**WRITEUP**

**Name Chall: Volatile Memory**

**Deskripsi:**
Kami berhasil menyita sebuah program biner dari server sindikat kriminal. Tim forensik kami mencoba menganalisis kodenya menggunakan disassembler, tetapi mereka bingung karena isinya terlihat seperti sampah data acak (gibberish).

Anehnya, saat program dijalankan, ia berfungsi normal dan meminta password. Sepertinya program ini memiliki mekanisme pertahanan diri yang menyembunyikan logika aslinya.

Format flag: LKS{...}

By: FailDeGaskar

Jadi dari challenge CTF Reverse Engineer dengan title Volatile Memory dan dari deskripsi nya yang menekankan bahwa program berhasil berjalan normal namun ketika di disasembler hanya tampil data acak, Dan ada penjelasan bahwa memiliki mekanisme pertahanan diri yang menyembunyikan logika aslinya maka bisa dipahami bahwa terdapat enkripsi di sini

baik langsung kita mulai dari menganalisa file attachment yang di berikan yaitu sebuah file binary "chall"

```
❯ file chall
chall: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d31165c01592f4cbd8feed8ace1ae4684135cf32, for
GNU/Linux 3.2.0, not stripped
```

file terlihat seperti ELF 64-bit normal tanpa stripped function name, langsung aja kita coba jalankan

```
❯ ./chall
Gunakan: ./chall <flag>
```

ohh ternyata file nya ketika di jalankan tidak meminta input flag namun meminta argument berupa flag ketika di jalankan

```
❯ ./chall LKS{TES}
Memeriksa flag...
Wrong!
```

hasilnya baru muncul, lanjut kita cek dengan strings apakah ada hal menarik. Nah hasilnya ada beberapa string readable namun tidak langsung menunjukkan flag nya

Gunakan: %s <flag>
Gagal alokasi memory
gagal mprotect
Memeriksa flag...
Correct!
Wrong!

ada string “gagal alokasi memory” dan “gagal mprotect” yang sebenarnya merupakan indikasi bahwa di program ini terdapat pemanggilan fungsi malloc untuk alokasi memory dan mprotect sendiri berfungsi untuk mengatur izin bagian memory tertentu seperti Read write yang menunjukkan program ini terdapat Self Modifying Code yang juga di perjelas dari deskripsi bahwa ketika di debug hanya memunculkan sampah data acak

kita coba dynamic analysis menggunakan GDB untuk mencoba intercept bahasa asembly yang di gunakan dalam program ini. untuk mengetahui cara kerja nya secara mendalam

di dalam GDB pertama kita bisa lihat daftar function yang ada di executable ini dengan mengetik info functions

Non-debugging symbols:
0x000000000040033c  _init
0x0000000000400370  puts@plt
0x0000000000400380  mmap@plt
0x0000000000400390  printf@plt
0x00000000004003a0  memcpy@plt
0x00000000004003b0  munmap@plt
0x00000000004003c0  mprotect@plt
0x00000000004003d0  perror@plt
0x00000000004003e0  _start
0x0000000000400410  _dl_relocate_static_pie
0x0000000000400420  deregister_tm_clones
0x0000000000400450  register_tm_clones
0x0000000000400490  __do_global_dtors_aux
0x00000000004004c0  frame_dummy
0x00000000004004c6  main
0x0000000000400630  _fini

dari hasil yang di dapat hanya menampilkan fungsi main sebagai program utamanya, namun dari hasil yang bisa dilihat memang mendukung teori kita sebelumnya yang mana program menggunakan fungsi2 manipulasi memory seperti mmap memcpy mprotect dan sejenisnya.

