



Reverse Engineering Challenges Documentation for Hexacon 2023

Windows Challenge

Linux Challenge

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.

Windows Challenge

In this challenge, we are presented with a Windows executable.

The goal in this challenge is to decrypt the flag that we will find inside the executable.

Description for the players:

- This challenge consists of a flag checker that takes a flag as input and uses internal checks to validate it.

Flag: FLAG{upNwCXbxOtID0ybEIsWEHJixsmgoreOznusN0}

Difficulty: Medium

Needed Software: A Windows debugger(IDA preferably) and VS code

Solution:

We start by running the program. At first, it waited for an input, and after we gave it a random one it gave us a message that indicated that our input was wrong.

```
C:\Users\arfao\OneDrive\Desktop\work\Yogosha\Hexacon\Windows_chall>.\Windows_chall.exe
Welcome to my flag checking tool
Give me the flag and i will check if it's correct or not
adssad
Wrong, try again.
```

So we dig right into Dynamic analysis using IDA:



Using Ida we notice something weird the program has only a few instructions and it ends with an `int3` instruction which is used to make a software breakpoint.

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.

```

.text:00000001400016DE mov     [rbp+40h+var_28], rax
.text:00000001400016E2 lea     rdx, Handle      ; Handler
.text:00000001400016E9 xor     ecx, ecx        ; First
.text:00000001400016EB call    cs:AddVectoredExceptionHandler
.text:00000001400016F1 int     3                ; Trap to Debugger
.text:00000001400016F2 ; -----
.text:00000001400016F2 lea     rcx, Handle      ; Handle
.text:00000001400016F9 call    cs:RemoveVectoredExceptionHandler
.text:00000001400016FF call    cs:IsDebuggerPresent
.text:0000000140001705 test    eax, eax
.text:0000000140001707 jz     short loc_140001712
.text:0000000140001709 xor     ecx, ecx        ; Code
.text:000000014000170B call    cs:__imp_exit
.text:000000014000170B ; -----
.text:0000000140001711 db     0CCh ; i
.text:0000000140001712 ; -----
.text:0000000140001712 loc_140001712:                ; CODE XREF: main+57↑j
.text:0000000140001712 mov     rax, gs:58h
.text:000000014000171B mov     rdi, [rax]
.text:000000014000171E mov     ecx, 1D8h
.text:0000000140001723 mov     eax, [rcx+rdi]

```

Changing to text view we find the rest of the program but according to IDA execution flow the program will never go there so it must be some kind of protection.

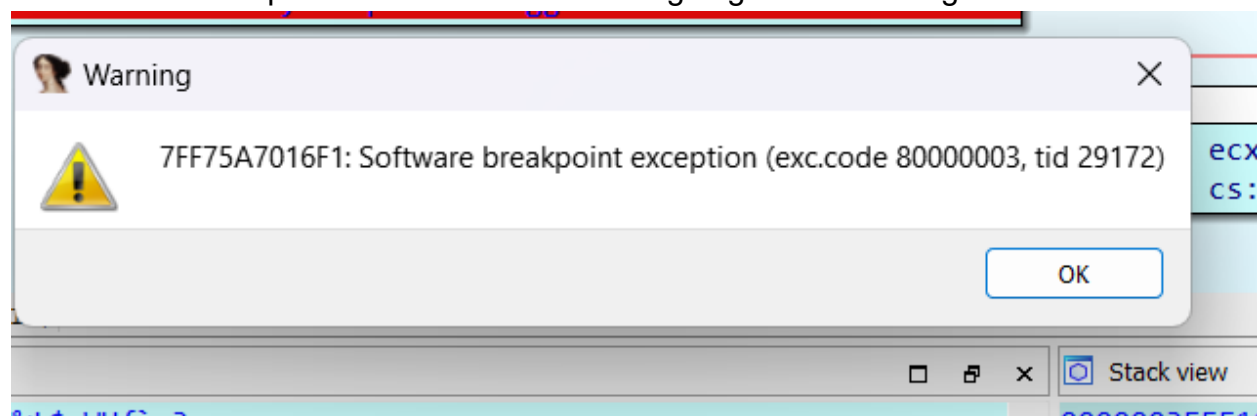
So now we go to debugging so we can understand more:

```

.text:00007FF75A7016D4 mov     rax, cs:__security_cookie
.text:00007FF75A7016DB xor     rax, rsp
.text:00007FF75A7016DE mov     [rbp+40h+var_28], rax
.text:00007FF75A7016E2 lea     rdx, Handle      ; Handler
.text:00007FF75A7016E9 xor     ecx, ecx        ; First
.text:00007FF75A7016EB call    cs:AddVectoredExceptionHandler
.text:00007FF75A7016F1 int     3                ; Trap to Debugger

