Attack Phase

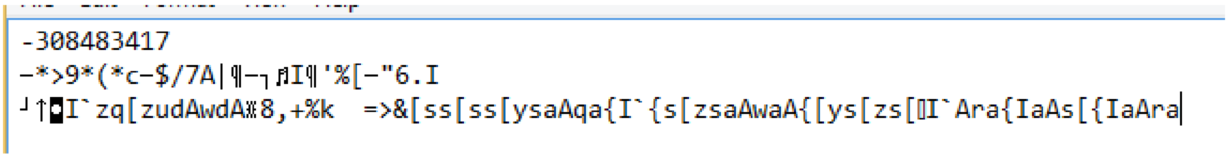
- Assumptions

I decided to pretend to be a player that owns the "game" and want to cheat by increasing my character stats to impossible levels (1 million dmg for example). I've already played the game a lot so I know some of the things that are going on.

- Process – Dynamic Analysis - Files

First, I find where the game saves the files. There is a player.char and an enemy.char which I believe to be the saved information for both the character I'm playing as, and the enemy I last fought.

Opening the files revels a large number on the first line, and below it a random slough of characters. Both files appear to have this structure. Playing the game several times and having it save different characters information reveals that the structure is always the same. Having it save the same character over and over again produces the same save file.



```
-308483417
-*>9*(*c-$/7A|¶-┐♬I¶'%[-"6.I
┘↑◘I`zq[zudAwdA¤8,+%k  =>&[ss[ss[ysaAqa{I`{s[zsaAwaA{[ys[zs[▯I`Ara{IaAs[{IaAra
```

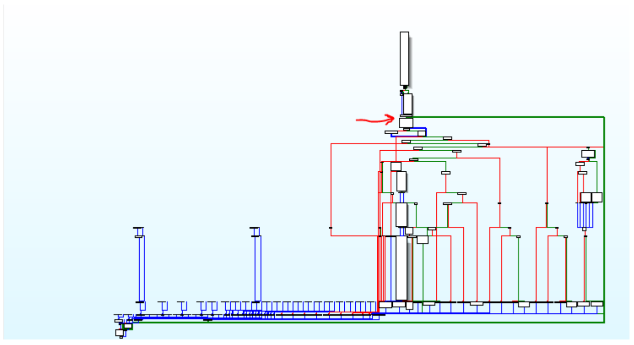
There doesn't appear to be any recognizable info that could distinguish what the character information pertains to. Therefore, the information must be encrypted. The first line is always a number and everything after it is random. These might be two distinct parts to the file.

Modifying the file in any way causes the file to not be loaded and start a default character when the game starts again – removing any progress I might have had for the character file, and creating a very difficult enemy for the enemy file.

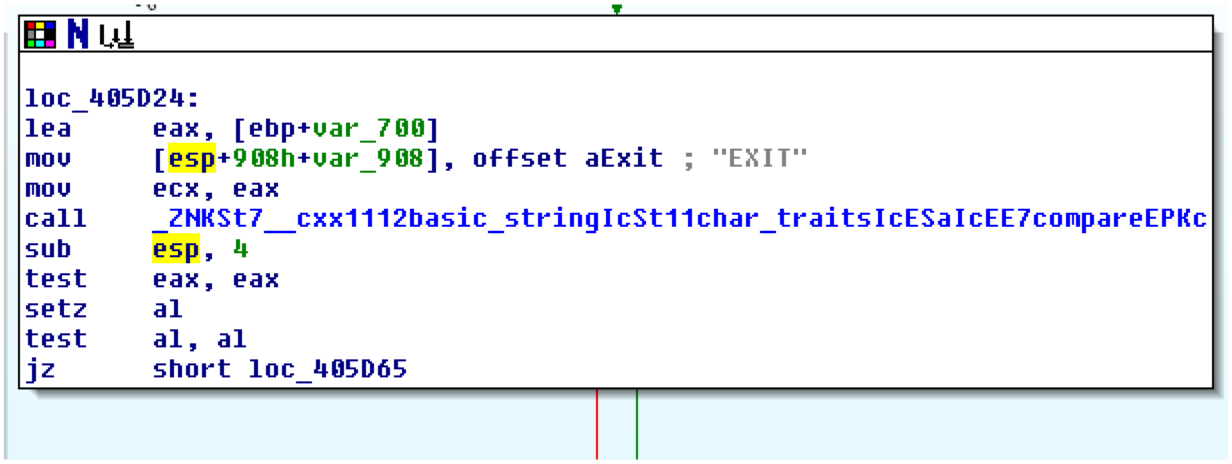
The files do not appear to change until after the game is closed

