

FIC 2021 - Challenge Naval Group

Pour le [#FIC2021](#), [Naval Group](#) a publié un challenge sur [Twitter](#) avec de nombreux lots à gagner !

Le QR code contient un lien vers la première étape du challenge :)



NAVAL GROUP

CHALLENGE CYBER

OSEREZ-VOUS RELEVER LE DÉFI ?



DE NOMBREUX LOTS À GAGNER



Un casque JBL Live 650BTNC pour la
réponse la plus rapide et pour la
réponse la plus qualitative

48 packs de goodies
Naval Group exclusifs

DÉBUT DU CHALLENGE LE 07/09 À 07H09	FIN DU CHALLENGE LE 09/09 À 09H09	ANNONCE DES RÉSULTATS ET REMISE DES LOTS LE 09/09 À 11H
--	--	---

Challenge 1.0

Rien à faire pour cette étape, on récupère juste le règlement du concours et un lien vers l'étape 1.1.

Challenge 1.1

Pour cette étape, on nous donne un petit binaire ELF 64 bits nommé **x**.

```
1 PS D:\FIC2021 Chall\ChallengeCERT#1.1> file .\x
2 .\x: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
   stripped
```

Une seule fonction intéressante nommée `_start`.

Voici le code renommé et nettoyé :

```
1 signed __int64 start()
2 {
3     int v0; // er14
4     signed __int64 v1; // rax
5     unsigned __int64 v2; // rcx
6
7     v0 = 0;
8     v1 = sys_read(0, &password, 0x48uLL);
9     v2 = 575LL;
10    do
11    {
12        if ( (((encrypted[v2 / 8] ^ password[v2 / 8]) >> (v2 % 8)) & 1) !=
13        bitstream[8 * (v2 / 8) + v2 % 8] )
14            v0 |= 1u;
15        --v2;
16    }
17    while ( v2 );
18    return sys_exit(v0);
```

Ça commence par lire le mot de passe de 72 caractères.


Puis, le flag est déchiffré avec ce mot de passe.

Chaque bit de **encrypted** est xorié avec son bit correspondant dans **password** et comparé au **bitstream**.

Le **bitstream** fait 576 bits, ce qui donne bien 8 bits par caractère.

On peut voir que la comparaison est faite du dernier caractère au premier.

Il suffit de xorer les bits d'**encrypted** avec ceux du **bitstream** pour retrouver bon **password**.

Ce qui donne en Python 

```
1 # coding=utf-8
2
3
4 def bin2str(s):
5     return "".join([chr(int(s[i:i + 8], 2)) for i in range(0, len(s), 8)])
6
7
8 def main():
9     data_400273 = open("400273", "rb").read()
```

```
10 data_40022B = open("40022B", "rb").read()
11
12 flag_bitstream = ""
13 for i in range(576)[::-1]:
14     flag_bitstream += "1" if (data_40022B[i // 8] >> (i % 8)) & 1 ^
data_400273[8 * (i // 8) + i % 8] else "0"
15
16 print(bin2str(flag_bitstream)[::-1])
17
18
19 if __name__ == '__main__':
20     main()
```

On obtient finalement l'URL vers la prochaine étape.

Tous les flags semblent être une URL [https://dropfile.naval-group.com/pfv2-sharing/sharings/\[random\].\[random\]](https://dropfile.naval-group.com/pfv2-sharing/sharings/[random].[random])

Challenge 2

Pour cette étape, nous avons un dump mémoire d'une machine infectée et nous devons trouver le hash sha256 du binaire malveillant.

Un programme **check_hash** est fourni pour vérifier le hash.

Dump mémoire

Avec [Volatility](#), on peut voir que l'image mémoire semble provenir d'un **Windows 10 x86**.

```

1 PS D:\FIC2021 Chall\ChallengeCERT#2> volatility_2.6_win64_standalone.exe
  imageinfo -f .\memory.img
2 Volatility Foundation Volatility Framework 2.6
3 INFO      : volatility.debug      : Determining profile based on KDBG search...
4           Suggested Profile(s) : win10x86_10586, win10x86, win81U1x86,
  win8SP1x86, win8SP0x86
5           AS Layer1 : IA32PagedMemoryPae (Kernel AS)
6           AS Layer2 : FileAddressSpace (D:\Torrents\Nouveau
  dossier\ChallengeCERT#2\memory.img)
7           PAE type : PAE
8           DTB : 0x1a8000L
9           KDBG : 0x82461820L
10          Number of Processors : 1
11          Image Type (Service Pack) : 0
12          KPCR for CPU 0 : 0x8248b000L
13          KUSER_SHARED_DATA : 0xffdf0000L
14          Image date and time : 2016-08-17 12:00:47 UTC+0000
15          Image local date and time : 2016-08-17 14:00:47 +0200

```

On peut maintenant lister les processus.

```

1 PS D:\FIC2021 Chall\ChallengeCERT#2> volatility_2.6_win64_standalone.exe
  pslist --profile=win10x86 -f .\memory.img
2 Volatility Foundation Volatility Framework 2.6
3 Offset(V) Name PID PPID Thds Hnds Sess Wow64
4 Start Exit
5 -----
6 0x868a7700 4 0 24...2 0 ----- 0
  2016-08-16 12:54:24 UTC+0000
7 0x8d2af5c0 `U?smss.exe 244 4 23...6 0 ----- 0
  2016-08-16 12:54:24 UTC+0000
8 0x8f7e3040 `?\?csrss.exe 324 316 33...2 0 0 0
  2016-08-16 12:54:27 UTC+0000
9 0x9487c640 388 244 24...8 ----- 1 0
  2016-08-16 12:54:28 UTC+0000 2016-08-16
10 0x8b9bf300 ▶???wininit.exe 396 316 33...6 0 0 0
  2016-08-16 12:54:28 UTC+0000
11 0x8f71d2c0 @???csrss.exe 408 388 33...2 0 1 0
  2016-08-16 12:54:28 UTC+0000
12 0x94863c40 ???winlogon.exe 460 388 33...8 0 1 0
  2016-08-16 12:54:28 UTC+0000

