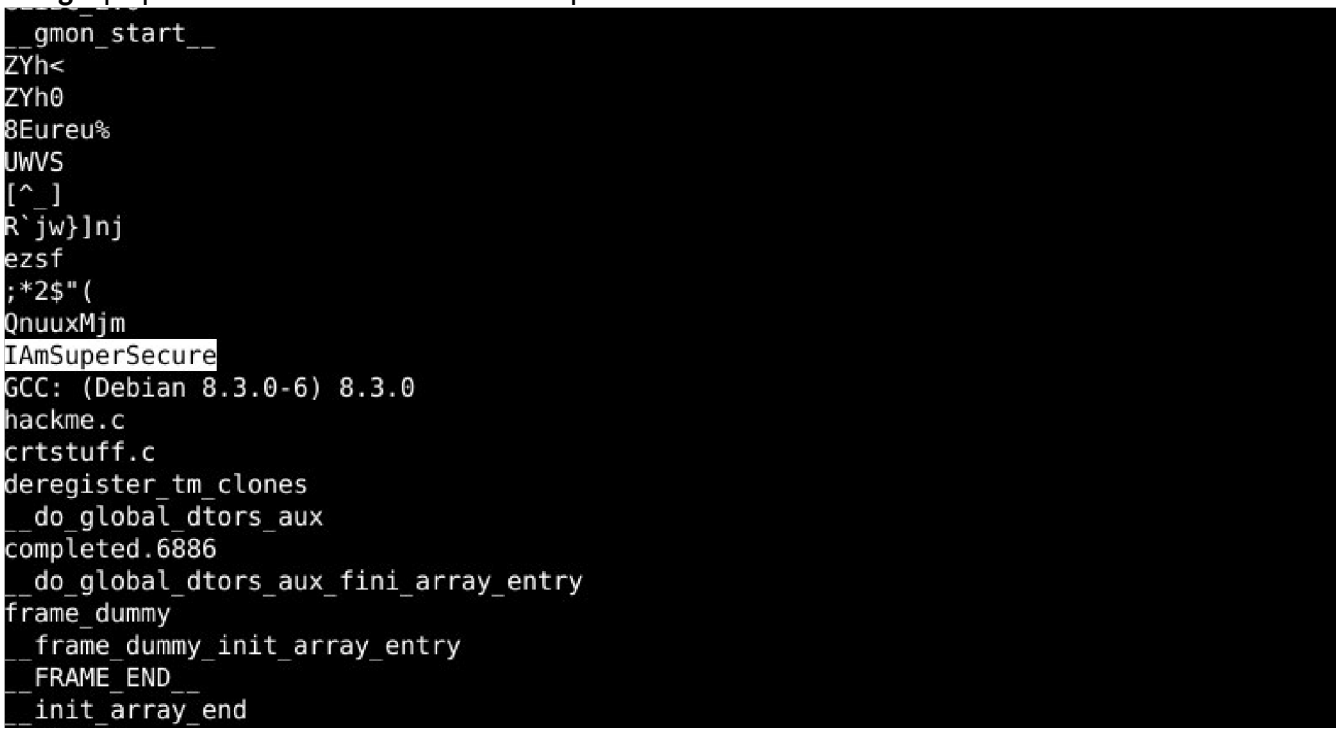
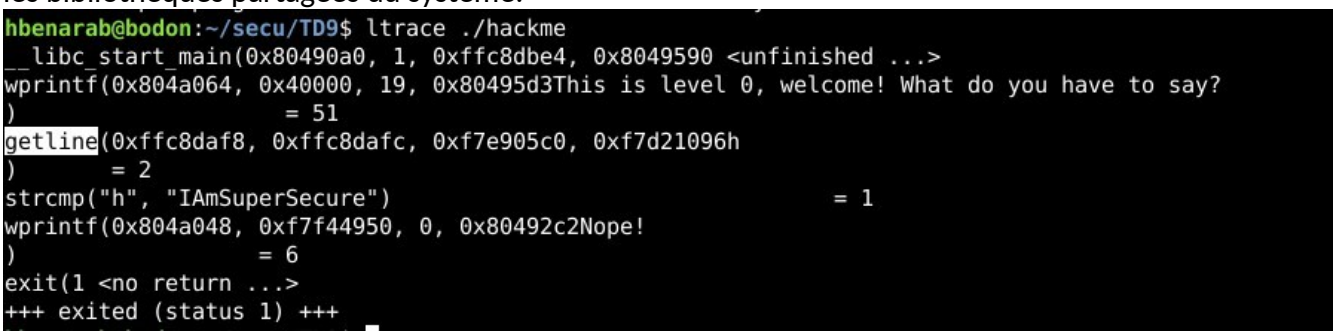
Level 0 :

Dans ce niveau afin de pouvoir trouver le mot de passe, on s'intéressera tout d'abord a la commande **strings** qui permet d'afficher les caractères qui sont affichables dans un fichier.



