black hat®
BRIEFINGS

AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS

# Dark Corners: How a Failed Patch Left VMware ESXi VM Escapes Open for Two Years

Yuhao Jiang, 0x140ce, Ezrak1e

# Who are we?

- Security researchers at Ant Group Light-Year Security Lab
- Escaped from virtual machine many times
- Won the Pwnie Awards🦄 in 2023


蚂蚁安全实验室 Ant Security Lab | 蚂蚁光年 Ant Light-Year

# Talk Roadmap

- Introduction
- Escape VM First
- Escape ESXi Sandbox
- Demo

# Introduction

# The Wake-Up Call

- VMware announced a 0day which has occurred in the wild.

**3a. VMCI heap-overflow vulnerability (CVE-2025-22224)**

**Description:**
VMware ESXi, and Workstation contain a TOCTOU (Time-of-Check Time-of-Use) vulnerability that leads to an out-of-bounds write. VMware has evaluated the severity of this issue to be in the Critical severity range with a maximum CVSSv3 base score of 9.3.

**Known Attack Vectors:**
A malicious actor with local administrative privileges on a virtual machine may exploit this issue to execute code as the virtual machine's VMX process running on the host.

**Resolution:**
To remediate CVE-2025-22224 apply the patches listed in the 'Fixed Version' column of the 'Response Matrix' found below.

**Workarounds:**
None.

**Additional Documentation:**
A supplemental FAQ was created for clarification. Please see: https://brcm.tech/vmsa-2025-0004

**Acknowledgements:**
VMware would like to thank Microsoft Threat Intelligence Center for reporting this issue to us.

**Notes:**
VMware by Broadcom has information to suggest that exploitation of CVE-2025-22224 has occurred in the wild.

- We exploited VMware ESXi on Tianfu Cup 2023.
- Let's share some interesting things behind that story.

# ESXi Architecture Overview

- Pretty same as VMware Workstation
- But the host OS is replaced as VMkernel
- Has sandbox

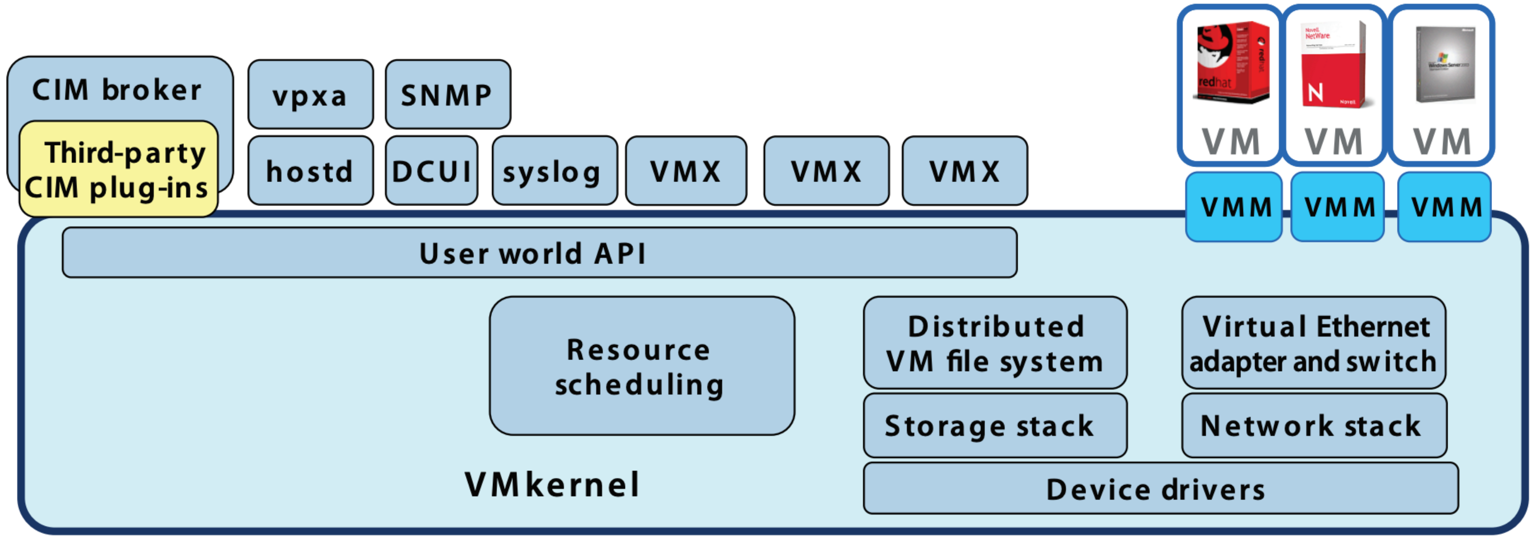



*Figure 1: The streamlined architecture of VMware ESXi eliminates the need for a service console.*

# Escape VM First

# Attack Surface

| | | | |
|---|---|---|---|
| Virtual Device | Hard Disk | LSI Logic | |
| | | PVSCSI | Pwn2Own 2025 Workstation (CVE-2025-41238) |
| | | NVME | |
| | Network Adapter | E1000/E1000e | |
| | | VMXNET3 | Pwn2Own 2025 ESXi (CVE-2025-41236) |
| | USB Controller | UHCI (USB 1) | Tianfu Cup 2021 Workstation (CVE-2021-22041),<br>Tianfu Cup 2023 Workstation (CVE-2024-22253, CVE-22255) |
| | | EHCI (USB 2) | GeekPwn 2022 Fusion (CVE-2022-31705) |
| | | XHCI (USB 3) | Tianfu Cup 2021 ESXi (CVE-2021-22040),<br>Tianfu Cup 2023 ESXi (CVE-2024-22252) |
| | USB Device | HID (mouse) | |
| | | Bluetooth | Pwn2Own 2023 Workstation (CVE-2023-20869, CVE-2023-20870),<br>Pwn2Own 2024 Workstation (CVE-2024-22267, CVE-2024-22269) |
| | | ... | |
| | GPU | SVGA 2D | |
| | | SVGA 3D | |
| | Sound Card | ES1371 | |
| | TPM | vTPM | |
| | VMCI | VMCI | Occurred in the wild (CVE-2025-22224),<br>Pwn2Own 2025 ESXi (CVE-2025-41237) |
| | | ... | |
| GuestRPC | | Backdoor | |
| | | HGFS | Pwn2Own 2024 Workstation (CVE-2024-22270),<br>Occurred in the wild (CVE-2025-22226) |
| VMM | | | |

# The “Ancient” Vulnerability

CVE-2021-22040 (Found by Wei of Kunlun Lab on Tianfu Cup 2021).

### 3a. Use-after-free vulnerability in XHCI USB controller (CVE-2021-22040)

**Description**

VMware ESXi, Workstation, and Fusion contain a use-after-free vulnerability in the XHCI USB controller.VMware has evaluated the severity of this issue to be in the Important severity range with a maximum CVSSv3 base score of 8.4.

# Diff the Patch

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Workstation | 16.x | Any | CVE-2021-22040, CVE-2021-22041 | 8.4 | important | 16.2.1 | KB87349 | FAQ |

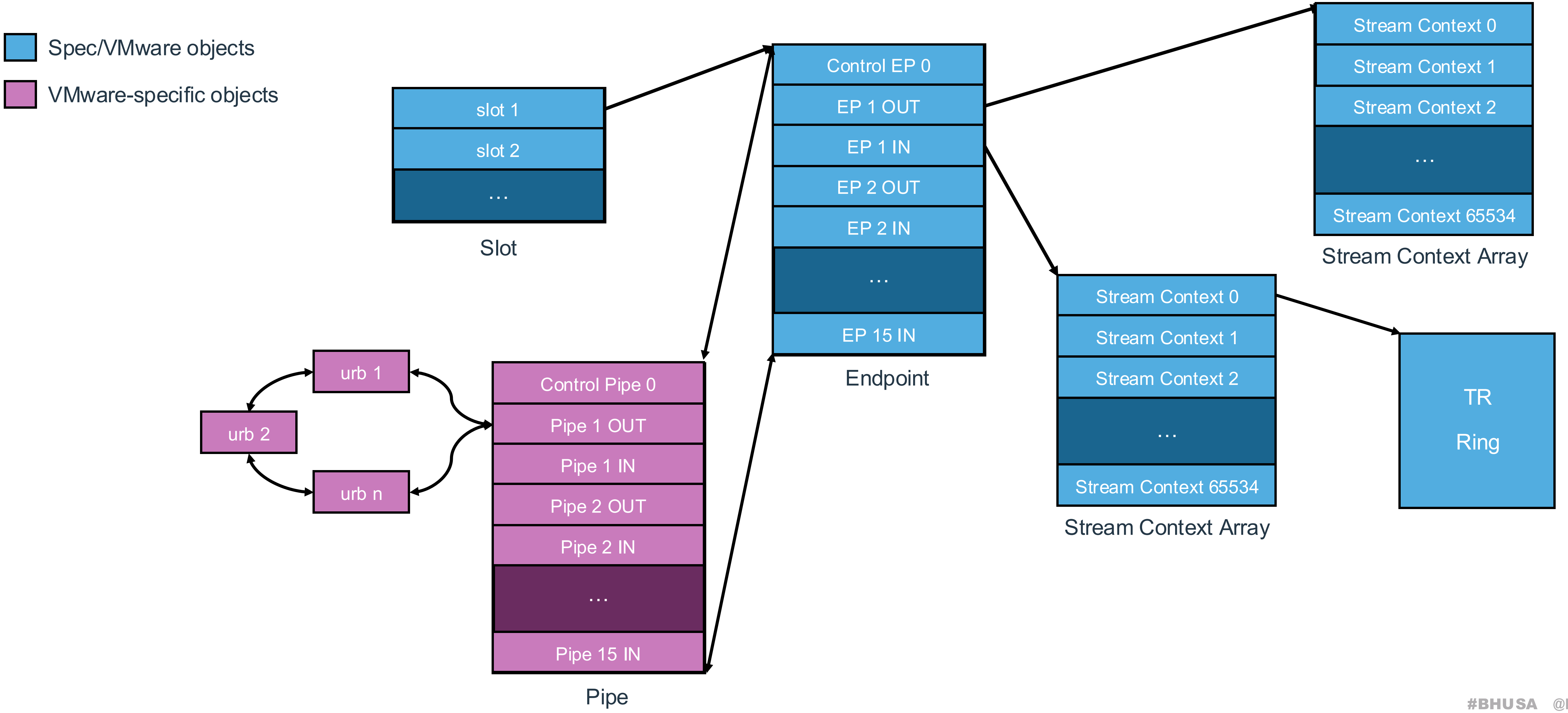
We diffed v16.2.1 with v16.2.0. Good, only 7 functions need to be analysis. 😎

