Mortal Kombat Writeup

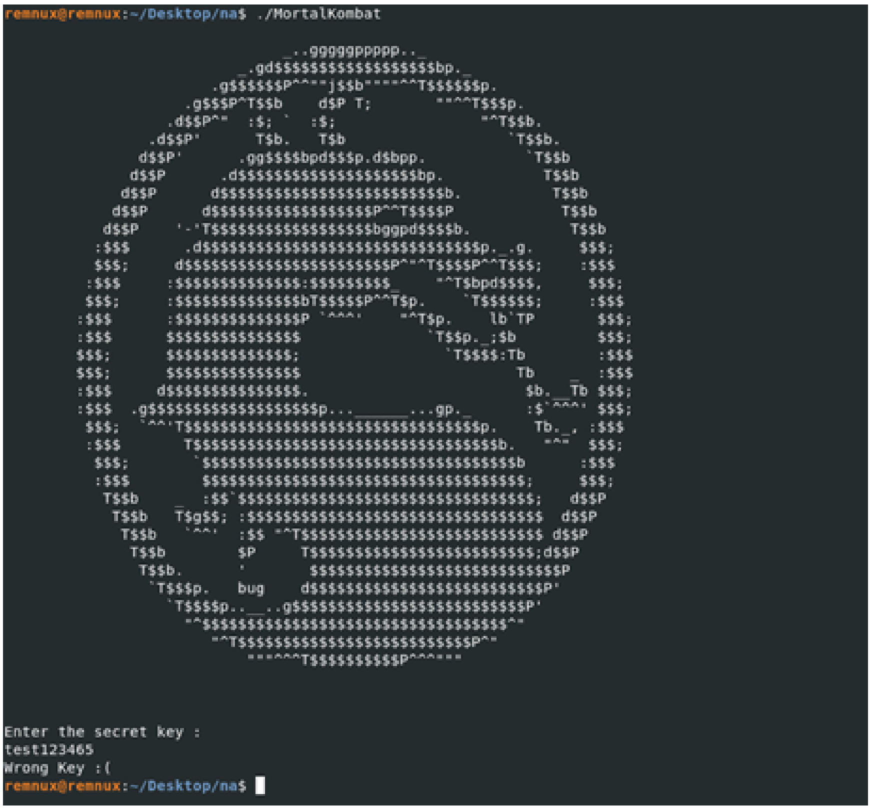
Challenge overview:

The challenge goal is to find the right secret which is the flag ,The challenge started with an ELF file which decrypts the second ELF file and executes it in-memory with memfd_create and fexec.

The second ELF creates a child process and attach the parent process to it to handle all the raised signals from the child process which is a regular nanomites behaviour.

To solve this challenge we have to have a good knowledge on what makes the process raise different signals and we have to analyze how the parent process handles these signals as this is how the secret is being decrypted and checked.

Step 1 : Discovery



We are given an x64 ELF file and our goal is to find the secret which is our flag.

Step 2 : Binary Analysis

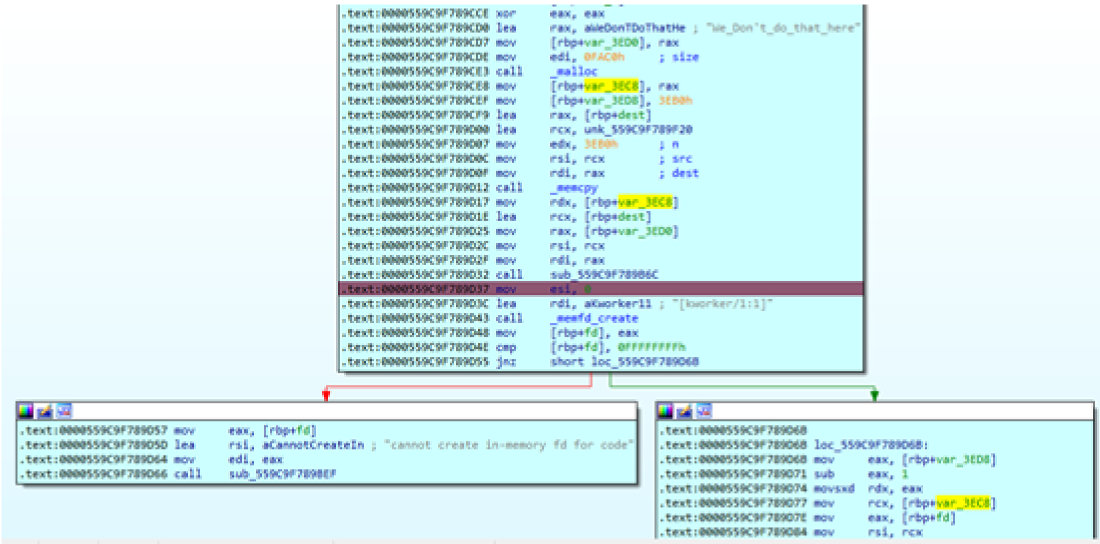
```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char *const *envp; // [rsp+8h] [rbp-3EF8h]
  unsigned int fd; // [rsp+2Ch] [rbp-3ED4h]
  void *v6; // [rsp+38h] [rbp-3EC8h]
  char dest; // [rsp+40h] [rbp-3EC0h]
  unsigned __int64 v8; // [rsp+3EF8h] [rbp-8h]

  envp = a3;
  v8 = __readfsqword(0x28u);
  v6 = malloc(0xFAC0uLL);
  memcpy(&dest, &unk_F20, 0x3EB0uLL);
  sub_B6C("We_Don't_do_that_here", &dest, v6);
  fd = memfd_create("[kworker/1:1]", 0LL);
  if ( fd == -1 )
    sub_BEF(0xFFFFFFFFLL, "cannot create in-memory fd for code");
  sub_C20(fd, v6, 16047LL);
  *a2 = "[kworker/1:1]";
  fexecve(fd, a2, envp);
  close(fd);
  return 0LL;
}
```

