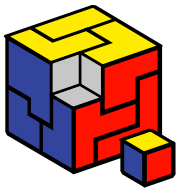
Prologin 2017



Concours National d'Informatique

Correction de la phase de sélection

Correction des challenges de pré-lancement

Alexandre Macabies
Victor Collod
Rémi Audebert

24 janvier 2017

Table des matières

1 Challenge 1

Énoncé Un lien vers un fichier avec l'extension `.archive` est fourni.

Résolution Téléchargeons le fichier et vérifions avec `file(1)` quel est son type :

```
$ wget https://prologin.org/static/ctf/2017/\
      chal1_fff9cfa45e0436e776e30a0c98b4ba2361eb87aa.archive
$ file chal1_fff9cfa45e0436e776e30a0c98b4ba2361eb87aa.archive
gzip compressed data, last modified: Sat Oct  21 2016, from Unix
```

Il s'agit d'une archive compressée au format gzip. Il existe plusieurs utilitaires pour extraire le contenu de ce type d'archive. Nous utiliserons `bsdtar(1)` qui a l'avantage d'être tout-en-un, en implémentant une grande variété de formats. Il est bien-entendu possible d'utiliser, à la place, un utilitaire dédié au format en question.

```
$ bsdtar -xvf chal1_fff9cfa45e0436e776e30a0c98b4ba2361eb87aa.archive
x challenge.rar
```

La sortie de l'utilitaire indique qu'un unique fichier `challenge.rar` a été extrait dans le dossier courant. Ce challenge est donc un exercice classique d'imbrication d'archives (compressées), à la manière de poupées russes. Extrayons cette nouvelle archive et les suivantes :

```
$ bsdtar -vxf challenge.rar
x challenge.tar.bz2
$ bsdtar -vxf challenge.tar.bz2
x challenge.iso
$ bsdtar -vxf challenge.iso
x CHALLENG.ZIP
$ bsdtar -vxf CHALLENG.ZIP
x flag.txt
```

Nous obtenons enfin le *flag* au terme de cinq imbrications et six formats de fichier différents !

```
$ cat flag.txt
prolo2017_rl4NlRGhhHyFAe21xTJWCRkdc9EZODik
```

Correction proposée par Alexandre Macabies (zopieux).

2 Challenge 2

Énoncé Un lien vers un fichier `.pyo` est fourni, pour différentes version du langage de programmation Python. Il est demandé d'exécuter ce programme, qui exposera un serveur TCP sur un certain port – non fourni. Le flag devra être trouvé en communiquant avec ce serveur.

Résolution Cette correction utilisera Python dans sa version 3.5[1]. Téléchargeons le fichier et vérifions son type :

```
$ wget https://prologin.org/static/ctf/2017/ \
       2_82b5b23bbbdcc4f37d5c5e0a141ce03ad3fcbfa4_py35.pyo
$ file chal2_82b5b23bbbdcc4f37d5c5e0a141ce03ad3fcbfa4_py35.pyo
python 3.5 byte-compiled
```

Il s'agit donc d'un programme Python byte-compilé. Un premier réflexe peut être de lister les chaînes de caractères ASCII que l'on peut trouver dans ce binaire : avec un peu de chance, le *flag* s'y trouvera. Nous pouvons à cette fin avoir recours à l'utilitaire `strings(1)` :

```
$ strings -a chal2_82b5b23bbbdcc4f37d5c5e0a141ce03ad3fcbfa4_py35.pyo
# sortie cachée par souci de concision
```

Nous n'y trouvons hélas que des noms de modules et fonctions Python. Rien de bien probant.

Approche 1 : suivre l'énoncé

L'énoncé indique qu'en lançant le programme, un serveur utilisant le protocole TCP est exposé. Il faudra s'y connecter pour récupérer le *flag*. Mais sur quel port TCP le serveur écoute-t-il ? Lançons[2] le programme :

```
$ python chal2_82b5b23bbbdcc4f37d5c5e0a141ce03ad3fcbfa4_py35.pyo
```

Utilisons l'utilitaire `ss(8)` pour lister les programmes ayant ouvert un port TCP en écoute (`LISTEN`, le comportement d'un serveur) :

```
$ ss -plant | grep LISTEN
# ...
LISTEN    0    5    *:42512    *:*    users:(("python",pid=7407,fd=5))
# ...
```

Entre autres lignes, nous trouvons celle qui indique l'utilisation de `python`. Le port qui nous intéresse est donc `42512`. Nous pouvons s'y connecter de plusieurs manières, le plus rapide étant certainement d'utiliser `nc(1)` :

```
$ nc localhost 42512
ACCESS SECURISE. VOUS AVEZ TRENTE ESSAIS POUR DETERMINER LE MOT
DE PASSE. CE NOMBRE SECRET FAIT ENTRE SEIZE ET VINGT-QUATRE BITS.
```

Le challenge est donc de deviner un nombre (positif, *a priori*) dont la taille en bits se situe entre 16 et 24 bits. Cela signifie que le nombre en question est dans l'intervalle $\left[2^{16}, 2^{24} - 1\right]$ soit $[65\,536, 16\,777\,215]$. Il y a donc 16 711 679 possibilités mais le programme ne nous laissera que… 30 essais. L'approche force brute qui consiste à tester tous les nombres ne sera donc pas exploitable, en plus d'être lente. L'astuce est ici d'exploiter le principe de la recherche dichotomique. Nous pouvons vérifier

1. La fin de vie de Python 2 est programmée.
2. Il n'est pas recommandé de lancer des programmes inconnus car il est difficile de vérifier que leur comportement n'est pas malveillant. La bonne pratique la plus courante serait de lancer le programme inconnu à l'intérieur d'une machine virtuelle. Dans notre cas, nous pouvons raisonnablement faire confiance à Prologin.

que la méthode fonctionnera dans la mesure où le nombre d'essais nécessaires, n, est (largement !) inférieur à ce que le programme nous permet :

$$n = \left\lceil \log_2 \left(2^{24} - 1 - 2^{16}\right) \right\rceil = 25 \leq 30$$

S'il est possible d'exécuter l'algorithme à la main en utilisant `nc`, cela reste fastidieux. Utilisons nos talents de programmeur·euse·s pour automatiser cette recherche :

```
import asyncio

async def connect(host, port):
    client_reader, client_writer = await asyncio.open_connection(host, port)
    # on consomme les deux lignes d'introduction
    await client_reader.readline()
    await client_reader.readline()

    # bornes initiales de la dichotomie
    the_min = 1 << 16
    the_max = 1 << 24

    tries = 0
    while True:
        # envoi de notre essai au serveur
        guess = (the_min + the_max) // 2
        client_writer.write('{}\n'.format(guess).encode())

        tries += 1
        # récupération de la réponse du serveur
        reply = (await client_reader.readline()).decode()

        if 'AUTORISE' in reply:
            print("Flag trouvé en {} essais :".format(tries))
            print((await client_reader.readline()).decode())
            return

        if 'FAIBLE' in reply:
            the_min = guess
        elif 'IMPORTANT' in reply:
            the_max = guess

loop = asyncio.get_event_loop()
# le port trouvé précédemment
loop.run_until_complete(connect('localhost', 42512))
```

Voici le résultat :

```
$ python chal2_client.py
Flag trouvé en 24 essais :
prolo2017_4FBbdiylB6YNU4O1xO2lMfMroAZFUW7Y
```

Approche 2 : ingénierie à rebours

Si le but de ce challenge était de nous faire travailler sur un classique d'algorithmique ainsi que sur la communication TCP, il était également possible de prendre un raccourci. Une certaine connaissance de Python est nécessaire pour s'en emparer. Il faut savoir que le bytecode généré par Python contient

suffisamment d'informations pour reconstruire en grande partie le code original. Plusieurs outils sont capables de faire cette traduction, par exemple `uncompyle2`. Si l'on exécute `uncompyle2` sur le fichier `.pyo` (pour Python 2) du challenge, nous obtenons un code relativement clair mais dans lequel les chaînes de caractères sont incompréhensibles [3]. Trouvons la ligne qui envoie le *flag*. Nous pouvons la repérer facilement car c'est celle qui correspond au cas où le nombre de l'utilisateur n'est ni trop grand (`if z < _`), ni trop petit (`elif _ < z`) mais bien égal (`else`) :

```
send(a('\x00\x00\x00*\xa7J\xf6y\xd0R\x0fuf\xe4^\x9d\xf6\x97\x10,\x1b'
       '\xf8\xa1\x14\xe8 6\x19\xb8\r\x1d\xce\x9f\xf2\xebd\xf0se]\xeb'
       'E\xd9\xfd\x7f:\x84\x00 \xadgW'))
```

Il nous faudrait donc le résultat de la fonction `a` appliquée à cette longue chaîne. Pour cela rien de plus simple : les fichiers de bytecode Python peuvent être importés comme n'importe quel module. Lançons un interpréteur dans le dossier courant :

```
# l'extension doit être .pyc pour que Python trouve le module
$ mv chal2_0def1833e3595e5ee36dc095c83f5658c0a7375b_py27.p{o,c}
$ python2

>>> import chal2_0def1833e3595e5ee36dc095c83f5658c0a7375b_py27 as chal
>>> chal.a
<function a at 0x7fddf746bea0>
>>> chal.a('\x00\x00\x00*\xa7J\xf6y\xd0R\x0fuf\xe4^\x9d\xf6\x97\x10,\x1b'
           '\xf8\xa1\x14\xe8 6\x19\xb8\r\x1d\xce\x9f\xf2\xebd\xf0se]\xeb'
           'E\xd9\xfd\x7f:\x84\x00 \xadgW')
'prolo2017_4FBbdiylB6YNU4O1x02lMfMroAZFUW7Y'
```

Pour finir sur le challenge 2, notons que l'utilisation d'un debugger eût été également possible pour récupérer facilement le *flag*.

Correction proposée par Alexandre Macabies (zopieux).

3. Précisément dans le but de ne pas pouvoir facilement trouver le *flag* via l'outil `strings`.

3 Challenge 3

Énoncé Un lien vers une capture réseau au format pcap est donné. Le *flag* est la concaténation d'informations que nous pouvons trouver en analysant le trafic réseau contenu dans cette capture.

