IRIS CTF 2025 Now this will run on my 486

Useful knowledge

XOR Operation Property

```
A * B = C
C * B = A
C * A = B
```

[MMAP DOCS](#)

[MPROTECT DOCS](#)

Tools Used

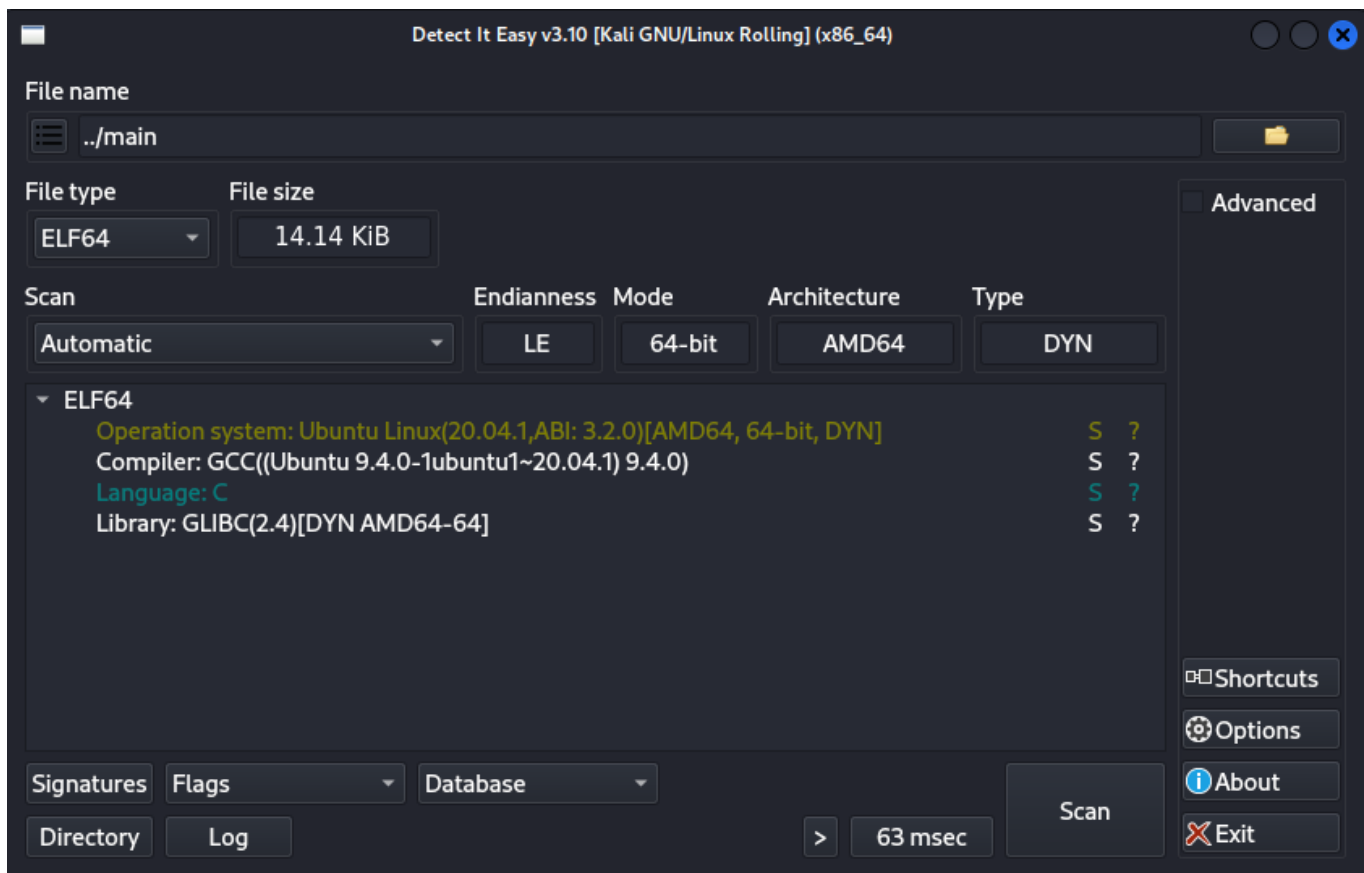[Detect it Easy](#)
[Ghidra](#)
Strings
GDB with [PWNDBG](#)