The first thing that seems interesting here is the call to **memfd_create** which is used to create an anonymous file and returns a file descriptor that refers to it and the call to fexecve which is similar to execve with the difference that the file to be executed is specified via a file descriptor.

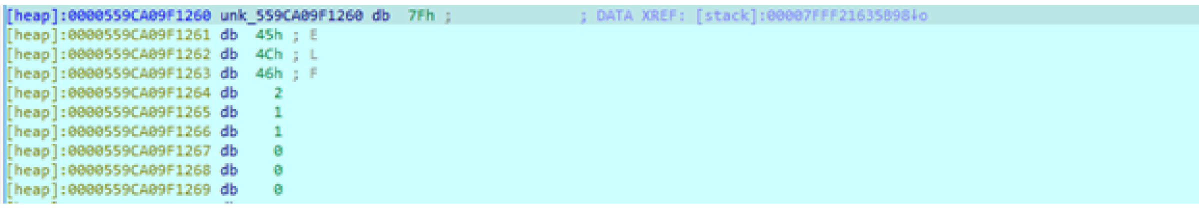
Doing further analysis we will find that **sub_B6C** is an **RC4** function which will be used to decrypt the encrypted second elf file and after that the decrypted data will be written to the created file and after that execute it using fexecve.

The easiest way to extract the second ELF file is to debug the program and put a breakpoint after the RC4 function then dump the decrypted ELF file from memory.

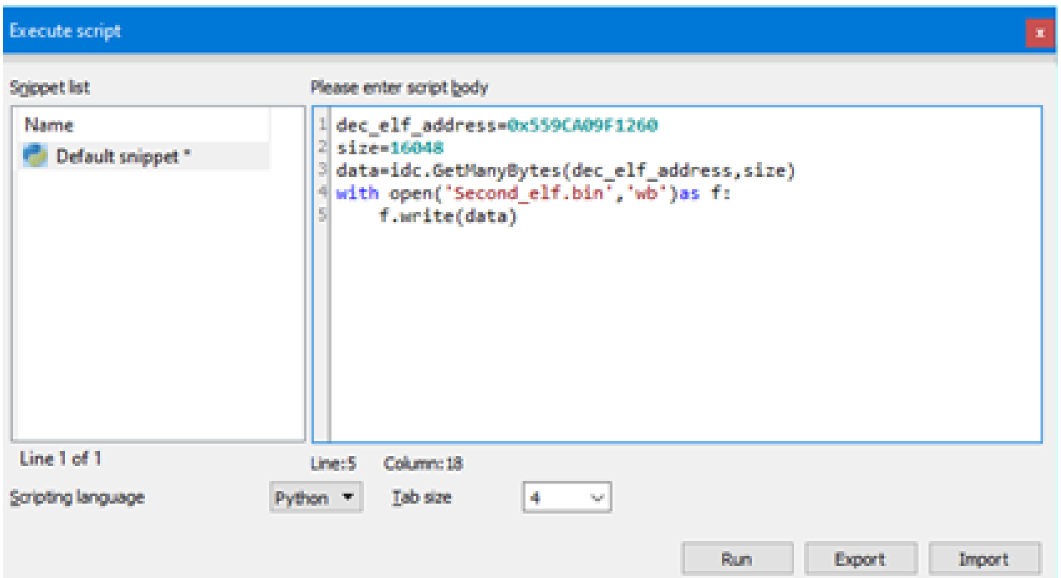
Step 3 : Dumping the ELF file



The decrypted ELF file content will be written in the allocated memory and the address of the allocated memory will be in **rbp+var_3EC8** .

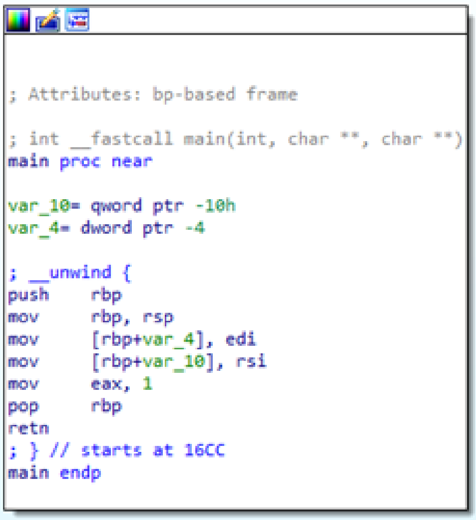


Following the location we will find the magic bytes for an ELF file so we can use ida python to dump the ELF file from memory.

