Flare-On 10 Challenge 12: HVM

By Dhanesh Kizhakkinan (@dhanesh_k)

Initial analysis

Given file is a Windows x64 executable. Loading into IDA provides clean disassembly and decompilation. Skimming through the decompiled code, one can quickly find the usage of Windows Hypervisor Platform API (https://learn.microsoft.com/en-us/virtualization/api/hypervisor-platform/hypervisor-platform) from WinHVPlatform.dll. This gives us a hint that the challenge uses Hyper-V for unknown functionality. Further reading the code and referring to API documentation, all the while renaming the variables and adding missing types, we can see that the Hyper-V platform APIs are used to create a VM and run a piece of shellcode (small OS). The shellcode itself is stored as a resource entry. We can use tools like CFF explorer to dump the shellcode and try to disassemble/decompile. There are 2 inputs from command line arguments to the binary and are passed to the VM by copying them to the mapped shellcode.

Is it 16 bit/32 bit/64 bit?

Once we try to load the shellcode to IDA, the question whether this a 16/32/64 shellcode pops up. Modern Intel based processors still start in 16-bit mode and require setting flags in control registers to move to 32 and 64-bit mode. So, for our initial disassembly, we start with 16-bits. Disassembling at the start provides us with
a small set of instructions, setting the stack pointer and loading GDT and setting the protected mode flag in CR0.

```
seg000:0000 BC 00 00
seg000:0003 FA
seg000:0004 0F 01 16 26 0D
seg000:0009 0F 20 C0
seg000:000C 66 83 C8 01
seg000:0010 0F 22 C0
seg000:0013 EA 18 00 00 00
mov sp, 8000h
cli
lgdt fword ptr ds:byte_D26
mov eax, cr0
or eax, 1
mov cr0, eax
jmp far ptr loc_98
```

**Figure 2: 16-bit startup code**

IDA is confused by the last jmp. The 0xEA points to a segment:offset(8:18) style far jmp. So, the jump sets CS as 8 and EIP as 0x18. Now, the instructions at 0x18 are 32-bit and we can either open the file again as 32-bit or create a new segment starting from 0x18 as a 32-bit segment.

```
sub_18 proc near
mov ax, 10h
mov ds, eax
assume ds:nothing
mov fs, eax
assume fs:nothing
mov gs, eax
assume gs:nothing
mov ss, eax
assume ss:nothing
call sub_37 ; setup 64 bit
lgdt fword ptr ds:byte_E44
jmp far ptr unk_D72 ; jmp to 64 bit
sub_18 endp
```

**Figure 3: 32-bit code**

32-bit code does more operations to move to 64-bit and we see another jump (wrongly calculated again by IDA). The real jump is once again segment:offset which sets CS as 8 and RIP as 0xCF2. We need to reload the shellcode as 64-bit to further analyze the shellcode.

```
;                                     mov ax, 10h
seg000:000000000000CF2 66 B8 10 00    mov ds, eax
seg000:000000000000CF6 6E DB           assume ds:nothing
seg000:000000000000CF8 BE E0           mov fs, eax
seg000:000000000000CFA assume fs:nothing
seg000:000000000000CFE BE EB           mov gs, eax
seg000:000000000000CFE assume gs:nothing
seg000:000000000000CFE 4B B8 EB AD DE EB    mov ss, eax
seg000:000000000000CFE assume ss:nothing
seg000:000000000000CFE 4B B8 EB AD DE EB    mov rax, 0DEADBEFDEADBEEFh
seg000:000000000000CFE AD DE
seg000:00000000000000DB EB A5 FE FF FF
seg000:00000000000000D4 hlt
```

**Figure 4: 64-bit code**

The 64-bit code is fairly simple. It sets RAX to 0xDEADBEFDEADBEEF and calls a function at 0xBB2.
Function at 0xBB2 starts with few valid instructions but soon the disassembly fails. One peculiar instruction stands out – the IN instruction. IN/OUT instructions are used for IO port access and are special for hypervisors as it causes VM exits. If we look back at the decompilation of HVM.EXE, we can see special handlers for the IO port access.