En voyant déjà le retour de la commande **strings,** la seule chaîne de caractère qui a du sens c'est **' IamSuperSecure',** on se doute déjà bien qu'il s'agit peut-être du mot de passe recherché.

On peut également utiliser la commande **ltrace** qui permet d'intercepter les appels dynamiques vers les bibliothèques partagées du système.



Avec cette commande, on voit bien que le programme fait appel a la fonction **strcmp()** qui permet d'effectuer une comparaison entre deux chaînes de caractères pour voir si elles sont identiques. Ici, **strcmp()** compare la chaîne de caractères donné par un utilisateur avec **' IamSuperSecure'** dans le cas ou, cette chaîne ne correspond pas a **' IamSuperSecure' .** Le programme s'arrête.
DONC le mot de passe est bien : **' IamSuperSecure'**

Level 1 :

Les programmes protègent leurs données en cryptant leurs contenues, donc les lires en utilisant **strings** est difficile. Or, dans la mémoire pendant l'exécution les données, sont en clair donc on peut

facilement les lires. En effet, une fois que l'appel à la fonction de déchiffrage est terminé, les données initialement chiffrées sont en clair maintenant en mémoire.

Pour trouver cette fois-ci le mot de passe, on met un breakpoint sur **strcmp@plt,** et on se met dans le contexte de la fonction qui a fait appel a **strcmp()** et on la désassemble.

```
0x08049363 <+19>:    pop    %edx
0x08049364 <+20>:    pop    %ecx
0x08049365 <+21>:    push   $0x804d030
0x0804936a <+26>:    push   %eax
0x0804936b <+27>:    mov    %eax,%ebx
0x0804936d <+29>:    call   0x8049030 <strcmp@plt>
```

On voit à travers le désassemblage, qu'on empile dans la pile une adresse ainsi que le contenu du registre **%eax**, qui correspondent aux paramètres de la fonction strcmp() et qu'ensuite en effectue l'appel a **strcmp()**.

Essayons à présent d'afficher le contenu du registre%eax et a quoi correspond l'adresse passé a strcmp
().

```
(gdb) p/x $eax
$1 = 0x804f580
(gdb) p (char *)0x804f580
$2 = 0x804f580 "okjpo"
(gdb) p (char *) 0x804d030
$3 = 0x804d030 <p> "HelloDad"
```

%eax contient la chaîne de caractère qui doit être passé par l'utilisateur, quant à l'adresse **0x804d030,** c'est la chaîne de caractère auquel la chaîne passé par l'utilisateur devrait être comparé. DONC Le mot de passe est : **HelloDad**

Maintenant, nous voudrions connaître le nom de la fonction qui effectue le déchiffrage. Pour ce faire, on met un **watchPoint** sur le premier caractère du mot de passe trouvé afin de traquer les modifications qu'il subît.

```
Hardware watchpoint 2: *(char * )0x804d030

Old value = 81 'Q'
New value = 72 'H'
0x08049339 in z ()
```

La fonction qui effectue le déchiffrage est la fonction z() comme mentionnée en dessus.
Si on effectue un break sur la fonction z () et qu'on essaie d'afficher le mot de passe (Donc avant le déchiffrage), on obtient cela :

```
(gdb) p (char *)0x804d030
$1 = 0x804d030 <p> "QnuuxMjm"
(gdb)
```

level 2 :

Les programmes soit évite de garder les données non encryptées dans la mémoire ou les gardent encryptées dans la mémoire et les déchiffrent seulement quand il faut les utiliser.

Cette fois-ci, en regardant les paramètres passés a la fonction strcmp(), ces derniers sont encryptés. On ne peut donc pas se ramener au niveau précédent.

```
(gdb) p (char *) 0x804adec
$1 = 0x804adec "R`jw}]nj"
```

En comparant le désassemblage de la fonction r1() du niveau précédant et la fonction r2 (), on constate que r2() effectue un appel supplémentaire a une fonction **x1()** qui sans doutes permet de crypter le mot de passe.