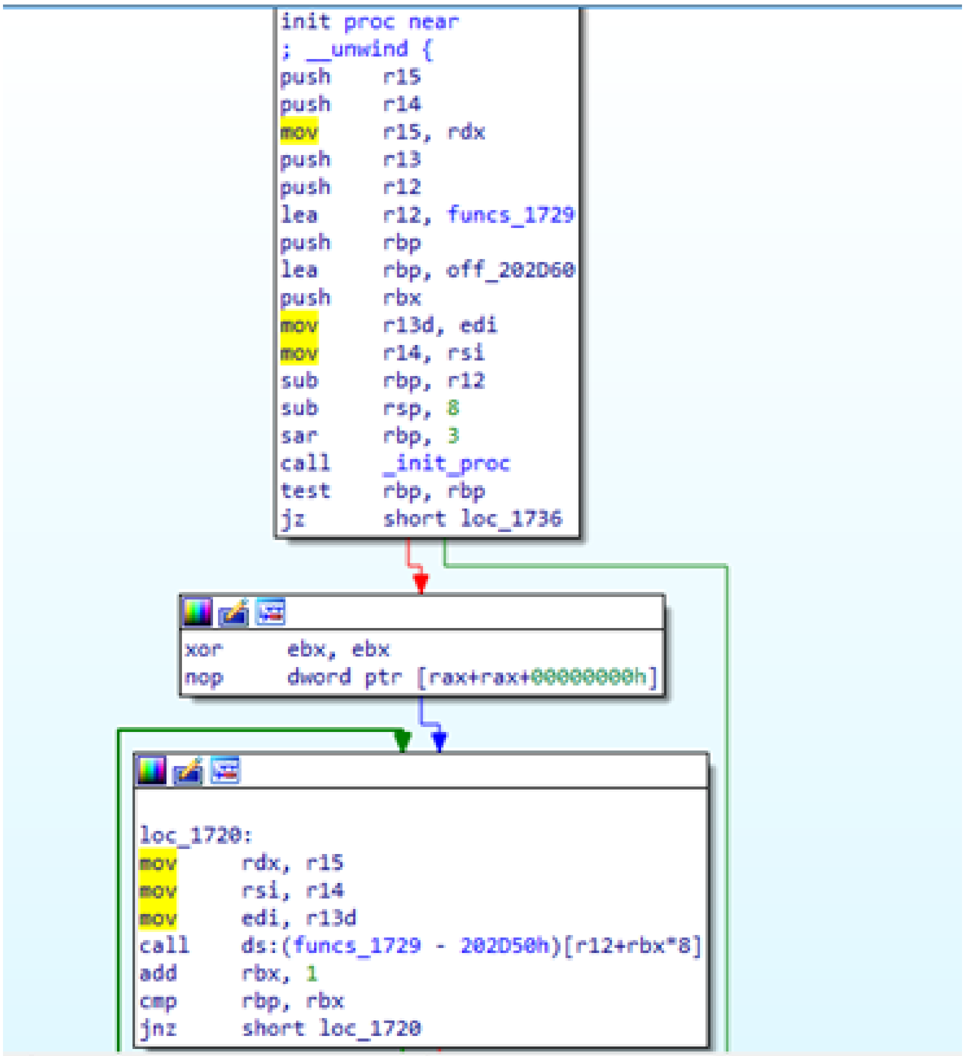


Step 4 : Analyzing the Second ELF file

Running the ELF file we will see the same **ascii** art and wait for our input so we can move the analysis part.

