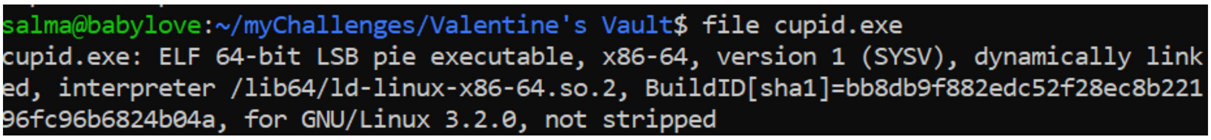
‘Valentine’s Vault’ Reversing Walkthrough

Tools i will be using :

- GDB
- Decompiler (IDA, Ghidra ...)
- Scripting Language (Python..)
- Text Editor for note taking

First reflexes :

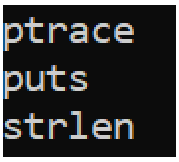
-File :



```
salma@babylove:~/myChallenges/Valentine's Vault$ file cupid.exe
cupid.exe: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically link
ed, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=bb8db9f882edc52f28ec8b221
96fc96b6824b04a, for GNU/Linux 3.2.0, not stripped
```

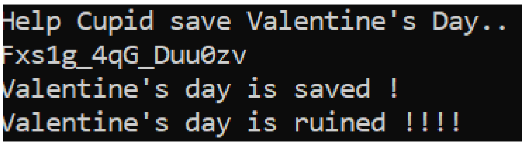
-Strings :

After executing *strings cupid.exe*, we will start noticing somethings :



```
ptrace
puts
strlen
```

There is ptrace among the functions.



```
Help Cupid save Valentine's Day..
Fxs1g_4qG_Duu0zv
Valentine's day is saved !
Valentine's day is ruined !!!!
```

There is a string that seems suspicious, it will surely be used to create the flag.

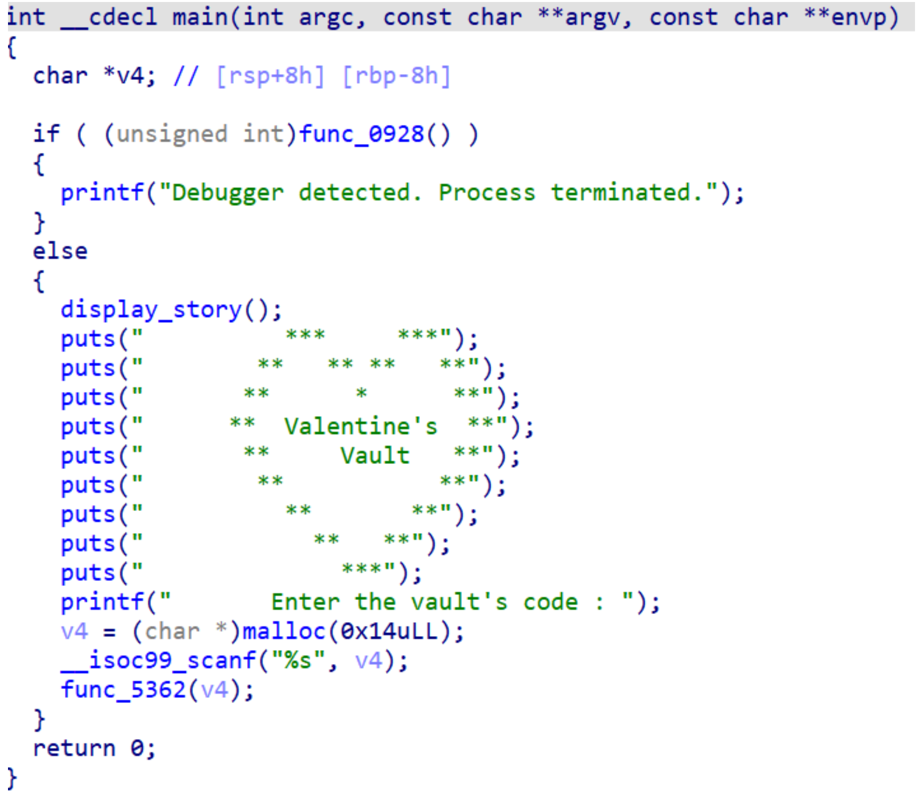
-Executing :

We proceed by giving the right to execute by running *chmod +x cupid.exe*
Then run the process using *./cupid.exe*

The process will then show the story line by line and will wait for an input : “The vault’s code”
Either that code is the flag or entering the right code will show the flag.

To get that information we are going to need another tool. Let’s try decompiling the exe file using IDA PRO decompiler.