Figure 5: Encrypted code

```
Figure 5: Encrypted code

Function at 0xBB2 starts with few valid instructions but soon the disassembly fails. One peculiar instruction stands out – the IN instruction. IN/OUT instructions are used for IO port access and are special for hypervisors as it causes VM exits. If we look back at the decompilation of HVM.EXE, we can see special handlers for the IO port access.
```
while ( cont_exec )
{
    if ( WHvRunVirtualProcessor(Partition, 0, &ExitContext, 0xE0u) >= 0 )
    {
        ExitReason = ExitContext.ExitReason;
        if ( ExitContext.ExitReason == WHvRunVmExitReasonX64IoPortAccess )
        {
            Get_RIP_R8_R9(Partition, &vm_rip);
            if ( (ExitContext.IoPortAccess.AccessInfo.AsUINT32 & 1) != 0 )
                RC4(shellcode, vm_rip - 16, vm_r9, vm_r9, vm_r8);
            else
                RC4(shellcode, vm_rip + 2, vm_r9, vm_r8);
            Add_RIP_2(Partition);
        }
        else if ( ExitReason == WHvRunVmExitReasonX64Halt )
        {
            success = Get_RAX(Partition);
            cont_exec = 0;
        }
        else
        {
            cont_exec = 0;
        }
    }

Figure 6: IO port VM exit handler

From the decompiled code, we see that the IO port access handler retrieves current RIP, R8 and R9 registers and passes them as arguments along with shellcode to an RC4 implementation. Further reading the RC4 implementation, we can deduce that the shellcode is decrypted with the key in R8 and shellcode length in R9 register. Looking at the length we can guess that it is not for the whole shellcode but only a small part – likely a function. We can also re-verify this dynamically by putting a breakpoint at the RC4 decryption routine and analyzing the resulting shellcode changes.

Another thing we see is the if..else condition which checks for IO port AccessInfo. The IO instructions can read from the port or write to the port. We see that the if..else condition handles the RC4 invocation differently based on whether the instruction is for read operation or a write operation. Further looking at the shellcode and debugging, we can see that after decrypting and running the decrypted shellcode, the OUT instruction re-encrypts the function. This blocks us from dumping the completely decrypted shellcode at the end of execution from memory.

There are multiple ways to decrypt the whole shellcode – either write a script to disassemble, find the IN/OUT instructions and decrypt or let the challenge run and dump decrypted shellcode at the IN instruction to a separate file. The second one requires fewer lines of code and likely can be achieved by writing a debugger script.

After dumping the decrypted shellcode, we have a much better looking program to analyze.
Figure 7: Decrypted function

Looking at function 0xBB2, we see a call to function 0xB3F with 2 specific parameters. Further analyzing 0xB3F and child functions, we see that 0xFE00 and 0xFC00 are supposed to be pointers to strings. Looking at the static shellcode, those locations are NULL. Looking at the shellcode while executing from HVM.EXE, we can see the two memory locations contain the data from the command line arguments. We can name those variables as name and serial.

```c
_int64 __fastcall sub_B3F(char *name, char *serial, __int64 a3, __int64 a4) {
    __int64 result;  // rax
    __int64 v5;      // [rsp-10h] [rbp-10h]

    __inbyte(3u);
    HIDWORD(v5) = CheckName(name);
    LODWORD(v5) = CheckSerial(((unsigned int *)name, serial);
    if ( v5 == 0x1337LL )
        result = 0x1337LL;  // success
    else
        result = 0LL;
    __outbyte(3u, result);
    return result;
}
```

Figure 8: Check function

Looking at the function, it calls two more functions. One of them checks the validity of the name and the other checks the validity of the serial.

The CheckName function xors two hard coded strings and compares the result to the name variable and returns the count of characters which are the same. This resulting count is compared to 0x24 (36 in decimal) to be a valid name. We can xor the strings and get the expected name.
We get the name: FLARE2023FLARE2023FLARE2023FLARE2023FLARE2023

Now onto the CheckSerial function. This function calls two different functions. The first one is a Base64 decode implementation. This can be deduced by either analyzing/debugging the code or looking at the lookup table which is used (0x40, 0x40...) in the code. The other cryptographic function is much more involved and needs proper analysis.
The decryption function creates a keystream using salsa20 algorithm with the first DWORD of name as the key. The serial (base64 decoded) is split into QWORDs. Two QWORDs and keystream passed to another function, DecryptBlock.

```c
__int64 __fastcall SomeDecryption(__int64 *decoded_serial, int decode_len, int name_first_dword)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    __inbyte(3u);
    v11 = v3;
    memset(out, 0, sizeof(out));
    for ( i = 0; i <= 15; ++i )
        in[i] = name_first_dword;
    salsa20_block((__int64*)out, (__int64*)in); // get salsa key stream
    chunk_len = decode_len / 8;
    decoded = decoded_serial;
    for ( index = 0; index < chunk_len )
    {
        result = (unsigned int)index;
        if ( index >= chunk_len )
            break;
        DecryptBlock(&decoded[index], &decoded[index + 1], out);
    }
    __outbyte(3u, index);
    return result;
}
```