Résolution Comme suggéré par l'énoncé, nous utiliserons Wireshark pour analyser le fichier de capture. Toutefois, à des fins pédagogiques, cette correction proposera une résolution en utilisant la ligne de commande plutôt que l'interface graphique. La version ligne de commande de Wireshark s'appelle `tshark(1)`.

Une fois le fichier `.pcap` téléchargé sous le nom `chal3.pcap`, nous pouvons trouver les réponses aux questions demandées.

Question 1

> Quelle est l'adresse IP du serveur web ?

Un serveur web utilise le protocole HTTP pour communiquer avec un navigateur. Nous pouvons donc filtrer la capture avec uniquement les requêtes HTTP. Le filtre à appliquer est `http.request` :

```
$ tshark -r chal3.pcap 'http.request'
  8   3.923585   10.42.0.52 -> 10.42.0.51   HTTP 140 GET / HTTP/1.1  10.42.0.51
 32  17.918269   10.42.0.53 -> 10.42.0.51   HTTP 140 GET / HTTP/1.1  10.42.0.51
```

Nous trouvons deux requêtes, issues de deux adresses IP différentes, mais qui s'adressent toutes deux au même serveur : `10.42.0.51`. Il n'y a donc aucune ambiguïté quant à la réponse à cette question.

Question 2

> Quelle est l'adresse MAC de la machine ayant pour adresse IP `10.42.0.53` ?

Il existe plusieurs manières de répondre à cette question. Nous pouvons notamment trouver n'importe quel paquet IP dont la source est l'IP demandée (`10.42.0.53`) ; il est probable que la couche liaison de ces paquets contienne l'adresse MAC correspondante. Ce sera le cas si les machines communiquent sans routage intermédiaire. Essayons :

```
$ tshark -r chal3.pcap -T fields -e mac.src 'ip.src == 10.42.0.53' | sort | uniq
08:00:27:3c:74:47
```

Pour vérifier cette réponse, nous pouvons également essayer de voir si dans ce réseau, une machine n'aurait pas demandé qui est `10.42.0.53` via le protocole ARP :

```
$ tshark -r chal3.pcap 'arp.src.proto_ipv4 == 10.42.0.53'
 19   8.396200 ARP 60 Who has 10.42.0.51? Tell 10.42.0.53
 26  13.407731 ARP 60 10.42.0.53 is at 08:00:27:3c:74:47
```

Le paquet № 26 confirme notre hypothèse.

Question 3

> Quel délai a mis le serveur de temps, en microsecondes, pour répondre à la requête qui lui a été faite (laps de temps entre le moment où la requête a été envoyée par le client et le moment où la réponse a été renvoyée par le serveur) ?

Par *serveur de temps* on entend un serveur permettant de donner l'heure grâce au protocole NTP. Filtrons !

```
$ tshark -r chal3.pcap 'ntp'
 21   8.396425   10.42.0.53 -> 10.42.0.51   NTP 90 NTP Version 4, client 10.42.0.51
 22   8.396550   10.42.0.51 -> 10.42.0.53   NTP 90 NTP Version 4, server 10.42.0.53
```

Pas d'ambiguïté possible, nous avons une seule paire requête/réponse. Si l'on lit la description des champs disponibles pour NTP, on sait que `ntp.org` est la date à laquelle le client à envoyé sa demande d'heure et `ntp.xml` est la date à laquelle le serveur a envoyé sa réponse. Affichons ces champs :

```
$ tshark -r chal3.pcap -T fields -e ntp.xmt -e ntp.org 'ntp'
Oct  6, 2016 10:15:24.469188000 CEST        Jan  1, 1970 01:00:00.000000000 CET
Oct  6, 2016 10:15:24.855015000 CEST        Oct  6, 2016 10:15:24.469188000 CEST
$ echo '( 855015000 - 469188000 ) / 1000' | bc
385827
```

Réponse finale

Assemblons les morceaux selon le format précisé dans l'énoncé :

```
$ q1='10.42.0.51'
$ q2='08:00:27:3c:74:47'
$ q3='385827'
$ echo -n "$q1 $q2 $q3" | sha1sum
2644a8d41adc0f7ab96a25cc6d7b7231306f2ca0
```

On peut vérifier que la réponse est correcte :

```
$ reponse=2644a8d41adc0f7ab96a25cc6d7b7231306f2ca0
$ curl -I https://prologin.org/static/ctf/2017/exos/$reponse.txt
HTTP/1.1 200 OK
```

Correction proposée par Alexandre Macabies (zopieux).

4 Challenge 4

Énoncé Un lien vers une capture réseau au format pcap est donné. Le *flag* est contenu dans un fichier `flag.txt` qu'il faudra dénicher dans cette capture.

Résolution Nous utilisons de nouveau Wireshark pour analyser cette capture réseau. Nous constatons rapidement que le trafic consiste exclusivement en des requêtes et réponses HTTP. La première requête est de type `HEAD`, ce qui permet de récupérer le type (`Content-Type`) et la taille (`Content-Length`) du contenu situé à l'adresse `/francolinuscapensis.zip`, sans récupérer le contenu lui-même. Les requêtes suivantes permettent de récupérer ce fichier, non pas d'une traite, mais par petits blocs consécutifs, grâce à l'en-tête `Range`.

Les plus courageux – et les moins informés ! – auront alors patiemment extrait le contenu de chacune des réponses puis assemblé le tout pour obtenir le fichier `francolinuscapensis.zip`. Mais cela n'est pas nécessaire ! Wireshark possède une fonctionnalité d'extraction pour les protocoles très répandus, tels que HTTP. Elle se cache dans le menu *File* > *Export Objects* > *HTTP*.

Maintenant que nous avons le fichier `zip` sous la main, nous pouvons essayer de l'extraire. Hélas :

```
$ unzip francolinuscapensis.zip
password:
```

L'archive est protégée par un mot de passe que nous ne connaissons pas !

4.1 Approche 1 : trouver le mot de passe

Où pourrait-on trouver cette information ? Si nous jetons à nouveau un œil à la capture réseau, nous constatons que les requêtes HTTP sont faites avec une identification *Basic*, autrement dit avec une paire d'identifiant et mot de passe. Avec un peu de chance, le mot de passe sera réutilisé pour l'archive !

```
Authorization: Basic cHJvbG9naW46QW4gaWxsdXNpb24/IFdoYXQgYXJlIHlvdSBoaWRpbmc/IA==
    Credentials: prologin:An illusion? What are you hiding?
```

Essayons :

```
$ unzip -P 'An illusion? What are you hiding? ' francolinuscapensis.zip
  inflating: francolinuscapensis/Francolinus_capensis_male.jpg
 extracting: francolinuscapensis/flag.txt
$ feh francolinuscapensis/Francolinus_capensis_male.jpg
```

```
$ cat francolinuscapensis/flag.txt
djGsCRi8npIchE2zxnetC53qvK
```

Les plus distraits n'auront pas fait attention à l'espace à la fin du mot de passe. Elle était pourtant bien présente si vous utilisiez la fonction de copie de Wireshark, en faisant un clic droit sur *Credentials* puis *Copy* > *Value*.

4.2 Approche 2 : attaque à texte clair connu

Une fois le fichier zip extrait de la capture réseau, un utilisateur malin peut réussir à le décompresser sans connaître le mot de passe. Comment fait-il ?

Le format de fichier PKZIP avec l'option de chiffrement activée est vulnérable à une attaque à texte clair connu. C'est-à-dire que si l'on connait le contenu exact d'un fichier présent dans l'archive protégée, on peut essayer de l'utiliser pour déchiffrer le reste de l'archive.

Regardons encore une fois le contenu de l'archive :

```
$ zipinfo francolinuscapensis.zip
Archive:  francolinuscapensis.zip
Zip file size: 159285 bytes, number of entries: 3
drwxrwxr-x  3.0 unx        0 bx stor 16-Oct-06 23:16 francolinuscapensis/
-rw-rw-r--  3.0 unx   158911 BX defN 13-Oct-07 08:52 francolinuscapensis/Francolinus_capensis_male.jpg
-rw-rw-r--  3.0 unx       27 TX stor 16-Oct-06 23:16 francolinuscapensis/flag.txt
3 files, 158938 bytes uncompressed, 158629 bytes compressed:  0.2%
```

En faisant une petite recherche pour le fichier `Francolinus_capensis_male.jpg` on arrive sur cette page : https://fr.wikipedia.org/wiki/Fichier:Francolinus_capensis_male.jpg. Téléchargeons le fichier d'origine.

On va ensuite utiliser `pkcrack`[4] pour tenter l'attaque.

Première étape : envoyer une carte postale à Peter Conrad pour le remercier d'avoir écrit ce logiciel. C'est vraiment important. Vous trouverez son adresse sur cette page.

Deuxième étape : extraire et compiler `pkcrack`.

```
$ tar xf pkcrack-1.2.2.tar.gz
$ make -C pkcrack-1.2.2/src
```

Troisième étape : créer un fichier zip contenant le fichier que l'on a téléchargé juste avant. Cette archive est requise pour lancer l'attaque.

```
$ zip francolinuscapensis_plain.zip Francolinus_capensis_male.jpg
  adding: Francolinus_capensis_male.jpg (deflated 0%)
```

Dernière étape : essayons `pkcrack`. On lui passe en arguments l'archive chiffrée, le fichier chiffré dont on connaît la version en clair, l'archive avec ledit fichier et son chemin.

```
$ ./pkcrack-1.2.2/src/pkcrack                                \
    -C francolinuscapensis.zip                               \
    -c francolinuscapensis/Francolinus_capensis_male.jpg     \
    -P francolinuscapensis_plain.zip                         \
    -p Francolinus_capensis_male.jpg                         \
    -a -d decrypted.zip
Files read. Starting stage 1 on Sat Dec 10 18:36:16 2016
Generating 1st generation of possible key2_158613 values...done.
Found 4194304 possible key2-values.
Now we're trying to reduce these...
Lowest number: 959 values at offset 151852
...
Done. Left with 98 possible Values. bestOffset is 125483.
Stage 1 completed. Starting stage 2 on Sat Dec 10 18:36:54 2016
Ta-daaaaa! key0=bdfea544, key1=6709c315, key2=a6959d91
Probabilistic test succeeded for 33135 bytes.
Stage 2 completed. Starting zipdecrypt on Sat Dec 10 18:36:55 2016
Decrypting francolinuscapensis/Francolinus_capensis_male.jpg (d648d8ad2082506031449646)... OK!
Decrypting francolinuscapensis/flag.txt (ff1b76a49d62cbd1eeb619ba)... OK!
Finished on Sat Dec 10 18:36:55 2016
```

4. https://www.unix-ag.uni-kl.de/~conrad/krypto/pkcrack.html

Ça a fonctionné, on peut décompresser le zip généré par `pkcrack` et lire le fichier `flag.txt` sans connaître le mot de passe :

```
$ unzip decrypted.zip
Archive:  decrypted.zip
   creating: francolinuscapensis/
  inflating: francolinuscapensis/Francolinus_capensis_male.jpg
 extracting: francolinuscapensis/flag.txt
$ cat francolinuscapensis/flag.txt
djGsCRi8npIchE2zxnetC53qvK
```

Correction proposée par Alexandre Macabies (zopieux) et Rémi Audebert (halfr).