28570 / 28570 Matched Functions

☑ Show structural changes ☑ Show only instructions changed ☑ Show identical

| Similarity | Confidence | Address | Primary Name | Type | Address | Secondary Name | Type | Basic Blocks | | | Jumps | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.64 | 0.90 | 000000... | sub_140331050 | Normal | 000000... | sub_140330FE0 | Normal | 0 | 1 | 0 | | | |
| 0.89 | 0.98 | 000000... | sub_140271750 | Normal | 000000... | sub_140271750 | Normal | 0 | 32 | 5 | 7 | 40 | 14 |
| 0.92 | 0.98 | 000000... | sub_1401ED670 | Normal | 000000... | sub_1401ED670 | Normal | 0 | 9 | 1 | 1 | 11 | 3 |
| 0.93 | 0.98 | 000000... | sub_1401ED710 | Normal | 000000... | sub_1401ED710 | Normal | 0 | 10 | 1 | 1 | 12 | 3 |
| 0.97 | 0.99 | 000000... | sub_140346220 | Normal | 000000... | sub_140346210 | Normal | 4 | 265 | 4 | 30 | 389 | 30 |
| 0.99 | 0.99 | 000000... | sub_140271500 | Normal | 000000... | sub_140271500 | Normal | 0 | 29 | 0 | 0 | 45 | 0 |
| 0.99 | 0.99 | 000000... | sub_140271D50 | Normal | 000000... | sub_140271D60 | Normal | 0 | 81 | 0 | 0 | 133 | 0 |
| 1.00 | 0.99 | 000000... | sub_1405C5FA0 | Normal | 000000... | sub_1405C5F90 | Normal | 0 | 9 | 0 | 0 | 11 | 0 |
| 1.00 | 0.99 | 000000... | sub_1406035F0 | Normal | 000000... | sub_1406035E0 | Normal | 0 | 6 | 0 | 0 | 7 | 0 |

# xHCI / USB3.x Controller



# The “Ancient” Vulnerability

The key changes were located at xHCI Command Ring handler functions.

The changes were reordering the execution sequence of slot context rewriting and invoking *xhci_clear_stream_ctx*.

```
if ( input_refer.drop_flags || input_refer.add_flags != 3 )
{
  Warning_0(
    "XHCI: COMMAND ADDRESS_DEVICE slot #%d, invalid drop/add flags %08x/%08x\n",
    v7 + 1,
    input_refer.drop_flags,
    (unsigned int)input_refer.add_flags);
  goto LABEL_25;
}
slot->dev_ctx.dev_ctx.slot = input_refer.slot_ctx;
*(_OWORD *)&slot->dev_ctx.dev_ctx.ep[0].ep_info = *(_OWORD *)&input_refer.ep_ctx_array[0].ep_info;
v11 = *(_OWORD *)&input_refer.ep_ctx_array[0].tx_info;
slot->dev_ctx.dirty = 0xFFFFFFFF;
*(_OWORD *)&slot->dev_ctx.dev_ctx.ep[0].tx_info = v11;
xhci_clear_stream_ctx(state, v7, 1u);
xhci_change_ep_state(state, v7, 1u, 1);
```

Before patch

```
}
if ( v21 || v22 != 3 )
{
  Warning_0("XHCI: COMMAND ADDRESS_DEVICE slot #%d, invalid drop/add flags %08x/%08x\n", v7 + 1, v21, v22);
  goto LABEL_25;
}
xhci_clear_stream_ctx(state, v7, 1u);
*(_OWORD *)(slot + 16) = v23;
*(_OWORD *)(slot + 32) = v24;
*(_OWORD *)(slot + 48) = v25;
v11 = v26;
*(_DWORD *)(slot + 8) = -1;
*(_OWORD *)(slot + 64) = v11;
xhci_change_ep_state(state, v7, 1u, 1);
```

After patch

In the older version, we can modify slot context before executing *xhci_clear_stream_ctx*.

What can we do with it?

black hat BRIEFINGS

# The "Ancient" Vulnerability

Let's see into the *xhci_clear_stream_ctx*

```
void __fastcall xhci_clear_stream_ctx(__int64 state, unsigned int slot_id_sub_1, unsigned int ep_id)
{
  __int64 v3; // rbx

  v3 = state + 8 * (ep_id + 162i64 * slot_id_sub_1);
  if ( *(_QWORD *)(v3 + 333560) )
  {
    hashmap_iterator(
      *(_QWORD *)(v3 + 333560),
      (__int64 (__fastcall *)(char *, char *, __int64))xhci_delete_stream_ctx,
      1,
      state);
    free_ptr_0(*(void ***)(v3 + 333560));
    *(_QWORD *)(v3 + 333560) = 0i64;
  }
}
```

```
void __fastcall xhci_delete_stream_ctx(XHCIStreamContext *stream_ctx, __int64 state)
{
  struct XHCIStreamContext *v4; // rdx
  struct XHCIStreamContext *v5; // rax
  struct XHCIStreamContext *next; // r8
  struct XHCIStreamContext *prev; // rdx
  struct XHCIStreamContext *v8; // rax
  xhci_endpoint endpoint; // [rsp+20h] [rbp-48h] BYREF

  if ( stream_ctx )
  {
    xhci_fetch_pipe((__int64)&endpoint, state, *(_DWORD *)&stream_ctx->ids, *(_QWORD *)&stream_ctx->struct2058);
    xhci_clean_pipe(&endpoint);
    if ( !endpoint.pipe || endpoint.pipe->urbList.next == &endpoint.pipe->urbList )
    {
```

```
void __fastcall xhci_clean_pipe(xhci_endpoint *endpoint)
{
  VUsbPipe *pipe; // rcx

  pipe = endpoint->pipe;
  if ( pipe )
  {
    cancel_pipe(pipe);
    endpoint->streamctx_struct2058->urb_size = 0;
    endpoint->streamctx_struct2058->urb_link_num = 0;
  }
}
```

If *xhci_fetch_pipe* fails, *cancel_pipe* won't be executed at all!

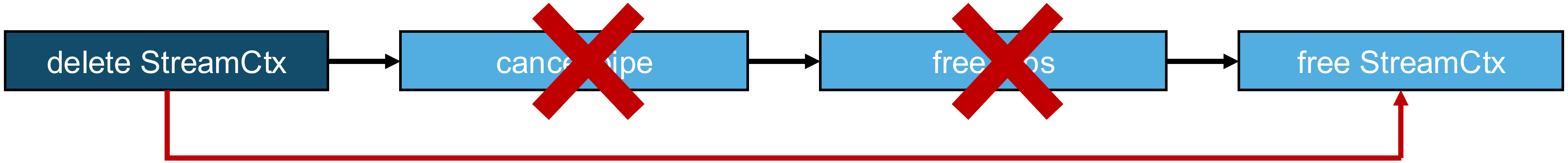
# The “Ancient” Vulnerability

What can we do in xhci_fetch_pipe function?

```
slot = (xhci_slot *)(1296i64 * (unsigned __int8)a3 + a2 + 332520);// 331224 + 1296 * slot_id
*(_QWORD *)(a1 + 8) = 0i64;
v7 = &slot->dev_ctx.dev_ctx.slot.dev_info + 8 * BYTE1(a3);
*(_QWORD *)(a1 + 24) = a4;
*(_QWORD *)(a1 + 16) = v7;
if ( ((*v7 >> 35) & 7) == 0 )                   // EP type == 0  ==>  Not Valid
  return 0;
v8 = *(_BYTE *)(a1 + 44) & 0xFE | (((*v7 >> 35) & 7ui64) >= 4);// Bidirectional or In
v9 = 0;
```

There's a check on the slot content, and if it fails, it directly returns 0.