- Process – Static Analysis - Disassemble

After compiling the code I disassembled it using ida-pro. Knowing the names of the files I looked for them in the strings window. Finding them they were only used once. Going to them we find ourselves at the top of "main". There is a lot here so lets skip it for now.

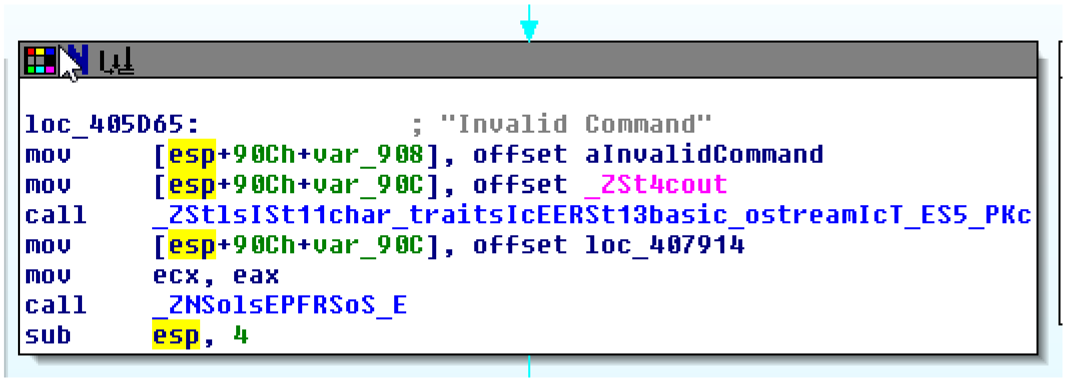


Since the game is using a command structure, and knowing that the files don't chance till the game exists I decided to look for the "EXIT" command. Finding it in the strings windows and going to it we find this



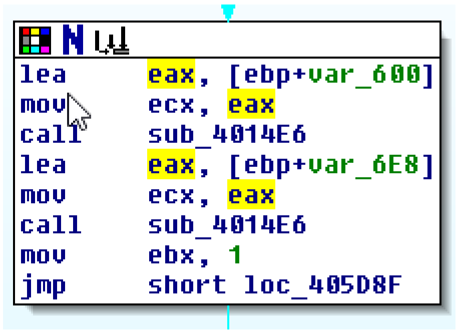
```
loc_405D24:
lea     eax, [ebp+var_700]
mov     [esp+908h+var_908], offset aExit ; "EXIT"
mov     ecx, eax
call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE7compareEPKc
sub     esp, 4
test    eax, eax
setz    al
test    al, al
jz      short loc_405D65
```

The call to _ZNK… in this function is called in numerous places and it is most likely comparing whether the given command matches "EXIT". Looking further it is also called for the other commands. Going along the "true" branch



```
loc_405D65:                 ; "Invalid Command"
mov     [esp+90Ch+var_908], offset aInvalidCommand
mov     [esp+90Ch+var_90C], offset _ZSt4cout
call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
mov     [esp+90Ch+var_90C], offset loc_407914
mov     ecx, eax
call    _ZNSolsEPFRSoS_E
sub     esp, 4
```

