
```

Ainsi, nous sommes à un point où rien n'est plus exploitable sur ce fichier mais il nous reste tout de même l'archive protégée par un mot de passe que nous ne connaissons pas. La première chose que nous pouvons faire face à cette situation est de tester un *Bruteforce* de mot de passe. Pour cela, nous avons utilisé l'utilitaire **fcrackzip** avec les options suivantes :

- **u** : qui force l'utilisation de l'utilitaire **unzip** pour la décompression ainsi lorsqu'un mot de passe incorrect sera utilisé, l'utilitaire le notifiera et ce dernier pourra être éliminé.
- **D** : qui permet l'utilisation d'un dictionnaire (liste) de mots de passe, pour ce cas nous avons utilisé la liste bien connue **rockyou.txt**.
- **p** : qui permet de spécifier le fichier contenant les mots de passe (spécifié juste avant)

Terminal

```
kali@kali:~$fcrackzip -uDp '/usr/share/wordlists/rockyou.txt' secrets.zip

PASSWORD FOUND!!!!: pw == finenuke
```

Parfait ! Nous avons trouvé le mot de passe de l'archive ! Maintenant, nous allons dézipper le fichier et voir ce qu'il comporte. Nous nous retrouvons avec deux fichiers, une image (**hint.png**) (voir 3.7) et un fichier texte (**info.txt**) contenant les indications suivantes :

```
Ange Albertini
key='\xce]`^+5w#\x96\xbbsa\x14\xa7\x0ei'
iv='\xc4\xa7\x1e\xa6\xc7\xe0\xfc\x82'
[3]4037402d4
```

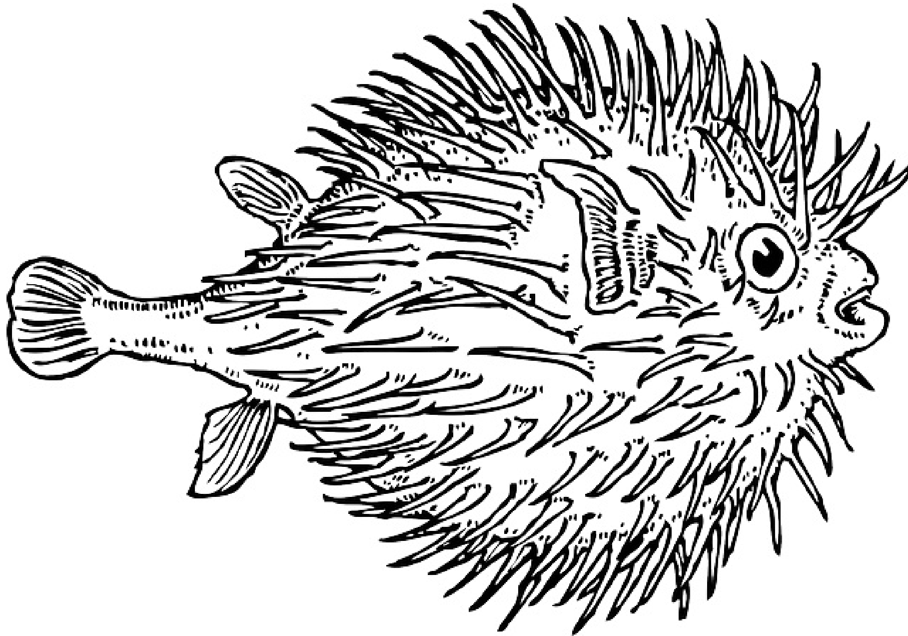


FIGURE 3.7 – Image **hint.png** de l'archive **secrets.zip**

Tout d'abord, nous remarquons qu'une troisième partie du flag est présente dans le fichier texte ainsi qu'une clé et un iv avec un nom : **Ange Albertini**. En effectuant quelques recherches, nous nous rendons compte que le nom fait référence à une technique mise au point par la personne du même nom : **Angecryption**. Sans rentrer dans les détails, elle permet de dissimuler dans un fichier conteneur (exemple : *PNG* ou encore *PDF*) un autre fichier qui n'est récupérable qu'après chiffrement du conteneur. Les premières recherches ont été effectuées en utilisant le chiffrement *AES-256-CBC*, or dans notre cas quand nous essayons de chiffrer le fichier **message.pdf** avec cette algorithme cela ne fonctionne pas. C'est là qu'intervient l'image **hint.png** qui représente un poisson mais pas n'importe lequel : un blowfish ; or c'est également le nom d'un algorithme de chiffrement. Avec cette nouvelle information, nous tentons de chiffrer le fichier avec cette algorithme et en utilisant les informations contenues dans le fichier **info.txt**, nous obtenons le résultat suivant :

Terminal