5 Challenge 5

Énoncé Un lien vers un exécutable est donné. Le *flag* dois être saisi sur l'entrée standard du programme.

Résolution Avant tout, il est nécessaire de comprendre comment le programme procède pour vérifier la validité de son entrée. Or, le programme fourni est déjà compilé : nous n'avons pour le moment à disposition que du langage machine, illisible en l'état.

On distingue plusieurs approches :

— convertir le langage machine en assembleur, sa représentation textuelle ;
— tenter de convertir le langage machine en un langage disposant d'un plus haut niveau d'abstraction, comme le C ;
— considérer que la compréhension du programme n'est pas nécessaire, et tenter une approche par force brute (essayer tous les mots possibles).

La dernière approche étant peu enrichissante, elle ne sera pas couverte ici.

5.1 Approche 1 : ingénierie à rebours classique

Cette approche est la manière conventionnelle de résoudre cette classe de problème, et nécessite de comprendre les bases de l'assembleur. Ici, les programmes fournis ont été compilés pour l'architecture `x86`, et sa variante `x86_64`. L'assembleur utilisé sera la variante Intel de la syntaxe de ce même assembleur.

On choisit ici de s'intéresser au *listing* assembleur de la version `x86_64`.

Dans cet exercice, on utilise le *framework* d'ingénierie à rebours `radare2`.

```
$ r2 challenge_linux_x86-64.bin
aaa     # analyser la structure du code
afl     # afficher la liste des symboles
s sym.main      # aller à la fonction principale
pdf     # afficher le listing assembleur de la fonction
```

Le *listing* de la fonction main, une fois commenté et les symboles renommés, est le suivant :

```
    ; arg int arg_5h @ rbp+0x5
    ; var int cmp_cnt @ rbp-0x4
    ; var int input_buffer @ rbp-0x200
    push rbp
    mov rbp, rsp
    sub rsp, 0x200
    mov edi, str.Entrez_le_mot_de_passe: ; "Entrez le mot de passe : "
    mov eax, 0
    call sym.imp.printf
    lea rax, [rbp - input_buffer]
    mov rsi, rax
    mov edi, 0x400792
    mov eax, 0
    call sym.imp.__isoc99_scanf
    lea rax, [rbp - input_buffer]
    mov rdi, rax
    call sym.imp.strlen
    cmp rax, 6                  ; échouer si strlen(input_buffer) != 6
,=< je 0x400670
|   mov eax, 0
|   call sym.fail
`-> mov dword [rbp - cmp_cnt], 0 ; compteur = 0
```

```
  ,=< jmp 0x4006ac
 ,--> mov eax, dword [rbp - cmp_cnt]       ; charger le compteur depuis la pile
 ||   movzx edx, byte [rbp + rax - 0x200] ; 200h est le décalage du buffer d'entrée:
 ||                                        ; on charge input_buffer[i]
 ||   mov eax, dword [rbp - cmp_cnt]
 ||   movzx eax, byte [rax + obj.k]        ; charger k[i]
 ||   xor edx, eax                         ; edx = k[i] ^ input_buffer[i]
 ||   mov eax, dword [rbp - cmp_cnt]
 ||   movzx eax, byte [rax + obj.r]
 ||   cmp dl, al                  ; comparer (k[i] ^ input_buffer[i]) avec r[i]
,===< je 0x4006a8                 ; échouer en cas de différence
|||   mov eax, 0
|||   call sym.fail
`---> add dword [rbp - cmp_cnt], 1 ; compteur++
 |`-> cmp dword [rbp - cmp_cnt], 5
 `==< jbe 0x400679                ; boucler tant que compteur != 5
      mov edi, str.Bravo          ; "Bravo !!!"
      call sym.imp.puts           ; afficher le message de réussite
      mov eax, 0
      leave
      ret
```

Après analyse, on constate que ce programme compare le n^{e}caractère saisi *xor* le n^{e}caractère de `obj.k` au n^{e} caractère de `obj.r`. Ainsi, chaque caractère de la chaîne demandée est égal au caractère correspondant de `k` *xor* son homologue dans `r`.

On veut donc :

— récupérer les six octets à l'adresse `obj.k` ;
— récupérer les six octets à l'adresse `obj.r` ;
— effectuer un ou exclusif bit à bit entre les deux entiers récupérés.

Dans la console *radare*, il est possible d'obtenir le résultat en saisissant [5] :

```
$ e cfg.bigendian = true
$ ? 0x`p8 6 @obj.k` ^ 0x`p8 6 @obj.r`
$
[...] "chaton"  [...]
```

La commande `p8 6 @address` affiche les 6 octets à l'adresse spécifiée, tandis que la commande `?` permet d'évaluer une expression mathématique. Ici, on évalue le résultat de `? 0xAABBCC ^ 0xDDEEFF`, où les valeurs hexadécimales sont celles précédemment récupérées.

Attention, l'architecture `x86` est dite *little-endian* : les données sont lues de « droite à gauche ». Étant donné que on s'intéresse à des données organisées dans le sens inverse, il faut l'indiquer à *radare*. Sinon, le sésame sera tout simplement affiché à rebours.

Il est également possible d'utiliser les outils standard de l'environnement *shell*, au prix d'une procédure plus complexe et moins lisible :

```
$ objdump -M intel -d challenge_linux_x86-64.bin
[...]
$ n1=$(hexdump -s 0x75a -n 6 -v -e '/1 "%02X"' challenge_linux_x86-64.bin)
$ n2=$(hexdump -s 0x754 -n 6 -v -e '/1 "%02X"' challenge_linux_x86-64.bin)
$ printf '0x%X\n' $((0x${n1} ^ 0x${n2})) | xxd -r -p
chaton
```

5. Les symboles ont été renommés pour plus de clarté, le nom peut être différent.

5.2 Approche 2 : décompilation

Cette approche utilise un service de décompilation en ligne dont l'objectif est de transformer un programme compilé en du code disposant d'un plus haut niveau d'abstraction, comme du C ou du python. Si ces outils manquent souvent de fiabilité, celle-ci va en s'améliorant et ils permettent la plupart du temps de se libérer d'une partie du travail d'interprétation.

Si le sujet vous intéresse, il existe un document d'Hex-Rays le traitant.

En utilisant un de ces outils, on obtiens un code C analogue au suivant :

```c
char * g1 = "\x42\x13\x37\x7e\xf4\x1f";
// la chaîne de caractère suivante est tronquée par soucis de concision
char * g2 = "\x21\x7b\x56\x0a\x9b\x71 [...]";

int32_t _fail(void) {
  int32_t puts_rc = puts("Mot de passe incorrect.");
  _exit(0);
  return puts_rc;
}

int main(int argc, char ** argv) {
  printf("Entrez le mot de passe :");
  int32_t str;
  scanf("%s", &str);
  int32_t len = strlen((char *)&str); // 0x401496
  int32_t v1 = 0;
  if (len != 6) {
    _fail();
    v1 = 0;
  }
  while (true) {
    char v2 = *(char *)(v1 + (int32_t)&str);
    char v3 = *(char *)(v1 + (int32_t)&g1);
    char v4 = *(char *)(v1 + (int32_t)&g2);
    if ((int32_t)(v3 ^ v2) != (int32_t)v4)
      _fail();
    int32_t v5 = v1 + 1;
    if (v5 >= 6)
      break;
    v1 = v5;
  }
  puts("Bravo !!!");
  return 0;
}
```

La lecture de ce code met en évidence plusieurs problèmes :

- le décompilateur n'a pas été capable de détecter le type de str, ni sa taille. Exécuter le programme dans l'état aboutira probablement à de la corruption mémoire voire à une erreur de segmentation ;
- il n'a pas non plus été capable de deviner la taille de g2 ;
- il considère deux variables distinctes v5 et v1 quand bien même cela n'est pas nécessaire.

On notera que même si ce code C n'est pas correct, il n'est pas nécessaire de le compiler pour comprendre le fonctionnement du programme et récupérer le *flag*.

Correction proposée par Victor Collod (multun).

6 Challenge surprise

Énoncé Un lien vers un fichier audio est donné. Le *flag* a été modulé dans le signal.

Résolution La résolution de cet exercice va se dérouler en deux parties. On va commencer par déterminer quel type de modulation a été utilisée. Une fois ceci fait on pourra démoduler le signal et ainsi obtenir le *flag*.

Ne soyez pas effrayés par un peu de traitement du signal [6] ! Vous allez voir que la solution se déduit en quelques étapes sans avoir à manipuler des formules compliquées.