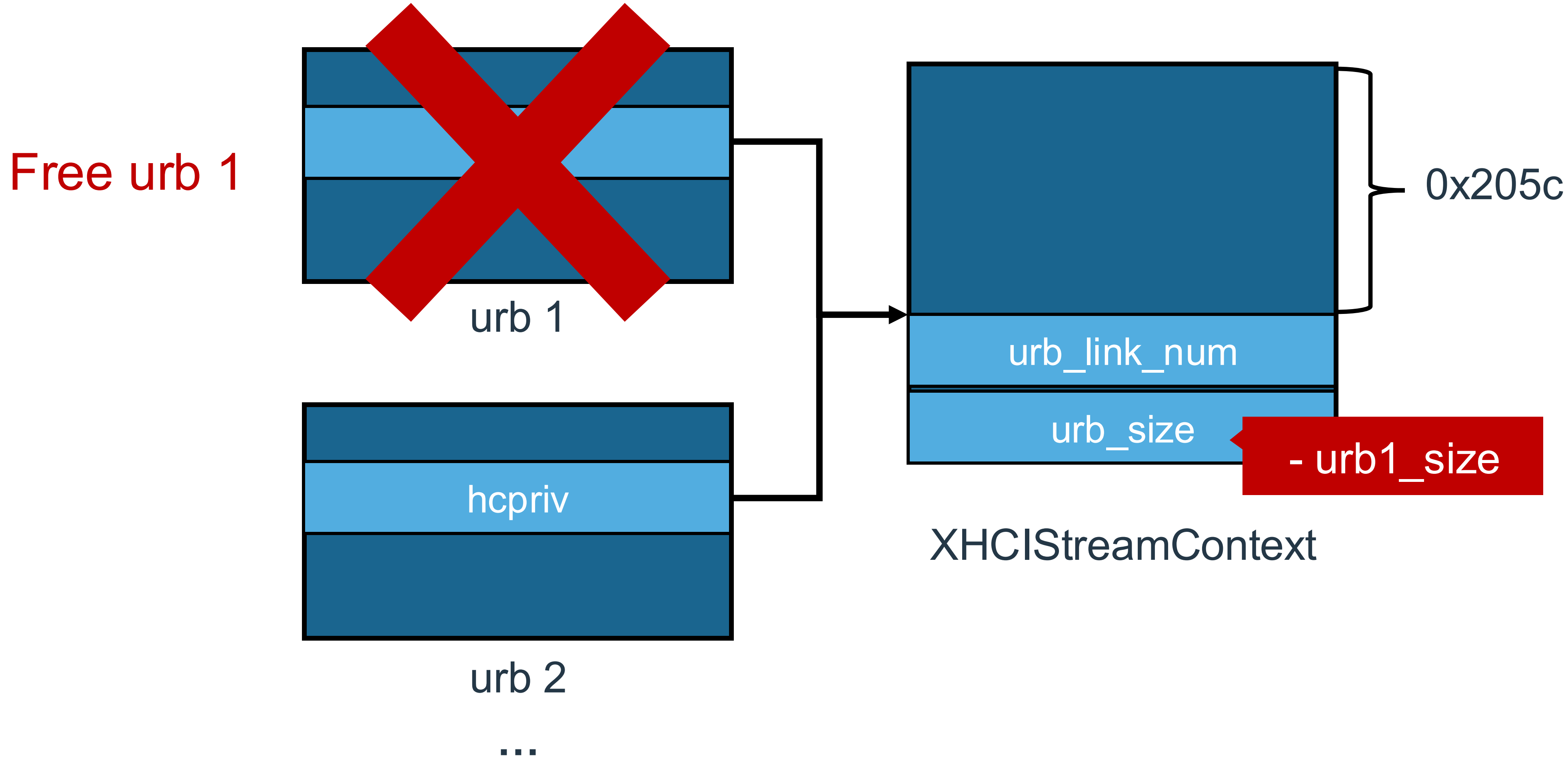
Then there won't be pipe on the endpoint!



# The use after free

Now we can leave the pipe not freed after stream context has been freed.

What can we do next?



# Resurrecting the “Ancient”

Some new code in xhci_fetch_pipe!

There was A new way for fetching pipe in xhci_fetch_pipe function.

1. For Slot State: Disabled/Enabled/Default → Find vusbDev in Root Hub
2. For other Slot States → Index via Dev State field in xHCI State's vsubDev table

```
dev_state = v6->dev_ctx.dc.slot_ctx.dev_state;
if ( (dev_state & 0xF8000000) < 0x10000000 )  // slot state为default, Disabled/Enabled状态
{
  v18 = (unsigned int)BYTE2(v6->dev_ctx.dc.slot_ctx.dev_info2) - 1;// Root Hub Port Number
  v19 = v6->dev_ctx.dc.slot_ctx.dev_info & 0xFFFFF;
  if ( (unsigned int)v18 < *(_DWORD *)(state + 1512) )// < 8
  {
    v20 = *(_QWORD *)(state + 1536) + 56 * v18;
    if ( v19 )
    {
      while ( 1 )
      {
        v21 = *(_QWORD *)(v20 + 32);
        v22 = (v19 & 0xF) - 1;
        if ( !v21 || v22 > *(int *)v21 )
          break;
        v19 >>= 4;
        v20 = *(_QWORD *)(v21 + 24) + 56i64 * v22;
        if ( !v19 )
          goto LABEL_18;
      }
    }
    else
    {
LABEL_18:
      if ( v20 )
      {
        vusbDev = get_vusbDev(v20);
        goto LABEL_21;
      }
    }
  }
LABEL_20:
  vusbDev = 0i64;
  goto LABEL_21;
}
if ( (unsigned __int8)dev_state >= 0x80u )
  goto LABEL_20;
vusbDev = *(__int64 **)(state + 8i64 * (unsigned __int8)dev_state + 0xC0);
LABEL_21:
if ( !vusbDev )
  return 0;
max_ep_size = get_max_ep_size(*vusbDev);      // 9000
epctx = endpoint->epctx;
if ( (endpoint->ep_type_info & 8) != 0 )
  max_ep_size = 196608;
endpoint->max_size = max_ep_size;
ep_info_high = HIWORD(epctx->ep_info);
```

1

2

002780E4 xhci_fetch_pipe:89 (278CE4) (Synchronized with IDA View-A)

# Resurrecting the "Ancient"

**Step 1:** Data transfer → finds pipe via second path

Root Hub Port Number incorrect

**Step 2:** Configure Endpoint → changes Slot State

Forces first path → no pipe found

**Step 3:** Disable Slot → triggers vulnerability

xhci_clean_pipe skipped

**URBs left dangling in pipe**

# Wait wait wait

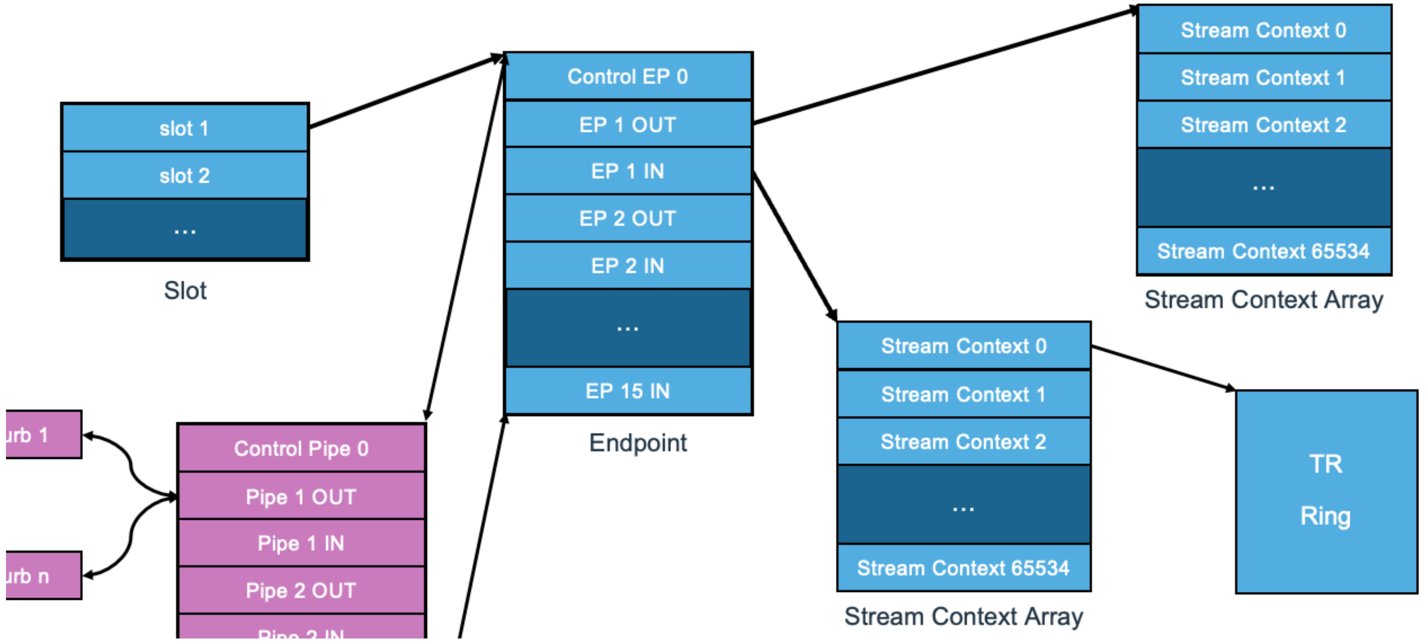
# It Never Really “Died”

Actually, we don't need new code to make the *xhci_fetch_pipe* function fail.