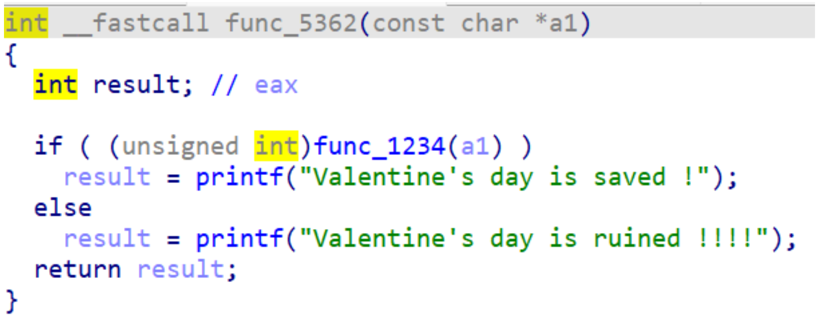
Decompiling + Static Analysis :



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char *v4; // [rsp+8h] [rbp-8h]

  if ( (unsigned int)func_0928() )
  {
    printf("Debugger detected. Process terminated.");
  }
  else
  {
    display_story();
    puts("          ***     ***");
    puts("        **   ** **   **");
    puts("       **      *      **");
    puts("      **  Valentine's  **");
    puts("       **     Vault   **");
    puts("        **           **");
    puts("          **       **");
    puts("            **   **");
    puts("              ***");
    printf("        Enter the vault's code : ");
    v4 = (char *)malloc(0x14uLL);
    __isoc99_scanf("%s", v4);
    func_5362(v4);
  }
  return 0;
}
```

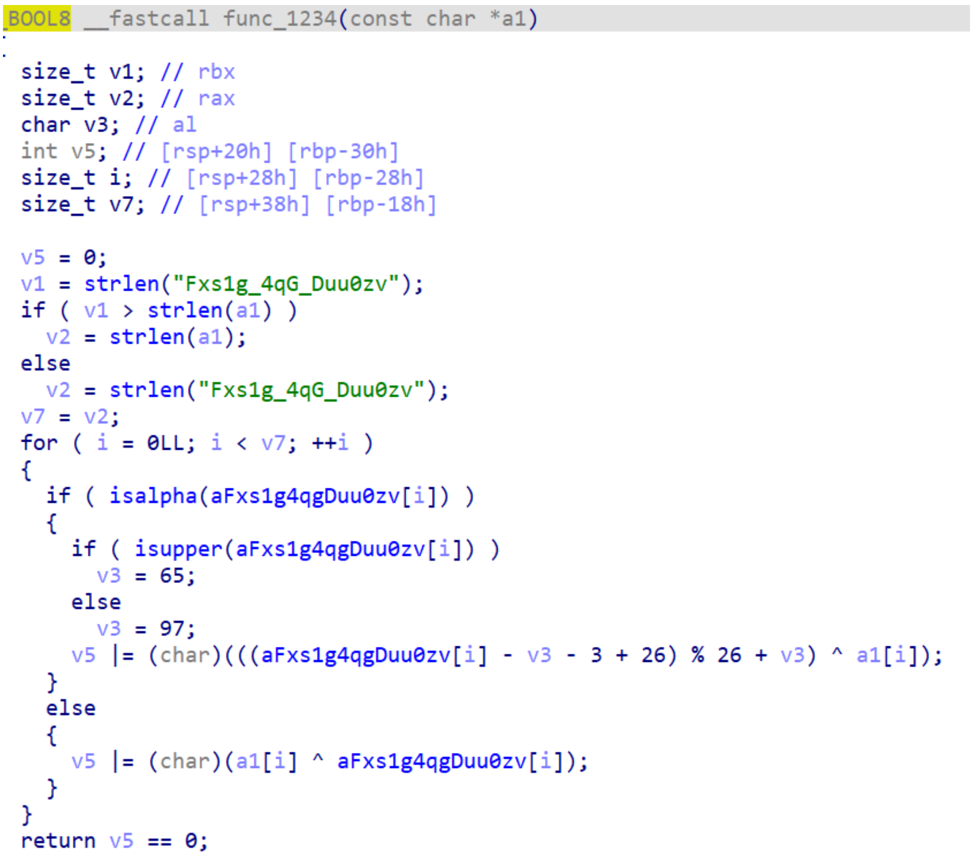
This is the main function decompiled. We notice that our input is given as an argument to the function *func_5362*

Here’s the function :