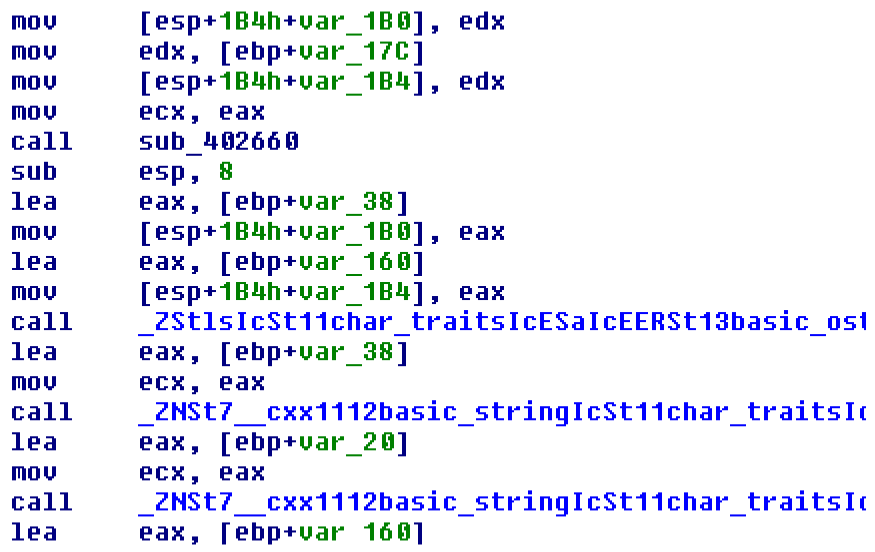
We find that it leads to the "Invalid Command" prompt for when no valid command was given. Going along the false instead



```
lea     eax, [ebp+var_600]
mov     ecx, eax
call    sub_4014E6
lea     eax, [ebp+var_6E8]
mov     ecx, eax
call    sub_4014E6
mov     ebx, 1
jmp     short loc_405D8F
```

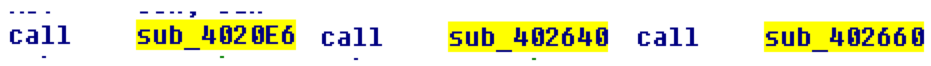
We find two calls to the same function. Since it is called twice with different "var_6xx" there is a decent chance that this is our save function. Jumping further down we find that it loops around back to the top for "commands".

Exploring the function, we find nothing very helpful at a first glance.



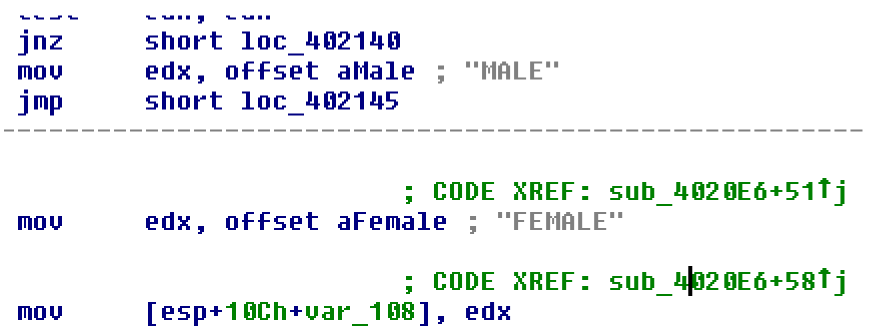
```
mov     [esp+1B4h+var_1B0], edx
mov     edx, [ebp+var_17C]
mov     [esp+1B4h+var_1B4], edx
mov     ecx, eax
call    sub_402660
sub     esp, 8
lea     eax, [ebp+var_38]
mov     [esp+1B4h+var_1B0], eax
lea     eax, [ebp+var_160]
mov     [esp+1B4h+var_1B4], eax
call    _ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ost
lea     eax, [ebp+var_38]
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIc
lea     eax, [ebp+var_20]
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIc
lea     eax, [ebp+var_160]
```

However, looking at the sub routines called there were three of note



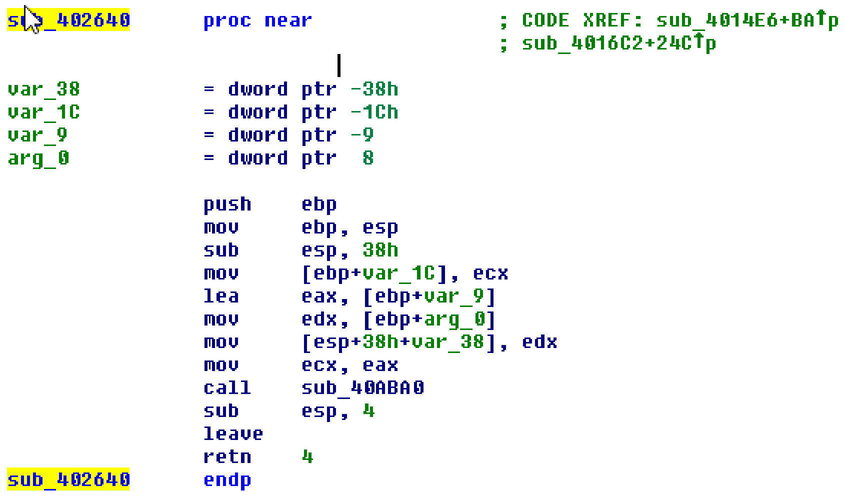
```
call    sub_4020E6   call    sub_402640   call    sub_402660
```