Lanjut kita lihat assembly dari fungsi main apakah ada hal2 menarik yang bisa di kulik lebih lanjut

```
pwndbg> disas main
Dump of assembler code for function main:
   0x00000000004004c6 <+0>:     push   rbp
   0x00000000004004c7 <+1>:     mov    rbp,rsp
   0x00000000004004ca <+4>:     sub    rsp,0x40
   0x00000000004004ce <+8>:     mov    DWORD PTR [rbp-0x34],edi
   0x00000000004004d1 <+11>:    mov    QWORD PTR [rbp-0x40],rsi
   0x00000000004004d5 <+15>:    cmp    DWORD PTR [rbp-0x34],0x1
   0x00000000004004d9 <+19>:    jg     0x4004fe <main+56>
   0x00000000004004db <+21>:    mov    rax,QWORD PTR [rbp-0x40]
   0x00000000004004df <+25>:    mov    rax,QWORD PTR [rax]
   0x00000000004004e2 <+28>:    mov    rsi,rax
   0x00000000004004e5 <+31>:    mov    edi,0x4012f0
   0x00000000004004ea <+36>:    mov    eax,0x0
   0x00000000004004ef <+41>:    call   0x400390 <printf@plt>
   0x00000000004004f4 <+46>:    mov    eax,0x1
   0x00000000004004f9 <+51>:    jmp    0x40062e <main+360>
   0x00000000004004fe <+56>:    mov    QWORD PTR [rbp-0x10],0x21f
   0x0000000000400506 <+64>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040050a <+68>:    mov    r9d,0x0
   0x0000000000400510 <+74>:    mov    r8d,0xffffffff
   0x0000000000400516 <+80>:    mov    ecx,0x22
   0x000000000040051b <+85>:    mov    edx,0x3
   0x0000000000400520 <+90>:    mov    rsi,rax
   0x0000000000400523 <+93>:    mov    edi,0x0
   0x0000000000400528 <+98>:    call   0x400380 <mmap@plt>
   0x000000000040052d <+103>:   mov    QWORD PTR [rbp-0x18],rax
   0x0000000000400531 <+107>:   cmp    QWORD PTR [rbp-0x18],0xffffffffffffffff
   0x0000000000400536 <+112>:   jne    0x40054c <main+134>
   0x0000000000400538 <+114>:   mov    edi,0x401304
   0x000000000040053d <+119>:   call   0x4003d0 <perror@plt>
   0x0000000000400542 <+124>:   mov    eax,0x1
   0x0000000000400547 <+129>:   jmp    0x40062e <main+360>
   0x000000000040054c <+134>:   mov    rdx,QWORD PTR [rbp-0x10]
   0x0000000000400550 <+138>:   mov    rax,QWORD PTR [rbp-0x18]
   0x0000000000400554 <+142>:   mov    esi,0x403060
   0x0000000000400559 <+147>:   mov    rdi,rax
   0x000000000040055c <+150>:   call   0x4003a0 <memcpy@plt>
   0x0000000000400561 <+155>:   mov    rax,QWORD PTR [rbp-0x18]
   0x0000000000400565 <+159>:   mov    QWORD PTR [rbp-0x20],rax
   0x0000000000400569 <+163>:   mov    DWORD PTR [rbp-0x4],0x0
   0x0000000000400570 <+170>:   jmp    0x400598 <main+210>
   0x0000000000400572 <+172>:   mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000400575 <+175>:   movsxd rdx,eax
   0x0000000000400578 <+178>:   mov    rax,QWORD PTR [rbp-0x20]
   0x000000000040057c <+182>:   add    rax,rdx
   0x000000000040057f <+185>:   movzx  edx,BYTE PTR [rax]
   0x0000000000400582 <+188>:   mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000400585 <+191>:   movsxd rcx,eax
   0x0000000000400588 <+194>:   mov    rax,QWORD PTR [rbp-0x20]
   0x000000000040058c <+198>:   add    rax,rcx
```

```
   0x000000000040058f <+201>:   xor    edx,0xffffffaa
   0x0000000000400592 <+204>:   mov    BYTE PTR [rax],dl
   0x0000000000400594 <+206>:   add    DWORD PTR [rbp-0x4],0x1
   0x0000000000400598 <+210>:   mov    eax,DWORD PTR [rbp-0x4]
   0x000000000040059b <+213>:   cdqe
   0x000000000040059d <+215>:   cmp    rax,QWORD PTR [rbp-0x10]
   0x00000000004005a1 <+219>:   jb     0x400572 <main+172>
   0x00000000004005a3 <+221>:   mov    rcx,QWORD PTR [rbp-0x10]
   0x00000000004005a7 <+225>:   mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004005ab <+229>:   mov    edx,0x5
   0x00000000004005b0 <+234>:   mov    rsi,rcx
   0x00000000004005b3 <+237>:   mov    rdi,rax
   0x00000000004005b6 <+240>:   call   0x4003c0 <mprotect@plt>
   0x00000000004005bb <+245>:   cmp    eax,0xffffffff
   0x00000000004005be <+248>:   jne    0x4005d1 <main+267>
   0x00000000004005c0 <+250>:   mov    edi,0x401319
   0x00000000004005c5 <+255>:   call   0x4003d0 <perror@plt>
   0x00000000004005ca <+260>:   mov    eax,0x1
   0x00000000004005cf <+265>:   jmp    0x40062e <main+360>
   0x00000000004005d1 <+267>:   mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004005d5 <+271>:   mov    QWORD PTR [rbp-0x28],rax
   0x00000000004005d9 <+275>:   mov    edi,0x401328
   0x00000000004005de <+280>:   call   0x400370 <puts@plt>
   0x00000000004005e3 <+285>:   mov    rax,QWORD PTR [rbp-0x40]
   0x00000000004005e7 <+289>:   add    rax,0x8
   0x00000000004005eb <+293>:   mov    rax,QWORD PTR [rax]
   0x00000000004005ee <+296>:   mov    rdx,QWORD PTR [rbp-0x28]
   0x00000000004005f2 <+300>:   mov    rdi,rax
   0x00000000004005f5 <+303>:   call   rdx
   0x00000000004005f7 <+305>:   mov    DWORD PTR [rbp-0x2c],eax
   0x00000000004005fa <+308>:   cmp    DWORD PTR [rbp-0x2c],0x1
   0x00000000004005fe <+312>:   jne    0x40060c <main+326>
   0x0000000000400600 <+314>:   mov    edi,0x40133a
   0x0000000000400605 <+319>:   call   0x400370 <puts@plt>
   0x000000000040060a <+324>:   jmp    0x400616 <main+336>
   0x000000000040060c <+326>:   mov    edi,0x401343
   0x0000000000400611 <+331>:   call   0x400370 <puts@plt>
   0x0000000000400616 <+336>:   mov    rdx,QWORD PTR [rbp-0x10]
   0x000000000040061a <+340>:   mov    rax,QWORD PTR [rbp-0x18]
   0x000000000040061e <+344>:   mov    rsi,rdx
   0x0000000000400621 <+347>:   mov    rdi,rax
   0x0000000000400624 <+350>:   call   0x4003b0 <munmap@plt>
   0x0000000000400629 <+355>:   mov    eax,0x0
   0x000000000040062e <+360>:   leave
   0x000000000040062f <+361>:   ret
End of assembler dump.
```