```
kali@kali:~$ openssl enc -e -bf-cbc -K ce5d605e2b35772396bb736114a70e69 -iv c4a71ea6c7e
0fc82 -in message.pdf -out message_dec.jpg
```

FIGURE 3.8 – Image **message_dec.jpg**

Après avoir regardé via l'utilitaire de manipulation de photo : **GIMP**, nous nous sommes rendu compte qu'aucun texte ou informations n'est caché dans cette image. Nous allons donc regarder plus en détail ce qui constitue cette image. Pour ce faire nous allons utilisé l'utilitaire **exiftool** et procéder à une analyse de l'image la plus complète possible (d'où la verbosité à 3).

Terminal

```
kali@kali:~$ exiftool -v3 message_dec.jpg
  ExifToolVersion = 12.07
  FileName = message_dec.jpg
  Directory = .
  FileSize = 89704
  FileModifyDate = 1604148053
  FileAccessDate = 1605015302
  FileInodeChangeDate = 1604148053
  FilePermissions = 33188
  FileType = JPEG
  FileTypeExtension = JPG
  MIMEType = image/jpeg
  [...]
JPEG EOI
Unknown trailer (17091 bytes at offset 0x11ba5):
   11ba5: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 [.ELF............]
   11bb5: 03 00 3e 00 01 00 00 00 40 11 00 00 00 00 00 00 [..>.....@.......]
   11bc5: 40 00 00 00 00 00 00 00 f8 3a 00 00 00 00 00 00 [@........:......]
   11bd5: 00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00 [....@.8...@.....]
   11be5: 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00 [........@.......]
   11bf5: 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 [@.......@.......]
    [snip 16899 bytes]
   15e08: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 [................]
   15e18: 00 00 00 00 00 11 00 00 00 03 00 00 00 00 00 00 [................]
   15e28: 00 00 00 00 00 00 00 00 00 00 00 00 00 dc 39 00 [..............9.]
   15e38: 00 00 00 00 00 1a 01 00 00 00 00 00 00 00 00 00 [................]
   15e48: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 [................]
   15e58: 00 00 00 00 00 01 02 02 49 f6 25 77 bc 05 c4 c7 [........I.%w....]
```

Parmi les différentes sections de l'image, nous constatons quelque chose d'intéressant, puisque nous avons pu détecter le **magic number** pour les fichiers **ELF**. Cela voudrait dire qu'un binaire se cache dans cette image. Pour en être certain, nous allons utiliser un autre outil spécialisé dans l'extraction de binaire : **binwalk**

Terminal

```
kali@kali:~$ binwalk --dd='.*' message_dec.jpg
```

Nous nous trouvons avec le binaire **11BA5**, qui lorsque nous l'exécutons, nous demande un mot de passe. Ainsi, nous allons donc devoir reverser ce fichier pour trouver le mot de passe attendu. Pour cela, nous avons utilisé l'outil **Ghidra** pour la décompilation du programme, et nous voyons la chose suivante dans la fonction **main()** :

fonction main binaire 11BA5

```
undefined8 main(void)

{
  size_t sVar1;
  byte *__dest;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Operation Soleil Atomique");
  printf("Entrez le mot de passe : ");
  fgets(local_28,0x10,stdin);
  sVar1 = strlen(local_28);
  __dest = (byte *)malloc(sVar1 + 1);
  strcpy((char *)__dest,local_28);
  checkpassword(__dest);
  if (((((((__dest[1] ^ *__dest) == 0x69) && ((__dest[2] ^ __dest[1]) == 0x6f)) &&
        ((__dest[3] ^ __dest[2]) == 0x38)) &&
       (((__dest[4] ^ __dest[3]) == 0x56 && ((__dest[5] ^ __dest[4]) == 0x50)))) &&
      (((__dest[6] ^ __dest[5]) == 0x57 &&
       (((__dest[7] ^ __dest[6]) == 0x50 && ((__dest[8] ^ __dest[7]) == 0x56)))))) &&
     (((__dest[9] ^ __dest[8]) == 6 && (__dest[9] == 0x34)))) {
    puts("Bravo");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  puts("Mauvais mot de passe");
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

Nous voyons tout de suite que le mot de passe qui est recherché et qui permet la validation finale est construit grâce à une succession de *XOR* entre chaque élément consécutif du mot de passe en partant d'une base fixe. Nous avons construit alors le tableau de résolution suivant :

Étape	Valeur A	Valeur B	Résultat C (A XOR C = B)	Valeur ASCII
9	Null	Null	0x34	'4'
8	0x32	0x6	0x32	'2'
7	0x32	0x56	0x64	'd'
6	0x64	0x50	0x34	'4'
5	0x34	0x57	0x63	'c'
4	0x63	0x50	0x33	'3'
3	0x33	0x56	0x65	'e'
2	0x65	0x38	0x5d	']'
1	0x5d	0x6f	0x32	'2'
table0	0x32	0x69	0x5b	'['