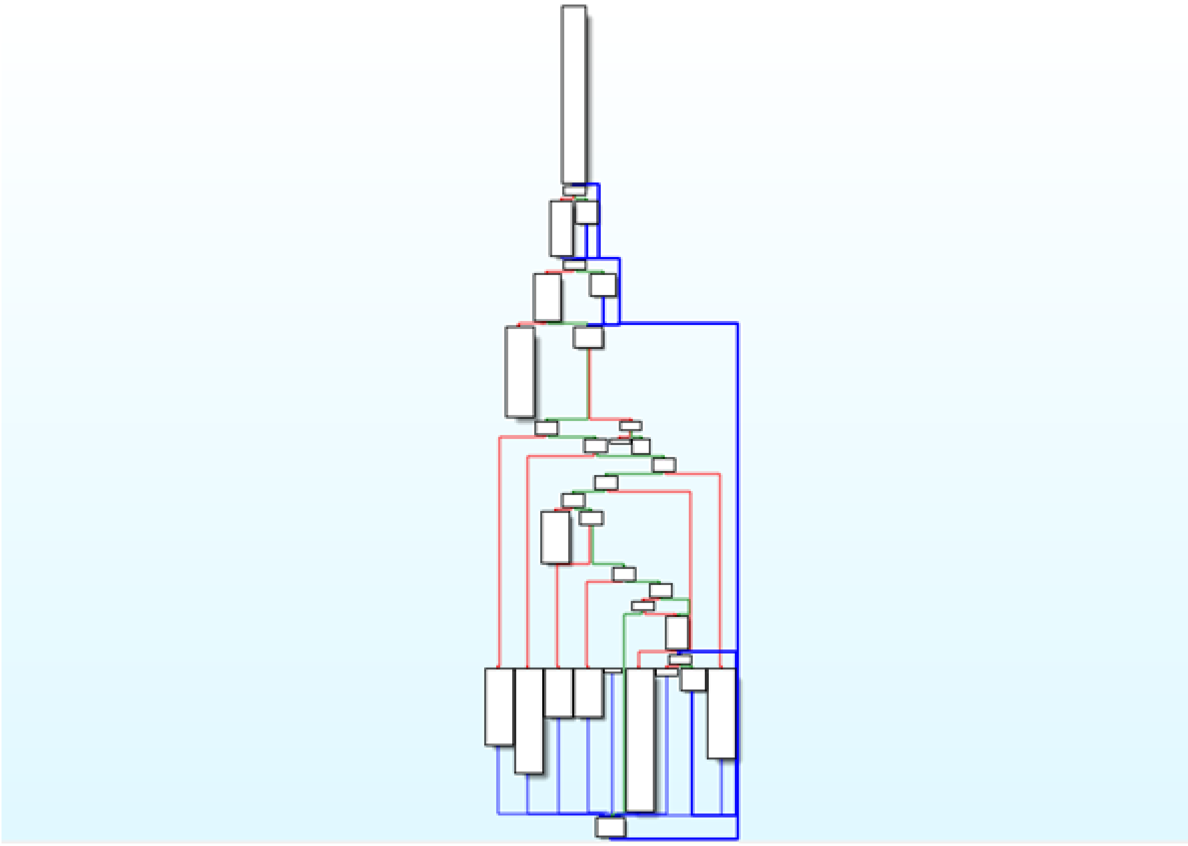


Checking the main function we will notice that there is no code here so in this case we can check the **init** function to see if there are any unregular functions that's being executed before the main function.

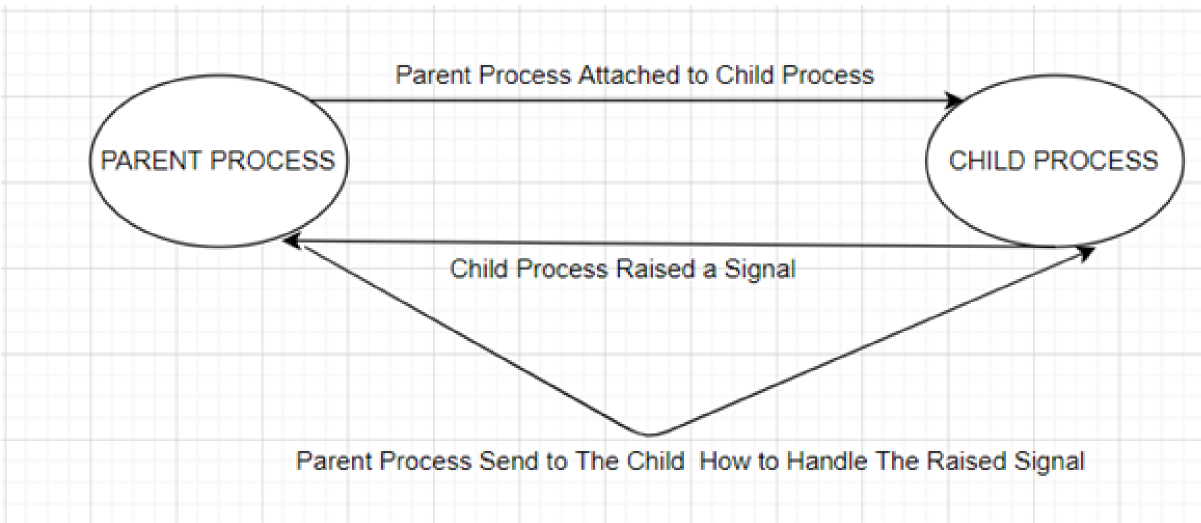



```
.init_array:0000000000202D50 funcs_1729      dq offset sub_A20        ; DATA XREF: LOAD:00000000000000F8↑o
.init_array:0000000000202D50                                          ; LOAD:0000000000000210↑o ...
.init_array:0000000000202D58                 dq offset sub_C4D
.init_array:0000000000202D58 _init_array     ends
.init_array:0000000000202D58
```

We can see that there is two functions is being executed in the init function **sub_A20** which is regular function and **sub_C4D** which is the function we are interesting in.



We see a lot of stuff happening here but before analyzing the function let's take a quick look on how **nanomites** work to give us a better understanding on what we are dealing with.



Nanomites usually consist of two or more processes

- one process will work as the debugger.

- The second process will be the debuggee process.

In this case we can easily manipulate the registers values, change the instruction pointer or inject code which is the same capabilities we have when working with any debugger and all of this can be done using **ptrace** function .

To be able to fully understand what is happening here you had to have a good knowledge of linux signals and what can cause the rise of these signals.

So now let's go back to our challenge.

```
for ( j = 0; j <= 23; ++j )
  *(&v27 + j) ^= 0x99u;
puts(&v27);
scanf("%38s", &input);
v9 = seccomp_init(SECCOMP_RET_ALLOW);
seccomp_rule_add(v9, SECCOMP_RET_TRAP, SYS_fanotify_init, 0LL);
seccomp_load(v9);
pid = fork();
if ( !pid )
{
  v0 = mmap(0LL, 0x6B5uLL, 7, 34, -1, 0LL);
  v11 = v0;
  *v0 = qword_203020[0];
  *(v0 + 1709) = *(&qword_203020[213] + 5);
  qmemcpy(
    ((v0 + 1) & 0xFFFFFFFFFFFFFFF8LL),
    (qword_203020 - (v0 - ((v0 + 1) & 0xFFFFFFFFFFFFFFF8LL))),
    8LL * (((v0 - ((v0 + 8) & 0xFFFFFFF8) + 1717) & 0xFFFFFFF8) >> 3));
  ptrace(PTRACE_TRACEME, 0LL, 0LL, 0LL);
  v12 = v11;
  v11(&input);
  exit(0);
}
```

The function starts by decrypting the ascii art we saw before and decrypt the message **"Enter the secret:"**, then wait for input from the user with len **38 char**.