Analysis Process

Step 1. Download Program

https://cdn.2025.irisc.tf/nowthiswillrunonmy486.tar.gz

Step 2. Open File in Detect It Easy

```
die ./main
```

Step 3. Run strings command on the program

```
strings ./main
```

Not too much to see here but we do find somewhat useful strings

Step 4. Open program in ghidra

Opening the file in ghidra we can go to the entry point for our program

```
void processEntry entry(undefined8 param_1,undefined8 param_2)

{
  undefined auStack_8 [8];


  __libc_start_main(main,param_2,&stack0x00000008,FUN_00101890,FUN_00101900,para
m_1,auStack_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

Here we can find libc_start_main with the first parameter holding the main function for our program

Main function decompiled with most variable names renamed

```
int main(void)
{
  int returnVal;
  long i;
  undefined8 *arrayPointer;
  undefined8 *arrayPointer2;
  long in_FS_OFFSET;
  byte zero;
  sigaction sigActionObj;
  undefined8 array113 [113];
  long stackCanary;

  zero = 0;
  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  sigActionObj.sa_flags = 4;
  sigActionObj.__sigaction_handler.sa_handler = sigillActionHandler;
  sigaction(4,&sigActionObj,(sigaction *)0x0);
  arrayPointer = (undefined8 *)&DAT_00102218;
  arrayPointer2 = array113;
  for (i = 0x6f; i != 0; i = i + -1) {
    *arrayPointer2 = *arrayPointer;
    arrayPointer = arrayPointer + (ulong)zero * -2 + 1;
    arrayPointer2 = arrayPointer2 + (ulong)zero * -2 + 1;
  }
  *(undefined *)arrayPointer2 = *(undefined *)arrayPointer;
  size = 889;
  mappedCode = (code *)mmap((void *)0x0,0x379,0x3,0x22,-1,0);
```

```
    returnVal = 1;
  if (mappedCode != (code *)0xffffffffffffffff) {
     mmapOut2 = mmap((void *)0x0,DAT_00104010,3,34,-1,0);
     if (mmapOut2 != (void *)0xffffffffffffffff) {
        memcpy(mappedCode,array113,size);
        returnVal = mprotect(mappedCode,size,5);
        if (returnVal == -1) {
          returnVal = 1;
        }
        else {
          returnVal = mprotect(mmapOut2,DAT_00104010,3);
          if (returnVal == -1) {
            returnVal = 1;
          }
          else {
            returnVal = (*mappedCode)(0,0,0,0,mmapOut2);
            if (returnVal == 0) {
              puts("Program returned \"correct\"!");
            }
            else {
              __printf_chk(1,"Program returned \"incorrect\"
 (%d).\n",returnVal);
            }
            munmap(mappedCode,size);
            munmap(mmapOut2,DAT_00104010);
          }
        }
     }
  }
  if (stackCanary == *(long *)(in_FS_OFFSET + 0x28)) {
     return returnVal;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

What this program main function does first is initializes a function handler to handle the SIGILL signal sent

```
The SIGILL signal is raised when an attempt is made to execute an invalid,
privileged, or ill-formed instruction.
```

After the function handler has been initialized some operations are done in order to prepare for the call to mmap.

The output of the first mmap call is held in the mappedCode variable which will be called later in main

If all goes well during setup the line of code below calls the mappedCode

```
returnVal = (*mappedCode)(0,0,0,0,mmapOut2);
```

If the value stored in returnVal = 0 we have gotten the correct flag input.

The sigill function handler is a larger function so I wont put the entire function decompilation in to this write up but I will state what it does.

The first thing it does is change the permission on the mapped code to be read/write with the function defined below

```
void setReadWriteMain(void)

{
  int returnCondition;

  returnCondition = mprotect(mappedCode,size,3); // 3 = Read/Write
  if (returnCondition != -1) {
    return;
  }
                  /* WARNING: Subroutine does not return */
  exit(1);
}
```

Then the sigill function handler uses the third parameter passed to it to determine the switch case to take. The switch cases are used to replace the next 8 instructions starting at the point in the mappedCode section we called when we hit a bad instruction

Here is an example of one of the switch cases

```
    case 0xd6:
      switchVar = byteArrayPointer[1];
      bVar1 = byteArrayPointer[2];
      byteArrayPointer[0] = 0x4a;
      byteArrayPointer[1] = 0x8d;
      byteArrayPointer[2] = (byte)((switchVar & 7) << 3) | 4;
      byteArrayPointer[3] = bVar1 & 7;
      byteArrayPointer[4] = 0x90;
      byteArrayPointer[5] = 0x90;
      byteArrayPointer[6] = 0x90;
```

```
            byteArrayPointer[7] = 0x90;
            break;
```

Once the instruction has been replaced by the different switch cases the program changes the permissions on the mappedCode section to read execute with the function definition below

```
void setReadExecMain(void)

{
  int iVar1;

  iVar1 = mprotect(mappedCode,size,5); // 5 = Read/Exec
  if (iVar1 != -1) {
    return;
  }
                /* WARNING: Subroutine does not return */
  exit(1);
}
```