TABLE 3.3 – Tableau de résolution du binaire de Polyglotte

En remettant le mot de passe dans le bon sens, nous obtenons la dernière partie du flag. Afin de valider le flag, utilisons-le lors de l'exécution du binaire *11BA5* :

Terminal

```
kali@kali:~$ ./11BA5
Operation Soleil Atomique
Entrez le mot de passe : [2]e3c4d24
Bravo
```

Ainsi, nous pouvons dire que la deuxième partie trouvée est correcte. Il nous reste plus qu'à concaténer les différents parties du flag afin de valider ce challenge :

`DGSESIEE{aa938a164d862d5ae3c4d244037402d4}`

3.5 MISC

3.5.1 Définition

Définition est l'unique challenge de la section *MISC* et consiste à exploiter une faille de sécurité basique et parfois oubliée par les administrateurs.

Tout machine connectée à Internet, comporte des services ouverts afin de fonctionner correctement. Pour pouvoir administrer ses machines, les administrateurs ont le choix de se connecter directement et physiquement sur ses dernières. Cependant, parfois il nécessaire d'accéder aux machines à distance. Pour cela, il existe plusieurs services permettant d'effectuer un connexion à distance : `ssh, netcat, rlogin, etc.`

Malheureusement, parfois ces derniers sont accessibles par tous et cela est un grosse faille de sécurité, car un attaquant peut essayer d'utiliser une vulnérabilité connue afin de compromettre le système d'authentification et avoir un accès sur la machine.

En se sens, dans le cadre de notre challenge, il nous est demandé de résoudre une énigme et d'envoyer le résultat à la machine distante suite à une connexion sur cette dernière via la commande suivante : `nc challengecybersec.fr 6660`. Nous remarquons ici, que la machine distante est bien disponible à tous, et cela sans même connaître le mot de passe. De ce fait, il faut limiter l'accès à la machine aux personnes autorisées via un filtrage d'adresse IP. Si le service ne sert à rien, il est recommandé de l'interrompre.

L'énigme est très simple et nous demande de renvoyer l'heure actuelle. Pour cela, nous pouvons utiliser la commande `date` avec le format `+%s` pour afficher les secondes depuis Unix Epoch (1970-01-01 00 :00 :00 UTC).

Pour envoyer cette commande directement lors de la connexion netcat, nous pouvons utiliser un pipe qui effectuera une redirection de la commande date vers la commande netcat comme suit :

Terminal

```
kali@kali:~$ date +%s | nc challengecybersec.fr 6660
Entrez la reponse :
 > Bravo ! Voici le flag : DGSESIEE{cb3b3481e492ccc4db7374274d23c659}
```

Nous remarquons directement que le flag pour valider le challenge nous est retourné :

`DGSESIEE{cb3b3481e492ccc4db7374274d23c659}`

Chapitre 4

Bibliographie

1. **Challenge Brigitte Friang - [En ligne]** ; https://www.challengecybersec.fr
   (Consulté le 24 Octobre 2020)

2. **Code Cesar - Chiffre de César - Déchiffrer, Coder, Décoder en Ligne, Dcode - [En ligne]** ;
   https://www.dcode.fr/chiffre-cesar
   (Consulté le 24 Octobre 2020)

3. **S3curConv - [En ligne]** ; https://www.challengecybersec.fr/chat
   (Consulté le 25 Octobre 2020)

4. **International Morse Code, Wikimedia - [En ligne]** ;
   https://upload.wikimedia.org/wikipedia/commons/b/b5/International_Morse_Code.svg
   (Consulté le 25 Octobre 2020)

5. **Résoudre une Equation Modulaire - Solveur de Congruence en Ligne, Dcode - [En ligne]** ;
   https://www.dcode.fr/solveur-equation-modulaire
   (Consulté le 25 Octobre 2020)

6. **Basics of UART Communication, Scott Campbell, DIY Electronics - [En ligne]** ;
   https://www.circuitbasics.com/basics-uart-communication/
   (Consulté le 5 Novembre 2020)

7. **Ascii Table - ASCII character codes and html, octal, hex and decimal chart conversion - [En ligne]** ;
   http://www.asciitable.com/
   (Consulté le 3 Novembre 2020)

8. **Keypad Pmod Controller (VHDL) - Logic - eewiki, Scott Larson, 17 Juillet 2019 - [En ligne]** ;
   https://www.digikey.com/eewiki/pages/viewpage.action?pageId=86278246
   (Consulté le 8 Novembre 2020)

9. **Finite field - Wikipedia - [En ligne]** ; https://en.wikipedia.org/wiki/Finite_field
   (Consulté le 25 Octobre 2020)