Essayons de comprendre ce que fait cette fonction x1 () :

```
080492f0 <x1>:
 80492f0:              8b 54 24 04             mov    0x4(%esp),%edx
 80492f4:              0f b6 02                movzbl (%edx),%eax
 80492f7:              84 c0                   test   %al,%al
 80492f9:        /---- 74 15                   je     8049310 <x1+0x20>
 80492fb:        |     8d 74 26 00             lea    0x0(%esi,%eiz,1),%esi
 80492ff:        |     90                      nop
 8049300:        |  /-> 83 c0 09               add    $0x9,%eax
 8049303:        |  |   83 c2 01               add    $0x1,%edx
 8049306:        |  |   88 42 ff               mov    %al,-0x1(%edx)
 8049309:        |  |   0f b6 02               movzbl (%edx),%eax
 804930c:        |  |   84 c0                  test   %al,%al
 804930e:        |  \-- 75 f0                  jne    8049300 <x1+0x10>
 8049310:        \----> c3                     ret
```

Elle met dans le registre %edx la valeur qui est contenue en mémoire à l'adresse 4+(%esp), ensuite elle met les 08 bits de poids faible de la valeur pointée par l'adresse contenu dans %edx et le reste a 0 dans le registre%eax.

Elle effectue ensuite un test pour savoir si ces 08 derniers bits sont nuls. Si c'est le cas on jump a ret. Sinon on rajoute 9 à %eax et 1 à %edx. Ensuite on reprend les 8 derniers bits de la valeur contenu dans %eax et on les met à l'emplacement mémoire pointée par %edx -1, on remet encore les 08 derniers bits de la valeur pointée par l'adresse contenu dans %edx et le reste a 0 dans le registre%eax et reteste si ces derniers sont nul si c'est le cas on jump a ret sinon on reboucle.
En résumé, il parcourt une chaîne de caractères, ajoute 9 à chaque valeur en ASCII du caractère parcouru et termine dès qu'il rencontre un caractère nul.

Donc pour retrouver le mot de passe, on prend pour chaque caractère de cette chaîne « **R`jw}]nj** », sa valeur en ASCII et on lui soustrait 9 et on regarde a quoi cette nouvelle valeur correspond en ASCII. LE MOT DE PASSE est donc : IwantTea

Level 3 :

L'utilisation de strcmp est trop évidente pour être traquée, donc les programmes préfèrent effectuer la comparaison eux-mêmes et ne pas l'appeler, donc le breakpooint sur strcmp ne fonctionnera pas cette fois-ci.

En regardant la bt, on voit que le code fait appel a main (), wut() et r() (wut () fait appel a r()). On sait que r() permet juste de récupérer la chaîne de caractère donné par l'utilisateur, désassemblant alors le code de la fonction wut() pour voir ce que cette dernière effectue :

```
(gdb) disas
Dump of assembler code for function wut:
   0x080493f0 <+0>:     push   %ebx
   0x080493f1 <+1>:     sub    $0x14,%esp
   0x080493f4 <+4>:     push   $0x804a37c
   0x080493f9 <+9>:     call   0x8049060 <wprintf@plt>
   0x080493fe <+14>:    call   0x8049220 <r>
=> 0x08049403 <+19>:    add    $0x10,%esp
   0x08049406 <+22>:    cmpl   $0x65727545,(%eax)
   0x0804940c <+28>:    jne    0x8049433 <wut+67>
   0x0804940e <+30>:    cmpl   $0x21614b,0x4(%eax)
   0x08049415 <+37>:    mov    %eax,%ebx
   0x08049417 <+39>:    jne    0x8049433 <wut+67>
   0x08049419 <+41>:    sub    $0xc,%esp
   0x0804941c <+44>:    push   $0x804a424
   0x08049421 <+49>:    call   0x8049060 <wprintf@plt>
   0x08049426 <+54>:    mov    %ebx,(%esp)
   0x08049429 <+57>:    call   0x8049050 <free@plt>
   0x0804942e <+62>:    add    $0x18,%esp
   0x08049431 <+65>:    pop    %ebx
   0x08049432 <+66>:    ret
   0x08049433 <+67>:    call   0x8049280 <f>
End of assembler dump.
```