The next thing is adding a **seccomp** rule for **sys_fanotify_init** so whenever this syscall is used it will raise **SIGSYS** **"Bad syscall"** which we will use later.

After that a child process is created and at the child it first allocates a region of memory with read,write and execute permissions then copy the shellcode content to the allocated region of memory and calls ptrace with the argument **PTRACE_TRACEME** to give the ability to the parent process to trace it and finally call the shellcode with input as the only argument.

Before checking the shellcode let's check the parent code first.

```
while ( waitpid(pid, &stat_loc, 0) != -1 )
{
  if ( stat_loc >> 8 == SIGTRAP )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v10 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip, 0LL) & 0xFFFFFFFF00000000LL | 0x90909090;
    v13.r11 = rotate_left(v13.r11, v13.r10);
    ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
  }
  else if ( stat_loc >> 8 == SIGILL )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v1 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip, 0LL);
    LOWORD(v1) = 0;
    v10 = v1 | 0x9090;
    ptrace(PTRACE_POKETEXT, pid, v13.rip, v1 | 0x9090);
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v13.r11 = rotate_right(LOBYTE(v13.r11), LODWORD(v13.r10));
    ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
  }
  else if ( stat_loc >> 8 == SIGFPE )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v13.r11 = add(v13.r11, v13.r10);
    ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
    v10 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip, 0LL) & 0xFFFFFFFFFF000000LL | 0x909090;
    ptrace(PTRACE_POKETEXT, pid, v13.rip, v10);
  }
```

The program waits for any raised signals from the child process.

When the child raises **SIGTRAP** signal the parent process get all the current registers values from the child process and change the value of **r11** register by calling the **rotate_left** function which do the same thing as `rol` assembly instruction **r11** register value will be rotated by the value of **r10** register and as we know `rol` can be reverse by using `ror`.

Finally as we know if we use the breakpoint instruction **int 3** it will raise the **SIGTRAP** signal.

When the child raises **SIGILL** signal the parent process get all the current registers values from the child process and change the value of **r11** register by calling the **rotate_right** function which do the same thing as `ror` assembly instruction **r11** register value will be rotated by the value of **r10** register and as we know `ror` can be reverse by using `rol` and also it patches the current instruction with two nop instruction.

Finally as we know if we use the instruction **UD2** it will raise the **SIGILL** signal.

When the child raises a **SIGFPE** signal the parent process gets all the current registers values from the child process and changes the value of the **r11** register by calling the add function, **r11** register value will be added to the value of **r10**.

Finally as we know **SIGFPE** can be raised by dividing by zero.

```
else if ( stat_loc >> 8 == SIGSEGV )
{
  ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
  v13.r11 = xor(v13.r11, v13.r10);
  ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
  v10 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip, 0LL) & 0xFFFFFFFF00000000LL | 0x90909090;
  ptrace(PTRACE_POKETEXT, pid, v13.rip, v10);
  v10 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip + 4, 0LL) & 0xFFFFFFFF00000000LL | 0x90909090;
  ptrace(PTRACE_POKETEXT, pid, v13.rip + 4, v10);
}
else
{
  if ( stat_loc >> 8 == SIGSYS )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v10 = ptrace(PTRACE_PEEKTEXT, pid, v13.rip, 0LL) & 0xFFFFFFFF00000000LL | 0x90909090;
    compare(LOBYTE(v13.r13));
  }
  if ( stat_loc >> 8 == SIGCONT )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v13.r11 = sub_AD7(v13.r14, v13.r10);
    ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
  }
  else if ( stat_loc >> 8 == SIGWINCH )
  {
    ptrace(PTRACE_GETREGS, pid, 0LL, &v13);
    v13.r11 = sub_B0B(LOBYTE(v13.r14), LOBYTE(v13.r10));
    ptrace(PTRACE_SETREGS, pid, 0LL, &v13);
  }
```

When the child raises a **SIGSEGV** signal the parent process gets all the current registers values from the child process and changes the value of the **r11** register by calling the xor function, **r11** register value will be **xored** to the value of **r10** and patch the instruction that raises the signal with nop instructions .

Finally as we know **SIGSEGV** can be raised by trying to access invalid memory region like this instruction **mov rax,[0]** .

Finally as we know **SIGSYS** in our case can be raised by calling the syscall **fanotify_init**.

When the child raises a **SIGCONT** signal the parent process gets all the current registers values from the child process and then the **r14** register value and the **r10** register value to the function **sub_AD7** which will simply **xor** the **r10** register value by **0×41** the rotate left the **r14** value by the resulted value .

Finally as we know **SIGCONT** in our case can be raised by calling the syscall kill with **SIGCONT** id.

When the child raises a **SIGWINCH** signal the parent process gets all the current registers values from the child process and then the **r14** register value and the **r10** register value to the function **sub_AD7** which will simply **xor** the **r10** register value by **0×44** the rotate right the **r14** value by the resulted value .