We found that the patch never succeeded. 😱

```
void __fastcall xhci_clear_stream_ctx(__int64 state, unsigned int slot_id_sub_1, unsigned int ep_id)
{
  __int64 v3; // rbx

  v3 = state + 8 * (ep_id + 162i64 * slot_id_sub_1);
  if ( *(_QWORD *)(v3 + 333560) )
  {
    hashmap_iterator(
      *(_QWORD *)(v3 + 333560),
      (__int64 (__fastcall *)(char *, char *, __int64))xhci_delete_stream_ctx,
      1,
      state);
    free_ptr_0(*(void ***)(v3 + 333560));
    *(_QWORD *)(v3 + 333560) = 0i64;
  }
}
```




*The xhci_clear_stream_ctx* function only delete stream context of a specific endpoint (ep).

But the content of the entire slot has already been modified by us!

Modify slot content → Clear non-essential endpoints → Issue disable slot command → UAF

# Exploit Time!

# The Exploitation Challenge

Constrained UAF:

- Only affects at offset +0x205c
- Operation: Subtract a value

The Problems:

1. If we want to change a 64-bit pointer alignment. We can only modify high 4 bytes

Meaningless for exploitation.

2. Massive offset distance. +0x205c = 8284 bytes. Need do better in heap fengshui.

# Finding Our Saving Grace

## HashMap

- Each element: value + key
- Controllable heap allocation size:

When storage exceeds capacity → reallocates to 2x size

stream_ctx hashmap:

- value: address, 8-byte
- key: id, 4-byte

Place 64-bit pointer at offset +0xc, perfect!

# The Arsenal of Primitives

1. **URB**
    - Controllable size, dynamic allocate and free
    - Has a data array and its length member, and some pointers.
- Modify the length member → out-of-bounds read.
- New finding: Use vmware-USBArbitrator in Linux version to get USB-related symbols.

2. **mob, Surface, and GMR**

Useful for heap spraying and heap grooming.

…


Yuhao Jiang & Xinlei Ying 2024


VMWARE EXPLOITATION THROUGH UNINITIALIZED BUFFERS

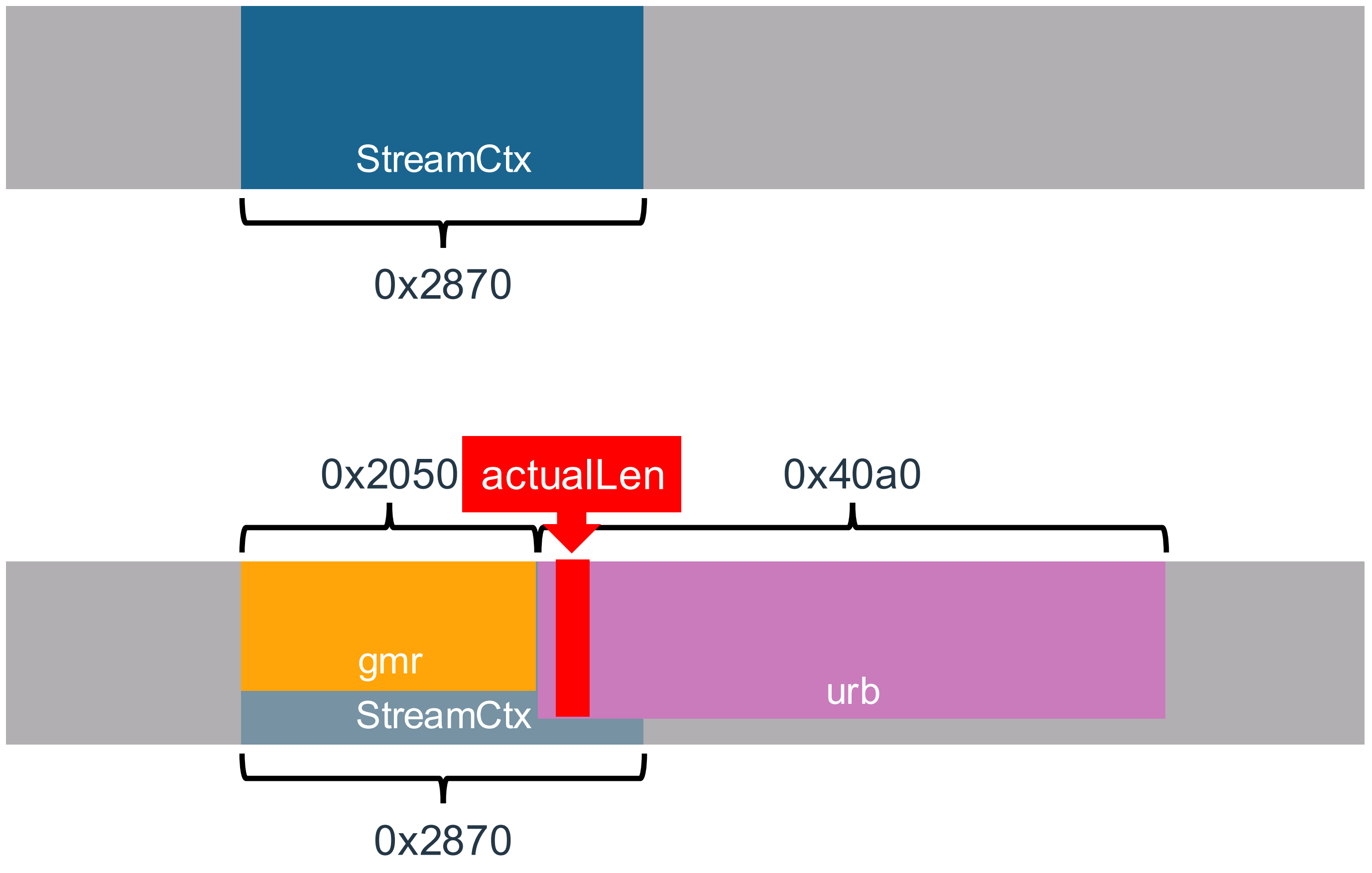
March 01, 2018 | Abdul-Aziz Hariri

As we approach Pwn2Own 2018, I'm reminded of some of the exploits we saw at last year's contest. Of course, the most interesting bugs we saw involved guest-to-host escalation in VMware. Recently, we presented "l'art de l'évasion: Modern VMware Exploitation techniques" at OffensiveCon in Berlin (Hats off for the great conference!). The talk focused on exploitation techniques that have been used in the past year to exploit different vulnerability types in VMware Workstation. I've previously discussed Use-After-Free (UAF) VMware exploits affecting drag-and-drop (DnD) and triggered by Remote Call Procedure requests. If you haven't read that blog yet, you may want to start there. Once you're done with that blog, check out former Mobile Pwn2Own winner Amat Cama's blog covering a heap-based buffer overflow exploit that Chaitin Security Research Lab planned to use at Pwn2Own last year. Unfortunately for them, the bug he describes was patched the day before the contest and thus couldn't be used. Still, it's a great write-up and should

