Reverse - DisassembleAndReassemble - 50 points

Waïl TAHMAOUI, Kévin DUVERGER

Table des matières

1 Résolution DisassembleAndReassemble : 2
1.1 Avec un décompilateur en ligne (et quelques autres outils) : 2
1.2 Avec Ghidra : 2

1 Résolution DisassembleAndReassemble :

Pour ce nouveau challenge de reverse, on peut encore une fois utiliser la commande `strings` et cette fois-ci vous remarquerez que l'on ne trouve rien (oh que c'est dommage).

1.1 Avec un décompilateur en ligne (et quelques autres outils) :

Nous pouvons donc envoyer cet exécutable dans un décompilateur de notre choix, vous pouvez prendre par exemple `ghidra`, mais un outil en ligne comme `https://dogbolt.org/` suffira pour ce challenge. La première étape consiste donc à trouver le `main`, je vous donne ici la version donnée par `HexRays` que je trouve la plus propre au niveau du code (mais n'hésitez pas à aller regarder les autres) :

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    FILE *v3; // eax
    char Buffer[1000]; // [esp+19h] [ebp-417h] BYREF
    char v6[18]; // [esp+401h] [ebp-2Fh] BYREF
    char v7[29]; // [esp+413h] [ebp-1Dh] BYREF

    __main();
    strcpy(&v7[17], "Password : ");
    strcpy(v7, "Good password !\n");
    strcpy(v6, "Wrong password !\n");
    printf(&v7[17]);
    v3 = ___acrt_iob_func(0);
    fgets(Buffer, 999, v3);
    if ( Buffer[strlen(Buffer) - 1] == 10 )
        Buffer[strlen(Buffer) - 1] = 0;
    if ( strlen(Buffer) == 20
        && Buffer[0] == 100
        && Buffer[1] == 49
        && Buffer[2] == 115
        && Buffer[3] == 97
        && Buffer[4] == 115
        && Buffer[5] == 115
        && Buffer[6] == 51
        && Buffer[7] == 109
        && Buffer[8] == 98
        && Buffer[9] == 108
        && Buffer[10] == 49
        && Buffer[11] == 110
        && Buffer[12] == 103
        && Buffer[13] == 95
        && Buffer[14] == 49
        && Buffer[15] == 115
        && Buffer[16] == 95
        && Buffer[17] == 102
        && Buffer[18] == 117
        && Buffer[19] == 110 )
    {
        printf(v7);
    }
    else
    {
        printf(v6);
    }
    return 0;
}
```

Ce que vous pouvez voir de ce code est qu'un mot de passe semble être demandé à l'utilisateur au niveau du `fgets` (qui demande à l'utilisateur une chaîne dans le terminal). Nous avons ensuite un bloc conditionnel qui affiche `v7 = "Good password !"` si une longue liste de conditions est vérifiée et `v6 = "Wrong password !"` sinon.
Comme vous pouvez le voir, cette condition vérifie si la taille du mot de passe est de 20 caractères, puis si le premier caractère vaut `100` (ce qui fait `'d'` en ASCII) et ainsi de suite pour les autres. On retrouve donc le vrai mot de passe qui est `d1sass3mbl1ng_1s_fun` ce que l'on peut utiliser pour récupérer le flag et valider le challenge !

Notez qu'il aurait aussi été possible d'utiliser une exécution pas à pas avec `gdb` mais pour ça il vous faut un environnement qui puisse exécuter des programmes Windows (soit un OS Windows complet, soit l'outil `Wine` sous Linux).

1.2 Avec Ghidra :

Encore une fois, le mot de passe est en dur dans l'exécutable mais on compare caractère par caractère au lieu d'utiliser un `memcmp` comme on aurait pu l'avoir pour `CheckMyStrings`. L'entrée est stockée à partir de "`acStack_428[1]`". Donc le premier caractère entré correspond à "`acStack_428[1]`", le deuxième à "`acStack_428[2]`", et le troisième

“acStack_428[3]”, ensuite les comparaisons sont faites avec les char “cStack_424, ..., cStack_414”. Il ne serait pas étonnant que ces variables “cStack_$$$” soient stockées en mémoire juste après “acStack_428[3]” (et donc que ce soient les caractères suivants de l'entrée utilisateur). Le code supprime ensuite un éventuel “\n” final (remplace par “\0”). On vérifie “strlen(acStack_428) == 0x14” i.e. la longueur attendue est 0x14 = 20 caractères. Ensuite il réalise une série de comparaisons caractère par caractère :

```
int __cdecl _main ( int _Argc , char ** _Argv , char ** _Env ) {
    FILE * pFVar1;
    int iVar2;
    char acStack_428 [4];
    char cStack_424;
    char cStack_423;
    char cStack_422;
    char cStack_421;
    char cStack_420;
    char cStack_41f;
    char cStack_41e;
    char cStack_41d;
    char cStack_41c;
    char cStack_41b;
    char cStack_41a;
    char cStack_419;
    char cStack_418;
    char cStack_417;
    char cStack_416;
    char cStack_415;
    char cStack_414;
    char local_3f [59];

    ___main ();
    builtin_strncpy ( local_3f + 0x23 ," Password : " ,0xc );
    builtin_strncpy ( local_3f + 0x12 ," Good password !\ n " ,0x11 );
    builtin_strncpy ( local_3f ," Wrong password !\ n " ,0x12 );
    printf ( local_3f + 0x23 );
    pFVar1 = ( FILE *) (*( code *) __imp____acrt_iob_func ) (0);
    fgets ( acStack_428 + 1 ,999 , pFVar1 );
    iVar2 = strlen ( acStack_428 + 1);
    if ( acStack_428 [ iVar2 ] == '\n') {
        iVar2 = strlen ( acStack_428 + 1);
        acStack_428 [ iVar2 ] = '\0';
    }
    iVar2 = strlen ( acStack_428 + 1);
    if ((((((( iVar2 == 0x14 ) && ( acStack_428 [1] == 'd') ) && ( acStack_428 [2] == '1') ) && (((
    acStack_428 [3] == 's' && ( cStack_424 == 'a') ) && (( cStack_423 == 's' && (( cStack_422 == 's'
     && ( cStack_421 == '3') ) ) ) ) ) ) ) && (( cStack_420 == 'm' && ((( cStack_41f == 'b' && (
    cStack_41e == 'l') ) && ( cStack_41d == '1') ) ) ) ) && ((( cStack_41c == 'n' && ( cStack_41b
    == 'g') ) && ((( cStack_41a == '_' && (( cStack_419 == '1' && ( cStack_418 == 's') ) ) ) && (
    cStack_417 == '_') ) ) ) ) ) && ((( cStack_416 == 'f' && ( cStack_415 == 'u') ) && ( cStack_414
    == 'n') ) ) ) {
        printf ( local_3f + 0x12 );
    } else {
        printf ( local_3f );
    }
    return 0;
}
```

Grâce à cela, on peut voir le flag qui est d1sass3mbl1ng_1s_fun !