Figure 12: Block decryption

Overall, the whole algorithm can be summarized as
1. Create a key stream using salsa20 with the first DWORD of the name as the key.
2. Split the base64 decoded serial into QWORDS and decrypt two QWORDS (16-byte block length) at a time.
3. Decryption is an 8 round xor sequence (7 to 0 loop) with the keystream.

To reverse this algorithm, we need to reverse the loop (0 to 7) and swap data1 and data2. The rest of the keystream generation remains the same.

Once all the validations are completed, the serial is used in an xor loop to calculate the final flag.

Final algorithm:

```python
import base64
import struct

def u32(b):
    return struct.unpack("<I", b)[0]

def u64(b):
    return struct.unpack("<Q", b)[0]

def p32(x):
    return struct.pack("<I", x)

def p64(x):
    return struct.pack("<Q", x)

def xor(a,b):
    return bytes([i^j for i,j in zip(a,b)])

def salsa20_step(state):
    x = state[:]
    def ROTL(a,b):
        return ((a << b) | (a >> (32 - b))) & 0xFFFFFFFF
    def QR(a,b,c,d):
        x[b] ^= ROTL((x[a] + x[d]) & 0xFFFFFFFF, 7)
```

```c
unsigned __int8 __fastcall DecryptBlock(__int64 *data1, __int64 *data2, __int64 *key_stream)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    result = __inbyte(3u);
    for ( i = 7; i >= 0; --i )
    {
        v5 = *data1;
        *data1 ^= RoundFunc(*data2, i, key_stream); // data1 ^= data2 ^ keystream[i]
        result = (unsigned __int8)data2;
        *data2 = v5;
        __outbyte(3u, result);
    }
    return result;
}
```
\[ x[c] \^= \text{ROTL}((x[b] + x[a]) \& 0xFFFFFFFF, 9) \\
x[d] \^= \text{ROTL}((x[c] + x[b]) \& 0xFFFFFFFF, 13) \\
x[a] \^= \text{ROTL}((x[d] + x[c]) \& 0xFFFFFFFF, 18) \]

for i in range(10):
    QR( 0,  4,  8, 12)
    QR( 5,  9, 13,  1)
    QR(10, 14,  2,  6)
    QR(15,  3,  7, 11)
    QR( 0,  1,  2,  3)
    QR( 5,  6,  7,  4)
    QR(10, 11,  8,  9)
    QR(15, 12, 13, 14)

out = b''
for i in range(16):
    out += p32((state[i] + x[i]) \& 0xFFFFFFFF)
return out

name = b'FLARE2023FLARE2023FLARE2023FLARE2023\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
salsa20_state = [u32(name[:4])] * 16
keystream = salsa20_step(salsa20_state)
keystream = [u64(keystream[i:i+8]) for i in range(0, len(keystream), 8)]
data = [u64(name[i:i+8]) for i in range(0, len(name), 8)]

... inverse of
for j in range(7, -1, -1):
    tmp = data[i]
    data[i] ^= data[i+1] ^ keystream[j]
    data[i+1] = tmp

... for i in range(0, len(data), 2):
    for j in range(8):
        tmp = data[i+1]
        data[i+1] ^= data[i] ^ keystream[j]
        data[i] = tmp

data = base64.b64encode(b''.join(p64(i) for i in data))
final_xor = b'\x19v7/=\x1d&?{\x069X\x12#%k*\x07<8\x18h\x16\x1c0\t4#\x08[!$6aj&j\x0fD]\x06'
print("Name:", name.decode("ascii"))
print("Serial:", data.decode("ascii"))
print("Flag:", xor(data, final_xor)[:len(final_xor)].decode('ascii') + '@flare-on.com')

Flag:
Name: FLARE2023FLARE2023FLARE2023FLARE2023
Serial: zBYpTBWlYvf9MUH4KtcYv7sdUVUPcj0CiU5G5i63bb+LLBZsAmEk9Y1NMplv5SiN
Flag: c4n_i_sh1p_a_vm_as_an_exe_ask1ng_4_a_frnd@flare-on.com