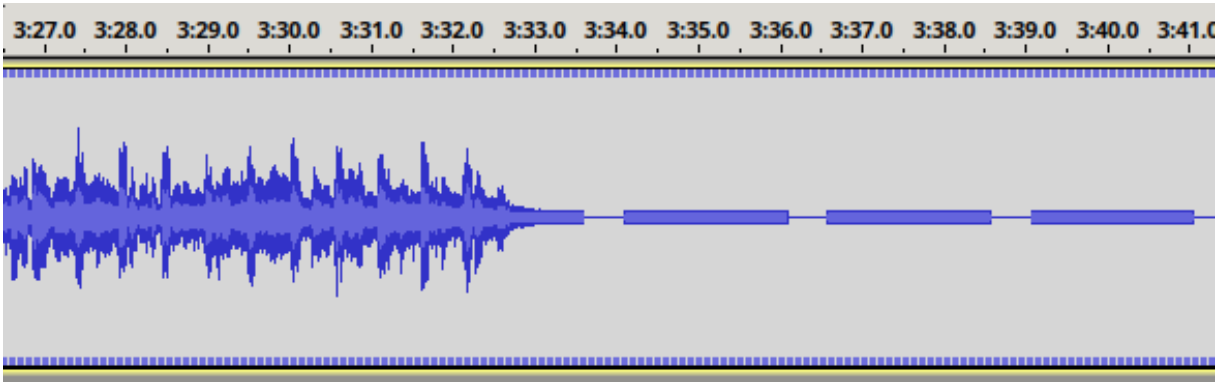
Puisqu'il faut commencer quelque part, dressons la liste des modulations numériques simples :

— la modulation en tout-ou-rien (OOK) ;
— la modulation par commutation d'amplitude (ASK) ;
— la modulation par commutation de phase (PSK) ;
— la modulation par commutation de fréquence (FSK).

Chaque possibilité est bien différente des autres et une recherche rapide vous fournira un aperçu de leur forme. Ne vous éternisez pas sur cette liste rudimentaire, sachez juste que la modulation utilisée pour cet exercice est dedans et qu'on aurait pu en utiliser une autre à la place sans rendre la difficulté de la détection et démodulation disproportionnée.

La première écoute du fichier vous aura peut-être fait grincer des dents. En effet, on entend un petit bruit en plus de la chanson. La fin du fichier est d'ailleurs composée d'une répétition de ce signal parasite.

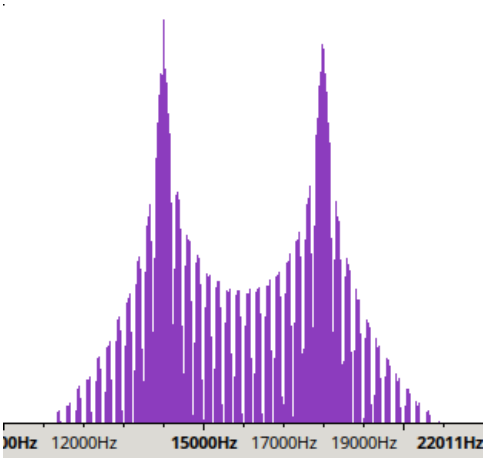
Pour connaître la nature du bruit, on va ouvrir le fichier dans Audacity, un logiciel libre de manipulation audio.



On voit sur cette image la fin de la chanson suivie du message qui se répète. Sans se poser trop de questions, c'est clairement là dedans que se trouve le *flag*. Contrairement à son collègue musical, le signal du « bruit » apparaît tout plat, assez loin de ce qu'on pourrait voir si le message était transmis par modulation d'amplitude (ASK). En zoomant, même beaucoup, on ne remarque pas non plus la présence de modulation tout-ou-rien (OOK) dans le signal, ce n'est pas ça non plus. Voilà ainsi deux modes rapidement éliminés.

En sélectionnant un exemplaire du message, on va maintenant effectuer une analyse fréquentielle, accessible dans Audacity par le menu *Analyse* > *Plot Spectrum*.

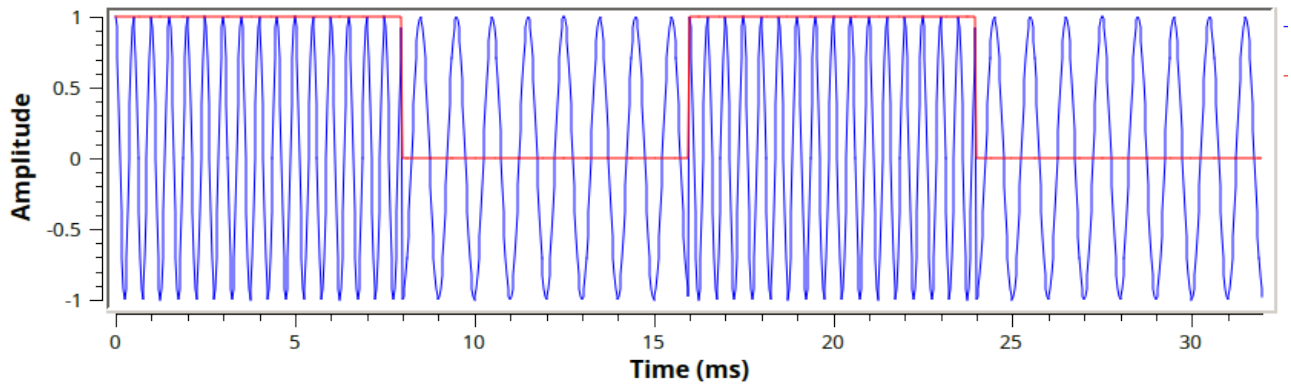
6. Cependant, si vous êtes amenés à en faire beaucoup, prévoyez d'avoir l'esprit détendu, parce que ce n'est pas facile...



Le graphique des fréquences nous révèle ceci : les fréquences 14 kHz et 18 kHz dominent dans ces messages. C'est un indice flagrant en faveur de la modulation en commutation de fréquences (FSK). Si on avait eu affaire à une modulation par commutation de phase (PSK), on aurait vu un gros pâté centré sur une seule fréquence plutôt que ces pics bien distincts.

On est donc plutôt confiant que la modulation utilisée est la FSK, mais pour la démoduler on va devoir retrouver les paramètres qui ont été utilisés. La modulation FSK, comme un certain nombre de modulations numériques, part d'une onde porteuse et lui applique des variations pour encoder de l'information. En connaissant précisément les paramètres des variations on pourra retrouver le message.

Le premier paramètre à déterminer sera la fréquence de l'onde porteuse, aussi appelée fréquence centrale. Le deuxième paramètre sera le décalage, en hertz, qui sera appliquée à la fréquence porteuse. Cette variation permet d'encoder une unité d'information, appelée un symbole dans le domaine de la communication numérique. Un symbole peut contenir un nombre de bits défini comme un troisième paramètre de la modulation. Bien qu'on note habituellement les bits avec les valeurs `0` et `1`, on va plutôt utiliser les valeurs `-1` et `+1` pour parler des symboles binaires. Avec cette notation on comprend bien que les fréquences que l'on va transmettre sont $fréquenceCentrale \pm 1 \times déviation$. Si on voulait moduler deux bits dans un symbole, on les noterait alors `-2`, `-1`, `+1` et `+2` et accompagnerait les fréquences de `±1` des fréquences $fréquenceCentrale \pm 2 \times déviation$. Ajoutons qu'il n'est pas rare de rencontrer des modulations sur plusieurs bits, par exemple quand vous composez un numéro sur le clavier d'un téléphone analogique [7], vous émettez un signal modulé en code DTMF. Cette FSK standardisée utilise 16 fréquences différentes pour encoder la touche sur laquelle vous avez tapé. Chaque symbole de la DTMF transporte donc 4 bits. Raccrochons la parenthèse téléphonique. Le dernier paramètre de la modulation est la durée de chaque symbole, c'est à dire le laps de temps où il sera observable. Le nombre de symboles transmis par seconde s'exprime en baud. Si on vous a un jour ennuyé avec la différence entre le *bit rate* et le *baud rate*, vous aurez maintenant un exemple en tête. Voici en exemple la courbe d'une modulation FSK binaire :

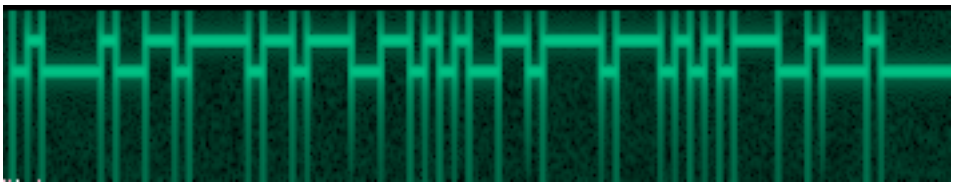


7. En ce début d'année 2017 du 21e siècle, il apparaît que certains d'entre vous ne l'auront jamais fait. Vous n'avez pas idée de la quantité de fun que vous avez raté.

On peut même dire que dans cet exemple la FSK est non-cohérente. Pourquoi cette particularité ? On voit que la modulation a été effectuée en générant deux signaux de fréquences différentes correspondant aux deux valeurs possibles des symboles. Lors du passage d'un symbole à l'autre on voit que la courbe entame une montée puis repars dans l'entre sens. À la limite du changement de symbole la phase du signal de sortie change brusquement. C'est une source de fréquences parasites en plus des fréquences « intéressantes ». On peut aussi les voir dans le spectre du signal qu'on étudie ici, au niveau des changements de symboles. La FSK est dite cohérente quand il n'y a pas de saut dans la phase du signal. On peut moduler une FSK cohérente (*Continuous-phase FSK*) dans *GNU Radio* avec le bloc *Voltage Controlled Oscillator*. Plus généralement, la plupart des variantes de la modulation FSK (GFSK et MFSK pour ne citer qu'elles) visent à éliminer les sauts de fréquence ou de phase dans le signal modulé.

Dans notre cas on peut s'orienter vers une modulation FSK binaire. L'onde porteuse est en 14 kHz et la déviation est de 2 kHz. Ces fréquences sont assez basses pour être audible par l'oreille humaine, faisant entrer la modulation dans la catégorie Audio FSK (AFSK). Les modulations AFSK étaient courantes à l'époque des modems branchés sur les lignes téléphoniques vocales, par exemple ceux utilisés par le Minitel [8].

Pour nous conforter un peu plus dans ce diagnostic de la modulation FSK, nous allons faire appel à un autre outil : baudline. Au delà de son interface austère c'est un très bon compagnon à Audacity pour l'inspection de signaux. Après avoir ouvert notre fichier on ne constate rien de plus, mais si on zoom sur un message, puis on réduit la taille de la fenêtre de l'analyse fréquentielle pour se rapprocher du *symbol rate* apparemment faible de la FSK en faisant *Clic droit* > *process* > *transform size* > *128*, on peut voir ceci :



On reconnaît clairement les symboles binaires du signal modulé par FSK où chaque symbole dure 4ms, soit un nombre de symboles par seconde de 250 bauds. Ah, c'est tout de même intéressant ! Seraient-ce des octets ? de l'ASCII ? Continuons pour le découvrir.