On voit qu'elle effectue une comparaison entre **$0x65727545,(%eax)** ensuite, **$0x21614b,0x4(%eax).** Essayons tout d'abord de voir ce que contient **%eax**

```
This is now level 3! How would you say?
haha

Breakpoint 1, 0x08049406 in wut ()
(gdb) p (char *) $eax
$3 = 0x804f580 "haha"
```

Le registre eax contient la chaîne de caractère donné par l'utilisateur et effectuer la comparaison avec (%eax) ceci revient a dire qu'on effectue la comparaison avec la représentation hexadecimale de la chaîne pointé par eax, c'est-à-dire la valeur ASCII de chaque caractère en hexadécimale et en little-endian.

Donc **$0x65727545** et **$0x21614b** peuvent être interprété comme des valeurs représenté en **ASCII** et en **little-endian**.

En effet, la fonction wut() effectue tout d'abord une comparaison des 04 premiers octets du mot de passe avec les 04 premiers octets de la chaîne donné par l'utilisateur (qui se trouve dans dans

l’espace mémoire pointé par l’adresse dans eax) si ce n’est pas égal, on jump a l’appel de la fonction f qui print (Nope!) et exit.

Sinon on continue la comparaison des 03 autres octets du mot de passe (**0x21614b** est représenté sur 3 octets) avec les autres octets de la chaîne donnée par l’utilisateur.

$**0x65727545 : donne** eruE et **$0x21614b : donne** !aK

Pour obtenir exactement ces valeurs en hexa il faut retourner la chaîne car elle sera écrite en little-endian.

Donc le mot de passe est : EureKa!

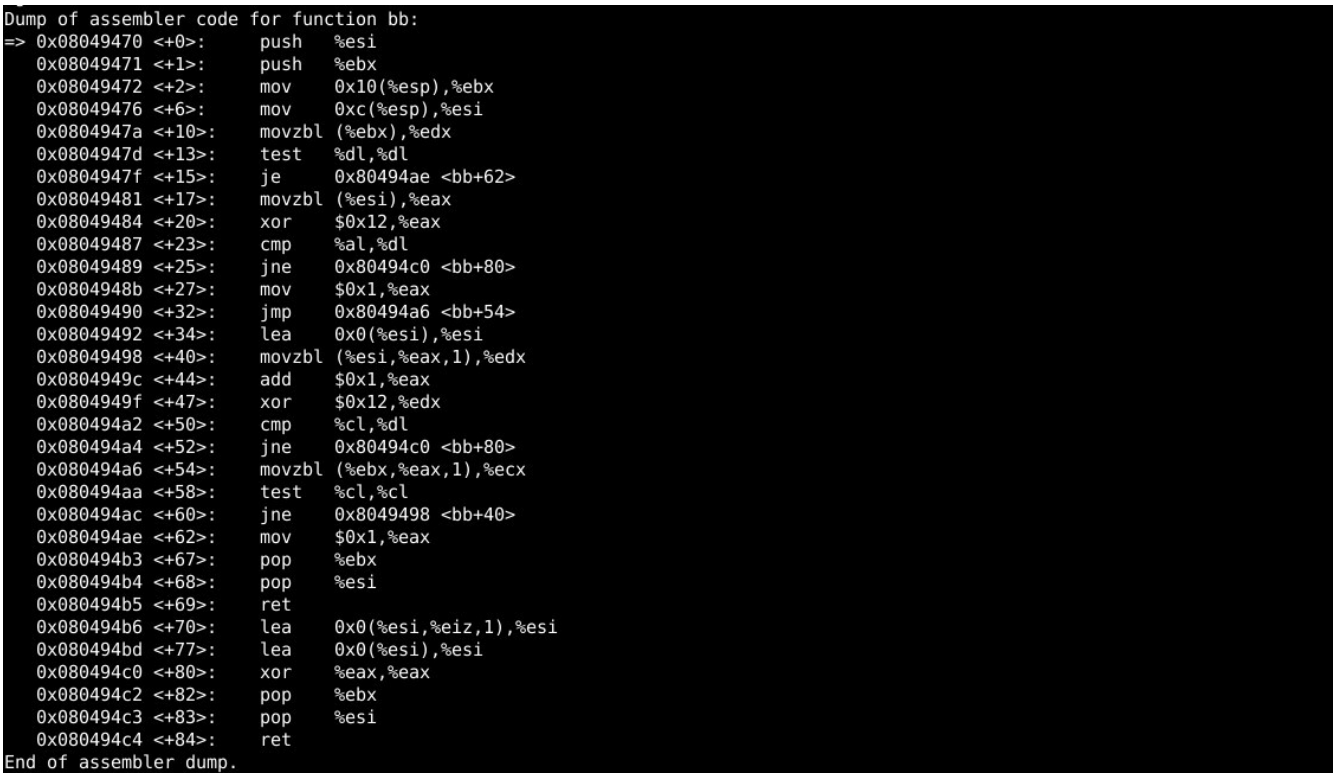
Level 4 :