```
int __fastcall func_5362(const char *a1)
{
  int result; // eax

  if ( (unsigned int)func_1234(a1) )
    result = printf("Valentine's day is saved !");
  else
    result = printf("Valentine's day is ruined !!!!");
  return result;
}
```

The parameter *a1* is an argument for the function *func_1234*

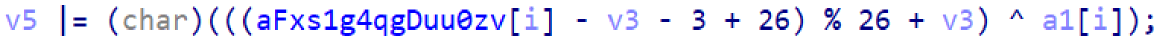


```
BOOL8 __fastcall func_1234(const char *a1)

  size_t v1; // rbx
  size_t v2; // rax
  char v3; // al
  int v5; // [rsp+20h] [rbp-30h]
  size_t i; // [rsp+28h] [rbp-28h]
  size_t v7; // [rsp+38h] [rbp-18h]

  v5 = 0;
  v1 = strlen("Fxs1g_4qG_Duu0zv");
  if ( v1 > strlen(a1) )
    v2 = strlen(a1);
  else
    v2 = strlen("Fxs1g_4qG_Duu0zv");
  v7 = v2;
  for ( i = 0LL; i < v7; ++i )
  {
    if ( isalpha(aFxs1g4qgDuu0zv[i]) )
    {
      if ( isupper(aFxs1g4qgDuu0zv[i]) )
        v3 = 65;
      else
        v3 = 97;
      v5 |= (char)(((aFxs1g4qgDuu0zv[i] - v3 - 3 + 26) % 26 + v3) ^ a1[i]);
    }
    else
    {
      v5 |= (char)(a1[i] ^ aFxs1g4qgDuu0zv[i]);
    }
  }
  return v5 == 0;
```

We can now say that we have found our box : *func_1234* !
This function is generating dynamically decrypting the flag from our string 'Fxs1g_4qG_Duu0zv' and comparing character by character.
We notice also that it is not using *strcmp* or == to compare but *xor* and *or*.

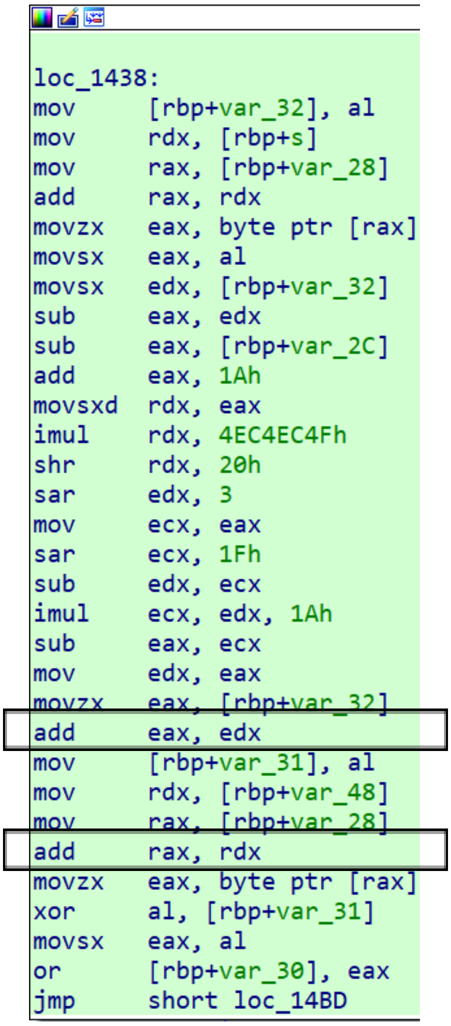
Let's remember that :
a ^ a = 0
a | 0 = a
So we will xor the characters from each string to see if they are equal (return 0) and we will 'or' all the elements so that if one isn't 0, the result isn't 0 too.

In the for block, there's a condition. If the character isn't a letter, it stays the same as we see in the else block : making direct comparison using xor between the argument and the string.
But if the character is a letter however, this is the line to notice :



```
v5 |= (char)(((aFxs1g4qgDuu0zv[i] - v3 - 3 + 26) % 26 + v3) ^ a1[i]);
```

So our code is hiding behind that line, more specifically behind the first part of it.
Let’s take a look at that function from the disassembler view of that part. Here’s th *if* part in the for loop