Looking at the first we find a large amount of movement of variables, and even some key strings such as MALE, FEMALE, PHYSICAL, MAGIC.



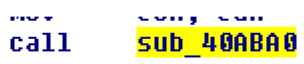
```
jnz     short loc_402140
mov     edx, offset aMale ; "MALE"
jmp     short loc_402145
-------------------------------------------------------

                         ; CODE XREF: sub_4020E6+51↑j
mov     edx, offset aFemale ; "FEMALE"

                         ; CODE XREF: sub_4020E6+58↑j
mov     [esp+10Ch+var_108], edx
```

The second function is very short



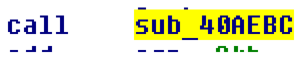
```
sub_402640      proc near               ; CODE XREF: sub_4014E6+BA↑p
                                        ; sub_4016C2+24C↑p

var_38          = dword ptr -38h
var_1C          = dword ptr -1Ch
var_9           = dword ptr -9
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 38h
                mov     [ebp+var_1C], ecx
                lea     eax, [ebp+var_9]
                mov     edx, [ebp+arg_0]
                mov     [esp+38h+var_38], edx
                mov     ecx, eax
                call    sub_40ABA0
                sub     esp, 4
                leave
                retn    4
sub_402640      endp
```

Exploring the only call inside



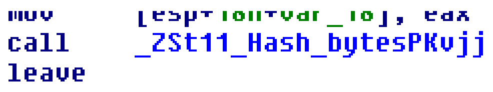
```
call    sub_40ABA0
```

And again, with the next one



```
call    sub_40AEBC
```

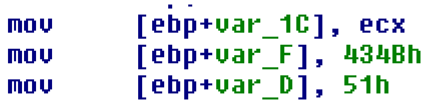
We find a hashing function



```
mov     [esp+18h+var_18], eax
call    _ZSt11_Hash_bytesPKvjj
leave
```

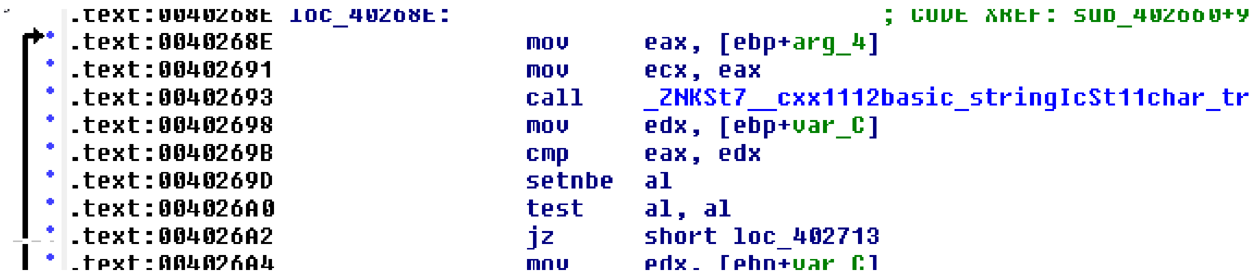
Thus, we can assume the second function is to hash something.