Once the instructions have been replaced the program returns execution flow to the mappedCode section where the bad instruction was hit

Step 5. Determine flag validity checks

Lets run the program in gdb and see what we can find

When we run the program we hit a bad instruction

File   Actions   Edit   View   Help

kali@kali: ~/Desktop ×   kali@kali: ~/Desktop/ghidra_11.2.1_PUBLIC ×   kali@kali: ~/Deskto

```
R9    0
R10   0×22
R11   0×213
R12   0
R13   0×7fffffffdeb8 —▶ 0×7fffffffe243 ◄— 'COLORFGBG=15;0'
R14   0×7ffff7ffd000 (_rtld_global) —▶ 0×7ffff7ffe2e0 —▶ 0×555555554000 ◄— 0×10102464
R15   0
RBP   1
RSP   0×7fffffffd958 —▶ 0×555555555806 ◄— mov ebx, eax
RIP   0×7ffff7fbf000 ◄— 0×1000000017
─────────────────────────────────────────────────────[ DISASM / x86-64 / set
Invalid instructions at 0×7ffff7fbf000
```

```
─────────────────────────────────────────────────────────[ STACK ]─
00:0000│ rsp 0×7fffffffd958 —▶ 0×555555555806 ◄— mov ebx, eax
01:0008│     0×7fffffffd960 —▶ 0×5555555552c8 ◄— endbr64
02:0010│     0×7fffffffd968 —▶ 0×7fffffffdab0 ◄— 0×c4
03:0018│     0×7fffffffd970 ◄— 0
... ↓        2 skipped
06:0030│     0×7fffffffd988 —▶ 0×7ffff7ffe2e0 —▶ 0×555555554000 ◄— 0×10102464c457f
07:0038│     0×7fffffffd990 —▶ 0×7ffff7ffe2e0 —▶ 0×555555554000 ◄— 0×10102464c457f
─────────────────────────────────────────────────────────[ BACKTRACE ]
 ▶ 0   0×7ffff7fbf000 None
   1   0×555555555806 None
   2   0×7ffff7ddbd68 __libc_start_call_main+120
   3   0×7ffff7ddbe25 __libc_start_main+133
   4   0×55555555518e None
pwndbg>
```

Lets examine the bytes at this location

```
pwndbg> x /8i 0×7ffff7fbf000
⇒ 0×7ffff7fbf000:          (bad)
   0×7ffff7fbf001:         add      BYTE PTR [rax],al
   0×7ffff7fbf003:         add      BYTE PTR [rax],dl
   0×7ffff7fbf005:         add      BYTE PTR [rax],al
   0×7ffff7fbf007:         add      BYTE PTR [rdi],dl
   0×7ffff7fbf009:         add      DWORD PTR [rax],eax
   0×7ffff7fbf00b:         data16 ins BYTE PTR es:[rdi],dx
   0×7ffff7fbf00d:          (bad)
pwndbg> x /8bx 0×7ffff7fbf000
0×7ffff7fbf000: 0×17     0×00      0×00      0×00      0×10      0×00      0×00      0×00
pwndbg>
```

We have a bad instruction and then some empty bytes now the sigill signal is sent and the
handler function will be ran.

Switch case handler for the bad instruction hit above as you can see 0x17 was hit

```
case 0x17:
    *byteArrayPointer = byteArrayPointer[1] & 7 | 0xb8;
    *(undefined4 *)(byteArrayPointer + 1) = *(undefined4 *)
(byteArrayPointer+3);
    byteArrayPointer[5] = 0x90;
    byteArrayPointer[6] = 0x90;
    byteArrayPointer[7] = 0x90;
    break;
```

If we let the program continue and wait for the next bad instruction we can see what the handler
does to the previous section

```
pwndbg> x /8i 0×7ffff7fbf000
   0×7ffff7fbf000:          mov      eax,0×1000
   0×7ffff7fbf005:          nop
   0×7ffff7fbf006:          nop
   0×7ffff7fbf007:          nop
⇒ 0×7ffff7fbf008:          (bad)
   0×7ffff7fbf009:         add      DWORD PTR [rax],eax
   0×7ffff7fbf00b:         data16 ins BYTE PTR es:[rdi],dx
   0×7ffff7fbf00d:          (bad)
pwndbg> x /8bx 0×7ffff7fbf000
0×7ffff7fbf000: 0×b8     0×00      0×10      0×00      0×00      0×90      0×90      0×90
```