Berikut hasil analisa code assembly tersebut

- Pada alamat <+82> hingga <+98> program memanggil mmap, tujuannya untukmeminta blok memori baru ke sistem operasi. Nilai 0x3 pada edx berarti memori tersebut awalnya diberik akses READ | WRITE
- pada alamat <+142> hingga <+150> program menggunakan memcpy, ia menyalin data dari alamat 0x403060 (data mentah yang ada di dalam file ELF) ke area memori yang baru saja di buat oleh mmap
- antara alammt <+170> hingga <+219> terdapat sebuah loop, “xor edx, 0xffffffaa” program melakukan operasi XOR pada setiap byte data yang di salin tadi dengan kunci 0xAA
- Setelah data di-XOR (di dekripsi), program memanggil mprotect pada alamat <+240>. Nilai 0x5 pada edx (prot) berarti READ | EXECUTE yang berarti memori yang tadinya hanya berisi data “tulisan”, sekarang di ubah izinnya agar bisa dijalankan sebagai instruksi CPU (kode program).
- Bagian paling krusial. Pada alamat <+303> “call rdx” program melompat dan menjalankan kode yang baru saja di dekripsi di memori tadi, dan setelah selesai ia akan membersihkan memori dengan munmap

Jadi intinya program ini adalah sebuah loader. Ia membawa payload yang terenkripsi (XOR 0xAA) -> membukanya di memory -> mengubah izin memorinya menjadi executable -> lalu menjalankannya.

oke setelah kita memahami cara kerja program selanjutnya kita bisa proses exploit dengan cara intercept alur program pada alamat 0x4005f5 (alamat call rdx) dengan memanfaatkan fitur breakpoint pada GDB “break *0x4005f5“

setelah berhasil memasang breakpoint kita hanya perlu menjalankan program nya dengan “run LKS{TEST}”