Exploring the last sub-routine we find an array with some values and a relatively large loop

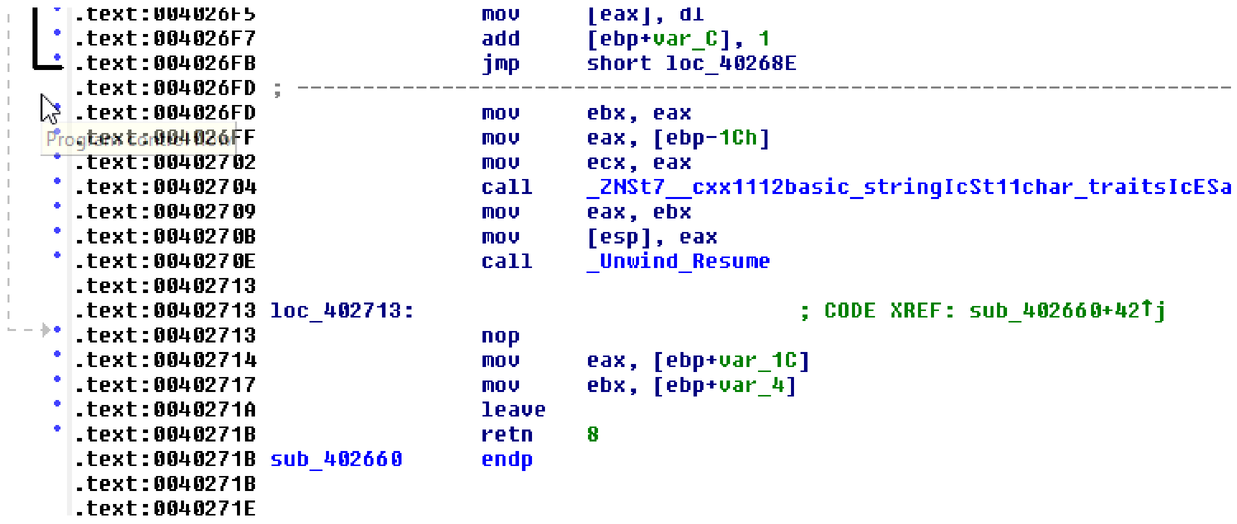


```
mov     [ebp+var_1C], ecx
mov     [ebp+var_F], 434Bh
mov     [ebp+var_D], 51h
```

CK, Q



```
.text:0040268E loc_40268E:                           ; CODE XREF: sub_402660+9
.text:0040268E                 mov     eax, [ebp+arg_4]
.text:00402691                 mov     ecx, eax
.text:00402693                 call    _ZNKSt7__cxx1112basic_stringIcSt11char_tr
.text:00402698                 mov     edx, [ebp+var_C]
.text:0040269B                 cmp     eax, edx
.text:0040269D                 setnbe  al
.text:004026A0                 test    al, al
.text:004026A2                 jz      short loc_402713
.text:004026A4                 mov     edx, [ebp+var_C]
```



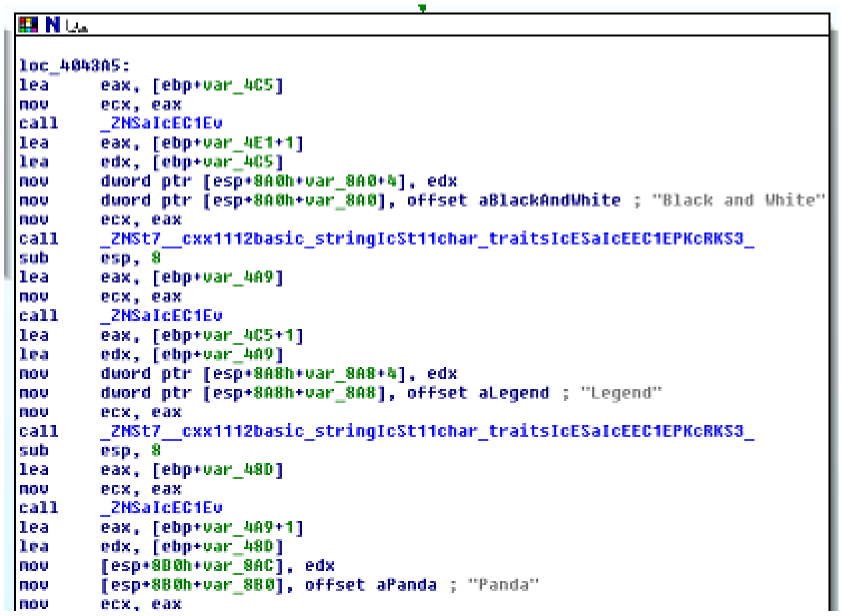
```
.text:004026F5                 mov     [eax], dl
.text:004026F7                 add     [ebp+var_C], 1
.text:004026FB                 jmp     short loc_40268E
.text:004026FD ; ---------------------------------------------------------------------------
.text:004026FD                 mov     ebx, eax
.text:004026FF                 mov     eax, [ebp-1Ch]
.text:00402702                 mov     ecx, eax
.text:00402704                 call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESa
.text:00402709                 mov     eax, ebx
.text:0040270B                 mov     [esp], eax
.text:0040270E                 call    _Unwind_Resume
.text:00402713
.text:00402713 loc_402713:                           ; CODE XREF: sub_402660+42↑j
.text:00402713                 nop
.text:00402714                 mov     eax, [ebp+var_1C]
.text:00402717                 mov     ebx, [ebp+var_4]
.text:0040271A                 leave
.text:0040271B                 retn    8
.text:0040271B sub_402660      endp
.text:0040271B
.text:0040271E
```