```

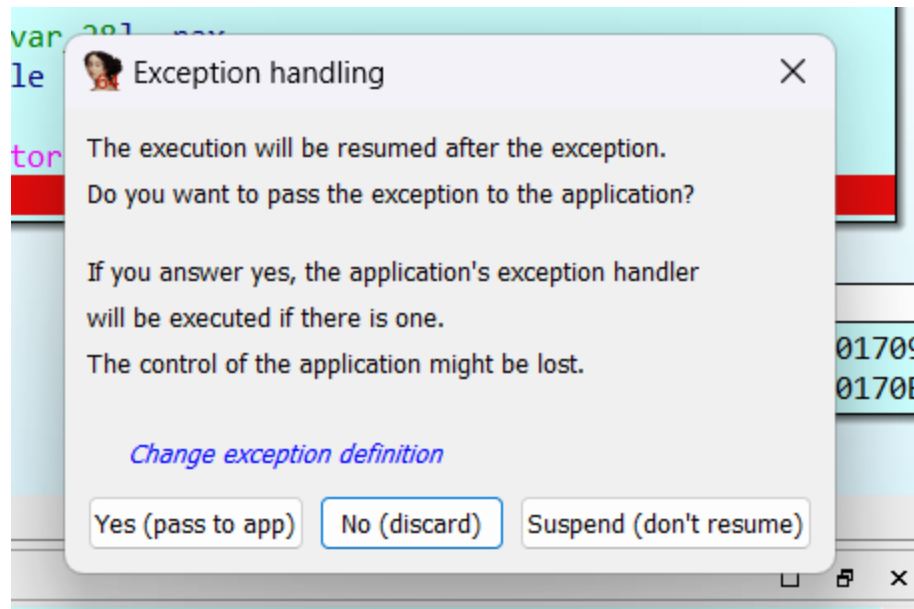
We've set a breakpoint at the int3 now we are going to follow along



After executing the int3 instruction an exception is invoked.

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.



We let the program handle the exception. And when we do that the app terminates so it must continue executing without doing anything so the solution we try to breakpoint before every exit we find.

After trying it out we find that this is the function that handles the exception:

```
.text:00007FF75A701450      push    rbx
.text:00007FF75A701452      sub     rsp, 20h
.text:00007FF75A701456      mov     rbx, [rcx+8]
.text:00007FF75A70145A      lea     rdx, aWelcomeToMyFla ; "Welcome to my flag checking tool"
.text:00007FF75A701461      mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@10A ; std::ostream std::cout
.text:00007FF75A701468      call    sub_7FF75A702DA0
.text:00007FF75A70146D      mov     rcx, rax
.text:00007FF75A701470      lea     rdx, sub_7FF75A702F70
.text:00007FF75A701477      call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z ; std::ostream::operator<<(s
.text:00007FF75A70147D      call    cs:IsDebuggerPresent
.text:00007FF75A701483      test    eax, eax
.text:00007FF75A701485      jz      short loc_7FF75A701490
.text:00007FF75A701487      xor     ecx, ecx ; Code
.text:00007FF75A701489      call    cs:__imp_exit
.text:00007FF75A701489 ; -----
```

It also has the message that prints out at the beginning.

So now we continue debugging:

```
.text:00007FF75A70146D      mov     rcx, rax
.text:00007FF75A701470      lea     rdx, sub_7FF75A702F70
.text:00007FF75A701477      call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z ; std::ostream::operator<<(s
.text:00007FF75A70147D      call    cs:IsDebuggerPresent
.text:00007FF75A701483      test    eax, eax
.text:00007FF75A701485      jz      short loc_7FF75A701490

007FF75A701487 xor     ecx, ecx ; Code
007FF75A701489 call    cs:__imp_exit

.text:00007FF75A701490      loc_7FF75A701490:
.text:00007FF75A701490      call    sub_7FF75A701510
.text:00007FF75A701495      mov     rcx, rax
.text:00007FF75A701495      sub     rcx, 10h
```

We run into our first anti-debugging technique but it's quite easy it calls the `IsDebuggerPresent` function and compares the result so an easy bypass is to zero the `EAX` register so we can pass the if statement

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.