As you can see the bad instruction has been replaced with the mov eax, 0x1000 instruction and
some trailing nop instructions

If we continue to follow this process and continue to examine the previous memory section we
can start to see the operations that actually check the flag validity

Here we can see that a value is taken from DWORD PTR and stored into eax then the value 0xFF is stored in ecx and the values have an and operation performed the result of that operation is stored into eax and then added to edx

```
0×7ffff7fbf0b0:        mov      eax,DWORD PTR [rax+r8*1]
0×7ffff7fbf0b4:        nop
0×7ffff7fbf0b5:        nop
0×7ffff7fbf0b6:        nop
0×7ffff7fbf0b7:        nop
0×7ffff7fbf0b8:        mov      ecx,0×ff
0×7ffff7fbf0bd:        nop
0×7ffff7fbf0be:        nop
0×7ffff7fbf0bf:        nop
0×7ffff7fbf0c0:        and      eax,ecx
0×7ffff7fbf0c2:        nop
0×7ffff7fbf0c3:        nop
0×7ffff7fbf0c4:        nop
0×7ffff7fbf0c5:        nop
0×7ffff7fbf0c6:        nop
0×7ffff7fbf0c7:        nop
0×7ffff7fbf0c8:        add      edx,eax
```

The DWORD PTR holds the value of the flag given by user input

The instruction above really take each character in the user supplied flag and then and's the character with the value 0xFF and added to a total held in edx

Later down the line if we continue this same method the value in edx is compared with 0xCFF

```
pwndbg> x /24i 0×7ffff7fbf100
0×7ffff7fbf100:        mov      eax,0×cff
0×7ffff7fbf105:        nop
0×7ffff7fbf106:        nop
0×7ffff7fbf107:        nop
0×7ffff7fbf108:        cmp      eax,edx
```

This is the first check on our flag. If the total value of the user supplied string with the and operations explained above results in the total 0xCFF we can continue on to further checks on the flag otherwise the program returns that our flag was incorrect

If we follow the same method of allowing the sig ill handler to continue to fix the bad instructions we get to the next check

The next check takes the user given flag 4 characters at a time converts the characters into hex and then applies a mask to it with an XOR operation then compares the value with the expected result. If it is the desired result it goes to the next 4 characters and repeats until the flag has

been validated

```
0x7ffff7fbf130:     mov     eax,DWORD PTR [rcx+r8*1]
0x7ffff7fbf134:     nop
0x7ffff7fbf135:     nop
0x7ffff7fbf136:     nop
0x7ffff7fbf137:     nop
0x7ffff7fbf138:     mov     edx,0xbf51b0d7
0x7ffff7fbf13d:     nop
0x7ffff7fbf13e:     nop
0x7ffff7fbf13f:     nop
0x7ffff7fbf140:     xor     edx,eax
0x7ffff7fbf142:     nop
0x7ffff7fbf143:     nop
0x7ffff7fbf144:     nop
0x7ffff7fbf145:     nop
0x7ffff7fbf146:     nop
0x7ffff7fbf147:     nop
0x7ffff7fbf148:     mov     eax,0xcc38c2be
0x7ffff7fbf14d:     nop
0x7ffff7fbf14e:     nop
0x7ffff7fbf14f:     nop
0x7ffff7fbf150:     cmp     eax,edx
```

If we take the mask and XOR it with the desired output we can reverse the operation and find out the correct 4 characters we need for the flag in reverse order

0xbf51b0d7 ^ 0xcc38c2be = 0x73697269 = 'siri'

The rest of the mask and desired outputs can be recovered if you follow this same method.

Lastly ill provide a python script I wrote in order to solve the challenge

Example Wrong Input

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./main
flag? blahblah
Program returned "incorrect" (1).
```

Example Correct Input

```
└─$ ./main
flag? irisctf{wow_very_optimal_code!!}
Program returned "correct"!
```

Solution Script

```python
def decrypt(input_value, target_value):
        hex_value = input_value ^ target_value
        bytes_value = hex_value.to_bytes(4, byteorder='big')

        ascii_string = bytes_value.decode('ascii')
```

```
        return ascii_string

maskArray =
[0xbf51b0d7,0x75cc547b,0x4f0fd83a,0xa2117744,0xecd0cec6,0x2e19f9fa,0x32ea83d9,
0xe5eb61e0]
comperativeArray =
[0xcc38c2be,0xeaa2018,0x1078b74d,0xdb631232,0x98a0a199,0x42789493,0x5685e086,0
x98ca4085]

flag = ""
for i in range(len(maskArray)):
        flag += decrypt(maskArray[i],comperativeArray[i])[::-1]

print(flag)
```