Abdul-Aziz Hariri 2018

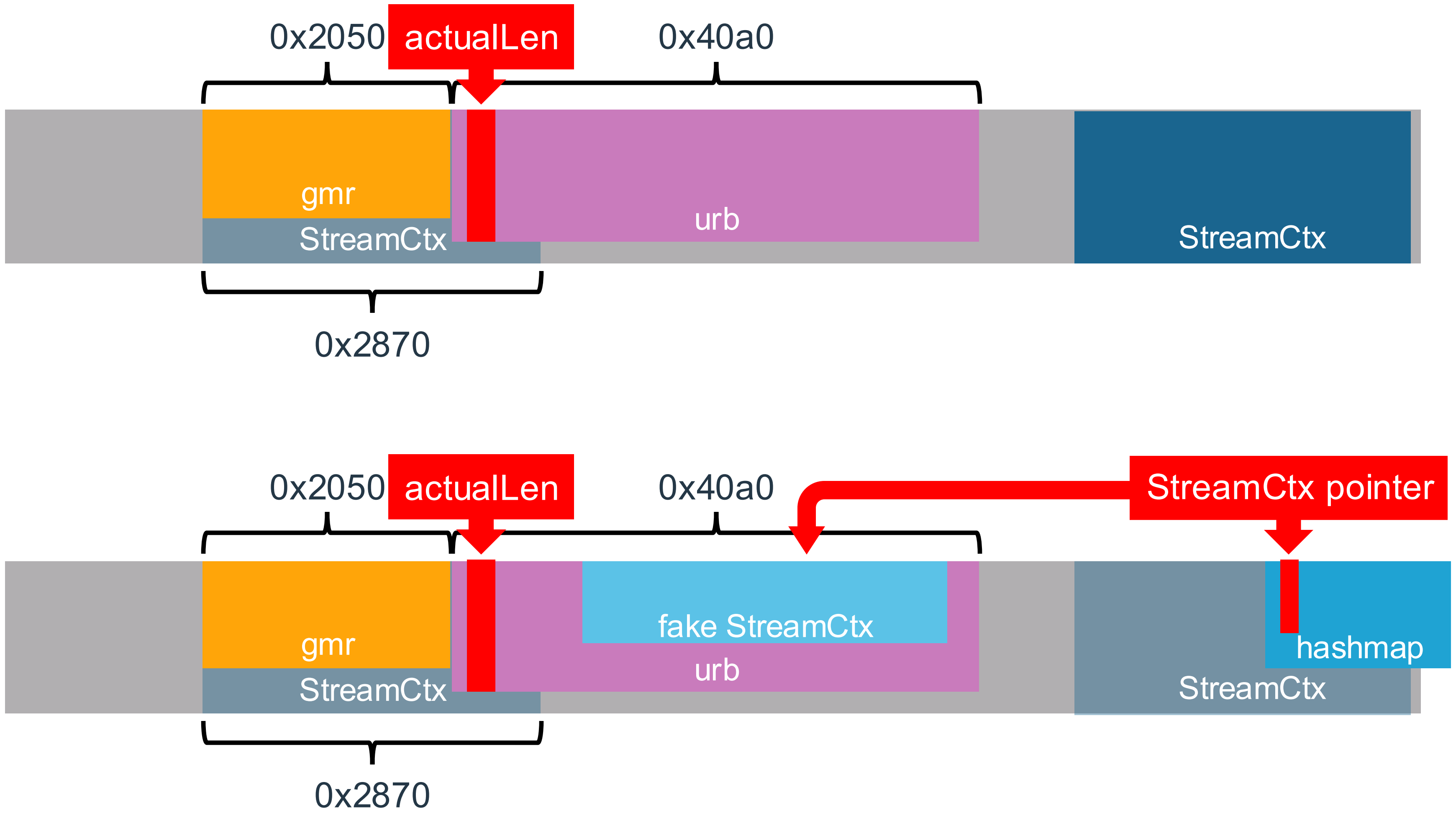
# The Exploitation Flow

1. Construct a UAF
2. Allocate a URB at the location of the original stream context
3. Trigger the use, causing urb->actualLen to integer underflow
4. Out-of-bounds read, obtain heap address that we placed afterwards



# The Exploitation Flow

1. Construct another UAF
2. Use hashmap to occupy
3. Trigger the use, causing the streamctx pointer to point to the URB located ahead
4. We can prepare a fake streamctx in the URB in advance
5. Pass streamctx check and free fake streamctx
6. Achieve heap overlapping



# Control Flow Hijacking

1. Before URB is freed, *vusbCompleteUrbAddBatch* function checks whether it's an xHCI URB. If so, it calls xhci_stream_ctx_sub_one_urb through the function pointer in vusbDev.

```
if ( urb->status == 6 )
{
  if ( urb->hcpriv )
    (*(void (__fastcall **)(vurb *))(*(_QWORD *)(*(_QWORD *)(dev + 8) + 160i64) + 32i64))(urb);// xhci_stream_ctx_sub_one_urb
  numPackets = urb->numPackets;
  bufferLen = urb->bufferLen;
  urb->stage = 2;
  urb_unlink_0((urb *)urb, bufferLen, numPackets);
  free_urb((urb *)urb);
  return 0;
}
```

2. Using our existing heap overlap capability, we can take over URB objects, modify their contents to craft fake vusbDev objects, and achieve arbitrary address calls.
3. But this is still not enough - we need the ability to execute shellcode.

black hat BRIEFINGS

# Execute Shellcode

1. We can obtain ROP capability via stack migration.
2. Replace rsp with the value at rdx address.
3. Subsequently, use ROP to allocate executable memory, copy shellcode, and execute it.

```
000000000117A08    mov     rdx, r12
000000000117A0B    mov     rdi, r14
000000000117A0E    call    qword ptr [rax+10h]
```

At the point of arbitrary address calls, the r12 register contains the same value as rdi - the URB address.

```
000000455F5          mov     rsp, [rdx+0A0h]
000000455FC          mov     rbx, [rdx+80h]
00000045603          mov     rbp, [rdx+78h]
00000045607          mov     r12, [rdx+48h]
0000004560B          mov     r13, [rdx+50h]
0000004560F          mov     r14, [rdx+58h]
00000045613          mov     r15, [rdx+60h]
00000045617          mov     rcx, [rdx+0A8h]
0000004561E          push    rcx
0000004561F          mov     rsi, [rdx+70h]
00000045623          mov     rdi, [rdx+68h]
00000045627          mov     rcx, [rdx+98h]
0000004562E          mov     r8, [rdx+28h]
00000045632          mov     r9, [rdx+30h]
00000045636          mov     rdx, [rdx+88h]
00000045636 ; } // starts at 455C0
0000004563D ; __unwind {
0000004563D          xor     eax, eax
0000004563F          retn
```

# The Dark Secret Revealed

- CVE-2021-22040 was not correctly patched.
- Until we reported the vulnerability at the end of 2023.
- Nobody pointed out that CVE-2021-22040 and CVE-2024-22252 are same.

# Reasons

- VMware’s bounty program is significantly lower than the vulnerability's true value.
- Closed source.
- Less technical sharing in the community.

# How About This Time?

**CVE-2025-22224**

- Directly fetch values from guest memory to perform packet size validation

**Step 1:** Use legitimate length → pass validation

**Step 2:** Immediately modify to oversized value

**Step 3:** Enter VMCIDatagramDispatch with malicious size

```
  }
LABEL_28:
  if ( (unsigned __int64)(buffer[2] + 24i64) > 0x11000 )
    v11 = -2;
  else
    v11 = VMCIDatagram(vmci_device_state->qword10[2], (unsigned int *)buffer);
  if ( !vmci_device_state->packet_flag )
    goto LABEL_36;
  packet = vmci_device_state->packet;
  goto end_packet;
}
```

Before patch

→

```
  buffer_1 = UtilSafeMalloc0(*(_QWORD *)(pval_1 + 16) + 24i64);
  memcpy(buffer_1, (const void *)pval_1, buffer_size);
  v20 = buffer_1[2];
  v21 = (unsigned int *)buffer_1;
  if ( v20 > 0x10FE8 || v20 + 24 != buffer_size )
  {
    v13 = -2;
LABEL_41:
    free(buffer_1);
    goto ERROR;
  }
LABEL_33:
  if ( (unsigned __int64)(*((_QWORD *)v21 + 2) + 24i64) > 0x11000 )
    v13 = -2;
  else
    v13 = VMCIDatagramDispatch(*(_DWORD *)(v7 + 32), v21);
  if ( *(_BYTE *)(v7 + 84) )
  {
```

After patch

# Escape ESXi Sandbox

# ESXI Sandbox Overview

- ESXi uses security domains to limit process access to files, networks, etc.

```
[root@localhost:~] secpolicytools -l
------------------------------
Valid Object Labels
------------------------------
appObj     2111
authObj     2113
authdBinObj     2114
certObj     2115
cimObj     2105
crxcliObj     2110
default     1
dhclientObj     2127
esxcfgInitObj     2128
esxcfgadvcfgObj     2129
esxtopObj     2126
etcdObj     2108
infravisorSphereletObj     2109
localcliObj     2131
opensslObj     2130
osfsdObj     2121
pluginObj     2106
pmemGCObj     2119
secpolicyObj     2104
sfcbVmwPluginObj     2107
shellObj     2118
sshdObj     2125
sslKeyObj     2112
supershellObj     2123
supportUtilObj     2124
swapobjdObj     2122
tardiskMountObj     2116
tpm2emuObj     2117
unlabeled     0
vdsVsipIoctl     2134
vmkloadmodObj     2133
vsanObserverObj     2120
vsishObj     2132
watchdogObj     2135
```

```
-----------------------------------------
Valid domains
-----------------------------------------
0   superDom
1   regularVMDom
2   lprDom
3   actionScriptDom
4   clomdDom
5   cmmdsTimeMachineDom
6   cmmdsdDom
7   dcuiDom
8   dhclientDom
9   driverVMDom
10   entropydDom
11   entropydEsxcfgInitDom
12   epdDom
13   esxioCommdDom
14   genericDom
15   genericDomLocalAuth
16   jumpstartDom
17   keypersistDom
18   kmxaDom
19   lacpDom
20   loadsecpolicyDom
21   nfsgssdDom
22   nvmf-authdDom
23   osfsdDom
24   vaainasdDom
25   vmkdevmgrDom
26   vmkeventdDom
27   vmsyslogdDom
28   vobdDom
29   vsanObserverDom
30   vsanmgmtdDom
31   vsantracedDom
```

```
[root@localhost:~] ps -Z
WID    CID    WorldName                        SecurityDomain
66184  66184  esxgdpd                          43
66185  66185  sandboxd                         82
66196  66184  esxgdpd-worker                   43
66197  66184  esxgdpd-fair                     43
66198  66184  esxgdpd-backend                  43
66201  66185  worker                           82
66202  66185  worker                           82
```