SO continuing debugging we notice that the function doesn't do anything besides taking us in a loophole so the best fix is to change the int3 with nop instruction so the exception doesn't happen and the program continues normally.

```

:6DB5716E2      lea     rdx, Handle      ; Handler
:6DB5716E9      xor     ecx, ecx         ; First
:6DB5716EB      call   cs:AddVectoredExceptionHandler
:6DB5716F1      nop
:6DB5716F2      lea     rcx, Handle      ; Handle
:6DB5716F9      call   cs:RemoveVectoredExceptionHandler
:6DB5716FF      call   cs:IsDebuggerPresent
:6DB571705      test    eax, eax
:6DB571707      jz      short loc_7FF6DB571712
:6DB571709      xor     ecx, ecx         ; Code
:6DB57170B      call   cs: __imp__exit

```

Now moving on we can see that the program is checking for a file existence:

```

.text:00007FF6DB57177C xor     byte ptr [rbx+6], 17h
.text:00007FF6DB571780 xor     byte ptr [rbx+7], 0D5h
.text:00007FF6DB571784 xor     byte ptr [rbx+8], 0E5h
.text:00007FF6DB571788 xor     byte ptr [rbx+9], 13h
.text:00007FF6DB57178C xor     byte ptr [rbx+0Ah], 7Dh
.text:00007FF6DB571790 xor     byte ptr [rbx+0Bh], 9Dh
.text:00007FF6DB571794 xor     byte ptr [rbx+0Ch], 77h
.text:00007FF6DB571798 mov     byte ptr [rbx+0Dh], 0
.text:00007FF6DB57179C loc_7FF6DB57179C: ; CODE XREF: main+B31j
.text:00007FF6DB57179C lea     rdx, [rbp+40h+var_C0] ; Stat
.text:00007FF6DB5717A0 mov     rcx, rbx ; FileName
.text:00007FF6DB5717A3 call   cs:_stat64i32
.text:00007FF6DB5717A9 test    eax, eax
.text:00007FF6DB5717AB jnz     loc_7FF6DB571712
.text:00007FF6DB5717B1 mov     ecx, 539h ; Seed
.text:00007FF6DB5717B6 call   cs:srand
.text:00007FF6DB5717BC movdqa xmm0, cs:xmmword_7FF6DB575770
.text:00007FF6DB5717C4 movdqa [rsp+140h+var_120], xmm0

```

A simple bypass is to change the register before the if statement.

Now everything is clear we can use the decompiler to see the code:

```

for ( i = v10; ; ++i )
{
    v15 = &v29;
    if ( v9 )
    {
        do
            ++v15;
        while ( *v15 );
    }
    if ( v12 >= (int)((_DWORD)v15 - (unsigned int)&v29) )
        break;
    if ( IsDebuggerPresent() )
        goto LABEL_38;
    *i = (unsigned __int8)*v13 + (~(unsigned __int8)*(v13 - 1) << 8);
    v12 += 2;
    v13 += 2;
}
v16 = v25;
v17 = (char *)v10 - (char *)v25;
while ( 1 )
{
    v18 = &v29;
    if ( v9 )
    {
        do
            ++v18;
        while ( *v18 );
    }
    if ( v11 >= (int)((_DWORD)v18 - (unsigned int)&v29) / 2 )
        break;
}

```

At first, it loads a vector with some values we can get from debugging and then checks the length which is 43 and the remaining is to decipher this code:

cipher[j] = (~(flag[i]) << 8) + flag[i + 1];

This is a simple script that does it:

```

enc = [-18100,-16825,-31627,-28850,-30653,-22686,-30897,-29844,-17585,-
31134,-17812,-29609,-17848,-19095,-30861,-28057,-28558,-26033,-31378,-
30093,-20176,-32256]
flag1=""
c = 0
for i in range(len(enc)):
    flag1+=chr(enc[i] & 0xFF)
    enc[i]=enc[i]-(enc[i] & 0xFF)
flag2=""
c=0
for i in range(len(enc)):
    flag2+= chr((~(enc[i]))>>8)
    c+=2

```

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.

```
flag=""  
for i in range(len(flag1)):  
    flag+=flag2[i]+flag1[i]  
print(flag)
```

Linux Challenge

The first layer of protection involves an obfuscated loader. The challenge lies in deciphering the code, which is not clear until execution time. Once you've uncovered the code, your task is to generate a flag based on the conditions specified within it.

Description for the players:

- This challenge is a keygen where you need to generate a flag that satisfies certain conditions. The binary is protected.