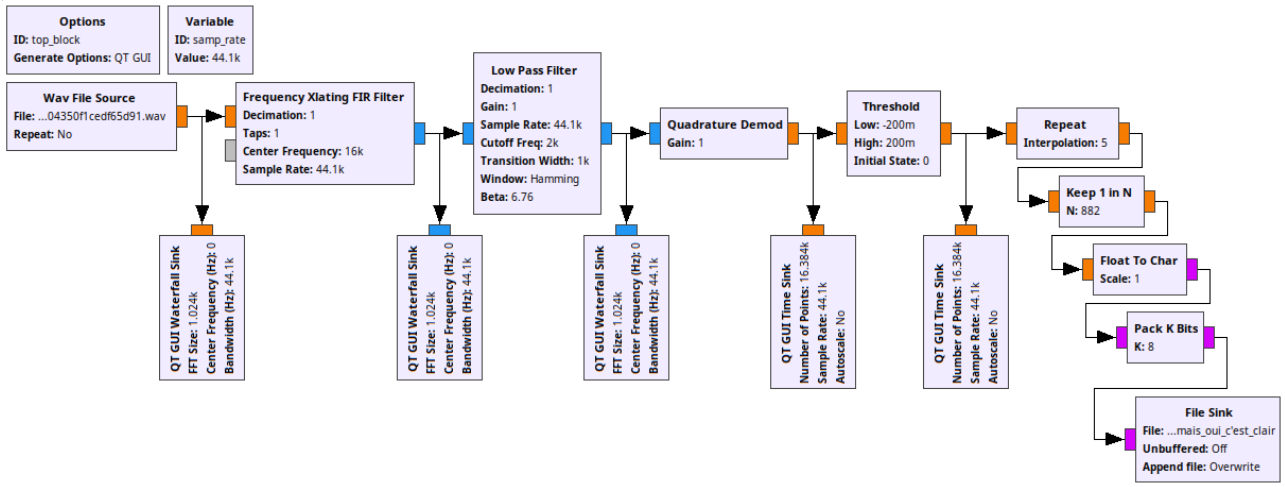
Nous allons maintenant nous atteler à démoduler ce signal à l'aspect prometteur. Pour cela il existe un logiciel libre parfaitement adapté : GNU Radio. On va utiliser son interface graphique : le *GNU Radio Companion*. C'est une boîte à outils fournissant des blocs de fonctions de traitement du signal. Avec lui on va connecter la sortie d'un bloc à l'entrée d'un autre. Une fois les blocs posés et reliés entre eux, on lancera l'exécution du schéma par *GNU Radio*.

On va proposer deux méthodes pour résoudre le challenge dans *GNU Radio*. Pour la première on va opérer de façon plutôt intuitive et pas parfaitement canonique tandis que la deuxième sera La bonne façon de faire [9] . Pas de panique, les deux nous mèneront au même résultat.

6.1 Approche 1 : résolution intuitive

Quelques blocs simples suffisent pour esquisser la démodulation FSK de notre fichier :

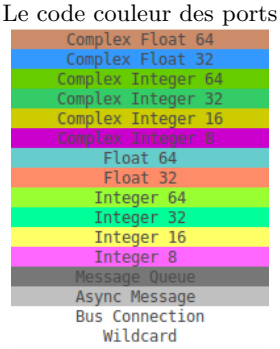
8. Si vous avez déjà utilisé un Minitel, levez votre canne !
9. Elle le restera en attendant qu'une Autre façon de faire soit proposée et déterminée objectivement Meilleure.



La première question à se poser est la suivante : que sont toutes ces couleurs sur les bords des blocs ? Ce sont les ports d'entrée/sortie. Chaque couleur correspond à un type de donnée, par exemple nombre flottant, complexe, octet, etc. Si vous reliez deux ports de types différents, *GNU Radio Companion* refusera de générer le code de votre schéma et, à juste titre, de l'exécuter.

Observons chaque bloc de ce schéma pour les expliquer et comprendre comment ils nous permettent d'extraire le *flag*.

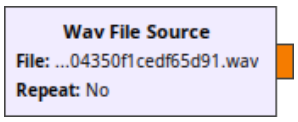
Le code couleur des ports



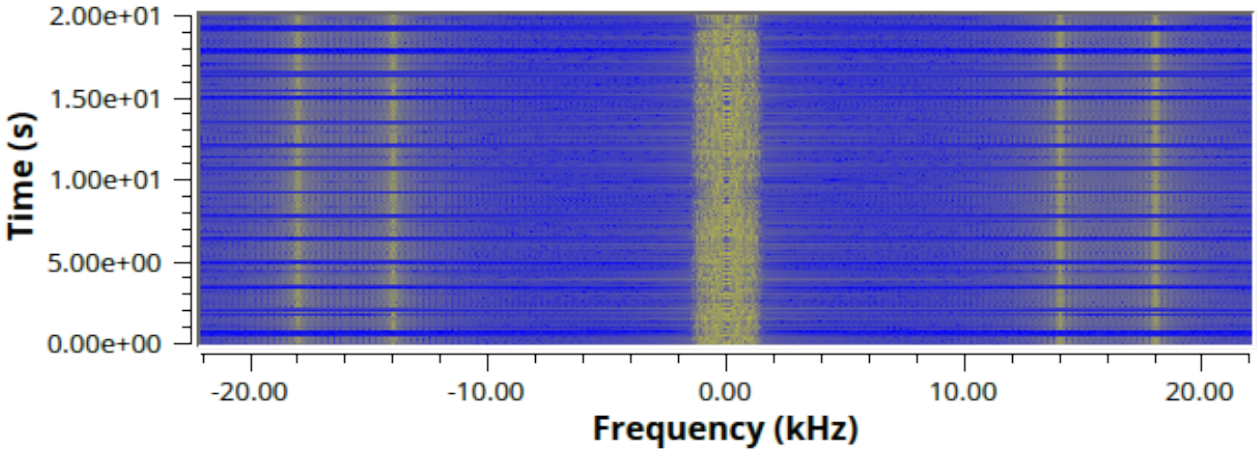
Le bloc *Options* nous permet de configurer *GNU Radio Companion* pour ce schéma. Le paramètre *ID* indique le nom du module python qui sera créé et *Generate Options* nous permet de demander à utiliser une interface mignonne en *Qt*.

Les différents blocs *QT GUI* nous permettent de visualiser le signal après les transformations qu'on lui fait subir. On les a reliés à la sortie de certaines fonctions pour effectuer les captures ci-dessous. Les blocs *Waterfall* sont les affichages des puissances des fréquences contenues dans le signal au cours du temps. Les blocs *Time* tracent la courbe du signal tel qu'il est. On a pas modifié les paramètres de ces blocs, les valeurs par défaut nous conviennent.

Le bloc *Wav File Source* insère le signal en entrée de notre schéma. On a pris le soin de convertir le fichier FLAC, non géré par *GNU Radio*, en WAV avec la commande `sox(1)`. L'attribut *Repeat* est passé à *No* pour éviter de lire en boucle le fichier, ce qui aurait pour conséquence de générer un fichier énorme en sortie. Il n'est pas actuellement possible de signaler automatiquement quand arrêter *GNU Radio* depuis le schéma. On devra donc le faire nous même en cliquant sur l'icône *Stop* de l'interface quand tous les *samples* sont passés à travers le schéma, ce qu'on verra quand les graphiques *Qt* ne se mettront plus à jour.



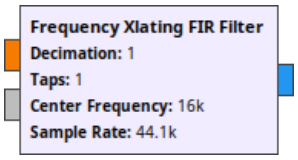
On lance l'exécution et le graphique ci-dessous apparaît, correspondant au bloc *QT GUI Waterfall Sink* branché sur le bloc *Wav File Source*.



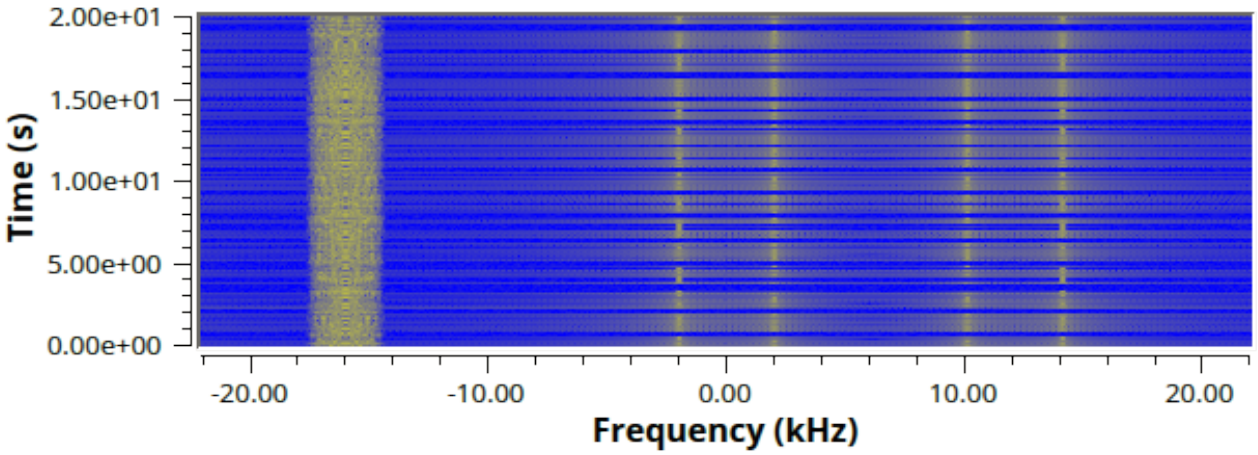
La *waveform* du signal en entrée. Au centre les fréquences de la chanson et sur les côtés le message à démoduler. Une interrogation peut vous être venues à l'esprit : pourquoi voit-on une symétrie dans le spectre affiché ? Audacity et baudline n'affichaient pas de fréquences négatives, eux.