```

12	0x8b9bc300)?services.exe	488	396	26...0	0	0	0
	2016-08-16 12:54:29 UTC+0000						
13	0x948c3040 ?-???sass.exe	516	396	33...0	0	0	0
	2016-08-16 12:54:29 UTC+0000						
14	0x948fb180 p???svchost.exe	576	488	24...6	0	0	0
	2016-08-16 12:54:30 UTC+0000						
15	0x94954380 ?I???svchost.exe	620	488	33...8	0	0	0
	2016-08-16 12:54:30 UTC+0000						
16	0x949bdc40 0?dwms.exe	716	460	33...0	0	1	0
	2016-08-16 12:54:31 UTC+0000						
17	0x949b08c0 xV???svchost.exe	764	488	33...2	0	0	0
	2016-08-16 12:54:31 UTC+0000						
18	0x9495d6c0 ????svchost.exe	800	488	35...0	0	0	0
	2016-08-16 12:54:31 UTC+0000						
19	0x949d3040 0S???svchost.exe	848	488	33...4	0	0	0
	2016-08-16 12:54:31 UTC+0000						
20	0x949d3c40 ????svchost.exe	856	488	41...2	0	0	0
	2016-08-16 12:54:31 UTC+0000						
21	0x949faac0???svchost.exe	896	488	41...8	0	0	0
	2016-08-16 12:54:31 UTC+0000						
22	0x94ca1700 ??s?svchost.exe	1068	488	26...2	0	0	0
	2016-08-16 12:54:32 UTC+0000						
23	0x94caf040 ????svchost.exe	1132	488	24...2	0	0	0
	2016-08-16 12:54:32 UTC+0000						
24	0x9a018040 ??@?spoolsv.exe	1212	488	31...8	0	0	0
	2016-08-16 12:54:32 UTC+0000						
25	0x9a039040 @?♥?svchost.exe	1380	488	26...2	0	0	0
	2016-08-16 12:54:34 UTC+0000						
26	0x9a118380 h?◀?svchost.exe	1540	488	33...4	0	0	0
	2016-08-16 12:54:34 UTC+0000						
27	0x9a10cb00 `???wlms.exe	1572	488	33...4	0	0	0
	2016-08-16 12:54:34 UTC+0000						
28	0x9c64f980 8?-?sihost.exe	688	800	33...4	0	1	0
	2016-08-16 12:55:35 UTC+0000						
29	0x8c13ea00 x!v?taskhostw.ex	268	800	33...0	0	1	0
	2016-08-16 12:55:36 UTC+0000						
30	0x8ad6c040	1556	460	23...6	-----	1	0
	2016-08-16 12:55:36 UTC+0000	2016-08-16					
31	0x8ac4a040 ????explorer.exe	2068	1556	35...4	0	1	0
	2016-08-16 12:55:36 UTC+0000						
32	0x8ad60940 xj???RuntimeBroke	2196	576	38...2	0	1	0
	2016-08-16 12:55:37 UTC+0000						
33	0x8ad5f040 ????SkypeHost.ex	2220	576	24...2	0	1	0
	2016-08-16 12:55:37 UTC+0000						
34	0x8ad22c40 l???ShellExperie	2432	576	33...8	0	1	0
	2016-08-16 12:55:39 UTC+0000						
35	0x8b8520c0 ?w???SearchIndexe	2532	488	33...0	0	0	0
	2016-08-16 12:55:40 UTC+0000						
36	0x8b8fb8c0 hx5?OneDrive.exe	3592	2068	31...0	0	1	0
	2016-08-16 12:55:57 UTC+0000						
37	0x9c68ec40 ????fontdrvhost.	4428	460	33...4	0	1	0
	2016-08-16 12:57:09 UTC+0000						
38	0x9c728480 H?v?svchost.exe	4900	488	24...6	0	1	0
	2016-08-16 12:57:21 UTC+0000						

39	0x8ad86c40 `E??Skype.exe	5128	4696	38...8	0	1	0
	2016-08-16 12:57:42 UTC+0000						
40	0x8c0c9240 ???TrustedInsta	6108	488	24...4	0	0	0
	2016-08-16 12:58:24 UTC+0000						
41	0x8c0ba9c0 0?@?Tiworker.exe	6140	576	24...2	0	0	0
	2016-08-16 12:58:25 UTC+0000						
42	0x9d489780 ?????SystemSettin	2144	576	26...4	0	1	0
	2016-08-16 12:59:36 UTC+0000						
43	0x9499ac40 ?i??ApplicationF	1696	576	24...4	0	1	0
	2016-08-16 12:59:48 UTC+0000						
44	0x9c629300 ??h?SystemSettin	5268	5252	33...6	0	1	0
	2016-08-16 12:59:51 UTC+0000						
45	0xb0d47780 ?o??svchost.exe	4888	4748	33...4	0	1	0
	2016-08-16 13:02:57 UTC+0000						
46	0x9d5e74c0 ???explorer.exe	4872	4748	33...8	0	1	0
	2016-08-16 13:02:58 UTC+0000						
47	0x9c7d7c40 ??*?svchost.exe	2168	5860	33...0	0	1	0
	2016-08-16 13:03:04 UTC+0000						
48	0xb0c96740 ?-[?update.exe	5172	5860	24...2	0	1	0
	2016-08-16 13:03:04 UTC+0000						
49	0xd0d9f600	1976	5172	35...6	-----	1	0
	2016-08-16 13:04:47 UTC+0000	2016-08-16					
50	0x9d5ba900	736	5172	35...6	-----	1	0
	2016-08-16 13:07:40 UTC+0000	2016-08-16					
51	0xbac89640 p???SystemSettin	4968	576	23...2	0	1	0
	2016-08-16 13:41:14 UTC+0000						
52	0xbad4b040	2748	5172	31...2	-----	1	0
	2016-08-16 13:50:51 UTC+0000	2016-08-16					
53	0xbf755c40	5280	5172	41...4	-----	1	0
	2016-08-16 14:17:24 UTC+0000	2016-08-16					
54	0x8b8c44c0	868	5172	35...2	-----	1	0
	2016-08-16 14:19:45 UTC+0000	2016-08-16					
55	0xd53d2c40	3540	5172	35...6	-----	1	0
	2016-08-16 14:23:05 UTC+0000	2016-08-16					
56	0xd5321480 ???SearchUI.exe	7360	576	31...6	0	1	0
	2016-08-16 18:13:21 UTC+0000						
57	0x9c6a8040 H ??audiodg.exe	18084	848	26...4	0	0	0
	2016-08-17 12:00:20 UTC+0000						
58	0xc8606c40 ??RamCapture.e	16740	2068	26...6	0	1	0
	2016-08-17 12:00:36 UTC+0000						
59	0xd53a3500 ?6o?conhost.exe	16756	16740	33...6	0	1	0
	2016-08-17 12:00:36 UTC+0000						
60	0x9c61a300 8)??SearchProtoc	15756	2532	33...4	0	-----	0
	2016-08-17 12:00:50 UTC+0000						
61	0xc7fa2a40 ?3p?SearchFilter	14288	2532	33...4	0	-----	0
	2016-08-17 12:00:50 UTC+0000						
62	0xe2df3040 ?\??MusNotificat	16968	800	25...4	0	-----	0
	2016-08-18 09:25:38 UTC+0000						
63	0xb0df7040	0	0	29...0	-----	-----	0

Binaire malveillant

Tout n'est pas très clair, mais le seul binaire avec un nom un peu suspect semble être **update.exe**. De toute façon, on pourrait très bien dumper tous les processus et essayer chaque hash :)

```

1 PS D:\FIC2021 Cha11\ChallengeCERT#2> volatility_2.6_win64_standalone.exe
   procdump --profile=win10x86 -f .\memory.img -D .\ -p 5172
2 Volatility Foundation Volatility Framework 2.6
3 Process(V) ImageBase Name Result
4 -----
5 0xb0c96740 0x01610000 ?-[?update.exe OK: executable.5172.exe