Regardons ce que fait la fonction aa ():

```
Dump of assembler code for function aa:
   0x080494d0 <+0>:     push   %ebx
   0x080494d1 <+1>:     sub    $0x14,%esp
   0x080494d4 <+4>:     push   $0x804a474
   0x080494d9 <+9>:     call   0x8049060 <wprintf@plt>
   0x080494de <+14>:    call   0x8049220 <r>
=> 0x080494e3 <+19>:    mov    %eax,(%esp)
   0x080494e6 <+22>:    mov    %eax,%ebx
   0x080494e8 <+24>:    call   0x8049080 <strlen@plt>
   0x080494ed <+29>:    add    $0x10,%esp
   0x080494f0 <+32>:    cmp    $0x4,%eax
   0x080494f3 <+35>:    jne    0x8049524 <aa+84>
   0x080494f5 <+37>:    sub    $0x8,%esp
   0x080494f8 <+40>:    push   $0x804adf5
   0x080494fd <+45>:    push   %ebx
   0x080494fe <+46>:    call   0x8049470 <bb>
   0x08049503 <+51>:    add    $0x10,%esp
   0x08049506 <+54>:    test   %eax,%eax
   0x08049508 <+56>:    je     0x8049524 <aa+84>
   0x0804950a <+58>:    sub    $0xc,%esp
   0x0804950d <+61>:    push   $0x804a4ec
   0x08049512 <+66>:    call   0x8049060 <wprintf@plt>
   0x08049517 <+71>:    mov    %ebx,(%esp)
   0x0804951a <+74>:    call   0x8049050 <free@plt>
   0x0804951f <+79>:    add    $0x18,%esp
   0x08049522 <+82>:    pop    %ebx
   0x08049523 <+83>:    ret
```

ce qu’on peut retenir de ce désassemblage, c’est que cette fonction fait appel a strlen() qui permet de calculer la longueur d’une chaîne de caractère, ensuite il teste si le contenu du registre %eax (retour de strlen () qui contient la taille de la chaîne de caractère donné par l’utilisateur) est sur 04 octets (on se doute bien que le mot de passe doit être sur 04 octets)
-si oui on empile dans la pile
$**0x804adf5** qui est l’adresse de la chaîne ‘ezsf’ et ebx qui est l’adresse de notre entrée (ils seront les paramètres de la fonction bb ()) . Et ensuite on appel la fonction bb (). Regardons a présent, ce que fait la fonction bb() :

-si non on jump a l’appel de fonction f() qui va print Nope! Et terminer le programme vu que le mot de passe est faux car pas la bonne taille.



```
Dump of assembler code for function bb:
=> 0x08049470 <+0>:     push   %esi
   0x08049471 <+1>:     push   %ebx
   0x08049472 <+2>:     mov    0x10(%esp),%ebx
   0x08049476 <+6>:     mov    0xc(%esp),%esi
   0x0804947a <+10>:    movzbl (%ebx),%edx
   0x0804947d <+13>:    test   %dl,%dl
   0x0804947f <+15>:    je     0x80494ae <bb+62>
   0x08049481 <+17>:    movzbl (%esi),%eax
   0x08049484 <+20>:    xor    $0x12,%eax
   0x08049487 <+23>:    cmp    %al,%dl
   0x08049489 <+25>:    jne    0x80494c0 <bb+80>
   0x0804948b <+27>:    mov    $0x1,%eax
   0x08049490 <+32>:    jmp    0x80494a6 <bb+54>
   0x08049492 <+34>:    lea    0x0(%esi),%esi
   0x08049498 <+40>:    movzbl (%esi,%eax,1),%edx
   0x0804949c <+44>:    add    $0x1,%eax
   0x0804949f <+47>:    xor    $0x12,%edx
   0x080494a2 <+50>:    cmp    %cl,%dl
   0x080494a4 <+52>:    jne    0x80494c0 <bb+80>
   0x080494a6 <+54>:    movzbl (%ebx,%eax,1),%ecx
   0x080494aa <+58>:    test   %cl,%cl
   0x080494ac <+60>:    jne    0x8049498 <bb+40>
   0x080494ae <+62>:    mov    $0x1,%eax
   0x080494b3 <+67>:    pop    %ebx
   0x080494b4 <+68>:    pop    %esi
   0x080494b5 <+69>:    ret
   0x080494b6 <+70>:    lea    0x0(%esi,%eiz,1),%esi
   0x080494bd <+77>:    lea    0x0(%esi),%esi
   0x080494c0 <+80>:    xor    %eax,%eax
   0x080494c2 <+82>:    pop    %ebx
   0x080494c3 <+83>:    pop    %esi
   0x080494c4 <+84>:    ret
End of assembler dump.
```