10. **EE4253 GF(2 ^ m) Calculator, Department of Electrical and Computer Engineering - University of New Brunswick, Fredericton, NB, Canada - [En ligne]** ;
    http://www.ee.unb.ca/cgi-bin/tervo/calc2.pl?num=1&den=inconnue&f=d&p=36&d=1&y=1
    (Consulté le 25 Octobre 2020)

Chapitre 5

Annexes

5.1 Script Sous l'océan

Sous_l_ocean_graph.py

```
#!/bin/python3

import matplotlib.pyplot as plt

def trace_graph(datas):

    x, y = zip(*datas)
    plt.plot(x, y, '-o')
    plt.show()

def main():

    data = None

    with open('gps_location', 'r') as file:
        data = file.readlines()

    if data is None:
        return -1, -1

# for loop to add every latitudes and longitudes in different list

    extract_data = coord_tmp = latitude = longitude = ""
    list_latitudes = []
    list_longitudes = []

    for line in data:

        extract_data = line.split('Location')[1]
        coord_tmp = extract_data.split('hAcc')[0].strip().replace('[gps ','').replace('
        \t', ' ')
        latitude = coord_tmp.split(' ')[0].replace(',','.')
        longitude = coord_tmp.split(' ')[1].replace(',','.')

# add an exception because we don't know if latitude or longitude
# can be cast into integer

        try:
            list_latitudes.append(float(latitude))
            list_longitudes.append(float(longitude))
        except Exception as e:
            raise(e)
            continue

# calcul of the mean of each list

    trace_graph([[list_latitudes[i], list_longitudes[i]] for i in range(len(list_latitu
    des))])

    return 0


if __name__ == '__main__':

    main()
```

5.2 Script Keypad Sniffer

keypad_sniffer_decrypt.py

```
#!/bin/python3

matrix = {
"11011110" : "B", "11100111" : "F", "11010111" : "3",
"10110111" : "2", "11101011" : "E", "11011011" : "6",
"10111011" : "5", "11101101" : "D", "11011101" : "9",
"10111101" : "8", "11101110" : "C", "10111110" : "0",
"01111110" : "A", "01111101" : "7", "01111011" : "4",
"01110111" : "1", "01111111" : "-1", "10111111" : "-2",
"11011111" : "-3", "11101111" : "-4"
}

res = []

file_in = open('input', 'r')

def ascii_Value(bin):
        try:
                ascii = matrix[bin]
        except KeyError:
                return None
        else:
                return ascii

compt_bloc = 0
line = file_in.readline()
binary = line[8:-1]+line[2:6]
ascii = ascii_Value(binary)
prev_key = ''

if ascii[0] != '-':
        prev_key = ascii
else:
        compt_bloc += 1

prev_ascii = ascii

for line in file_in:

        binary = line[8:-1]+line[2:6]
        ascii = ascii_Value(binary)

        if prev_ascii==ascii:
                continue
        elif compt_bloc==4 and prev_key!='':
                compt_bloc += 1
                res.append(prev_key)
                prev_key = ''
        elif ascii[0]=='-':
                compt_bloc += 1
        else:
                prev_key = ascii
                compt_bloc = 0

        prev_ascii = ascii


file_in.close()

print('DGSESIEE{'+"".join(res)+'}')
```

Chapitre 6

Index

6.1 Liste des figures :

[2.1] : Site internet du challenge Brigitte Friang
[2.2] : Mail envoyé à Eve Descartes
[2.3] : Code Morse capturé par Audacity
[2.4] : Plateforme de Capture The Flag
[3.1] : Structure d'un paquet de données en UART
[3.2] : Configuration d'importation du fichier
[3.3] : Extraction des données du 1er paquet
[3.4] : Tableau des connexions de la matrice 4x4 du keypad
[3.5] : Vue macro des données
[3.6] : Graphique de réponse au challenge
[3.7] : Image **hint.png** de l'archive **secrets.zip**
[3.8] : Image **message_dec.jpg**

6.2 Liste des tableaux :

[2.1] : Transcription des micro-fusibles à l'état ON et OFF en bits
[2.2] : Transcription inversé des micro-fusibles à l'état ON et OFF en bit
[2.3] : Résolution de l'inverse dans le corps de Galois
[3.1] : Tableau récapitulatif de la communication UART
[3.2] : Matrice du keypad
[3.3] : Tableau de résolution du binaire de Polyglotte

6.3 Liste des flags :

[3.1.1] : Chatbot
[3.2.1] : ASCII UART
[3.2.2] : Keypad Sniffer
[3.3.1] : Sous l'océan
[3.4.1] : Polyglotte
[3.5.1] : Définition

6.4 Liste des scripts :

[5.1] : Script de création du nuage de points
[5.2] : Script de récupération du message UART