```

Puis on calcule le hash sha256.

```

1 PS D:\FIC2021 Cha11\ChallengeCERT#2> sha256sum.exe .\executable.5172.exe
2 \166d8eb95ac704b6dc2bad8ffa8fb492e84fde52801a3dec551cc79e9c644e50
   *.\\executable.5172.exe

```

Et on passe sous Linux (dans Windows ♡) pour vérifier le hash.

```

1 PS D:\FIC2021 Cha11\ChallengeCERT#2> bash
2 debian@osef:/mnt/ChallengeCERT#2$ ./check_hash
3 Please input the SHA256 hash:
4 166d8eb95ac704b6dc2bad8ffa8fb492e84fde52801a3dec551cc79e9c644e50
5 -----
6 well done !
7 -----

```

Bonus

Le binaire est en fait [Xtreme RAT](#) et il est fait en [Delphi](#) ! Mais alors il y a une autre personne qui aime Delphi en 2021 ? 😊

On peut aller voir le `timestamp` du binaire et on trouve `2A425E19` ce qui correspond au `1992-06-19 22:22:17` ... un peu étrange parce que l'aventure Delphi a commencé vers 1995 😊.

En fait, il semblerait que les anciens compilateurs Delphi (4 à 2006) aient un "bug" où le `timestamp` soit une valeur fixe préconfigurée au lieu de la date de compilation 🙄. On a donc tout un tas de binaire avec un `timestamp` à `2A425E19` et il suffit de chercher sur [Google](#) pour voir pas mal de ces binaires dans des rapports de crash.

Mais alors il date de quand ce binaire ? 🤖

Pour ce binaire, il y a d'autres timestamps dans la partie des ressources (`_IMAGE_RESOURCE_DIRECTORY`). On trouve la valeur `446B4C0F` qui correspond au `2006-05-17 16:15:11`.

Plus d'infos sur : https://0xc0decafe.com/malware-analyst-guide-to-pe-timestamps/#Delphi_timestamps

Challenge 3

On passe maintenant à un reverse Windows avec un fichier nommé **cell.exe**.

(J'ai renommé quelques fonctions pour que ce soit plus simple)

Un petit test rapide pour voir ce qu'il fait.

```
1 PS D:\FIC2021 Cha11\ChallengeCERT#3> .\cell.exe
2 Enter the password:
3 123456
4 Fail!
```

Alors peut-être avec une autre URL.

```
1 PS D:\FIC2021 Cha11\ChallengeCERT#3> .\cell.exe
2 Enter the password:
3 https://dropfile.naval-group.com/pfv2-sharing/sharings/aaaaaaaa.aaaaaaa
4 b13220621fd3a7f32e2f163df5c01cbb06f7d8e4951ac01d9cb650b70272396fcb8f6c422be5c7
5 f729420051df07c5b2c960294ffb0a5514b4be071790b82f451830663ffdaa16b8
6 54108f28cfe1c3f116974a9efadfcc5d737bea600a4d5fce8c5a265b783098b7c5a7b51895e2d3
7 fb1098ff24ef73d2d140af9087fd642a48585ef34b865b97924b97229ffed4cb5a
8 29c6271247ecd9f44b0b850f7d6fc52eb1bdf42fe406afc7252c822dbb960a5bd2c3da4a0ae961
9 fd460a7f0077b1e1649e578633fea194032b2f7185b22d8b80058b090fff6845ac
10 90d2834013f468f90565b267beb7d21754def997e1f357d302143896dd8ae42de159ec04e570ac
11 fe92e43f7f3bd4eca00e2bb291ff4c89f91517b4b2d096a5bffa2a56067ffb312d5
12 8661399fc1f9327872a2d023df5be0cba86f788becf1abe178c99a0b6ea56196eca8f5f062b654
13 7f01619fbf1de8744fe695d104ff8420f84a4bd851660b42dff142af23ffd1416a
14 322c188fdcf8103b31496789efadee45d337ba25f474d5ecba408ce5b742ac8b74527af7295a29
15 3f7cac8fdf4ef33907e30ae4707fb18e7b0405eb24a2e5996ff4995709ffe49cb5
16 Fail!
```

C'est déjà un peu mieux.

On peut aller voir le code qui fait ça (*sub_401B9D*).

```
1 int __cdecl sub_401B9D(char a1)
2 {
3     FILE *Stream; // eax
4     DWORD v3; // eax
5     CHAR Name[30]; // [esp+1Fh] [ebp-449h] BYREF
6     char state[515]; // [esp+3Dh] [ebp-42Bh] BYREF
7     char Str[500]; // [esp+240h] [ebp-228h] BYREF
8     BOOL v7; // [esp+434h] [ebp-34h]
9     HANDLE hHandle; // [esp+438h] [ebp-30h]
10    int m; // [esp+43Ch] [ebp-2Ch]
11    char v10[4]; // [esp+440h] [ebp-28h]
12    int k; // [esp+444h] [ebp-24h]
13    int j; // [esp+448h] [ebp-20h]
14    int i; // [esp+44Ch] [ebp-1Ch]
15    char *v14; // [esp+458h] [ebp-10h]
16
17    v14 = &a1;
```

```

18 sub_401F60();
19 memset(&state[15], 0, 500);
20 puts("Enter the password:");
21 fgets(Str, 500, stdin);
22 Str[strcspn(Str, "\n")] = 0;
23 if ( strlen(Str) == 72 )
24 {
25     for ( i = 0; i <= 71; ++i )
26         Str[i] ^= key1[i];
27     for ( j = 0; j <= 71; ++j )
28         printf("%02x", Str[j]);
29     putchar(10);
30     initEvent(Str);
31     for ( k = 0; k <= 4; ++k )
32     {
33         memset(&state[15], 0, 0x48u);
34         transformPass();
35         *v10 = 0;
36         while ( *v10 <= 575 )
37         {
38             strcpy(state, "Global\\cell_%d");
39             printflike(Name, 0x1Eu, state, v10[0]);
40             hHandle = OpenEventA(0x1F0003u, 0, Name);
41             v3 = WaitForSingleObject(hHandle, 0xAu);
42             if ( v3 != WAIT_TIMEOUT )
43                 state[*v10 / 8 + 15] |= 1 << (7 - *v10 % 8);
44             ++*v10;
45         }
46         for ( m = 0; m <= 71; ++m )
47             printf("%02x", state[m + 15]);
48         putchar(10);
49     }
50     if ( !memcmp(&unk_404080, &state[15], 0x48u) )
51         puts("Congratulations!");
52     else
53         puts("Fail!");
54     return 0;
55 }
56 else
57 {
58     puts("Fail!");
59     return 0;
60 }
61 }

```

Ça commence par demander le mot de passe et le stocke à **&state[15]**.

```

1  memset(&state[15], 0, 500);
2  puts("Enter the password:");
3  fgets(Str, 500, stdin);

```

Ensuite, on vérifie que le mot de passe fait bien 72 caractères.

```

1  if ( strlen(Str) == 72 )
2  {
3  ...
4  }
5  else
6  {
7      puts("Fail!");
8      return 0;
9  }

```

Première transformation

Si la taille est bonne, le mot de passe est xorré avec une clé située @00404020 (key1).

```

1  D9 46 54 12 6C E9 88 DC 4A 5D 79 4D 93 A9 70 DE
2  28 99 B9 92 F4 76 ED 7A EE D9 25 C7 2C 11 56 02
3  E4 FF 0A 34 19 C8 B4 9F 48 30 69 3F B8 28 B6 DA
4  A8 12 40 21 9C 79 7A 75 D5 DF 66 76 F1 D9 4E 6B
5  79 51 07 5E 9C CB 77 D9

```

On a bien une clé de 72 caractères.