Il y a une ambigüité dans le signe de la fréquence du signal que l'on a envoyé en entrée du bloc *waterfall*. Elle est due au fait qu'on a un signal audio réel comme source. Pour la lever il faudrait travailler avec un signal I/Q et c'est exactement ce qui est fait quand on capture un signal électromagnétique [10].

Le bloc *Frequency Xlating FIR Filter* cache sous son nom effrayant une fonction qui va « décaler » le signal dans le domaine des fréquences pour le centrer sur la partie qui nous intéresse, à savoir celle qui est autour de 16 kHz. On a configuré le taux d'échantillonnage (*sample rate*) dans la variable *samp_rate* à partir de la valeur qui est indiquée dans le fichier FLAC source. C'est un des paramètres importants dans tout schéma *GNU Radio* car les flux qu'il traite en interne ne contiennent pas cette information. De plus il n'est pas rare de faire varier le taux d'échantillonnage à certaines étapes du schéma, par exemple pour réduire la quantité de calcul à effectuer.



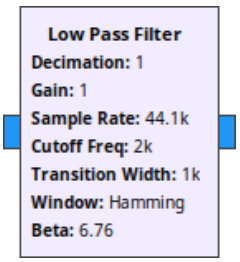
Les paramètres *Decimation* et *Taps* servent respectivement à rééchantillonner le signal en entrée en moyennant les *samples* en entrée et à intégrer des filtres supplémentaires en sortie du bloc. Ce sont des outils qui permettent de simplifier et réduire la charge de calcul nécessaire à l'exécution du schéma et nous n'en avons pas particulièrement besoin pour l'instant donc nous les laissons à leur valeur par défaut. Le port gris de ce bloc est de type *Async Message* et peut être laissé non connecté sans problème car il n'est pas utile dans notre cas.

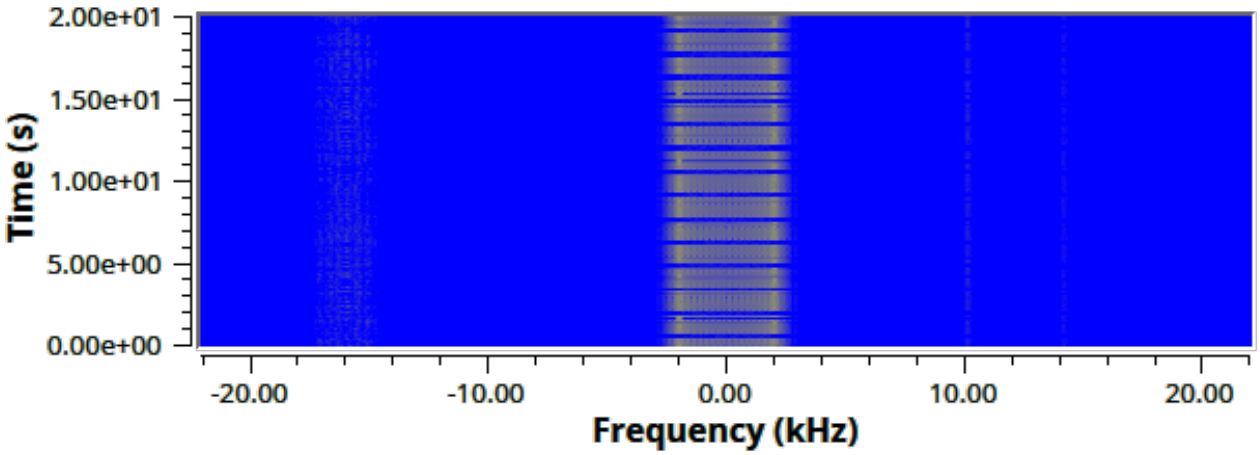


10. On les appelle les données I/Q, pour plus d'informations à ce sujet vous pouvez consulter l'excellent article I/Q Data for Dummies.

La *waveform* du signal en sortie du *Frequency Xlating FIR Filter*. Le message modulé est maintenant centré et les autres signaux ont été décalés. Cependant, comme pour la *waveform* précédente, il y quelque chose qui ne va pas. On voit que le filtre a effectué une « rotation » dans le domaine des fréquences alors qu'on avait demandé un « décalage ». C'est un exemple d'*aliasing* : un même signal peut être identifié à plusieurs fréquences différentes. Ainsi le décalage est bien effectué mais fait apparaître d'autres images des fréquences. L'*aliasing* n'aurait pas été visible si on avait eu un signal complexe en entrée.

Le bloc *Low Pass Filter* (filtre passe-bas) laisse passer les basses fréquences et atténue les fréquences plus élevées que la fréquence limite. Il nous permet d'effacer la chanson originale, maintenant située dans les hautes fréquences, et de ne garder que le signal qui nous intéresse, convenablement situé dans les basses fréquences. Les deux paramètres que l'on a dû modifier sont *Cutoff Freq* et *Transition Width*. Le premier correspond à la fréquence à partir de laquelle le filtre commence à faire effet et le deuxième l'intensité dudit effet.

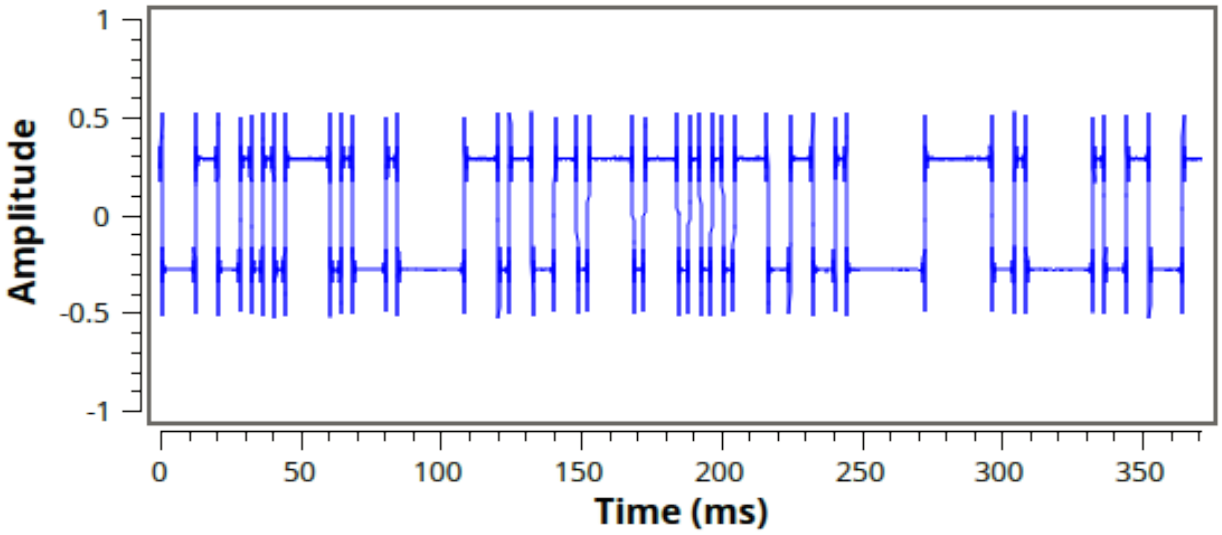




On remarque des résidus de fréquences plus hautes que celle indiquée dans le filtre passe-bas, on les doit à l'implémentation numérique finie du filtre. Ils ne sont pas embêtants et on peut les négliger car la puissance du signal intéressant est dominante. Autre point de détail notable : les « basses » fréquences sont considérées en valeur absolue, ce qui explique pourquoi les fréquences négatives sont aussi filtrées.

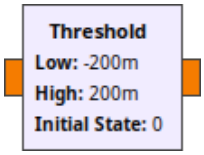
Le bloc *Quadrature Demod* est le bloc le plus important de ce schéma car c'est lui qui va effectuer la démodulation FSK. C'est un module utilisable pour le décodage FM analogique [11] et il fonctionnera tout aussi bien pour notre modulation numérique. Son effet est, de façon simple, le suivant : il transforme les variations du domaine fréquentiel dans le domaine temporel. En d'autres mots, si la fréquence dominante en entrée est positive alors le signal en sortie sera positif et inversement avec une fréquence dominante négative. Il existe une formule [12] pour calculer le gain qui va rendre le décodage optimal, mais dans notre cas un gain de 1 fonctionne plutôt bien.

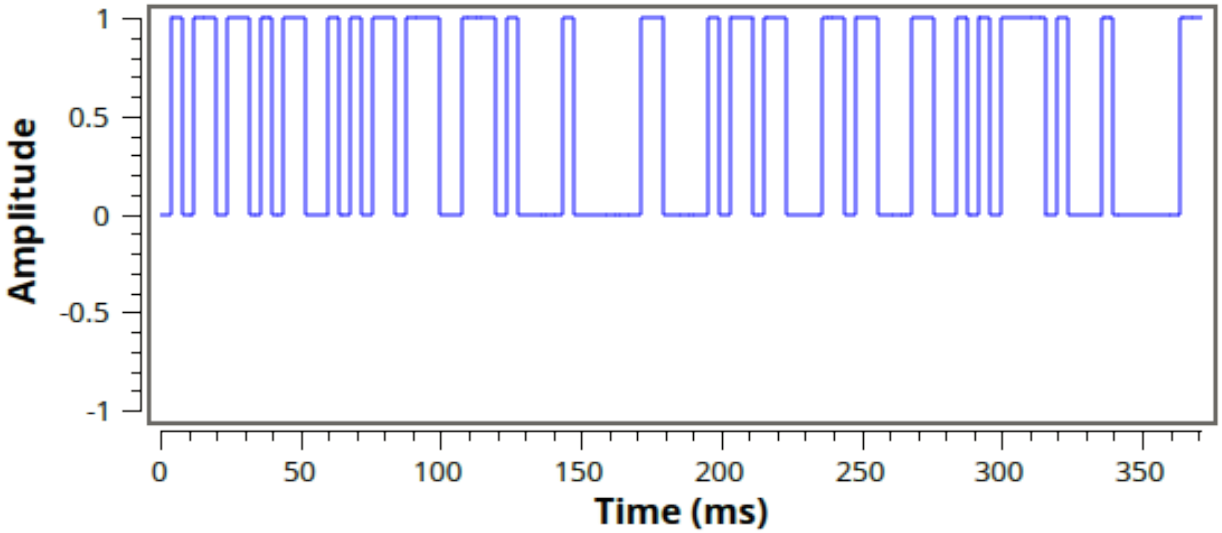
11. On préfèrera cependant les blocs *FM Demod*, *WBFM Receive* ou *NBFM Receive, implémentés avec des filtres adaptés pour démoduler la radio.* pour écouter la radio car ils sont plus adaptés.
12. Qu'on ne citera pas pour l'instant, comme promis.