Flag: FLAG{DC_I_th1nk_yOu_mad3_4_B1G_mil3ston3_R3V3RS3R_K33p_G01ng}

Difficulty: Hard

Needed Software: A Linux debugger(IDA preferably) and VS code and intel pintool

Solution:

this challenge consists of two main part the first is the loader bypass:

the first thing to do to bypass the loader is to disable aslr in your system.

then you need strace the binary to trace system calls because there is heavy use of mmap indicating some kind of unpacker used.

Using intelpintool:

```
0xb0002049c: 262(0x3, 0xb00029b55, 0x7fffffffdb0, 0x1000, 0x0, 0xb00000301, returns: 0x0
0xb00021175: 9(0x0, 0x17b2f, 0x1, 0x2, 0x3, 0x0) returns: 0x7ffe44bc000
0xb00020e99: 3(0x3, 0x17b2f, 0x1, 0x2, 0x3, 0x0) returns: 0x0
0xb00020fcf: 257(0xffffffff9c, 0xb00034f80, 0x80000, 0x0, 0x7fffffff2f7, 0x0) returns: 0x3
0xb00021026: 0(0x3, 0x7fffffff318, 0x340, 0x0, 0x7fffffff2f7, 0x0) returns: 0x340
0xb0002105c: 17(0x3, 0x7fffffff30, 0x310, 0x40, 0x7fffffff2f7, 0x0) returns: 0x310
0xb00020d9c: 262(0x3, 0xb00029b55, 0x7fffffffdb0, 0x1000, 0xb00034f80, 0xb000342c0) returns: 0x0
0xb00021175: 9(0x0, 0x2000, 0x3, 0x22, 0xffffffff, 0x0) returns: 0x7ffe43dc000
0xb0002105c: 17(0x3, 0x7ffffffce00, 0x310, 0x40, 0xffff, 0x0) returns: 0x310
0xb00021175: 9(0x0, 0x1e6d70, 0x1, 0x802, 0x3, 0x0) returns: 0x7ffe4145000
0xb00021175: 9(0x7ffe4167000, 0x15a000, 0x5, 0x812, 0x3, 0x22000) returns: 0x7ffe4167000
0xb00021175: 9(0x7ffe42c1000, 0x58000, 0x1, 0x812, 0x3, 0x17c000) returns: 0x7ffe42c1000
0xb00021175: 9(0x7ffe4319000, 0x6000, 0x3, 0x812, 0x3, 0x1d4000) returns: 0x7ffe4319000
0xb00021175: 9(0x7ffe431f000, 0xcd70, 0x3, 0x32, 0xffffffff, 0x0) returns: 0x7ffe431f000
0xb00020e99: 3(0x3, 0x7ffe4145378, 0x3, 0xffff80001ebacbc, 0xc0008002, 0x50) returns: 0x0
0xb00021175: 9(0x0, 0x2000, 0x3, 0x22, 0xffffffff, 0x0) returns: 0x7ffe3e3000
0xb0001ba93: 158(0x1002, 0x7ffe43dd640, 0x7ffe3fe5000, 0x22, 0xffffffff, 0x0) returns: 0x0
0xb0001278c: 218(0x7ffe43dd910, 0x7ffe43dd640, 0xb000340b0, 0x22, 0xffffffff, 0x0) returns: 0x11a9
0xb000127e8: 273(0x7ffe43dd920, 0x18, 0xb000340b0, 0x22, 0xffffffff, 0x0) returns: 0x0
0xb0001281f: 334(0x7ffe43ddf60, 0x20, 0x0, 0x53053053, 0x0, 0x0) returns: 0x0
0xb000211e9: 10(0x7ffe4319000, 0x4000, 0x1, 0x7ffe431cb10, 0xc157, 0x0) returns: 0x0
0xb000211e9: 10(0x800004000, 0x1000, 0x1, 0x800004f18, 0x3abf8d2, 0x7ffe4194990) returns: 0x0
0xb000211e9: 10(0xb00031000, 0x2000, 0x1, 0xb00032f20, 0x15c0adc, 0x0) returns: 0x0
0x7ffe4240fb2: 302(0x0, 0x3, 0x0, 0x7fffffffdcf0, 0x0, 0xc) returns: 0x0
0xb000211b9: 11(0x7ffe44bc000, 0x17b2f, 0xb00033000, 0x7fffffffdcf0, 0x0, 0xc) returns: 0x0
0x7ffe4242b6c: 101(0x0, 0x0, 0x1, 0x0, 0xffffffff, 0xb00004d70) returns: 0x0
0x7ffe41d7343: 318(0x7ffe43242b8, 0x8, 0x1, 0x0, 0xffffffff, 0xb00004d70) returns: 0x8
0x7ffe4241419: 12(0x0, 0x7ffe431daa0, 0x0, 0x7ffe431db00, 0x2, 0x0) returns: 0xa06000
```

This is the dump file of strace

Know we look for the executable part

This documentation is the property of Binary Gecko GmbH and is intended for educational use only.