Une fois xorrée, le résultat est affiché. Dans les tests fais plus haut, cela correspond à :

```

1  b13220621fd3a7f32e2f163df5c01cbb06f7d8e4951ac01d9cb650b70272396fcb8f6c422be5c7f
   729420051df07c5b2c960294ffb0a5514b4be071790b82f451830663ffdaa16b8

```

Boucle de transformations

Maintenant, ça se complique !

On voit bien les 5 boucles qui vont transformer et afficher notre mot de passe, mais difficile de dire comment comme ça.

En tout cas, à la fin, on vérifie que le résultat correspond bien à celui attendu qui est stocké à @404080.

```

1  if ( !memcmp(&unk_404080, &state[15], 0x48u) )
2      puts("Congratulations!");
3  else
4      puts("Fail!");

```

Revenons à notre boucle.

Initialisation

On commence par la fonction @401564 (initEvents).

```

1  int __cdecl initEvents(int a1)
2  {
3      CHAR Name[30]; // [esp+1Bh] [ebp-4Dh] BYREF
4      char v3[16]; // [esp+39h] [ebp-2Fh] BYREF
5      char bInitialState[19]; // [esp+49h] [ebp-1Fh] BYREF

```

```

6   char v5[4]; // [esp+5Ch] [ebp-Ch]
7
8   strcpy(bInitialState, "Global\\cell_%d");
9   strcpy(v3, "Global\\ncell_%d");
10  *v5 = 0;
11  while ( *v5 <= 575 )
12  {
13      printflike(Name, 0x1Eu, bInitialState, v5[0]);
14      *&bInitialState[15] = ((*v5 / 8 + a1) >> (7 - *v5 % 8)) & 1;
15      CreateEventA(0, 1, *&bInitialState[15], Name);
16      printflike(Name, 0x1Eu, v3, v5[0]);
17      CreateEventA(0, 1, 0, Name);
18      ++*v5;
19  }
20  return 0;
21  }

```

On retrouve une boucle de 576 tours. Apparemment, il y aura encore de la comparaison bit à bit 🤔.

Pour chaque bit de **password**, on va créer un Event nommé **Global\cell_x** avec ce bit en paramètre.

```

1  *&bInitialState[15] = ((*v5 / 8 + a1) >> (7 - *v5 % 8)) & 1;

```

Quant à l'autre Event, **Global\ncell_0**, il est créé, mais avec un paramètre nul (0).

Au final, cette fonction va donc créer 576 Events nommés **Global\cell_0**, **Global\cell_1**, ... avec les bits de

password, puis 576 Events "vides" nommés **Global\ncell_0**, **Global\ncell_1**, ...

Les **Global\ncell_x** serviront à stocker le résultat d'une operation/transformation.

Transformation

On revient à la fonction précédente avec nos 5 boucles.

```

1  for ( k = 0; k <= 4; ++k )
2  {
3      memset(&state[15], 0, 0x48u);
4      transformPass();
5      *v10 = 0;
6      while ( *v10 <= 575 )
7      {
8          strcpy(state, "Global\\cell_%d");
9          printflike(Name, 0x1Eu, state, v10[0]);
10         hHandle = OpenEventA(0x1F0003u, 0, Name);
11         v3 = WaitForSingleObject(hHandle, 0xAu);
12         if ( v3 != WAIT_TIMEOUT )
13             state[*v10 / 8 + 15] |= 1 << (7 - *v10 % 8);
14         ++*v10;
15     }
16     for ( m = 0; m <= 71; ++m )
17         printf("%02x", state[m + 15]);
18     putchar(10);
19 }

```

A chaque boucle, on va transformer le **password**.

Puis on récupère les nouveaux bits depuis **Global\cell_x** en faisant un **OpenEvent** et on met à jour le bit correspondant de **state**.

À chaque boucle, on affiche le nouvel état.

```

1  54108f28cfe1c3f116974a9efadfcc5d737bea600a4d5fce8c5a265b783098b7c5a7b51895e2d3f
   b1098ff24ef73d2d140af9087fd642a48585ef34b865b97924b97229ffed4cb5a
2  29c6271247ecd9f44b0b850f7d6fc52eb1bdf42fe406afc7252c822dbb960a5bd2c3da4a0ae961f
   d460a7f0077b1e1649e578633fea194032b2f7185b22d8b80058b090fff6845ac
3  90d2834013f468f90565b267beb7d21754def997e1f357d302143896dd8ae42de159ec04e570acf
   e92e43f7f3bd4eca00e2bb291ff4c89f91517b4b2d096a5bff2a56067ffb312d5
4  8661399fc1f9327872a2d023df5be0cba86f788becf1abe178c99a0b6ea56196eca8f5f062b6547
   f01619fbf1de8744fe695d104ff8420f84a4bd851660b42dff142af23ffd1416a
5  322c188fdcf8103b31496789efadee45d337ba25f474d5ecba408ce5b742ac8b74527af7295a293
   f7cac8fdf4ef33907e30ae4707fb18e7b0405eb24a2e5996ff4995709ffe49cb5

```

Il faut que le dernier état soit égal à celui harcodé en @404080.

```

1  f22c188fdcf8103b31496789efadee45d337ba25f474d5ecba408ce5b742ac8b74527af7295a293
   f7cac8fdf4ef33907e30ae4707fb18e2596de344bc8befc1518c8b2897f61096a

```

On peut voir qu'une bonne partie correspond déjà. Mais ni la fin (normal) et ni le premier octet 🤔

On verra après pourquoi ...

Event et bits

Sans trop rentrer dans les détails, un Event a un nom et deux états possibles.

Les Events sont un moyen de faire de la synchronisation. Ils sont même partagés entre les processus.

Un processus pour créer un Event et attendre cet Event pour lire quelque chose. Et un autre processus peut écrire quelque part, puis faire un **SetEvent** pour signaler à l'autre processus qu'il peut commencer à lire.

Plus d'infos ici: <https://docs.microsoft.com/en-us/windows/win32/sync/event-objects>

On le crée avec **CreateEvent** et on modifie son état avec **SetEvent** pour le mettre à "Vrai" ou **ResetEvent** pour l'état "Faux".