Disini program akan berhenti pada “'Memeriksa flag...\n'”

nah disini berdasarkan alur program sebelumnya pada kondisi ini program itu sudah mengalokasikan dan mendekripsi payload, yang berarti disini kita bisa mengintip data mentahnya dengan melihat assembly pada alamat rdx dengan cara “x/200i $rdx” yang berarti kita melihat 200 line yang ada di alamat rdx hasilnya sebagai berikut

```
pwndbg> x/200i $rdx
   0x7ffff7fbe000:      push   rbp
   0x7ffff7fbe001:      mov    rbp,rsp
   0x7ffff7fbe004:      mov    QWORD PTR [rbp-0x8],rdi
   0x7ffff7fbe008:      mov    rax,QWORD PTR [rbp-0x8]
   0x7ffff7fbe00c:      movzx  eax,BYTE PTR [rax]
   0x7ffff7fbe00f:      cmp    al,0x4c
   0x7ffff7fbe011:      je     0x7ffff7fbe01d
   0x7ffff7fbe013:      mov    eax,0x0
   0x7ffff7fbe018:      jmp    0x7ffff7fbe21d
   0x7ffff7fbe01d:      mov    rax,QWORD PTR [rbp-0x8]
   0x7ffff7fbe021:      add    rax,0x1
```

```
0x7ffff7fbe025:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe028:    cmp    al,0x4b
0x7ffff7fbe02a:    je     0x7ffff7fbe036
0x7ffff7fbe02c:    mov    eax,0x0
0x7ffff7fbe031:    jmp    0x7ffff7fbe21d
0x7ffff7fbe036:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe03a:    add    rax,0x2
0x7ffff7fbe03e:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe041:    cmp    al,0x53
0x7ffff7fbe043:    je     0x7ffff7fbe04f
0x7ffff7fbe045:    mov    eax,0x0
0x7ffff7fbe04a:    jmp    0x7ffff7fbe21d
0x7ffff7fbe04f:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe053:    add    rax,0x3
0x7ffff7fbe057:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe05a:    cmp    al,0x7b
0x7ffff7fbe05c:    je     0x7ffff7fbe068
0x7ffff7fbe05e:    mov    eax,0x0
0x7ffff7fbe063:    jmp    0x7ffff7fbe21d
0x7ffff7fbe068:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe06c:    add    rax,0x4
0x7ffff7fbe070:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe073:    cmp    al,0x68
0x7ffff7fbe075:    je     0x7ffff7fbe081
0x7ffff7fbe077:    mov    eax,0x0
0x7ffff7fbe07c:    jmp    0x7ffff7fbe21d
0x7ffff7fbe081:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe085:    add    rax,0x5
0x7ffff7fbe089:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe08c:    cmp    al,0x31
0x7ffff7fbe08e:    je     0x7ffff7fbe09a
0x7ffff7fbe090:    mov    eax,0x0
0x7ffff7fbe095:    jmp    0x7ffff7fbe21d
0x7ffff7fbe09a:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe09e:    add    rax,0x6
0x7ffff7fbe0a2:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe0a5:    cmp    al,0x64
0x7ffff7fbe0a7:    je     0x7ffff7fbe0b3
0x7ffff7fbe0a9:    mov    eax,0x0
0x7ffff7fbe0ae:    jmp    0x7ffff7fbe21d
0x7ffff7fbe0b3:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe0b7:    add    rax,0x7
0x7ffff7fbe0bb:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe0be:    cmp    al,0x64
0x7ffff7fbe0c0:    je     0x7ffff7fbe0cc
0x7ffff7fbe0c2:    mov    eax,0x0
0x7ffff7fbe0c7:    jmp    0x7ffff7fbe21d
0x7ffff7fbe0cc:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe0d0:    add    rax,0x8
0x7ffff7fbe0d4:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe0d7:    cmp    al,0x33
```

```
0x7ffff7fbe0d9:    je     0x7ffff7fbe0e5
0x7ffff7fbe0db:    mov    eax,0x0
0x7ffff7fbe0e0:    jmp    0x7ffff7fbe21d
0x7ffff7fbe0e5:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe0e9:    add    rax,0x9
0x7ffff7fbe0ed:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe0f0:    cmp    al,0x6e
0x7ffff7fbe0f2:    je     0x7ffff7fbe0fe
0x7ffff7fbe0f4:    mov    eax,0x0
0x7ffff7fbe0f9:    jmp    0x7ffff7fbe21d
0x7ffff7fbe0fe:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe102:    add    rax,0xa