On obtient un signal qui ressemble à une suite de valeurs binaires. Il ne reste plus qu'à nettoyer le signal, extraire les bits que l'on a sous les yeux, recomposer les octets et on devrait obtenir le *flag*.

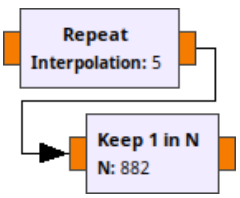
Le bloc *Threshold* nous permet d'améliorer la qualité du signal en le contraignant à deux valeurs : 0 et 1. Les deux paramètres du bloc, *Low* et *High*, correspondent aux valeurs limites à partir desquelles le signal est considéré comme un 0 ou un 1. On a déterminé ces valeurs en regardant l'amplitude sur le graphique précédent.





Nous voici avec une magnifique suite de bits ! Ou en réalité un signal qui y ressemble, composé d'une grande quantité de nombres flottants. Comment transformer ce signal en octets dans un fichier ? C'est exactement le but des derniers blocs du schéma. On va extraire un *sample* représentatif au milieux de chacun des symboles. On aurait pu détecter le début des messages, mais on va faire simple en échantillonnant un *sample* périodiquement, quitte à avoir des bits qui n'ont pas de sens.

Les blocs *Repeat* et *Keep 1 in N* vont extraire un nombre flottant par bit présent dans le signal en entrée. Les valeurs de *Interpolation* et N ont été déterminées de la façon suivante : tout d'abord on a constaté en regardant les bits qu'ils ont une période de 4ms. Compte tenu du taux d'échantillonnage de notre fichier en entrée, 44,1 kHz, il faudrait extraire un bit tous les $samp\_rate \times symbol\_period = 176{,}4\ samples$. Ceci n'est pas un nombre entier ! Or il est



impossible d'extraire exactement un *sample* à un intervalle non-entier depuis les schémas de *GNU Radio*[13]. Si on arrondissait à l'entier supérieur ou inférieur on arriverait à extraire quelques octets à la suite, mais on se désynchroniserait assez rapidement. Pour s'en sortir on va augmenter artificiellement le taux d'échantillonnage en répétant 5 fois chaque *sample* avec le bloc *Repeat*. Ainsi avec un *sample rate* de 44,1 kHz × 5 = 220,5 kHz on peut extraire un bit exactement tous les 176,4 × 5 = 882 *samples* avec le bloc *Keep 1 in N*.

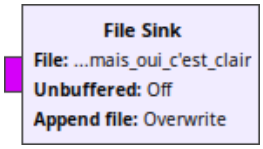
Il ne reste qu'à recomposer les octets à partir des bits.

Le bloc *Float To Char* convertit les *samples* de flottant en octet. Le 0.0_{flottant} devient $0000\ 0000_2$ et 1.0_{flottant} devient $0000\ 0001_2$. On a besoin d'effectuer cette transformation pour pouvoir connecter le bloc suivant.

Le bloc *Pack K Bits* va accumuler $K = 8$ *samples* en entrée, les convertir en bits et composer un octet en sortie avec ces valeurs. Par exemple, les 8 octets suivants arrivent en entrée : `0x00`, `0x00`, `0x00`, `0x01`, `0x00`, `0x00`, `0x01`, `0x00`. Le bloc produira un octet en sortie : `0x48` ($0100\ 1000_2$). Au passage, on peut dire que le *sample rate* est divisé par $K = 8$, mais ce n'est plus important à cette étape du schéma.

Le bloc *File Sink* fait sortir les octets du monde de *GNU Radio* en les écrivant dans le fichier spécifié.

Après avoir exécuté le schéma on va regarder le contenu du fichier que nous avons créé. On a effectué la démodulation sur l'intégralité du signal en entrée, sans se soucier de trouver le début et la fin des messages. La démodulation a donc été effectuée en partie sur des résidus de signaux non modulés. Ce n'est pas grave, on se retrouve juste avec du non-sens autour des messages. On peut améliorer le schéma en ajoutant un bloc *Simple Squelch* après le *Low Pass Filter* permettant de couper le signal quand le signal en entrée est trop faible[14] et ainsi se débarrasser du bruit.



Pour ne garder que les octets représentant des caractères affichables on utilise `strings(1)`.

```
$ strings -10 mais_oui_c\'est_clair
p`Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
HBonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.8
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.q
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.&
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.Q
3Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.L
wBonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
$
```

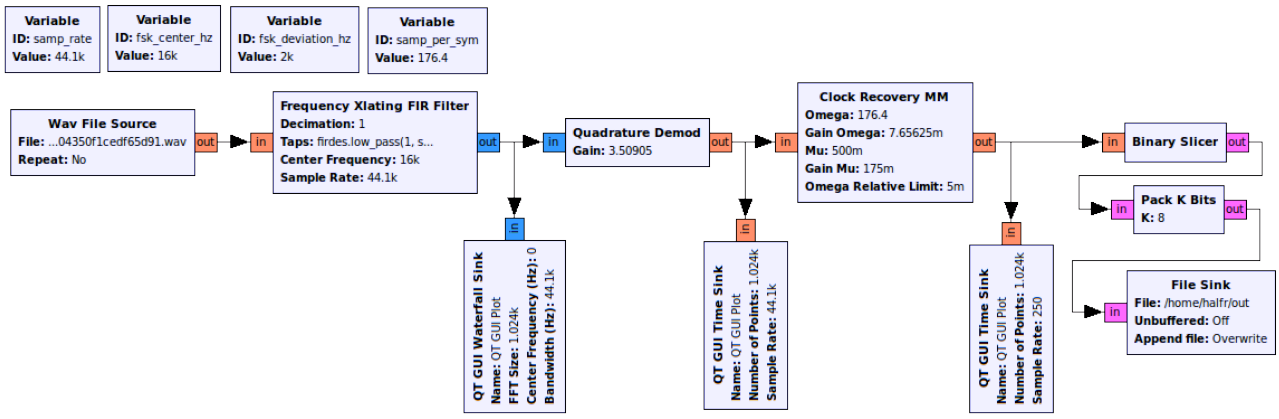
Bravo ! Le *flag* est `AfskIsAwesome`.

13. Ce n'est pas possible pour une bonne raison : notre manière de procéder au décodage ne correspond pas à ce qui est conventionnellement fait. Ce signal ne nous fournit pas de moyen explicite de nous synchroniser automatiquement sur l'horloge de l'émetteur et permettant de détecter automatiquement le début de la transmission. On voit souvent une suite de 0 et de 1 en préambule des messages pour permettre une synchronisation. À la place on s'est débrouillé en regardant préalablement le signal pour déterminer la période des symboles et s'assurer qu'ils sont répartis uniformément. La deuxième correction pour cet exercice évite cette gymnastique étrange en utilisant un bloc dédié à la synchronisation.

14. Dans notre cas une valeur qui fonctionne bien pour le seuil de bruit est -80dB.

6.2 Approche 2 : résolution avancée

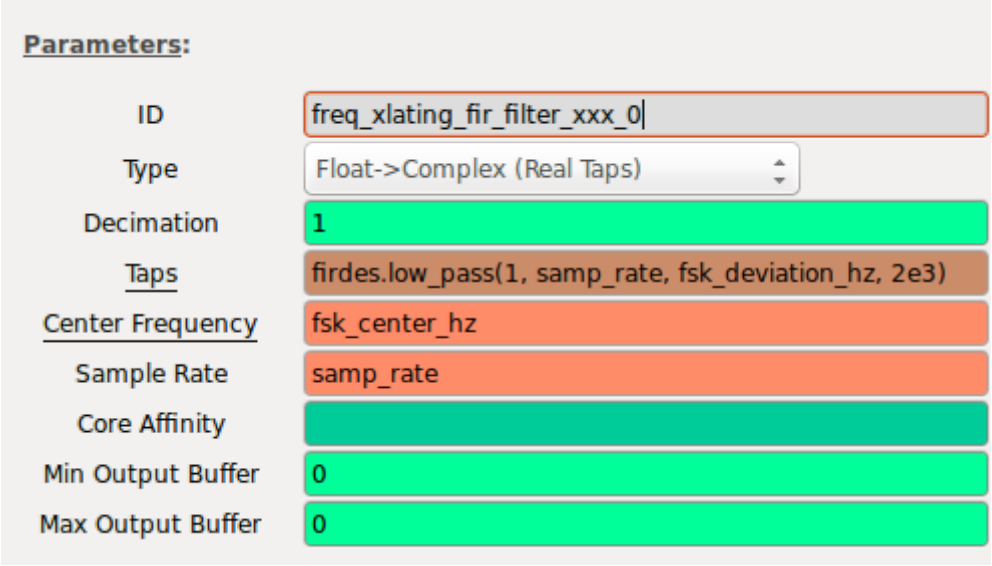
On a dit que la première façon de faire était peut être plus aisée à comprendre, mais elle n'est pas la façon canonique de démoduler une FSK binaire. Le schéma suivant est plus proche de cette méthode :



La principale différence est que le schéma utilise moins de blocs, eh oui, quand on maitrise nos outils, on fait moins d'effort. Comme pour le premier schéma, reprenons chaque bloc.

Le bloc *Wav File Source* ne change pas.