Pour connaître l'état d'un Event, on peut utiliser **WaitForSingleObject** et regarder le résultat. Si on a **WAIT_TIMEOUT**, l'Event n'est toujours pas passé à "Vrai" sinon ... il est passé à "Vrai".

Je n'ai pas trouvé de meilleure traduction avec les "Vrai" et "Faux". Mais dans la doc Microsoft, c'est "*signaled*" et "*non-signaled*".

Bref, il y a deux états!

On revient au challenge et on passe à la transformation en elle-même :

```

1  int transformPass()
2  {

```

```

3  HANDLE Thread; // eax
4  CHAR Name[30]; // [esp+23h] [ebp-965h] BYREF
5  char v3[16]; // [esp+41h] [ebp-947h] BYREF
6  char lpThreadId[2323]; // [esp+51h] [ebp-937h] BYREF
7  HANDLE hEvent; // [esp+964h] [ebp-24h]
8  BOOL v6; // [esp+968h] [ebp-20h]
9  HANDLE hHandle; // [esp+96Ch] [ebp-1Ch]
10 char v8[4]; // [esp+970h] [ebp-18h]
11 LPVOID lpParameter; // [esp+974h] [ebp-14h]
12 DWORD nCount; // [esp+978h] [ebp-10h]
13 int i; // [esp+97Ch] [ebp-Ch]
14
15 for ( i = 0; ; ++i )
16 {
17     nCount = 0;
18     for ( lpParameter = (i << 6); lpParameter < sub_40190B(576, (i + 1) << 6);
19         lpParameter = lpParameter + 1 )
20     {
21         ++nCount;
22         Thread = CreateThread(0, 0, StartAddress, lpParameter, 0,
23             &lpThreadId[15]);
24         *&lpThreadId[4 * lpParameter + 19] = Thread;
25         if ( !*&lpThreadId[4 * lpParameter + 19] )
26         {
27             printf("Can't create thread %d\n", lpParameter);
28             return 0;
29         }
30         WaitForMultipleObjects(nCount, &lpThreadId[256 * i + 19], 1, 0xFFFFFFFF);
31         if ( sub_40190B((i + 1) << 6, 576) == 576 )
32             break;
33     }
34     *v8 = 0;
35     while ( *v8 <= 575 )
36     {
37         strcpy(lpThreadId, "Global\\cell_%d");
38         strcpy(v3, "Global\\ncell_%d");
39         printflike(Name, 0x1Eu, v3, v8[0]);
40         hHandle = OpenEventA(0x1F0003u, 0, Name);
41         v6 = WaitForSingleObject(hHandle, 0xAu) != 258;
42         printflike(Name, 0x1Eu, lpThreadId, v8[0]);
43         hEvent = OpenEventA(0x1F0003u, 0, Name);
44         if ( v6 )
45             SetEvent(hEvent);
46         else
47             ResetEvent(hEvent);
48         ++*v8;
49     }
50     return 0;
51 }

```

La première boucle va calculer le prochain état bit à bit par bloc de 64.

Pourquoi 64 ? A priori c'est juste pour éviter de créer d'un coup 576 threads 🤖

Chaque calcul sera fait dans un thread. À la fin de la boucle, on attend que chaque thread soit terminé avec **WaitForSingleObject** comme pour les Events.

Ensuite, pour chaque bit, on récupère son état depuis **Global\ncell_x** et on modifie **Global\cell_x** en conséquence.

On arrive finalement au calcul des nouveaux bits, la seule partie vraiment utile.

```

1  DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
2  {
3      unsigned int v1; // edx
4      CHAR Name[30]; // [esp+1Fh] [ebp-69h] BYREF
5      char v4[16]; // [esp+3Dh] [ebp-4Bh] BYREF
6      char v5[19]; // [esp+4Dh] [ebp-3Bh] BYREF
7      int v6; // [esp+60h] [ebp-28h]
8      BOOL v7; // [esp+64h] [ebp-24h]
9      BOOL v8; // [esp+68h] [ebp-20h]
10     BOOL v9; // [esp+6Ch] [ebp-1Ch]
11     HANDLE v10; // [esp+70h] [ebp-18h]
12     HANDLE v11; // [esp+74h] [ebp-14h]
13     HANDLE hHandle; // [esp+78h] [ebp-10h]
14     char v13[4]; // [esp+7Ch] [ebp-Ch]
15
16     strcpy(v5, "Global\\cell_%d");
17     strcpy(v4, "Global\\ncell_%d");
18     *v13 = lpThreadParameter;
19     printflike(Name, 0x1Eu, v5, lpThreadParameter);
20     hHandle = OpenEventA(0x1F0003u, 0, Name);
21     if ( *v13 )
22         v1 = (*v13 - 1) % 0x240u;
23     else
24         LOBYTE(v1) = 63;
25     printflike(Name, 0x1Eu, v5, v1);
26     v11 = OpenEventA(0x1F0003u, 0, Name);
27     printflike(Name, 0x1Eu, v5, (*v13 + 1) % 0x240u);
28     v10 = OpenEventA(0x1F0003u, 0, Name);
29     v9 = WaitForSingleObject(hHandle, 0xAu) != 258;
30     v8 = WaitForSingleObject(v11, 0xAu) != 258;
31     v7 = WaitForSingleObject(v10, 0xAu) != 258;
32     v6 = v8 ^ (!v9 && !v7);
33     v6 = v6 != 0;
34     printflike(Name, 0x1Eu, v4, v13[0]);
35     *&v5[15] = OpenEventA(0x1F0003u, 0, Name);
36     if ( v6 )
37         SetEvent(*&v5[15]);
38     else
39         ResetEvent(*&v5[15]);
40     return 0;
41 }

```

Ca lit les bits depuis **Global\cell_x**, calcule le prochain bit puis met à jour **Global\ncell_x**.

Le plus important, c'est cette ligne :

```
1 | v6 = v8 ^ (!v9 && !v7);
```

- **v6** correspond au prochain état du bit courant.
- **v8** correspond au bit précédent. Et si on est au bit 0, cela correspond au dernier bit, le 575.
- **v9** correspond au bit courant.
- **v7** correspond au bit suivant.

Pour résumer, cela donne :

$$\text{next_bit}_x = \text{bit}_{x-1} \text{ xor } (!\text{bit}_x \ \&\& \ !\text{bit}_{x+1})$$

Et c'est tout ce qu'il nous faut !

Le reste, c'est juste de la copie de bits entre Events, ...

D'ailleurs tous ces threads et Events, ça prend du temps. Il faut environ 15 secondes pour tester un mot de passe sur mon PC. Et si on voulait bruteforcer, ça ne pourrait pas marcher car tout les Events sont partagés, lancer plusieurs processus en même temps ne servirait à rien.

Par contre, on pourrait créer plusieurs binaires avec des noms d'Event différents en changeant un caractère ou plus.