0x7ffff7fbe106:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe109:    cmp    al,0x5f
0x7ffff7fbe10b:    je     0x7ffff7fbe117
0x7ffff7fbe10d:    mov    eax,0x0
0x7ffff7fbe112:    jmp    0x7ffff7fbe21d
0x7ffff7fbe117:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe11b:    add    rax,0xb
0x7ffff7fbe11f:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe122:    cmp    al,0x31
0x7ffff7fbe124:    je     0x7ffff7fbe130
0x7ffff7fbe126:    mov    eax,0x0
0x7ffff7fbe12b:    jmp    0x7ffff7fbe21d
0x7ffff7fbe130:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe134:    add    rax,0xc
0x7ffff7fbe138:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe13b:    cmp    al,0x6e
0x7ffff7fbe13d:    je     0x7ffff7fbe149
0x7ffff7fbe13f:    mov    eax,0x0
0x7ffff7fbe144:    jmp    0x7ffff7fbe21d
0x7ffff7fbe149:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe14d:    add    rax,0xd
0x7ffff7fbe151:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe154:    cmp    al,0x5f
0x7ffff7fbe156:    je     0x7ffff7fbe162
0x7ffff7fbe158:    mov    eax,0x0
0x7ffff7fbe15d:    jmp    0x7ffff7fbe21d
0x7ffff7fbe162:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe166:    add    rax,0xe
0x7ffff7fbe16a:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe16d:    cmp    al,0x72
0x7ffff7fbe16f:    je     0x7ffff7fbe17b
0x7ffff7fbe171:    mov    eax,0x0
0x7ffff7fbe176:    jmp    0x7ffff7fbe21d
0x7ffff7fbe17b:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe17f:    add    rax,0xf
0x7ffff7fbe183:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe186:    cmp    al,0x34
0x7ffff7fbe188:    je     0x7ffff7fbe194
0x7ffff7fbe18a:    mov    eax,0x0
```

```
0x7ffff7fbe18f:    jmp    0x7ffff7fbe21d
0x7ffff7fbe194:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe198:    add    rax,0x10
0x7ffff7fbe19c:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe19f:    cmp    al,0x6d
0x7ffff7fbe1a1:    je     0x7ffff7fbe1aa
0x7ffff7fbe1a3:    mov    eax,0x0
0x7ffff7fbe1a8:    jmp    0x7ffff7fbe21d
0x7ffff7fbe1aa:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe1ae:    add    rax,0x11
0x7ffff7fbe1b2:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe1b5:    cmp    al,0x5f
0x7ffff7fbe1b7:    je     0x7ffff7fbe1c0
0x7ffff7fbe1b9:    mov    eax,0x0
0x7ffff7fbe1be:    jmp    0x7ffff7fbe21d
0x7ffff7fbe1c0:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe1c4:    add    rax,0x12
0x7ffff7fbe1c8:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe1cb:    cmp    al,0x36
0x7ffff7fbe1cd:    je     0x7ffff7fbe1d6
0x7ffff7fbe1cf:    mov    eax,0x0
0x7ffff7fbe1d4:    jmp    0x7ffff7fbe21d
0x7ffff7fbe1d6:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe1da:    add    rax,0x13
0x7ffff7fbe1de:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe1e1:    cmp    al,0x37
0x7ffff7fbe1e3:    je     0x7ffff7fbe1ec
0x7ffff7fbe1e5:    mov    eax,0x0
0x7ffff7fbe1ea:    jmp    0x7ffff7fbe21d
0x7ffff7fbe1ec:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe1f0:    add    rax,0x14
0x7ffff7fbe1f4:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe1f7:    cmp    al,0x7d
0x7ffff7fbe1f9:    je     0x7ffff7fbe202
0x7ffff7fbe1fb:    mov    eax,0x0
0x7ffff7fbe200:    jmp    0x7ffff7fbe21d
0x7ffff7fbe202:    mov    rax,QWORD PTR [rbp-0x8]
0x7ffff7fbe206:    add    rax,0x15
0x7ffff7fbe20a:    movzx  eax,BYTE PTR [rax]
0x7ffff7fbe20d:    test   al,al
0x7ffff7fbe20f:    je     0x7ffff7fbe218
0x7ffff7fbe211:    mov    eax,0x0
0x7ffff7fbe216:    jmp    0x7ffff7fbe21d
0x7ffff7fbe218:    mov    eax,0x1
0x7ffff7fbe21d:    pop    rbp
0x7ffff7fbe21e:    ret
```