```
loc_1438:
mov     [rbp+var_32], al
mov     rdx, [rbp+s]
mov     rax, [rbp+var_28]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
movsx   edx, [rbp+var_32]
sub     eax, edx
sub     eax, [rbp+var_2C]
add     eax, 1Ah
movsxd  rdx, eax
imul    rdx, 4EC4EC4Fh
shr     rdx, 20h
sar     edx, 3
mov     ecx, eax
sar     ecx, 1Fh
sub     edx, ecx
imul    ecx, edx, 1Ah
sub     eax, ecx
mov     edx, eax
movzx   eax, [rbp+var_32]
add     eax, edx
mov     [rbp+var_31], al
mov     rdx, [rbp+var_48]
mov     rax, [rbp+var_28]
add     rax, rdx
movzx   eax, byte ptr [rax]
xor     al, [rbp+var_31]
movsx   eax, al
or      [rbp+var_30], eax
jmp     short loc_14BD
```

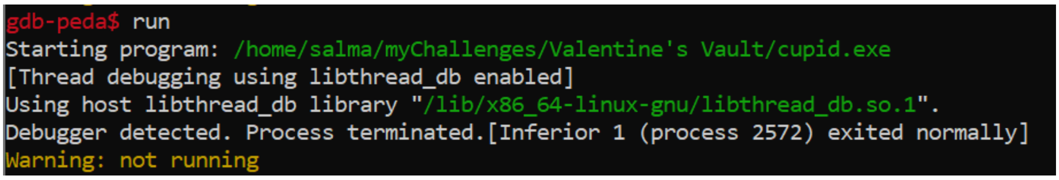
Since our last operation in the block that we are looking into is *add*, let’s find the last *add* operations after a *sub*

We shouldn’t forget that the results of calculations and functions are always in the eax register

Debugging + Dynamic Analysis :

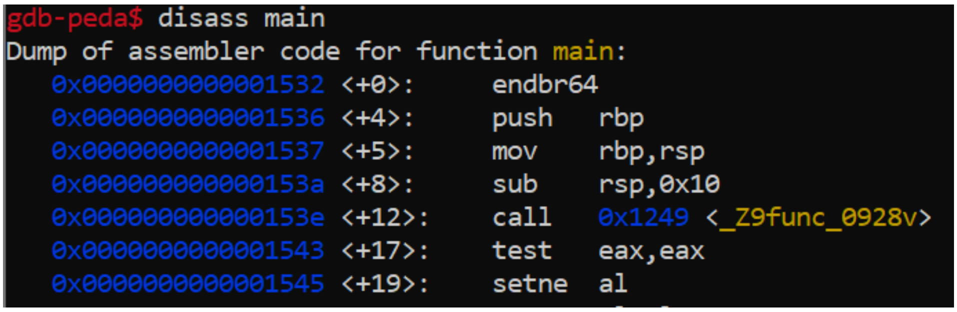
-Disabling *ptrace* :

If we try running the exe :



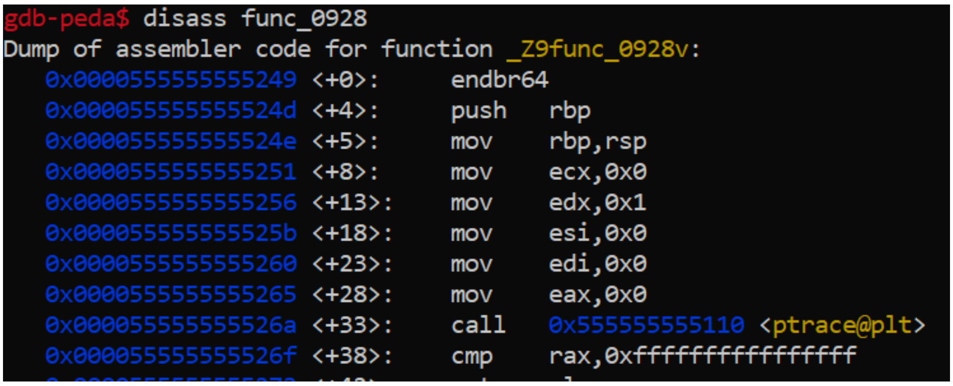
```
gdb-peda$ run
Starting program: /home/salma/myChallenges/Valentine's Vault/cupid.exe
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Debugger detected. Process terminated.[Inferior 1 (process 2572) exited normally]
Warning: not running
```

Let’s take a look at the main function using *disass main* in *gdb*.