Mais comme dans le Python tout est bon ou presque, on passe au Python.

Voilà la transformation:

```
1 | def transformation(b):
2 |     res = []
3 |     for i in range(576):
4 |         bit_prev = b[(i - 1) % 576 if i != 0 else 575]
5 |         bit_cur = b[i]
6 |         bit_next = b[(i + 1) % 576]
7 |         res.append(bit_prev ^ ((bit_cur == 0) and (bit_next == 0)))
8 |     return res
```

Il y a peut-être un moyen de reverser cette transformation, mais un peu de bruteforce c'est bien aussi 😊.

Solution

Pour chaque indice, on teste différents caractères et on regarde combien d'octets à la fin sont bons en comparant avec le résultat attendu.

Avant de bruteforcer, il faut trouver le dernier caractère du flag.

Comme on prend le bit précédent, le premier octet final dépend aussi du dernier octet du mot de passe.


```

1 def find_last_char():
2     best = (0, None)
3     for c1 in charset:
4         s1[-1] = c1
5         r = score(test(xor(s1, key)), reference)
6         if r >= best[0]:
7             best = (r, c1)
8     print("Last char is", best[1]) # I
9     s1[-1] = best[1]
10    print("".join(map(chr, s1)))

```

On trouve que le dernier caractère est I.

Maintenant on peut bruteforcer chaque caractère en testant les deux suivants, au cas où on ait besoin du bit du caractère suivant.

```

1 i = 70
2 best = [0, set()]
3 for c1 in charset:
4     s1[i] = c1
5     for c2 in charset:
6         s1[i + 1] = c2
7         r = score(test(xor(s1, key)), reference)
8         if r > best[0]:
9             best = [r, {c1}]
10        elif r == best[0]:
11            best[1].add(c1)
12    print("Best char for", i, "is", list(map(chr, best[1])))

```

Cette fonction teste toutes les combinaisons de 2 caractères et garde les meilleurs résultats.

On trouve généralement un ou deux caractères possibles à chaque fois. Mais en testant les suivants, on voit qu'il n'y a, en fin de compte, qu'un seul caractère possible.

Ça prend un peu de temps de faire les 16 caractères qui nous manquent pour l'URL, mais on arrive facilement à la solution !

Il y a sûrement une solution plus optimisée et ça devrait pouvoir aussi se faire avec des outils comme [z3](#).

Remarque

Vu les URLs précédentes, le charset est a priori `[A-Za-z0-9]`.

Mais on ne trouve plus de solution possible au bout d'un moment.

En fait, il faut rajouter `-` au charset. Un petit détail, mais il faut y penser 🤖.

On peut passer au prochain (et dernier) challenge.

Challenge 4

Pour celui-là, on a un binaire ARM avec les bibliothèques qu'il faut pour le lancer.

```
1 PS D:\FIC2021 Chall\ChallengeCERT#4> file .\AT-AT.bin
2 .\AT-AT.bin: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.3,
BuildID[sha1]=9475d7b94edd198234e7be148c8acaec728987f5, for GNU/Linux 3.2.0,
stripped
```

Pour le lancer, on peut utiliser `qemu-arm-static`.

```
1 debian@debian:/mnt/d/FIC2021 Chall/ChallengeCERT#4$ qemu-arm-static ./AT-AT.bin
2 welcome to ATATATATATATATAT
3 Please enter the password to decrypt the secret plans
4 Password: secretpassword
5
6 Invalid password
7 Decryption failure:
8 [random chars]
```

Ça lit un mot de passe et déchiffre les plans secrets, peu importe le mot de passe, puis les affiche.

Le binaire n'est pas petit (526 Ko) mais n'a pas l'air d'avoir beaucoup de code 🤖.

Machine virtuelle

Voilà le `main` après renommage.

```
1 void __fastcall __noreturn main(int a1, char **a2, char **a3)
2 {
3     char *v3; // r10
4     int v4; // r2
5     int v5; // r3
6     __int64 v6; // [sp-28h] [bp-44h]
7     int v7; // [sp-20h] [bp-3ch]
8     int v8; // [sp-1ch] [bp-38h]
9     int instrOffset; // [sp+10h] [bp-ch]
10
11     instrOffset = 5482;
12     printf("welcome to %s\n", instr);
13     mprotect(instr, 0x1000u, 7);
14     v6 = 0LL;
15     v7 = 0;
16     v8 = 0;
17     while ( 1 )
18     {
19         do
20             instrOffset = offset(instrOffset);
21         while ( check_exists(instrOffset >> 1) );
22         usedOffset[usedOffsetIdx++] = instrOffset >> 1;
```

```

23     decryptInstr(instr, &unk_23000 + 16 * (instrOffset >> 1),
    HIWORD(instrOffset));
24     v3 = instr;
25     if ( (instrOffset & 1) != 0 )
26         v3 = &instr[1];
27     v6 = (v3)(v6, HIWORD(v6), v7, v8);
28     v7 = v4;
29     v8 = v5;
30     strcpy(instr, src);
31 }
32 }

```

Nous avons une boucle qui va calculer un offset, déchiffrer quelque chose avec les données de `unk_23000` et l'exécuter ... une VM !

Dump du code

Pour dumper le code de la VM, on peut utiliser `gdb`, mettre un breakpoint à 22000 (v3) et afficher les instructions.

Par exemple, avec le script `gdb` suivant :

```

1  set logging on
2  b *0x22000
3  commands
4  x/2i 0x22000
5  end

```

Seule la première instruction est vraiment utile, la seconde, c'est presque toujours `bx 1r` pour revenir au `main` ou encore la modification du registre `r3` après une comparaison.

Sinon on peut récupérer les instructions en recodant le décodeur avec un script en Python.

Pour les décoder correctement, il faut voir que suivant le dernier bit de l'offset, ça switch en mode [Thumb](#). Mais c'est fait exprès 😊

Après, il reste à coder un peu.

```

1  # coding=utf-8
2  import binascii
3  import string
4
5  from capstone import *
6
7  def _ror(val, bits, bit_size):
8      return ((val & (2 ** bit_size - 1)) >> bits % bit_size) | \
9              (val << (bit_size - (bits % bit_size)) & (2 ** bit_size - 1))
10
11  __ROR4__ = lambda val, bits: _ror(val, bits, 32)
12
13  HIBYTE = lambda val: (val & 0xff00) >> 8
14  HIWORD = lambda val: (val & 0xffff0000) >> 16
15