yap disini kita ssudah menemukan disasembler dari payload sumber nya, program mengecek karakter pada indeks tertentu (add rax, [indeks]) lalu membandingkannya (cmp al, [hex]). Jadi disini kita bisa menerjemahkan hex nya satu persatu dan sudah bisa menemukan flag nya sebagai berikut

| Alamat | Indeks | Hex | Karakter (ASCII) |
|---|---|---|---|
| 0x7ffff7fbe00f | 0 | 0x4c | **L** |
| 0x7ffff7fbe028 | 1 | 0x4b | **K** |
| 0x7ffff7fbe041 | 2 | 0x53 | **S** |
| 0x7ffff7fbe05a | 3 | 0x7b | **{** |
| 0x7ffff7fbe073 | 4 | 0x68 | **h** |
| 0x7ffff7fbe08c | 5 | 0x31 | **1** |
| 0x7ffff7fbe0a5 | 6 | 0x64 | **d** |
| 0x7ffff7fbe0be | 7 | 0x64 | **d** |
| 0x7ffff7fbe0d7 | 8 | 0x33 | **3** |
| 0x7ffff7fbe0f0 | 9 | 0x6e | **n** |
| 0x7ffff7fbe109 | 10 | 0x5f | **_** |
| 0x7ffff7fbe122 | 11 | 0x31 | **1** |

| | | | |
|---|---|---|---|
| 0x7ffff7fbe13b | 12 | 0x6e | **n** |
| 0x7ffff7fbe154 | 13 | 0x5f | **_** |
| 0x7ffff7fbe16d | 14 | 0x72 | **r** |
| 0x7ffff7fbe186 | 15 | 0x34 | **4** |
| 0x7ffff7fbe19f | 16 | 0x6d | **m** |
| 0x7ffff7fbe1b5 | 17 | 0x5f | **_** |
| 0x7ffff7fbe1cb | 18 | 0x36 | **6** |
| 0x7ffff7fbe1e1 | 19 | 0x37 | **7** |
| 0x7ffff7fbe1f7 | 20 | 0x7d | **}** |
| 0x7ffff7fbe20d | 21 | 0x00 | (Null terminator) |