Finally as we know **SIGWINCH** in our case can be raised by calling the syscall kill with **SIGWINCH** id.

```
      else if ( !(stat_loc & 0x7F) || stat_loc >> 8 == SIGCHLD )
      {
        v14 = -51;
        v15 = -15;
        v16 = -8;
        v17 = -19;
        v18 = -66;
        v19 = -22;
        v20 = -71;
        v21 = -50;
        v22 = -4;
        v23 = -16;
        v24 = -21;
        v25 = -3;
        v26 = -103;
        for ( k = 0; k <= 12; ++k )
          *(&v14 + k) ^= 0x99u;
        puts(&v14);
      }
    }
    ptrace(PTRACE_CONT, pid, 0LL, 0LL);
  }
  return __readfsqword(0x28u) ^ v58;
```

The last condition checks if the child process is dead or not and finally the parent process calls ptrace with **PTRACE_CONT** as argument to make the child process continue the execution.

So now let's go back to the child code and check the shellcode to see how we can decrypt the flag.

```
.data:0000000000203020 loc_203020:                             ; DATA XREF: sub_C4D+28A↑o
.data:0000000000203020                 mov     r9, rdi
.data:0000000000203023                 xor     r12, r12
.data:0000000000203026                 xor     r13, r13
.data:0000000000203029                 mov     r12b, 91h
.data:000000000020302C                 xor     r11, r11
.data:000000000020302F                 rol     r12b, 4
.data:0000000000203033                 xor     r12b, 2Ch
.data:0000000000203037                 mov     r11b, [r9]
.data:000000000020303A                 mov     r10, 1Ch
.data:0000000000203041                 ud2
.data:0000000000203043 ; ---------------------------------------------------------------------------
.data:0000000000203043                 inc     r9
.data:0000000000203046                 xor     r11b, r12b
.data:0000000000203049                 or      r13b, r11b
.data:000000000020304C                 mov     r12b, 0Eh
.data:000000000020304F                 xor     r11, r11
.data:0000000000203052                 rol     r12b, 4
.data:0000000000203056                 xor     r12b, 2Ch
.data:000000000020305A                 mov     r11b, [r9]
.data:000000000020305D                 mov     r14b, [r9]
.data:0000000000203060                 mov     r10, 6
.data:0000000000203067                 mov     rax, 3Eh ; '>'
.data:000000000020306E                 mov     rdi, 0
.data:0000000000203075                 mov     rsi, 1Ch
.data:000000000020307C                 syscall                 ; LINUX - sys_kill
.data:000000000020307E                 inc     r9
.data:0000000000203081                 xor     r11b, r12b
.data:0000000000203084                 or      r13b, r11b
```

The shellcode starts by moving the rdi value to **r9** which contains the address for the input we passed as it’s passed as argument to the shellcode. And after that some instructions that will decrypt the value that our input after encryption will be compared with.

These set of instructions are:

- **mov r12b, `value`**
- **rol r12b,4**
- **xor r12b, 2Ch**

After that a value is moved to the r10 register which will be used to encrypt the flag char before the check and after that instruction differ depending on the desired signal to be raised which will differ on the encryption technique to be used.

The full list of the instructions used to raise signal:

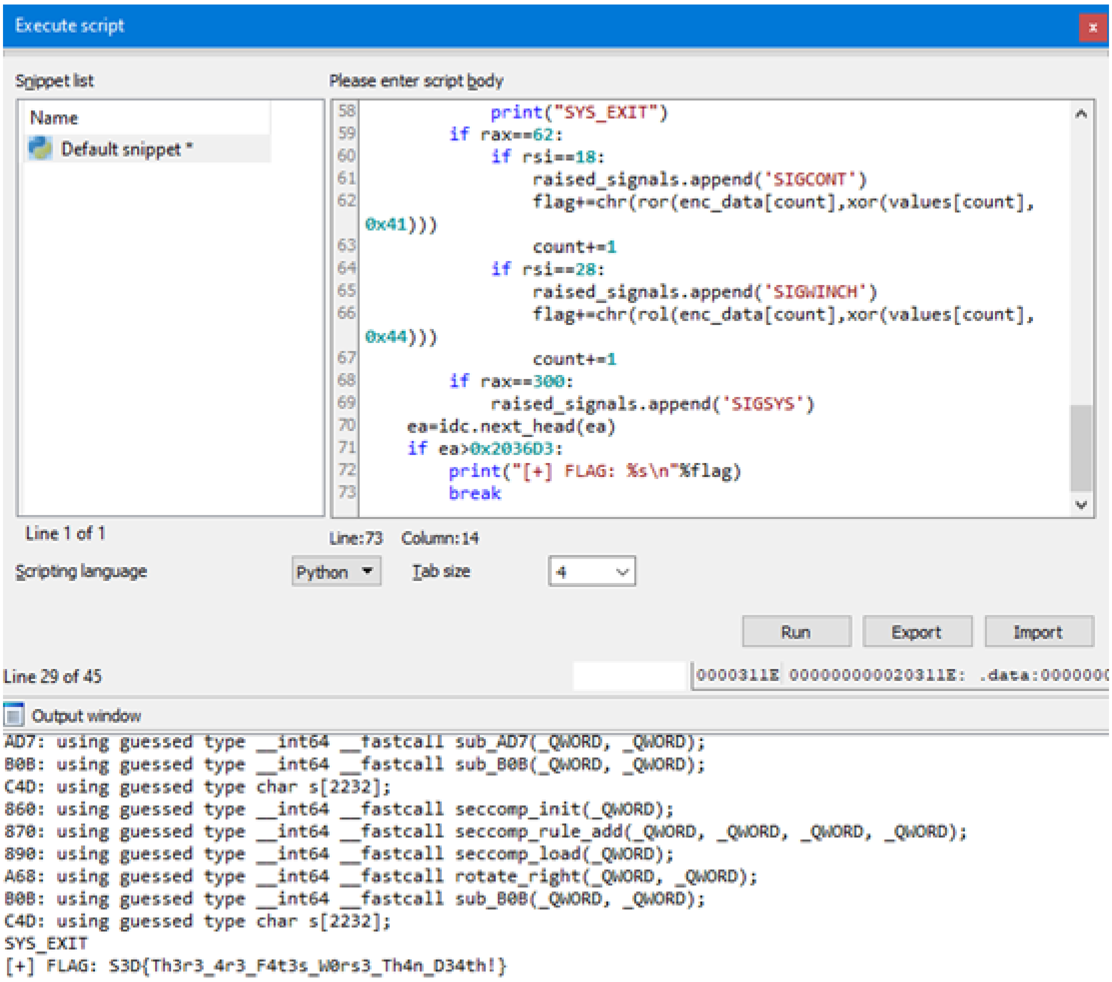
```
SIGTRAP: 'int 3'
SIGILL: 'UD2'
SIGFPE: 'xor rax,rax ; xor rdi,rdi ; idiv rcx'
SIGSEGV: 'mov rax, qword ptr ds:dword_0'
SIGSYS: 'mov rax,300 ; syscall'
SIGCONT: 'mov rax,62 ; mov rdi,0 ; mov rsi,18 ; syscall'
SIGWINCH: 'mov rax,62 , mov rdi,0 , mov rsi,28, syscall'
```