```

```

16 buf_base = list(b"ATATATATATATAT")
17 data_23000 = list(open("data_23000", "rb").read())
18
19 def nextint(i):
20     return __ROR4__(i + 1337, 25) ^ 0xDEADBEEF
21
22 def decrypt(base, key, flag):
23     for i in range(15):
24         if i & 1:
25             base[i] = HI BYTE(flag) ^ key[i]
26         else:
27             base[i] = (flag & 0xff) ^ key[i]
28
29 def disasm(f, mode=CS_MODE_THUMB):
30     md = Cs(CS_ARCH_ARM, CS_MODE_ARM | mode)
31     code = binascii.unhexlify(f)
32     for i in md.disasm(code, 0x22000, 2):
33         if i.mnemonic == "bx" and i.op_str == "lr":
34             break
35         print("%-8s\t\t%s\t%s" % (binascii.hexlify(i.bytes).decode(),
36 i.mnemonic, i.op_str))
37
38 def main():
39     instr = 0x156a
40     tab = set()
41     while True:
42         while True:
43             instr = nextint(instr)
44             offset = (instr & 0xffff) >> 1
45             if offset not in tab:
46                 tab.add(offset)
47                 break
48
49     buf = buf_base[:]
50     decrypt(buf, data_23000[offset * 16:], HIWORD(instr))
51     code = "".join(list(map(lambda x: "%02x" % x, buf)))
52     if code[:4] == "fecc": # End of code
53         break
54     if (instr & 1) != 0:
55         disasm(code, CS_MODE_THUMB)
56     else:
57         disasm(code, CS_MODE_ARM)
58
59 if __name__ == '__main__':
60     main()

```

On récupère une liste d'environ 2000 instructions.

Analyse du code

Après analyse, il y a 4 parties dans ce code. On peut les trouver facilement en cherchant où se trouvent les modifications

du registre `r7` qui contrôle le syscall à appeler.

- On écrit le message *"Please enter the password to decrypt the secret plans"* sur Stdout.

```

1  # write challenge message
2
3  4ff00100      mov.w   r0, #1           # stdout
4  0120a0e3      mov    r2, #1           # 1 byte to write
5  0470a0e3      mov    r7, #4           # WRITE syscall
6  06f15001      add.w   r1, r6, #0x50    # Please enter the password to
decrypt the secret plans
7  000000ef      svc    #0
8  06f16c01      add.w   r1, r6, #0x6c
9  00df          svc    #0
10 ...

```

- On lit le password (42 caractères)

```

1  # Read 42 chars (password) to r4 = @A3004
2
3  0000a0e3      mov    r0, #0
4  0410a0e1      mov    r1, r4
5  2a20a0e3      mov    r2, #0x2a
6  4ff00307      mov.w   r7, #3           # READ syscall
7  00df          svc    #0
8
9  # Set last char to '\0'
10
11 000020e0      eor    r0, r0, r0
12 84f82900      strb.w  r0, [r4, #0x29]

```

- Vérification du password.

Une erreur met un bit de `r3` à 1. À la fin, suivant ce bit, le bon message est affiché ou pas.

```

1  4ff00100      mov.w   r0, #1
2  55f82310      ldr.w   r1, [r5, r3, lsl #2]
3  4ff02e02      mov.w   r2, #0x2e
4  0470a0e3      mov    r7, #4           # WRITE syscall
5  000000ef      svc    #0

```

Revenons à la vérification.

Pour chaque test, j'ai mis la signification avant.

- `pdw[x] => password[x:x+4]`
- `pw[x] => password[x:x+2]`
- `pb[x] => password[x]`

Le code:

```

1  # pdw[0] == pdw[10]
2  2246          mov    r2, r4
3  000092e5      ldr    r0, [r2]
4  d2f80a10      ldr.w   r1, [r2, #0xa]
5  010050e1      cmp    r0, r1
6  01308313      orrne   r3, r3, #1
7

```

```

8  # pdw[4] == pdw[14]
9  02f10402      add.w   r2, r2, #4
10 1068          ldr  r0, [r2]
11 0a1092e5      ldr  r1, [r2, #0xa]
12 010050e1      cmp  r0, r1
13 01308313      orrne  r3, r3, #1
14
15 # pw[8] == pw[18]
16 02f10402      add.w   r2, r2, #4
17 1088          ldrh   r0, [r2]
18 ba10d2e1      ldrh   r1, [r2, #0xa]
19 010050e1      cmp  r0, r1
20 01308313      orrne  r3, r3, #1
21
22 # pdw[28] & unk_21050[20] = unk_21050[12]
23 1c0094e5      ldr  r0, [r4, #0x1c]
24 d9f81410      ldr.w  r1, [sb, #0x14]
25 00ea0100      and.w  r0, r0, r1
26 0c2099e5      ldr  r2, [sb, #0xc]
27 020050e1      cmp  r0, r2
28 01308313      orrne  r3, r3, #1
29
30 # (((pw[5]*2)+pw[5])*2)+pw[5] = unk_21050[32]
31 b4f80500      ldrh.w r0, [r4, #5]
32 0146          mov  r1, r0
33 0944          add  r1, r1
34 001081e0      add  r1, r1, r0
35 0944          add  r1, r1
36 001081e0      add  r1, r1, r0
37 202099e5      ldr  r2, [sb, #0x20]
38 020051e1      cmp  r1, r2
39 01308313      orrne  r3, r3, #1
40
41 # pb[40] == pb[39]
42 282084e2      add  r2, r4, #0x28
43 0000d2e5      ldrb   r0, [r2]
44 a2f10102      sub.w  r2, r2, #1
45 0010d2e5      ldrb   r1, [r2]
46 010050e1      cmp  r0, r1
47 01308313      orrne  r3, r3, #1
48
49 # pb[40] == pb[38]
50 012042e2      sub  r2, r2, #1
51 0010d2e5      ldrb   r1, [r2]
52 010050e1      cmp  r0, r1
53 01308313      orrne  r3, r3, #1
54
55 # pdw[20]+pdw[24] = unk_21050[16]
56 6069          ldr  r0, [r4, #0x14]
57 181094e5      ldr  r1, [r4, #0x18]
58 010080e0      add  r0, r0, r1
59 102099e5      ldr  r2, [sb, #0x10]
60 020050e1      cmp  r0, r2
61 01308313      orrne  r3, r3, #1
62