Looking at it, it is not very clear as to what it is doing.

However, looking at the combination of the three functions, we can probably assume that the first function gets the character information, the 2nd hashes it, and then the 3rd encrypts it. This also seems to follow the structure of the file we found. Where the hash value is the first line, and the encrypted info is the rest.

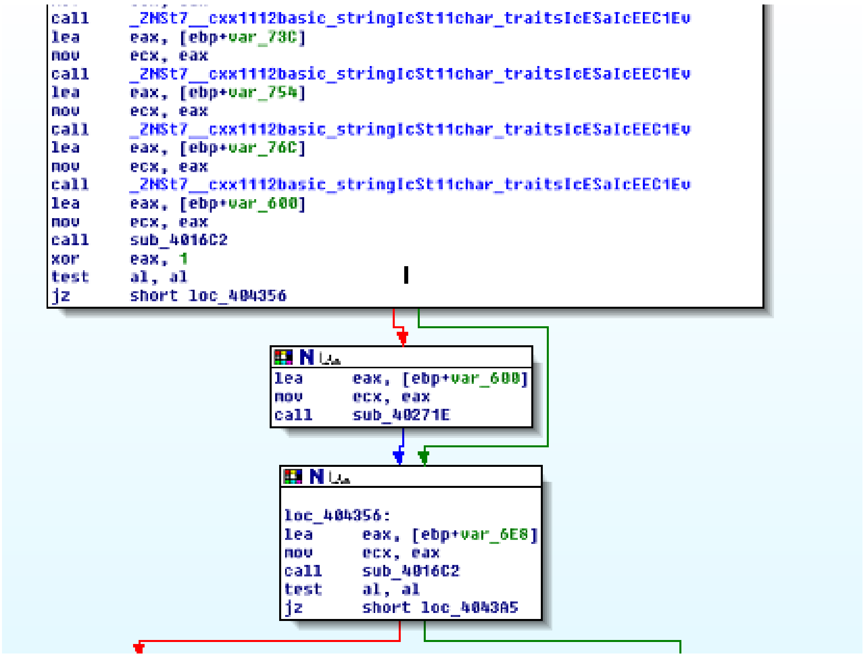
So far we've only looked at the saving function, and since the files are loaded we can see how it might be decrypted. Since the character seems to be loaded at the start of the program we go to the top of main. We do this easily by finding string "player.char" as before.

Since the loading is most likely complete before commands are taken from the user, we only need to look at the sections between "player.char" and "Commands". Looking through we find the very recognizable legendary panda the is made when the enemy fails to load.



```
loc_4043A5:
lea     eax, [ebp+var_4C5]
mov     ecx, eax
call    _ZNSaIcEC1Ev
lea     eax, [ebp+var_4E1+1]
lea     edx, [ebp+var_4C5]
mov     dword ptr [esp+8A0h+var_8A0+4], edx
mov     dword ptr [esp+8A0h+var_8A0], offset aBlackAndWhite ; "Black and White"
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1EPKcRKS3_
sub     esp, 8
lea     eax, [ebp+var_4A9]
mov     ecx, eax
call    _ZNSaIcEC1Ev
lea     eax, [ebp+var_4C5+1]
lea     edx, [ebp+var_4A9]
mov     dword ptr [esp+8A8h+var_8A8+4], edx
mov     dword ptr [esp+8A8h+var_8A8], offset aLegend ; "Legend"
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1EPKcRKS3_
sub     esp, 8
lea     eax, [ebp+var_48D]
mov     ecx, eax
call    _ZNSaIcEC1Ev
lea     eax, [ebp+var_4A9+1]
lea     edx, [ebp+var_48D]
mov     [esp+8B0h+var_8AC], edx
mov     [esp+8B0h+var_8B0], offset aPanda ; "Panda"
mov     ecx, eax
```