# Sandbox for Syscall

- Looking at the rules, we can see restrictions on Syscalls

```
-s genericSys grant
-s vmxSys grant
-s ioctlSys grant
-s getpgidSys grant
-s getsidSys grant
-s vobSys grant
-s vsiReadSys grant
-s rpcSys grant
-s killSys grant
-s sysctlSys grant
-s syncSys grant
-s forkSys grant
-s forkExecSys grant
-s cloneSys grant
-s openSys grant
-s mprotectSys grant
-s iofilterSys grant
-s crossfdSys grant
-s pmemGenSys grant
-s keyCacheGenSys grant
-s vmfsGenSys grant
```

# Sandbox for Syscall

- In order to know which specific syscalls can be used,it is time to analyze the vmkernel
- The vmkernel binary with symbols can be extracted from k.b00 in the system

```
[root@localhost:~] find /|grep k.b00
/vmfs/volumes/7cda6fab-a54aa197-79a1-0fb2c415f5f2/k.b00
[root@localhost:~]
```

# Sandbox for Syscall

- syscall number < 0x400 — Linux64_SyscallTable
- 0x400 < syscall number < 0x4000 — UW64VMKSyscall_HandlerTable
- syscall number > 0x4000 — UW64VMKPrivateSyscall_HandlerTable

```
UW64VMKSyscall_HandlerTable dq offset UW64VMKSyscallUnpackGetSyscallVersion
                                        ; DATA XREF: User_UWVMK64SyscallHandler+156↓o
                dq offset UW64VMKSyscallUnpackForkExec
                dq offset UW64VMKSyscallUnpackTdataInit
                dq offset UW64VMKSyscallUnpackGetSMBIOS
                dq offset UW64VMKSyscallUnpackGetSMBIOSLen
                dq offset UW64VMKSyscallUnpackGetCPUModelName
                dq offset UW64VMKSyscallUnpackLockPage
                dq offset UW64VMKSyscallUnpackProbeMPN
                dq offset UW64VMKSyscallUnpackGetNextAnonPage
                dq offset UW64VMKSyscallUnpackGetVMKStackInfo
                dq offset UW64VMKSyscallUnpackReadVMKStack
                dq offset UW64VMKSyscallUnpackGetMPNContents
                dq offset UW64VMKSyscallUnpackNumaGetSystemInfo
                dq offset UW64VMKSyscallUnpackNumaGetNodeInfo
                dq offset UW64VMKSyscallUnpackSetMPNContents
                dq offset UW64VMKSyscallUnpackLiveCoreDump
                dq offset UW64VMKSyscallUnpackRPCRegister
                dq offset UW64VMKSyscallUnpackRPCGetMsg
}
```

```
public UW64VMKPrivateSyscall_HandlerTable
yscall_HandlerTable dq offset UW64VMKPrivateSyscallUnpackGetPrivateSyscallVersion
                                ; DATA XREF: User_UWVMK64SyscallHandler+CB↓o
                                ; User_UWVMK64SyscallHandler+D2↓r
dq offset UW64VMKPrivateSyscallUnpackUNUSED_PR1972171_AddPage64
dq offset UW64VMKPrivateSyscallUnpackProcessBootstrap
dq offset UW64VMKPrivateSyscallUnpackMigrateRestoreDone
dq offset UW64VMKPrivateSyscallUnpackUNUSED_PR1972171_GetVMMPageRoot
dq offset UW64VMKPrivateSyscallUnpackGetNextAnonPages
dq offset UW64VMKPrivateSyscallUnpackUNUSED_PR1972171_GetSharedRegion
)
```

# Sandbox for Syscall

- Sandbox restrictions on syscall are mainly implemented in VmkAccessSyscallCheck

```
EnforcementLevel = DomainObject->EnforcementLevel;
if ( !EnforcementLevel
  || _bittest64(&DomainObject->SyscallMask, (unsigned int)a2)
  || DomainObject->PrivilegeLevel == 3
  && !_interlockedbittestandset64((volatile signed __int32 *)&DomainObject->SyscallMask, (unsigned int)a2) )
{
  return 0LL;
}
if ( EnforcementLevel != 2 )
  return 0xBAD0117LL;
Log(
  (unsigned int)"VmkAccess: %d: %s: %s:: dom:%s(%d), sysClass:%s(%d)\n",
  81,
  __readgsqword(0x10u) + 3024,
  (unsigned int)"access warning",
  (_DWORD)DomainObject + 233,
  DomainObject->dword0,
  sysClassIdentifiers[(unsigned int)a2],
  a2);
return 0LL;
```

# Sandbox for Syscall

```
sysClassIdentifiers dq offset aGenericsy

                    dq offset aVmxsys
                    dq offset aVmkacsys
                    dq offset aMountsys
                    dq offset aUmountsys
                    dq offset aTimesys
                    dq offset aIoctlsys
                    dq offset aSetpgidsys
                    dq offset aGetpgidsys
                    dq offset aGetsidsys
                    dq offset aAdminsys
                    dq offset aVobsys
                    dq offset aVsireadsys
                    dq offset aVsiwritesys
                    dq offset aModulesys
```

```
-s genericSys grant
-s vmxSys grant
-s ioctlSys grant
-s getpgidSys grant
-s getsidSys grant
-s vobSys grant
-s vsiReadSys grant
-s rpcSys grant
-s killSys grant
-s sysctlSys grant
-s syncSys grant
-s forkSys grant
-s forkExecSys grant
-s cloneSys grant
-s openSys grant
-s mprotectSys grant
-s iofilterSys grant
-s crossfdSys grant
-s pmemGenSys grant
-s keyCacheGenSys grant
-s vmfsGenSys grant
```

**Example :genericSys | vmxSys**

**1<<0 | 1<<1 = 3**
**Domain AccessMask=3**

**GetPrivateSyscallVersion belongs to genericSys**

**access check succeed**

# Domain Transition

- There are two ways to change domains
- By adding SecurityDom to the parameters in the exec system call, the sandbox domain can be switched.

```
__int64 __fastcall UserParamParseSecurityDom(__int64 a1, __int64 a2)
{
  __int64 result; // rax
  int v3; // edx
  int v4; // ecx
  unsigned __int64 v5; // r8
  int v6; // [rsp+4h] [rbp-54h] BYREF
  char v7[8]; // [rsp+8h] [rbp-50h] BYREF
  char v8[72]; // [rsp+10h] [rbp-48h] BYREF

  if ( (unsigned int)UserParamParseGetString(v8, 64LL, a2, v7) )
  {
    v4 = 962;
    v5 = __readgsqword(0x10u) + 3024;
  }
```

```
  result = VmkAccessDomain_LookupName((__int64)v8, (unsigned int *)&v6);
  if ( !(_DWORD)result )
  {
L_4:
    Domain_Index = v6;
    *(_BYTE *)(a1 + 0x4C10) = 1;
    *(_DWORD *)(a1 + 0x4C14) = Domain_Index;
    return result;
  }
```

If you want to test your own programs in a sandbox domain, there is an easy way example ./test ++securitydom=51

# Domain Transition

Only privileged domains and arbitraryTransitionDomains can use this method to transition domains.

```
if ( v2 )
{
  result = 0LL;
  if ( !a1->IsarbitraryTransitionDomains )
  {
    result = a1->EnforcementLevel;
    if ( (_DWORD)result == 1 || a1->PrivilegeLevel )
    {
      return 0xBAD0117LL;
    }
    else if ( (_DWORD)result )
    {
      Log(
        (unsigned int)"VmkAccess: %d: Allow %s(%u) -> %s(%u) transition as %s(%u) is not enforcing\n",
        1335,
```

```
arbitraryTransitionDomains db 'hostprofilesDom',0
                                        ; DATA XREF: VmkAccessDomain_Create+2D
                align 40h
aJumpstartdom   db 'jumpstartDom',0
                align 40h
aSettingsddom   db 'settingsdDom',0
                align 40h
aSuperdom_0     db 'superDom',0
                align 40h
aGenericdom     db 'genericDom',0
                align 40h
aGenericdomloca db 'genericDomLocalAuth',0
                align 40h
aHostddom       db 'hostdDom',0
                align 40h
```

# Domain Transition

- Another way to transition to a sandbox domain is when the binary has the vmware.security xattr attribute.

```
v5 = 4LL;
v4 = &v3;
result = vmk_StringCopy(v6, "vmware.security", 256LL);
if ( !(_DWORD)result )
{
  result = (*(__int64 (__fastcall **)(__int64, char *, int **))(*(_QWORD *)(a1 + 64) + 328LL))(a1, v6, &v4);// getxattr
  if ( (_DWORD)result == 0xBAD0020 )
```

- The label of the domain obtained through vmware.security is used to find the domain object

```
if ( (unsigned int)VmkAccessTransition_Lookup(*(_QWORD *)(a2 + 0x1AE0), *a1, 0, v8) )
{
  v7 = 0;
  VmkAccessDomainTransitionAtomic(a2, *(_QWORD *)(a2 + 6880), a3);
}
else
{
  v4 = VmkAccessDomain_Find((unsigned int)v8[0]);
  v5 = v4;
  if ( v4 )
  {
    VmkAccessDomainTransitionAtomic(a2, v4, a3);
```

# Domain Transition

- Is it possible to directly escape the sandbox by setting the xattr of a binary file?