Le bloc *Frequency Xlating FIR Filter* permet toujours de décaler le spectre pour placer la fréquence centrale de la FSK au niveau du 0 ; autrement dit : annuler la fréquence constante de 16 kHz pour ne garder que la partie variable. L'utilisation avancée de ce bloc consiste à configurer le paramètre *Taps* pour inclure le filtre passe-bas, qu'on avait pris la peine de placer dans un bloc supplémentaire. La configuration exacte du bloc est :

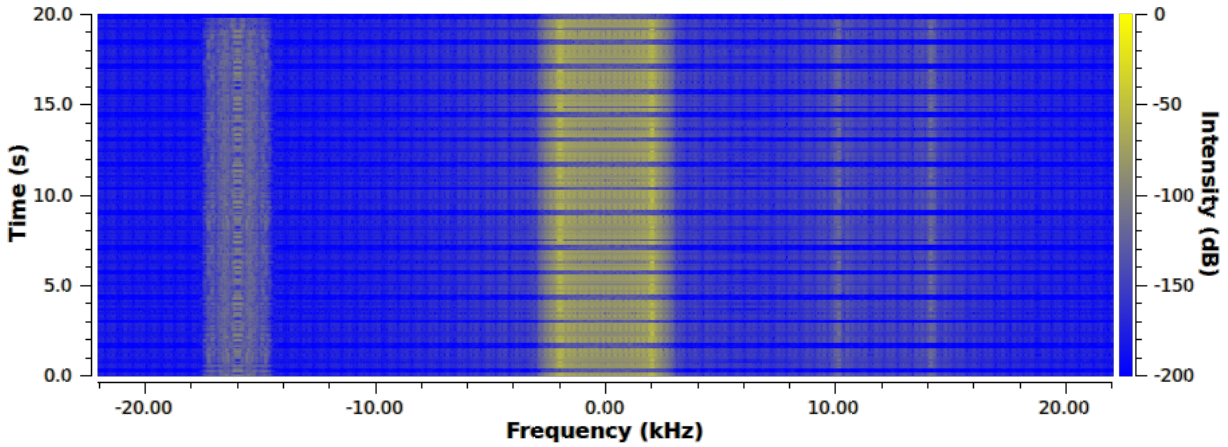


Le paramètre *Taps*, comme la plupart des autres champs, est évalué comme une expression Python, le langage dans lequel est généré le code *GNU Radio* généré par *GNU Radio Companion*. On peut donc utiliser des fonctions disponibles dans la bibliothèque Python de *GNU Radio*, qui est en grande partie un *proxy* vers les fonctions implémentées en C++. Nous avons donc utilisé la fonction `filter::low_pas` permettant de générer un filtre passe-bas et dont le prototype est :

```
gr::filter::firdes::low_pass(
  double    gain, // Le gain du filtre, on utilise 1 pour ne rien faire
```

```
  double     sampling_freq, // On l'appelle aussi sample rate
  double     cutoff_freq, // La fréquence où le filtre prend effet
  double     transition_width, // La largeur de la fenêtre du filtre
  win_type   window = WIN_HAMMING,
  double     beta = 6.76
);
```

On a laissé les deux derniers arguments à leurs valeurs par défaut, elles nous iront très bien. En sortie de ce filtre, on obtient un signal avec le spectre suivant :

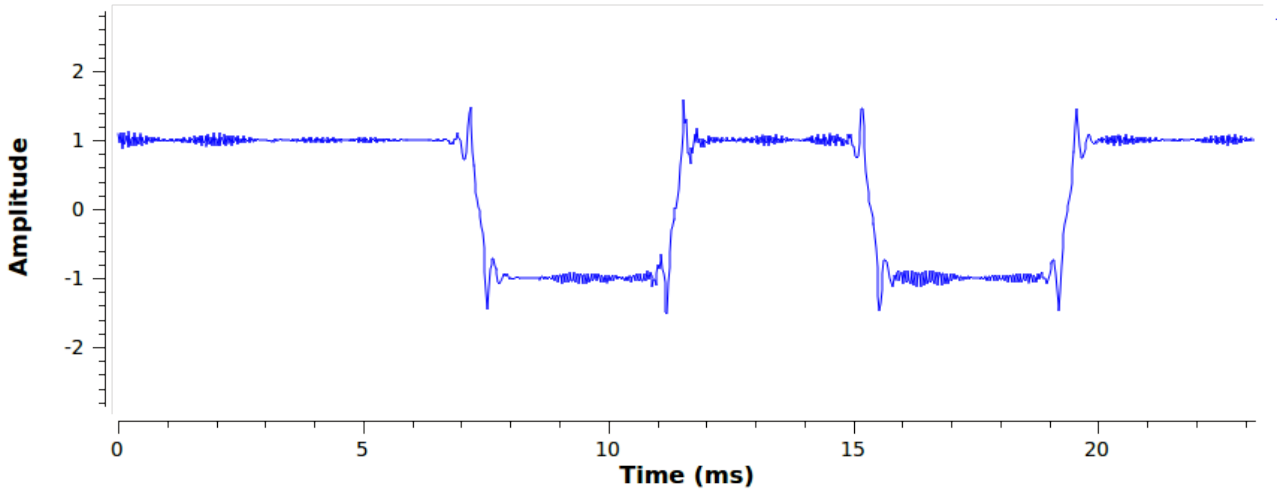


L'atténuation des hautes fréquences n'est pas aussi forte qu'avec le bloc dédié mais vous verrez que ça ne pose aucun problème pour la suite. L'important était d'annuler la fréquence porteuse pour retrouver le signal modulé sans sa composante constante.

Le bloc *Quadrature Demod* va nous permettre de démoduler les symboles `-1` et `+1` dont on parlait dans la première partie. La documentation de `quadrature_demod_cf` explique la formule du gain qui nous permet de les retrouver [15] :

$$\text{Gain} = \frac{\text{sample\_rate}}{2\pi \times \text{fsk\_deviation\_hz}}$$

On obtient en sortie du bloc un magnifique signal qui ressemble à :



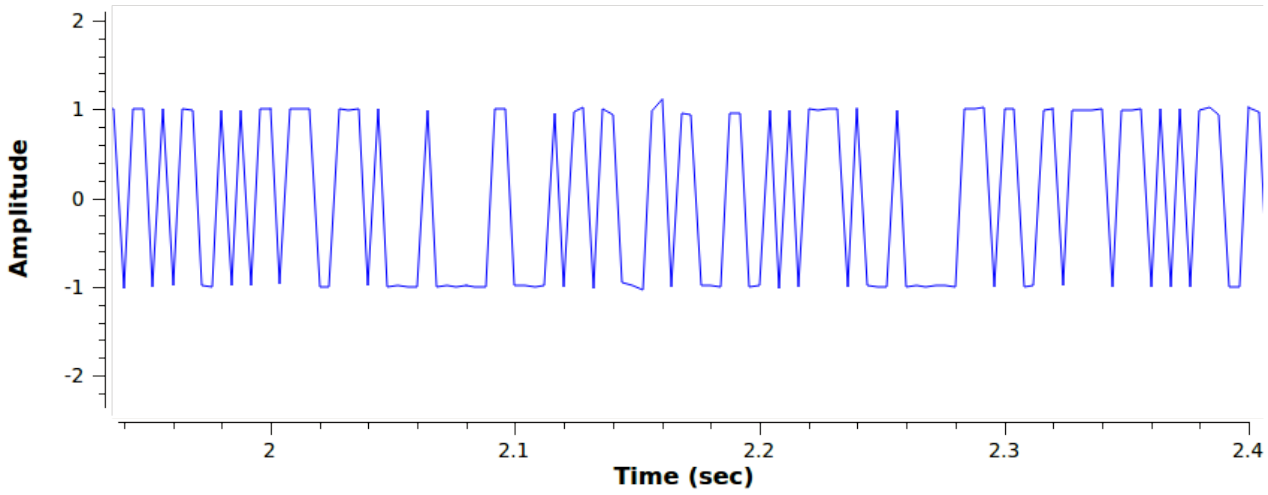
Comme voulu, les symboles `-1` et `+1` sont parfaitement présents. Le petit bruit visible correspond aux résidus du filtrage des hautes fréquences, comme on vous disait ils sont négligeables.

15. Ceci dit une approche empirique fonctionne aussi, mais on a une formule alors autant l'utiliser.

Le bloc *Clock Recovery MM* [16] est la clef de l'analyse des symboles et probablement le bloc le plus « magique » que l'on a utilisé dans le schéma. Son but est d'émettre un *sample* par symbole en entrée, celui-ci étant modulé dans une série de *samples* contigus. L'algorithme implémenté dans ce bloc va se synchroniser automatiquement sur le flux de *samples* en entrée et, connaissant la durée d'un symbole et prenant comme hypothèse une distribution équitable de ceux-ci, déterminer la valeur de chaque symbole, et ce même si les symboles se décalent petit à petit dans le temps. Cette phrase était bien longue et tordue, nous vous conseillons de la relire plusieurs fois pour une meilleure compréhension. Un cycle complet sera une série de 100.

Le seul paramètre que l'on va modifier est *Omega*, correspondant au nombre de *samples* dans un symbole, que l'on a calculé précédemment comme ceci :

$$samp\_per\_sym = samp\_rate \times symbol\_period = 176{,}4$$



Chaque *sample* est extrait d'un ensemble de *samples* en entrée déterminés comme un symbole par l'algorithme. Croyez-nous, ça fonctionne.

Le bloc *Binary Slicer* émet pour chaque *sample* en entrée :

$$\begin{cases} 1_2 & \text{si } sample_{flottant} > 0 \\ 0_2 & \text{si } sample_{flottant} < 0 \end{cases}$$

On a bien l'idée de l'entrée qui est découpée au niveau du 0 pour obtenir des bits en sortie.

Pour terminer on retrouve les blocs *Pack K bits* et *File Sink* du premier schéma, leur but n'a pas changé, ni leurs paramètres.

```
$ strings -10 out
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
@Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
 Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
0        Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
```

16. Implémentation de l'algorithme *Mueller and Müller (M&M) discrete-time error-tracking synchronizer*.

```
@R<Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
N\1DP{$S<0x
Bonjour ! Comment allez vous ? La reponse est 'AfskIsAwesome'.
```

Comme pour la première méthode, le manque de filtrage de début et de fin des messages nous donne des octets bons à jeter autour des informations pertinentes, mais l'objectif est atteint et on peut lire le flag !

De Guiche
Merci. Je vais avec ce bout d'étoffe claire,
Pouvoir faire un signal, — que j'hésitais à faire.

Cyrano de Bergerac, acte IV
Edmond Rostand

Correction proposée par Rémi Audebert (halfr), avec une relecture extensive de Jean-Michel.

*
**

Nous espérons que la lecture de ces corrections vous a été aussi plaisainte qu'a été leur rédaction pour nous. Vive Prologin !