```
gdb-peda$ disass main
Dump of assembler code for function main:
   0x0000000000001532 <+0>:	endbr64
   0x0000000000001536 <+4>:	push   rbp
   0x0000000000001537 <+5>:	mov    rbp,rsp
   0x000000000000153a <+8>:	sub    rsp,0x10
   0x000000000000153e <+12>:	call   0x1249 <_Z9func_0928v>
   0x0000000000001543 <+17>:	test   eax,eax
   0x0000000000001545 <+19>:	setne  al
```

There’s a call of a function in the first lines followed by a comparison of eax



```
gdb-peda$ disass func_0928
Dump of assembler code for function _Z9func_0928v:
   0x0000555555555249 <+0>:	endbr64
   0x000055555555524d <+4>:	push   rbp
   0x000055555555524e <+5>:	mov    rbp,rsp
   0x0000555555555251 <+8>:	mov    ecx,0x0
   0x0000555555555256 <+13>:	mov    edx,0x1
   0x000055555555525b <+18>:	mov    esi,0x0
   0x0000555555555260 <+23>:	mov    edi,0x0
   0x0000555555555265 <+28>:	mov    eax,0x0
   0x000055555555526a <+33>:	call   0x555555555110 <ptrace@plt>
   0x000055555555526f <+38>:	cmp    rax,0xffffffffffffffff
```

We clearly see that this is the function that uses ptrace. So what we are going to do is that we will set a breakpoint before the *test eax,eax* and set *eax* register to zero so we will be executing this :

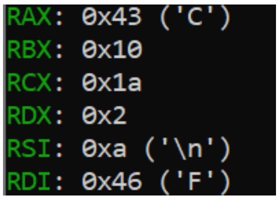
break main
*break *0x1543*
run
set $eax=0
continue

And just like that we have taken down our first obstacle, the ptrace function.

-Debugging with GDB :

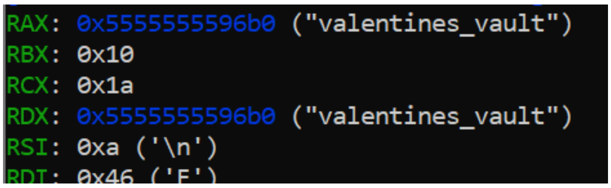
Now we can debug our program normally. Let's set breakpoints on the two ‘add’ that we saw earlier and run and inspect registers to see which one contains the letter of the flag.

We run *disass func_1234* then set the breakpoints after the add . Then we run *continue*



```
RAX: 0x43 ('C')
RBX: 0x10
RCX: 0x1a
RDX: 0x2
RSI: 0xa ('\n')
RDI: 0x46 ('F')
```

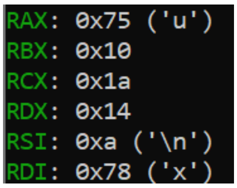
Here’s the registers on the first breakpoint



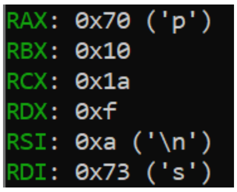
```
RAX: 0x5555555596b0 ("valentines_vault")
RBX: 0x10
RCX: 0x1a
RDX: 0x5555555596b0 ("valentines_vault")
RSI: 0xa ('\n')
RDI: 0x46 ('F')
```

Here’s the registers on the second one

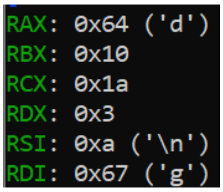
As we see *rax* has an unknown letter on the first breakpoint, it should be the first letter of the code. and there’s the letter that symbolizes it in our string stored in *rdi* register.
Obviously, the second doesn’t have any information to give.
We continue through iteration and get the string (be careful the numbers and ‘_’ stay the same)



```
RAX: 0x75 ('u')
RBX: 0x10
RCX: 0x1a
RDX: 0x14
RSI: 0xa ('\n')
RDI: 0x78 ('x')
```



```
RAX: 0x70 ('p')
RBX: 0x10
RCX: 0x1a
RDX: 0xf
RSI: 0xa ('\n')
RDI: 0x73 ('s')
```



```
RAX: 0x64 ('d')
RBX: 0x10
RCX: 0x1a
RDX: 0x3
RSI: 0xa ('\n')
RDI: 0x67 ('g')
```

...etc

We will have the flag then.

-Other method :

From the decompiled code, you can notice that it is caesar encryption using *shift=3* so you can decrypt it using the given string and a simple python program like this :

```
def caesar_decrypt(ciphertext, shift):
        plaintext = ""
        for char in ciphertext:
            # Check if the character is a letter
            if char.isalpha():
                base = ord('A') if char.isupper() else ord('a')
                # Apply the inverse Caesar shift
                decrypted_char = chr(((ord(char) - base - shift + 26) % 26) + base)
                plaintext += decrypted_char
            else:
                # If the character is not a letter, keep it as is
                plaintext += char
        return plaintext

# Test string
ciphertext = "Fxs1g_4qG_Duu0zv"
shift = 3

# Decrypt the ciphertext
plaintext = caesar_decrypt(ciphertext, shift)

# Print the decrypted plaintext
print("Decrypted plaintext:", plaintext)
```

Thank you <3