namun hengker itu jarang melakukan tugasnya secara manual dan memilih mengautomasi menggunakan script, berikut script yang bisa digunakan untuk langsung mengambil hexa dari shellcode yang di enkripsi dengan XOR lalu di dekripsi dengan XOR lagi dan hasilnya bisa langsung di dapat FLAG nya

```
#!/usr/bin/env python3
"""
Solver untuk challenge binary dengan self-decrypting code
Binary menggunakan XOR 0xAA untuk menyembunyikan logika verifikasi flag
"""
```

```
import sys
import re
from pathlib import Path

def get_section_info(binary_path):
    """Parse ELF header untuk mendapatkan info section .data"""
    with open(binary_path, 'rb') as f:
        data = f.read()

    # ELF header parsing sederhana
    if data[:4] != b'\x7fELF':
        raise ValueError("Bukan file ELF valid")

    is_64bit = data[4] == 2
    if not is_64bit:
        raise ValueError("Hanya support ELF 64-bit")

    # Section header offset dan info
    e_shoff = int.from_bytes(data[0x28:0x30], 'little')
    e_shentsize = int.from_bytes(data[0x3a:0x3c], 'little')
    e_shnum = int.from_bytes(data[0x3c:0x3e], 'little')
    e_shstrndx = int.from_bytes(data[0x3e:0x40], 'little')

    # String table section
    shstr_offset = e_shoff + e_shstrndx * e_shentsize
    shstr_sh_offset =
int.from_bytes(data[shstr_offset+0x18:shstr_offset+0x20], 'little')

    # Cari section .data
    for i in range(e_shnum):
        sh_offset = e_shoff + i * e_shentsize
        sh_name_idx = int.from_bytes(data[sh_offset:sh_offset+4],
'little')

        # Baca nama section
        name_end = data.find(b'\x00', shstr_sh_offset + sh_name_idx)
        name = data[shstr_sh_offset +
sh_name_idx:name_end].decode('ascii')

        if name == '.data':
            sh_addr =
int.from_bytes(data[sh_offset+0x10:sh_offset+0x18], 'little')
            sh_offset_file =
int.from_bytes(data[sh_offset+0x18:sh_offset+0x20], 'little')
            sh_size =
int.from_bytes(data[sh_offset+0x20:sh_offset+0x28], 'little')
            return {
                'vaddr': sh_addr,
                'offset': sh_offset_file,
                'size': sh_size
            }

    raise ValueError("Section .data tidak ditemukan")

def find_encrypted_code_params(binary_path):
    """Cari parameter encrypted code dari disassembly main()"""
```

```
    with open(binary_path, 'rb') as f:
        data = f.read()

    # Pattern untuk mencari: movq $size, -0x10(%rbp) dan mov $addr,
%esi (memcpy source)
    # Kita cari pattern XOR key juga: xor $0xNN, %edx

    # Cari XOR key - pattern: 83 f2 XX (xor $XX, %edx)
    xor_pattern = re.compile(rb'\x83\xf2(.)', re.DOTALL)
    xor_match = xor_pattern.search(data)
    xor_key = xor_match.group(1)[0] if xor_match else 0xAA

    # Cari size - pattern: 48 c7 45 f0 XX XX 00 00 (movq $size,
-0x10(%rbp))
    size_pattern = re.compile(rb'\x48\xc7\x45\xf0(....)' , re.DOTALL)
    size_match = size_pattern.search(data)
    size = int.from_bytes(size_match.group(1), 'little') if
size_match else 0x21f

    # Cari source address untuk memcpy - pattern: be XX XX XX XX (mov
$addr, %esi)
    # Biasanya setelah mov -0x10(%rbp), %rdx
    memcpy_pattern = re.compile(rb'\xbe(....)\x48\x89\xc7\xe8',
re.DOTALL)
    memcpy_match = memcpy_pattern.search(data)
    src_addr = int.from_bytes(memcpy_match.group(1), 'little') if
memcpy_match else 0x403060

    return {
        'src_addr': src_addr,
        'size': size,
        'xor_key': xor_key
    }

def decrypt_shellcode(binary_path):
    """Dekripsi shellcode dari binary"""
    with open(binary_path, 'rb') as f:
        binary_data = f.read()

    # Dapatkan info
    section_info = get_section_info(binary_path)
    params = find_encrypted_code_params(binary_path)

    # Hitung file offset dari virtual address
    file_offset = section_info['offset'] + (params['src_addr'] -
section_info['vaddr'])

    # Ekstrak dan dekripsi
    encrypted = binary_data[file_offset:file_offset + params['size']]
    decrypted = bytes([b ^ params['xor_key'] for b in encrypted])

    return decrypted, params

def extract_flag_from_shellcode(shellcode):
    """Ekstrak flag dari shellcode dengan parsing instruksi CMP"""
    flag_chars = []
    i = 0
```

```
    while i < len(shellcode) - 2:
        # Pattern: 3c XX (cmp $XX, %al)
        if shellcode[i] == 0x3c:
            char_val = shellcode[i + 1]
            # Skip jika bukan printable ASCII
            if 0x20 <= char_val <= 0x7e:
                flag_chars.append(chr(char_val))
            i += 2
        # Pattern: 84 c0 (test %al, %al) - null terminator check
        elif shellcode[i:i+2] == b'\x84\xc0':
            break
        else:
            i += 1

    return ''.join(flag_chars)

def solve(binary_path):
    """Main solver function"""
    print(f"[*] Analyzing: {binary_path}")

    # Dekripsi shellcode
    shellcode, params = decrypt_shellcode(binary_path)
    print(f"[+] Found encrypted code at 0x{params['src_addr']:x}")
    print(f"[+] Size: {params['size']} bytes")
    print(f"[+] XOR key: 0x{params['xor_key']:02x}")

    # Ekstrak flag
    flag = extract_flag_from_shellcode(shellcode)
    print(f"\n[✓] FLAG: {flag}")

    return flag

if __name__ == "__main__":
    binary = sys.argv[1] if len(sys.argv) > 1 else "chall"

    if not Path(binary).exists():
        print(f"[-] File tidak ditemukan: {binary}")
        sys.exit(1)

    solve(binary)
```

FLAG : LKS{h1dd3n_1n_r4m_67}