Knowing the enemy has already failed to load we look before the panda enemy is created



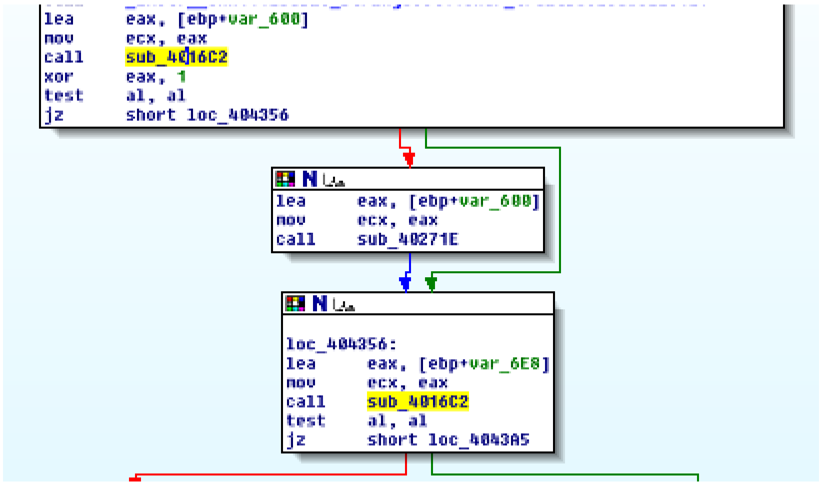
```
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
lea     eax, [ebp+var_73C]
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
lea     eax, [ebp+var_754]
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
lea     eax, [ebp+var_76C]
mov     ecx, eax
call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev
lea     eax, [ebp+var_600]
mov     ecx, eax
call    sub_4016C2
xor     eax, 1
test    al, al
jz      short loc_404356
```

```
lea     eax, [ebp+var_600]
mov     ecx, eax
call    sub_40271E
```

```
loc_404356:
lea     eax, [ebp+var_6E8]
mov     ecx, eax
call    sub_4016C2
test    al, al
jz      short loc_4043A5
```

Where we have the beginning of main, and two if statements, where the final green goes to the panda creation

Looking at the subroutines we find that one is called twice



```
lea     eax, [ebp+var_600]
mov     ecx, eax
call    sub_4016C2
xor     eax, 1
test    al, al
jz      short loc_404356
```

```
lea     eax, [ebp+var_600]
mov     ecx, eax
call    sub_40271E
```

```
loc_404356:
lea     eax, [ebp+var_6E8]
mov     ecx, eax
call    sub_4016C2
test    al, al
jz      short loc_4043A5
```

Looking inside we find two notable subroutines

```
call    sub_402660 call    sub_402640
```

Which are the encryption and hashing functions we identified previously. This means that loading and saveing file use the same method to both encrypt and decrypt the character information.

- Final Thoughts

With the given analysis above, we know that the first line of each file is a hash of the data, and that the remaining characters after it is the encrypted data of the character. Encryption/Decryption seem to belong to the same function, and both this and the hash function are used for loading and saving the character info.

Since most hashes are irreversable we can assume this is a type of "check" to ensure that the information in the files can't be tampered with. This follows how a diffiuclt enemy or a default character shows up when we attempted to modify the files.

Since the encypriton/decryption method is the same for both save and load, we can also assume it is something like an XOR encryption.

- Breaking the files

If we just want to see the encrypted information we can just run the file through the method again and print it out somehow.

If we want to modify the data for our use we would need to decrypt the data, modify any values if possible, and get the correct hash of the new data. However it is unclear if the hash is of the encrypted or decyprted version.

Attempting to modify values and calls and such as proven unsuccessful (due to my lack of knowledge)