1- On met dans le registre %ebx, l'adresse de la chaîne de caractères « ezsf » $**0x804adf5** et dans %esi l'adresse de notre entrée

2- On met les 8 derniers bits (8 bits de poids faible) de valeur pointée par %ebx dans edx (8 bits = 1 octet = premier caractère de la chaîne)

3- il teste si ces bits sont nuls.

4- si oui, on retourne 0

5- on met les 8 derniers bits (8 bits de poids faible) de valeur pointée par %esi dans eax (8 bits = 1 octet = premier caractère de la chaîne)

6- on fait un XOR entre 12 et la valeur de eax.

7- il teste si la valeur contenue dans eax (aprés le xor) est égale a %edx (qui contient le premier octet de « ezsf »)

8- si pas égale, on ret 0

9- sinon, on met 1 dans%eax et on jump a bb+54

10- on met les 8 derniers bits de la valeur contenu a l'adresse ebx+eax*1 dans%ecx

11- teste si l'octet contenu dans%ecx n'est pas null, si oui on ret 1

12- si non, on jump a bb+40

13-on met les 8 derniers bits de la valeur contenu a l'adresse esi+eax*1 dans%edx

14- il rajoute 1 a eax et effectue un xor entre edx et 0x12

15- 7- il teste si la valeur contenue dans edx (aprés le xor) est égale a %ecx (qui contient le eax iéme octet de « ezsf »)

16- si c'est pas égal, on ret 0, si non on boucle (on se re ramène a ligne 10 tant qu'on arrive pas a la finde la chaîne)

Pour résumer, bb() prend en paramètres l'adresse de la chaîne d'entrée et l'adresse de la chaîne de caractères « ezsf » **0x804adf5.** Elle effectue un XOR entre chaque caractère de notre chaîne et 0x12 pour les comparer ensuite, à la chaine « ezsf ». l'algorithme de chiffrement de ce programme est juste effectuer un XOR entre 0x12 et chaque caractère de la chaîne.

En effet, afin de trouver le mot de passe, il suffit de trouver une chaîne de caractères dont ses caractères XOR 0x12, donneront « ezsf ».

```
(gdb) p/x 'e'
$1 = 0x65
(gdb) p/x 'z'
$2 = 0x7a
(gdb) p/x 's'
$3 = 0x73
(gdb) p/x 'f'
$4 = 0x66
```

On XOR chacune des valeurs hexadécimal avec 0x12 et obtient : 0x77,0x68,0x61,0x74

```
(gdb) p (char) 0x77
$5 = 119 'w'
(gdb) p (char) 0x68
$6 = 104 'h'
(gdb) p (char) 0x61
$7 = 97 'a'
(gdb) p (char) 0x74
$8 = 116 't'
```

Le mot de passe est donc « what ».