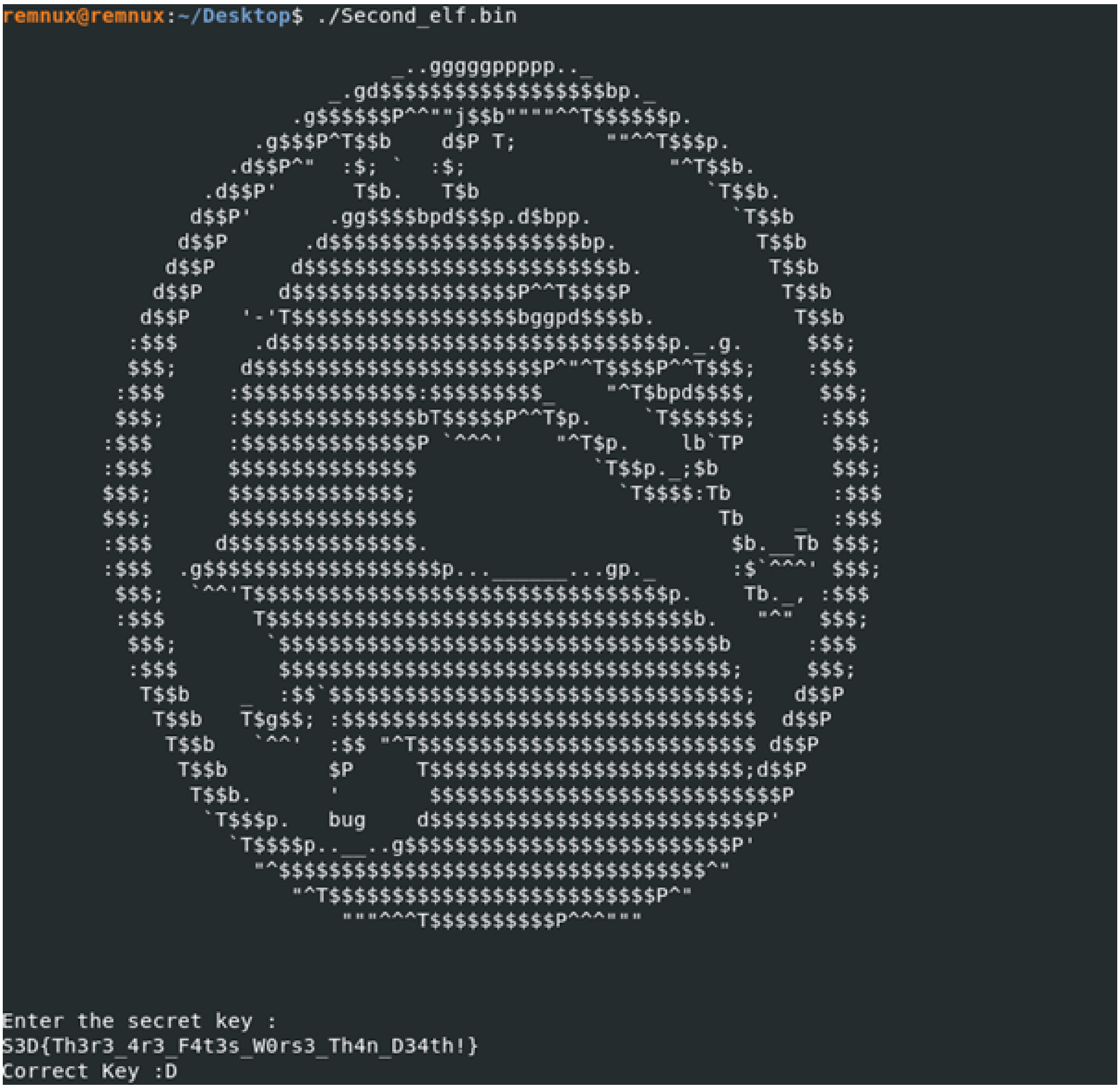
Using these information we can write an ida python script to parse the assembly instruction and decrypt the flag depending on the raised signal.

```
"""
This script will work on ida versions under 7.5
"""
import idautils
import idaapi
import idc

def xor(data,val):
    return data ^ val
def add(data,val):
    return data+val
def sub(data,val):
    return (data-val)%256
rol = lambda val, r_bits: \
    (val << r_bits%8) & (2**8-1) | \
    ((val & (2**8-1)) >> (8-(r_bits%8)))

ror = lambda val, r_bits: \
    ((val & (2**8-1)) >> r_bits%8) | \
    (val << (8-(r_bits%8)) & (2**8-1))

start_add=0x203020
enc_data=[]
values=[]
count=0
raised_signals=[]
rax=0
rsi=0
flag=''
#print(idc.generate_disasm_line(start_add, 0))
ea=idc.next_head(0x203020)
while True:
    inst= idc.generate_disasm_line(ea, 0)
    #print(inst)
    if idc.GetMnem(ea)=='mov' and idc.print_operand(ea,0)=='r12b':
        enc_data.append(xor(rol(idc.GetOperandValue(ea,1),4),0x2c) )
    elif idc.GetMnem(ea)=='mov' and idc.print_operand(ea,0)=='r10':
        values.append(idc.GetOperandValue(ea,1))
    elif idc.GetMnem(ea)=='mov' and idc.print_operand(ea,0)=='rax':
        if idc.print_operand(ea,1)=="qword ptr ds:dword_0":
            raised_signals.append('SIGSEGV')
            flag+=chr(xor(enc_data[count],values[count]))
            count+=1
        else:
            rax=idc.GetOperandValue(ea,1)
    elif idc.GetMnem(ea)=='mov' and idc.print_operand(ea,0)=='rsi':
        rsi=idc.GetOperandValue(ea,1)
    elif idc.GetMnem(ea)=='int':
        raised_signals.append('SIGTRAP')
```

```
        flag+=chr(ror(enc_data[count],values[count]))
        count+=1
    elif idc.GetMnem(ea)=='ud2':
        raised_signals.append('SIGILL')
        flag+=chr(rol(enc_data[count],values[count]))
        count+=1
    elif idc.GetMnem(ea)=='idiv':
        raised_signals.append('SIGFPE')
        flag+=chr(sub(enc_data[count],values[count]))
        count+=1

    elif idc.GetMnem(ea)=='syscall':
        if rax==60:
            print("SYS_EXIT")
        if rax==62:
            if rsi==18:
                raised_signals.append('SIGCONT')
                flag+=chr(ror(enc_data[count],xor(values[count],0x41)))
                count+=1
            if rsi==28:
                raised_signals.append('SIGWINCH')
                flag+=chr(rol(enc_data[count],xor(values[count],0x44)))
                count+=1
        if rax==300:
            raised_signals.append('SIGSYS')
    ea=idc.next_head(ea)
    if ea>0x2036D3:
        #print("Done")
        #print(raised_signals)
        print("[+] FLAG: %s\n"%flag)
        break
```