© 2023 Binary Gecko. All rights reserved.


```

0x200e80: 10(0x800000000, 0x778, 0x1, 0x778, 0x0, 0x800000000)returns: 0x0
0x200e17: mmap(0x800001000, 0x1065, 0x2, 0x22, 0xffffffff, 0x0)returns: 0x80000
0x200e80: 10(0x800001000, 0x1065, 0x5, 0x1065, 0x0, 0x800001000)returns: 0x0
0x200e17: mmap(0x800003000, 0x1b8, 0x2, 0x21, 0xffffffff, 0x0)returns: 0x800003
0x200e80: 10(0x800003000, 0x1b8, 0x1, 0x1b8, 0x0, 0x800003000)returns: 0x0
0x200e17: mmap(0x800004000, 0x1018, 0x2, 0x22, 0xffffffff, 0x0)returns: 0x80000
0x200e80: 10(0x800004000, 0x1018, 0x3, 0x1018, 0x0, 0x800004000)returns: 0x0
0x200d90: 2(0xa02270, 0x0, 0x0, 0x0, 0x0, 0xa02270)returns: 0x3
0x200d2d: 0(0x3, 0x7fffffffcf40, 0x40, 0x40, 0x3, 0x7fffffffcf40)returns: 0x40
0x200d5d: 8(0x3, 0x40, 0x0, 0x0, 0x3, 0x40)returns: 0x40
0x200d2d: 0(0x3, 0x7fffffffcf00, 0x38, 0x38, 0x3, 0x7fffffffcf00)returns: 0x38
0x200e17: mmap(0xb00000000, 0xeb8, 0x1, 0x12, 0x3, 0x0)returns: 0xb00000000

```

This region has execute permission

So the solution here is to dump that memory region at runtime after the program has done unpacking:

We run the binary and we cat the code:

```

ps -edf | grep 'rev1'
0:00:00 ./rev1.bin
0:00:00 grep rev1
cat /proc/4578/maps

```

```

00a06000-00a27000 rw-p 00000000 0
[heap]
8000000000-800001000 r--p 00000000
800001000-800003000 r-xp 00000000
800003000-800005000 r--p 00000000

```

This is the region +

After we dump we will have a normal binary readable by IDA

```

xx dest[30] + dest[27] == 1608
&& dest[30] + dest[27] + 1337 + dest[34] == 1608
&& dest[31] + 2 * dest[34] - dest[32] - dest[33] == -3
&& dest[10] == dest[14]
&& dest[2] + dest[35] + 1337 + dest[33] == 1637
&& dest[37] - dest[2] == -44
&& dest[36] + dest[15] + 1337 + *dest == 1596
&& dest[37] + dest[36] + 1337 * dest[2] == 127148
&& dest[34] == dest[39]
&& dest[14] == dest[19]
&& dest[37] + dest[34] - dest[38] == 16
&& 2 * dest[39] + dest[36] == 184
&& (dest[40] ^ (dest[39] >> 1) & (2 * dest[38])) != (dest[1] == 25)
&& dest[39] + 2 * dest[41] - dest[40] - dest[43] == 53
&& dest[45] + dest[42] + 1337 * dest[37] == 68313
&& dest[41] == 83
&& dest[46] == 51
&& dest[19] == dest[21]
&& dest[55] == 125
&& (char)(dest[39] ^ dest[27] ^ dest[18] ^ dest[21] ^ dest[52] ^ dest[31] ^ dest[25] ^ dest[14] ^ dest[49]) != (dest[53] == 66)
&& dest[55] - dest[47] == 74
&& dest[45] == 75
&& dest[21] == dest[25]
&& (dest[49] ^ (dest[48] >> 1) & (2 * dest[46])) != (dest[47] == 76)
&& ~(49374 * dest[48]) + dest[55] == -5529764
&& dest[49] + 2 * dest[51] - dest[50] - dest[53] == 10
&& dest[54] + dest[55] - dest[52] == 179
&& 2 * dest[55] + dest[52] == 299
&& dest[25] == dest[35]
&& dest[54] + dest[46] + 1337 * dest[51] == 64330
&& dest[53] + dest[55] + 1337 + *dest == 1640 )

```

We can see now all the conditions with IDA know using Z3 we write a solution (in the files you will find the solver.py)