```

```

63 # pdw[7] xor 0xffffffff = unk_21050[8]
64 d700c4e1      ldrd    r0, r1, [r4, #7]
65 0020a0e3      mov    r2, #0
66 012042e2      sub    r2, r2, #1
67 80ea0200      eor.w   r0, r0, r2
68 021021e0      eor    r1, r1, r2
69 082099e5      ldr    r2, [sb, #8]
70 020050e1      cmp    r0, r2
71 01308313      orrne   r3, r3, #1
72
73 # pdw[11] xor 0xffffffff = unk_21050[0]
74 d9f80020      ldr.w   r2, [sb]
75 020051e1      cmp    r1, r2
76 01308313      orrne   r3, r3, #1
77
78 # pb[32] == pb[31]+36
79 1f2084e2      add    r2, r4, #0x1f
80 1178          ldrb    r1, [r2]
81 01f12401      add.w   r1, r1, #0x24
82 02f10102      add.w   r2, r2, #1
83 1078          ldrb    r0, [r2]
84 010040e0      sub    r0, r0, r1
85 003083e1      orr     r3, r3, r0
86
87 # pb[33] == pb[31]+36+10
88 0a1081e2      add    r1, r1, #0xa
89 012082e2      add    r2, r2, #1
90 0000d2e5      ldrb    r0, [r2]
91 010040e0      sub    r0, r0, r1
92 003083e1      orr     r3, r3, r0
93
94 # pb[34] == pb[31]+36+10-59
95 a1f13b01      sub.w   r1, r1, #0x3b
96 02f10102      add.w   r2, r2, #1
97 0000d2e5      ldrb    r0, [r2]
98 010040e0      sub    r0, r0, r1
99 43ea0003      orr.w   r3, r3, r0
100
101 # pb[35] == pb[31]+36+10-59+32
102 01f12001      add.w   r1, r1, #0x20
103 02f10102      add.w   r2, r2, #1
104 0000d2e5      ldrb    r0, [r2]
105 a0eb0100      sub.w   r0, r0, r1
106 43ea0003      orr.w   r3, r3, r0
107
108 # pb[36] == pb[31]+36+10-59+32-30
109 1e1041e2      sub    r1, r1, #0x1e
110 012082e2      add    r2, r2, #1
111 0000d2e5      ldrb    r0, [r2]
112 010040e0      sub    r0, r0, r1
113 003083e1      orr     r3, r3, r0
114
115 # pb[37] == pb[31]+36+10-59+32-30+42
116 01f12a01      add.w   r1, r1, #0x2a
117 012082e2      add    r2, r2, #1

```

```

118 0000d2e5      ldrb    r0, [r2]
119 010040e0      sub    r0, r0, r1
120 003083e1      orr    r3, r3, r0
121
122 # pb[38] == pb[31]+36+10-59+32-30+42-62
123 a1f13e01      sub.w   r1, r1, #0x3e
124 012082e2      add    r2, r2, #1
125 0000d2e5      ldrb    r0, [r2]
126 a0eb0100      sub.w   r0, r0, r1
127 003083e1      orr    r3, r3, r0
128
129 # Set r3 to 1 if last tests failed
130 4ff00000      mov.w   r0, #0
131 000053e1      cmp    r3, r0
132 0130a013      movne   r3, #1
133
134 # pdw[28] & unk_21050[4] == unk_21050[24]
135 e069         ldr    r0, [r4, #0x1c]
136 041099e5      ldr    r1, [sb, #4]
137 010000e0      and    r0, r0, r1
138 d9f81820      ldr.w   r2, [sb, #0x18]
139 020050e1      cmp    r0, r2
140 01308313      orrne   r3, r3, #1
141
142 # pdw[24] ror 11 == unk_21050[28]
143 180094e5      ldr    r0, [r4, #0x18]
144 e005a0e1      ror    r0, r0, #0xb
145 d9f81c10      ldr.w   r1, [sb, #0x1c]
146 010050e1      cmp    r0, r1
147 01308313      orrne   r3, r3, #1

```

À partir de ces contraintes, on pourrait utiliser [z3](#) et autres, ou faire ça en codant un peu !

Solution

Chaque test nous donne une partie du flag moyennant un peu de C et de bruteforce pour quelques octets, on arrive à la solution assez vite.

```

1  #include <iostream>
2
3  char unk_21050[36] = { 0 };
4  char flag[43] = { 0 };
5
6  int load_unknown_21050()
7  {
8      FILE* f;
9      fopen_s(&f, "unk_21050", "rb");
10     if (f)
11     {
12         fread(unk_21050, 36, 1, f);
13         fclose(f);
14     }
15     else
16     {

```



```

17     printf("Failed to read unk_21050");
18     return 0;
19 }
20 return 1;
21 }
22
23 uint32_t unk(int i)
24 {
25     return *((uint32_t*)(unk_21050 + i));
26 }
27
28 int main()
29 {
30     int i;
31
32     if (!load_unknown_21050())
33         return 1;
34
35     *((uint32_t*)(flag + 7)) = unk(8) ^ 0xffffffff;
36     *((uint32_t*)(flag + 11)) = unk(0) ^ 0xffffffff;
37
38     *((uint32_t*)(flag + 0)) = *((uint32_t*)(flag + 10));
39     *((uint32_t*)(flag + 4)) |= *((uint32_t*)(flag + 14));
40
41     *((uint16_t*)(flag + 18)) = *((uint16_t*)(flag + 8));
42
43
44     for (i = 0; i <= 0xffff; i++) {
45
46         if (((i * 2) + i) * 2 + i == unk(32))
47         {
48             *((uint16_t*)(flag + 5)) = i;
49         }
50     }
51
52     *((uint32_t*)(flag + 14)) = *((uint32_t*)(flag + 4));
53
54     *((uint32_t*)(flag + 24)) = _rotl(unk(28), 11);
55
56     *((uint32_t*)(flag + 20)) = unk(16) - *((uint32_t*)(flag + 24));
57
58     i = 1;
59     uint32_t unk4 = unk(4);
60     uint32_t unk12 = unk(12);
61     uint32_t unk20 = unk(20);
62     uint32_t unk24 = unk(24);
63
64     while (i++)
65     {
66         if (((i & unk4) == unk24) && ((i & unk20) == unk12))
67         {
68             *((uint32_t*)(flag + 28)) = i;
69             break;
70         }
71     }

```

```

72
73     flag[32] = flag[31] + 36;
74     flag[33] = flag[32] + 10;
75     flag[34] = flag[33] - 59;
76     flag[35] = flag[34] + 32;
77     flag[36] = flag[35] - 30;
78     flag[37] = flag[36] + 42;
79     flag[38] = flag[37] - 62;
80
81
82     for (i = 39; i < 42; i++)
83     {
84         flag[i] = flag[38];
85     }
86
87     for (i = 0; i < 42; i++) printf("%c", flag[i] ? flag[i] : '_');
88     printf("\n");
89
90     return 0;
91 }

```

On obtient finalement le plan secret qui me rappelle quelque chose 🤔.

```

1  Here are the secret plans:
2
3      _____
4      _,-Y | | Y-._
5      .~" | | | | "-.
6      I" ""=""|" !""! "|"[]"""|
7      L_ [] |..-----|: _[----I" .-{"-
8      I_| ..| 1_____|1_ [_L]_[I_/r(=)--P
9      [L_____L_[_____]_____j~ '-=C_]/=-^
10     \_I_j.--.\==I|I==/_.--L_
11     [_((==)[`-----"])(==)j
12     I--I"~~""~"I--I
13     |[]| |[]|
14     1__j 1__j
15     |!!| |!!|
16     |..| |..|
17     ([]) ([])
18     ]--[ ]--[
19     [_L] [_L]
20     /|..|\ /|..|\
21     `= }--{ = `= }--{ =
      .-^--r-^-. .-^--r-^-.

```

Ainsi qu'un lien vers la partie 5.

Challenge 5

Cette partie n'est pas un autre challenge, mais juste les instructions pour prévenir Naval Group qu'on a fini le challenge.