Running the script in ida will reveal the flag.


Execute script
Snippet list
Name
Default snippet *
Please enter script body
print("SYS_EXIT")
if rax==62:
if rsi==18:
raised_signals.append('SIGCONT')
flag+=chr(ror(enc_data[count],xor(values[count],
0x41)))
count+=1
if rsi==28:
raised_signals.append('SIGWINCH')
flag+=chr(rol(enc_data[count],xor(values[count],
0x44)))
count+=1
if rax==300:
raised_signals.append('SIGSYS')
ea=idc.next_head(ea)
if ea>0x2036D3:
print("[+] FLAG: %s\n"%flag)
break
Line 1 of 1
Line:73 Column:14
Scripting language
Python
Tab size
4
Run
Export
Import
Line 29 of 45
0000311E 000000000020311E: .data:000000
Output window
AD7: using guessed type __int64 __fastcall sub_AD7(_QWORD, _QWORD);
B0B: using guessed type __int64 __fastcall sub_B0B(_QWORD, _QWORD);
C4D: using guessed type char s[2232];
860: using guessed type __int64 __fastcall seccomp_init(_QWORD);
870: using guessed type __int64 __fastcall seccomp_rule_add(_QWORD, _QWORD, _QWORD, _QWORD);
890: using guessed type __int64 __fastcall seccomp_load(_QWORD);
A68: using guessed type __int64 __fastcall rotate_right(_QWORD, _QWORD);
B0B: using guessed type __int64 __fastcall sub_B0B(_QWORD, _QWORD);
C4D: using guessed type char s[2232];
SYS_EXIT
[+] FLAG: S3D{Th3r3_4r3_F4t3s_W0rs3_Th4n_D34th!}



Flag: **S3D{Th3r3_4r3_F4t3s_W0rs3_Th4n_D34th!}**

© The Crafters - twitter.com/CTFCreators