- First, the sandbox restricts the use of the Setxattr syscall (need vmkacSys)

- Second, the sandbox defines what kind of domain each domain can transition to.

```
-r /bin/tpm2emu rx
-d tpm2emuObj tpm2emuDom file_exec grant
```

```
for ( i = domain_object + 0xD8; v8 != i; v8 = *(_QWORD *)(v8 + 8) )
{
  v10 = *(_DWORD *)(v8 + 16);
  if ( v10 == a2 )
  {
    if ( *(_DWORD *)(v8 + 20) == a3 )
    {
      v4 = 0;
      *a4 = *(_DWORD *)(v8 + 24);
      break;
    }
```

# ESXI Sandbox Overview

- Now we can fully understand the sandbox rules returned by secpolicytools

## Socket

```
-c dgram_vsocket_bind grant
-c dgram_vsocket_create grant
-c dgram_vsocket_send grant
-c dgram_vsocket_trusted grant
-c inet_dgram_socket_create grant
-c inet_stream_socket_create grant
-c opaque_net_connect grant
-c stream_vsocket_bind grant
-c stream_vsocket_connect grant
-c stream_vsocket_create grant
-c stream_vsocket_trusted grant
-c unix_dgram_socket_bind grant
-c unix_dgram_socket_connect grant
-c unix_socket_create grant
-c unix_stream_socket_bind grant
-c unix_stream_socket_connect grant
-c unix_vmklink_socket_connect grant
-c vsocket_provide_service grant
```

## File

```
-r /.vmware r
-r /bin/remoteDeviceConnect rx
-r /bin/tpm2emu rx
-r /bin/vmx rx
-r /bin/vmx-debug rx
-r /bin/vmx-stats rx
-r /dev/PMemDisk rw
-r /dev/cbt rw
-r /dev/cdrom/mpx.vmhba64:C0:T0:L0 rw
-r /dev/char rw
-r /dev/char/vmkdriver/vprobe
-r /dev/deltadisks rw
-r /dev/svm rw
```

## Syscall

```
-s genericSys grant
-s vmxSys grant
-s ioctlSys grant
-s getpgidSys grant
-s getsidSys grant
-s vobSys grant
-s vsiReadSys grant
-s rpcSys grant
-s killSys grant
-s sysctlSys grant
-s syncSys grant
-s forkSys grant
-s forkExecSys grant
```

## Transition

```
-d tpm2emuObj tpm2emuDom file_exec grant
```

# Target Selection

```
-r /bin/vmx-stats rx
-r /dev/PMemDisk rw
-r /dev/cbt rw
-r /dev/cdrom/mpx.vmhba64:C0:T0:L0 rw
-r /dev/char rw
-r /dev/char/vmkdriver/vprobe
```

- Changed Block Tracking (CBT) is a VMkernel feature that keeps track of the storage blocks of virtual machines as they change over time. The VMkernel keeps track of block changes on virtual machines, which enhances the backup process for applications that have been developed to take advantage of VMware's vStorage APIs.

```
==+Module :
  |----Name..............................................cbt
  |----File Name.........................................cbt
  |----File Path.........................................../usr/lib/vmware/vmkmod/cbt
  |----Module Id.........................................85
  |----ReadOnly Load Address.............................0x000042002a2b9000
  |----ReadOnly Length...................................12288
  |----Writable Load Address.............................0x000041ffd5000000
  |----Writable Length...................................4096
```

# Bug Discovery

- The CBT driver is a File Device Service driver, which is registered into the kernel through FDS_RegisterDriver

```
v2 = FDS_RegisterDriver(
       "cbt",
       cbtOps,
       (unsigned int)cbtModuleID,
       &v7,
       *(unsigned int *)(*(_QWORD *)(__readgsqword(0x10u) + 6896) + 4LL),
       *(unsigned int *)(*(_QWORD *)(__readgsqword(0x10u) + 6896) + 16LL),
       1023LL);
```

```
cbtOps          dq offset CBT_OpenDevice
                                        ; DATA XREF: ini
                                        ; init_module+20
                dq offset CBT_CloseDevice
                dq offset CBT_AsyncIO
                dq offset CBT_Ioctl
                dq offset FDS_NotSupported
                dq offset CBT_MakeDev
                align 20h
                dq offset FDS_NotSupported
                dq offset FDS_NotSupported
                dq offset CBT_RemoveDev
```

1.open("/dev/cbt/control")+ioctl -> CBT_Ioctl

2.UW64VMKSyscallUnpackFDSMakeDev->CBT_MakeDev

- For example, create the /dev/cbt/pwn1
- open("/dev/cbt/pwn1")+ioctl -> CBT_Ioctl

# Bug Discovery

- CBT_MakeDev creates a CbtDev object.
- CbtDev stores the file handle entered by the user.
- Use FSS_GetFileAttributesByFH to get the file size by file handle
- Create a bitmap object based on the file size value

```
*v1 = a1->FileHandle;
FileAttributesByFH = FSS_GetFileAttributesByFH(*(_QWORD *)v22, v23);
if ( !FileAttributesByFH )
{
  v3 = v22;
  v4 = v23[0];
  *(_QWORD *)(v22 + 16) = v23[0];
  BitmapAlign = a1->BitmapAlign;
  *(_DWORD *)(v3 + 32) = 2;
  *(_DWORD *)(v3 + 8) = BitmapAlign;
  v6 = (v4 + (unsigned __int64)(unsigned int)(8 * BitmapAlign) - 1) / (unsigned int)(8 * BitmapAlign);
  v7 = CBTAlloc(24LL);
  if ( v7 )
  {
    v8 = CBTAlloc((unsigned int)v6);
```

# Bug Discovery

```
result = FSS_IoctlByFH(*v6, 0xBE9LL, &v16, &v16, 0LL);
if ( !(_DWORD)result )
{
  CBTUpdateBitmap((__int64)v5, *(_QWORD *)a3[2], *(_QWORD *)a3[2] + *(_QWORD *)(a3[2] + 8LL));
  result = 0LL;
}
```

- The vulnerability occurs in CBTUpdateBitmap, which causes an out-of-bounds write based on the offset and size entered by the user.

```
do
{
  v7 = (_BYTE *)(*BitmapPtr + (StartOffset >> 3));
  v8 = StartOffset & 7;
  v9 = (unsigned __int8)*v7;
  if ( !_bittest(&v9, v8) )
    *v7 |= 1 << v8;
  ++StartOffset;
}
while ( (unsigned int)EndOffset >= StartOffset );
```

# Check Bypass

```
result = FSS_IoctlByFH(*v6, 0xBE9LL, &v16, &v16, 0LL);
if ( !(_DWORD)result )
{
  CBTUpdateBitmap((__int64)v5, *(_QWORD *)a3[2], *(_QWORD *)a3[2] + *(_QWORD *)(a3[2] + 8LL));
  result = 0LL;
}
```

- FSS_IoctlByFH -> Fil3_FileBlockUnmap
- Check  the offset and size cannot be larger than the file size.