Level 5 :

Regardons ce que fait la fonction yay () qui fait appel a r() pour récupérer la chaîne de caractère donné par l'utilisateur :

```
Dump of assembler code for function yay:
   0x08049530 <+0>:     push   %ebx
   0x08049531 <+1>:     sub    $0x14,%esp
   0x08049534 <+4>:     push   $0x804a584
   0x08049539 <+9>:     call   0x8049060 <wprintf@plt>
   0x0804953e <+14>:    call   0x8049220 <r>
=> 0x08049543 <+19>:    mov    %eax,(%esp)
   0x08049546 <+22>:    mov    %eax,%ebx
   0x08049548 <+24>:    call   0x8049080 <strlen@plt>
   0x0804954d <+29>:    add    $0x10,%esp
   0x08049550 <+32>:    cmp    $0x3,%eax
   0x08049553 <+35>:    jbe    0x804958b <yay+91>
   0x08049555 <+37>:    movzbl (%ebx),%eax
   0x08049558 <+40>:    mov    %ebx,%edx
   0x0804955a <+42>:    xor    %ecx,%ecx
   0x0804955c <+44>:    test   %al,%al
   0x0804955e <+46>:    je     0x804958b <yay+91>
   0x08049560 <+48>:    add    $0x1,%edx
   0x08049563 <+51>:    xor    %eax,%ecx
   0x08049565 <+53>:    movzbl (%edx),%eax
   0x08049568 <+56>:    test   %al,%al
   0x0804956a <+58>:    jne    0x8049560 <yay+48>
   0x0804956c <+60>:    cmp    $0x43,%cl
--Type <RET> for more, q to quit, c to continue without paging--
   0x0804956f <+63>:    jne    0x804958b <yay+91>
   0x08049571 <+65>:    sub    $0xc,%esp
   0x08049574 <+68>:    push   $0x804a624
   0x08049579 <+73>:    call   0x8049060 <wprintf@plt>
   0x0804957e <+78>:    mov    %ebx,(%esp)
   0x08049581 <+81>:    call   0x8049050 <free@plt>
   0x08049586 <+86>:    add    $0x18,%esp
   0x08049589 <+89>:    pop    %ebx
   0x0804958a <+90>:    ret
   0x0804958b <+91>:    call   0x8049280 <f>
```

On voit bien que cette fonction fait appel également a la fonction strlen(), le retour de cette fonction est stocké dans %eax, donc contient la taille de la chaîne de caractère de l'entrée, on compare cette taille a 0x3, si la taille de la chaîne est inférieur ou égale a 3, on jump vers la fonction f() qui print Nope! (MOT DE PASSE FAUX), le mot de passe est donc sur 04 caractères minimum.

1- On met dans eax, les 8 bits de poids faible (1 octet) de valeur pointée par ebx. (correspond a la valeur ASCII du premier caractère de la chaine passé en entrée)
2- on met l'adresse de la chaine de l'entrée dans%edx
3- on effectue un XOR entre %ecx et %ecx (remettre le registre ecx a zéro)
4- on teste si l'octet qu'on a mit dans %eax est nul, si oui on jump vers la fonction f() (termine le programme)
5- sinon, on ajoute 1 à %edx (qui contient l'adresse de la chaîne de l'entrée)
6- effectue un XOR entre %eax et %ecx et on garde le résultat dans ecx.
7- On met dans eax, les 08 derniers bit (1 octet) de la valeur pointée par %edx.
8- on teste si l'octet qu'on a mit dans %eax est nul, si pas null on jump vers ligne 5 (yay + 48)

9- si nul, on teste si la valeur des 8 bits de poids faible de la valeur contenu dans ecx (résultat du xor eax ecx) est égale a 0x43 en ASCII qui vaut « C», sinon jump vers f ().

En résumé, il parcourt la chaîne de l'utilisateur octet par octet effectue un xor sur les caractères de la chaîne de caractères pour obtenir un caractère unique et vérifie si ce caractère est un caractère "C".

Pour passer le niveau, il suffit de prendre un mot de passe a 04 caractères ou plus dont le xor entre chaque caractère donne 0x43, la façon le plus simple de le faire est de former ce MDP de tel sorte qu'on obtienne un XOR nul puis on concatène la lettre C a ce dernier donc le XOR entre 0 et C donnera C qui vaut 0x43 ainsi, on passera les vérifications effectuées par yay(). **Par exemple:** aaaaC

Pour le rendre l'algorithme plus efficace et pas facilement cassable, il serait intéressant d'interdire les mots de passes de taille impair. En effet, avec une chaîne de taille paire on peut facilement obtenir un XOR nul, restera qu'a concaténer la lettre C pour passer le niveau. Si on interdit les chaînes de taille impaire, ceci deviendra impossible, il faudrait donc résoudre une équation d'au moins 4 inconnus a fin d'arriver à trouver un xor qui donne 0x43.