- check can be bypassed by writing more content to the file

# Analysis

- Now we can trigger an out-of-bounds write on a heap object

- Which object can we overwrite?

```
v0 = configOption[587];
Log("CBT: %d: Currently the max memory size for CBT bitmap allocation is %u MB.\n", 221LL, (unsigned int)v0);
CBTHeapID = Heap_Create("cbt", vmk_ModuleGroupID, 0LL, (unsigned int)((_DWORD)v0 << 20), 0LL);
if ( !CBTHeapID )
{
  Warning("CBT: %d: Unable to create heap for cbt driver\n", 229LL);
  Warning("CBT: %d: CBT specific initialization for the cbt driver failed\n", 266LL);
  return 0xFFFFFFFFLL;
}
```

- It seems that we cannot find any exploitable objects from vmkernel
- The cbt driver has only 15 functions and 24kb in size, which may not be as big as the problem in ctf

# Analysis

- Fortunately, we still have bitmap_object that can be used for exploitation

```
v6 = (struct_v6 *)CBTAlloc(0x18uLL);
if ( v6 )
{
  v7 = CBTAlloc((unsigned int)v5);
  v6->BitmapPtr = v7;
  if ( v7 )
  {
    v6->BitmapSize = v5;
  }
```

- By modifying the BitmapSize field with an out-of-bounds write, we can get an out-of-bounds read to leak the kernel address.

```
v8 = User_CopyOut(v43 + 16, Bitmap_content, Size);
if ( !v8 )
{
  v8 = User_CopyOut(v43, a2 + 40, 8LL);
  if ( !v8 )
    v8 = User_CopyOut(v43 + 8, a2 + 48, 8LL);
}
```

# GET AAW

By modifying bitmapptr, we can obtain arbitrary address write primitive

```
do
{
  v7 = (_BYTE *)(*BitmapPtr + (v6 >> 3));
  v8 = v6 & 7;
  v9 = (unsigned __int8)*v7;
  if ( !_bittest(&v9, v8) )
    *v7 |= 1 << v8;
  ++v6;
}
```

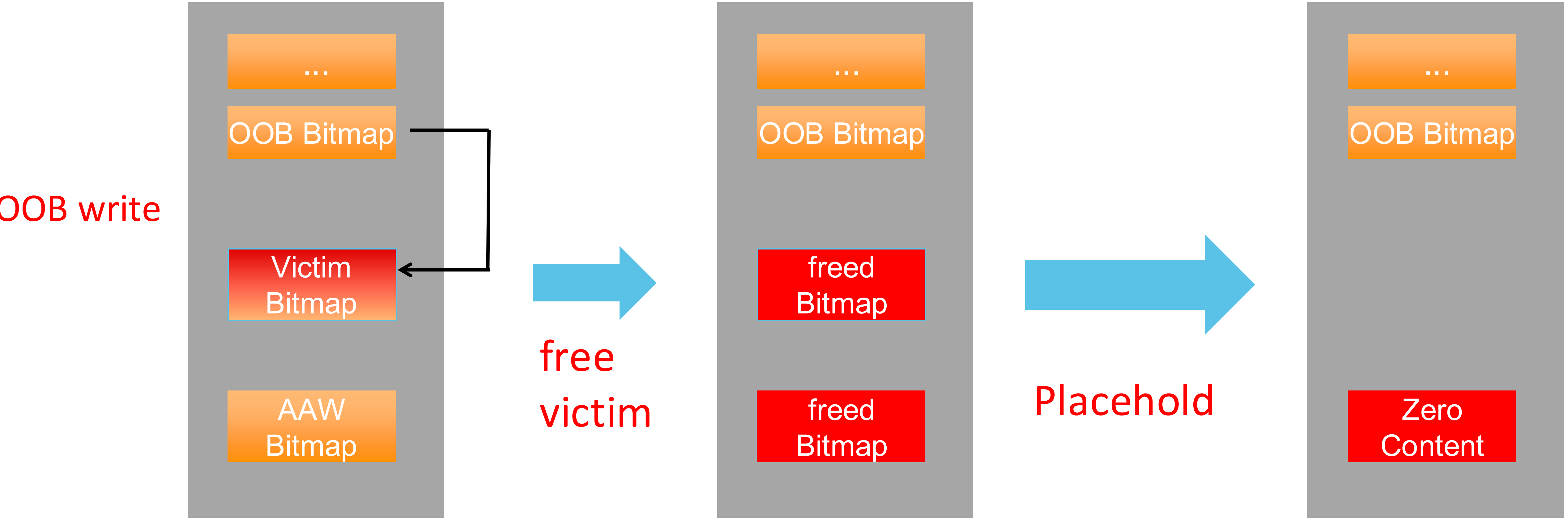
But out-of-bounds write primitive cannot modify a pointer address to another pointer address.

Example:

0x41(01000001) -> 0xff  (11111111) ✓

0x41(01000001) -> 0x42(01000010) ✗

# Vmkernel Heap Exploitation




bitmap_ptr = 0 !

Now we got AAR/W primitive

# Expolit Overview

Step1:OverWrite Victim->BitmapSize to get OOB READ primitive and leak kernel address

Step2:OverWrite Victim->ChunkSize and then release the chunk to control the BitmapPtr pointer to obtain AAW primitive

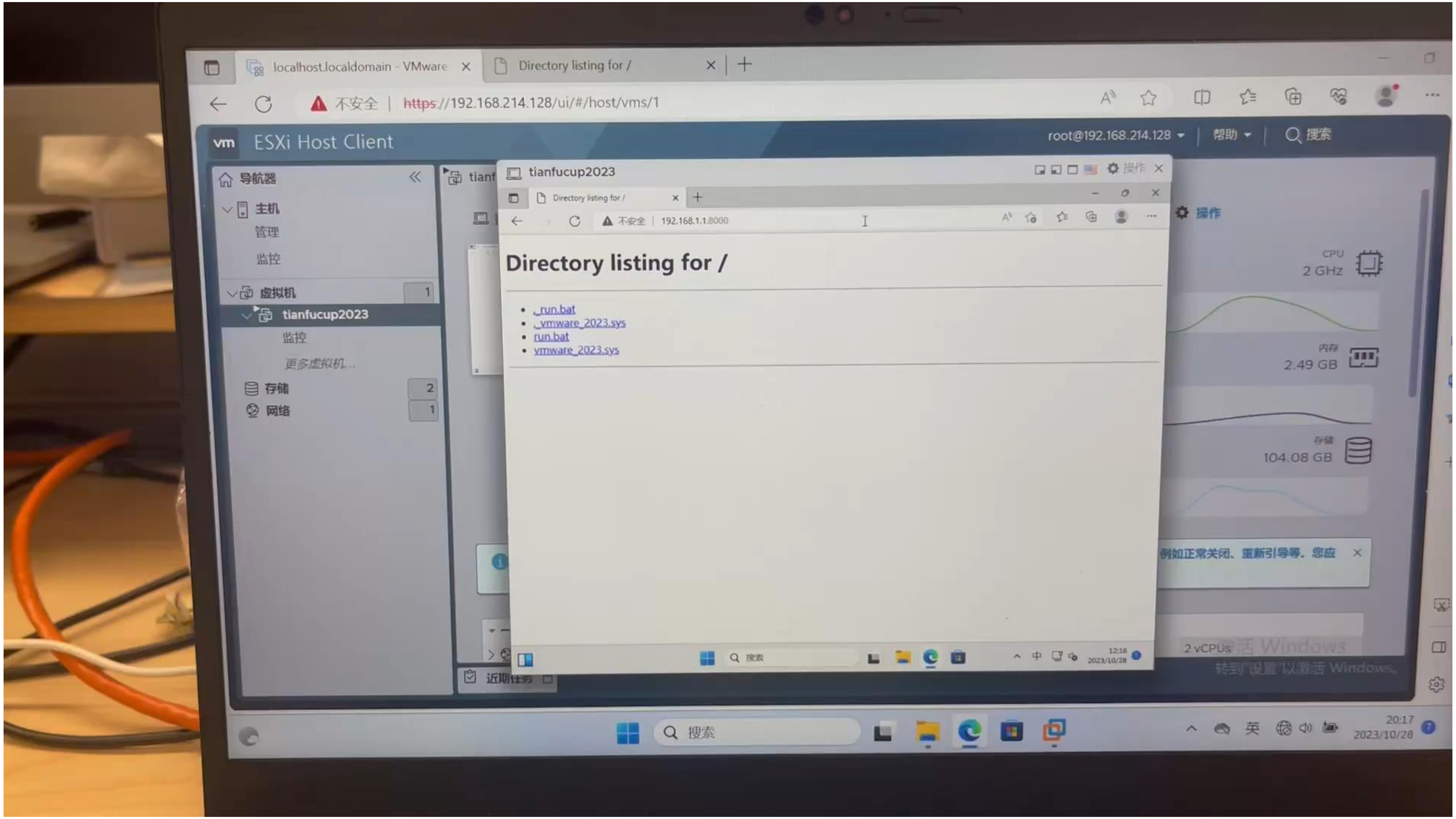
Step3:Use AAW primitive to modify SyscallMask_table and call VmkAccessEnableDomain to close the sandbox

# Summary

# Summary

- How the ESXi sandbox works
- Found a bug in the CBT driver (CVE-2024-22254)
- Used OOB write + heap tricks to Escaped the sandbox and got full control
- Small drivers can be dangerous

# Demo Video


localhost.localdomain - VMware
Directory listing for /
https://192.168.214.128/ui/#/host/vms/1
ESXi Host Client
root@192.168.214.128
导航器
主机
管理
监控
虚拟机
tianfucup2023
存储
网络
tianfucup2023
192.168.1.1:8000
Directory listing for /
_run.bat
_vmware_2023.sys
run.bat
vmware_2023.sys
2 GHz
2.49 GB
104.08 GB
2 vCPUs
2023/10/28

# Thank you

# Questions?