Hardening Hyper-V through offensive security research

Jordan Rabet, Microsoft OSR

Note: all vulnerabilities mentioned in this talk have been addressed
Hyper-V 101
Host OS

Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: accessing hardware resources from Guest OS
Hyper-V architecture: virtualization providers can be in user-mode
vmbus internals: small packet

Physical memory

System Physical Addresses (SPA)

Host physical memory

Shared virtual ringbuffer

vmbus

Packet

Guest physical memory

System Virtual Addresses (SVA)

Kernel mode

Host OS

Guest OS

Host physical memory

Shared virtual ringbuffer

vmbusr

VSP

VSC

vmbus

Guest Virtual Addresses (GVA)

Guest Physical Addresses (GPA)

Physical memory

Host OS

Guest OS

vmbus

vmbusr

VSP

Packet

System Virtual Addresses (SVA)

Physicial addresses (PA)
vmbus internals: small packet

Physical addresses (PA)

Guest Physical Addresses (GPA)

Guest Virtual Addresses (GVA)

System Physical Addresses (SPA)

System Virtual Addresses (SVA)

Host physical memory

Guest physical memory

Host OS

Guest OS

Shared virtual ringbuffer

Packet

vmbus

vmbusr

VSP

VSC

Kernel mode
vmbus internals: small packet
vmbus internals: small packet passing a direct mapping (GPADL)
What about security? Host OS mitigations

Host OS kernel

• Full KASLR
• Kernel Control Flow Guard
  • Optional
• Hypervisor-enforced code integrity (HVCI)
  • Optional
• No sandbox

VM Worker Process

• ASLR
• Control Flow Guard (CFG)
• Arbitrary Code Guard (ACG)
• Code Integrity Guard (CIG)
• Win32k lockdown
VSP case study: vmswitch
vmswitch emulates a network card through the RNDIS protocol.

vmswitch is a VSP, lives in host kernel.

netVSC tunnels traffic over to vmswitch.

vmswitch: virtualized network provider.
vmswitch: initialization sequence
vmswitch: initialization sequence
vmswitch: initialization sequence
vmswitch: initialization sequence

- Host physical memory
- Guest physical memory
- Receive Buffer
- Kernel mode
- Host OS
- Guest OS

vmbus messages

NVSP_PROTOCOL_VERSION_5

OK!

NDIS v6.30

OK!

Receive buffer GPADL

OK!
vmswitch: initialization sequence
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
Guest OS

Host OS

Kernel mode

Send buffer

Receive buffer

RNDIS QUERY

vmswitch

vmusbus messages

netVSC

Kernel mode

Guest OS

vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: sending RNDIS packets
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
Initialization sequence vulnerability
Messing with the initialization sequence
Messing with the initialization sequence

NVSP_PROTOCOL_VERSION_5
OK!

NDIS v6.30
OK!

Receive Buffer Pointer
Messing with the initialization sequence

Host physical memory

Receive Buffer Pointer

GPADL 0

NVSP_PROTOCOL_VERSION_5
OK!

NDIS v6.30
OK!

Receive buffer: GPADL 0
OK!

Guest physical memory

Physical memory

Host OS

Guest OS

Kernel mode
Messing with the initialization sequence
Messing with the initialization sequence
Messing with the initialization sequence
Receive buffer update isn’t atomic
1. Updates the pointer to the buffer
2. Generates and updates sub-allocations

No locking on the receive buffer
• It could be used in parallel
vmswitch receive buffer update
vmswitch receive buffer update
Host OS

Kernel mode

vmswitch

Receive buffer: GPADL 0

vmbus channel

GPADL 0

GPADL 1

Receive Buffer Pointer

vmswitch receive buffer update
Kernel mode

Host OS

vmswitch

vmbus channel

Receive buffer: GPADL 0

Receive Buffer Pointer

GPADL 0

GPADL 1
Host OS

Kernel mode

vmswitch

vmbus channel

Receive buffer: GPADL 0

OK!

Receive Buffer Pointer

GPADL 0

GPADL 1

vmmswitch receive buffer update
vmswitch receive buffer update
1. Update pointer to receive buffer

vmswitch receive buffer update
Kernel mode

1. Update pointer to receive buffer

Host OS

vmswitch receive buffer update

vmswitch

vmbus channel

Receive buffer: GPADL 0

OK!

Receive buffer: GPADL 1

Receive Buffer Pointer

GPADL 0

GPADL 1

Update pointer to receive buffer
2. Generate bounds of sub-allocations

vmswitch receive buffer update
Host OS

Kernel mode

Receive Buffer Pointer

GPADL 0

GPADL 1

vmswitch

vmdbus channel

3. Update bounds of sub-allocations

vmmswitch receive buffer update
Host OS

Kernel mode

Receive buffer: GPADL 0

vmswitch receive buffer update

vmswitch

vmbus channel

GPADL 0

GPADL 1

Receive Buffer Pointer

Update bounds of sub-allocations

1. Generate bounds of sub-allocations
2. Update bounds of sub-allocations
3. OK!
Receive buffer race condition

• During this short window, we can have out-of-bound sub-allocations
• This results in a useful out-of-bounds write if:
  1. We can control the data being written
  2. We can win the race
  3. We can place a corruption target adjacent to the receive buffer

1. Update pointer to receive buffer
2. Generate bounds of sub-allocations
3. Update bounds of sub-allocations
Exploiting the vulnerability

- Controlling what’s written out-of-bounds
- Winning the race
- Finding a reliable corruption target
Exploiting the vulnerability

- Controlling what’s written out-of-bounds
- Winning the race
- Finding a reliable corruption target
Controlling the OOB write contents

- OOB write contents: RNDIS control message responses
- RNDIS_QUERY_MSG messages can return large buffers of data

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>MessageType</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>MessageLength</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>RequestId</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Status</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>InformationBufferLength</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>InformationBufferOffset</td>
</tr>
</tbody>
</table>
Controlling the OOB write contents

- OOB write contents: RNDIS control message responses
- RNDIS_QUERY_MSG messages can return large buffers of data

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>MessageType</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>MessageLength</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>RequestId</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Status</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>InformationBufferSize</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>InformationBufferOffset</td>
</tr>
</tbody>
</table>
Exploiting the vulnerability

- Controlling what’s written out-of-bounds
- Winning the race
- Finding a reliable corruption target
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
vmswitch: handling RNDIS messages is asynchronous, but not really
Winning the race: delaying one RNDIS message?

• Can’t have RNDIS messages continuously write to the receive buffer
  • But we don’t need continuous RNDIS messages – we just need one
  • Can we send an RNDIS message and have it be processed in a delayed way?

• No by-design way of delaying RNDIS messages...

• ...but not all messages require an ack from the guest
  • Example: malformed RNDIS_KEEPALIVE_MSG message

• Idea: “cascade of failure”
  • Block off all RNDIS worker threads
  • Chain $N$ malformed RNDIS_KEEPALIVE_MSG messages
  • Append a single valid RNDIS message
The Cascade Of Failure: making the host race itself
The Cascade Of Failure: making the host race itself
The Cascade Of Failure: making the host race itself
The Cascade Of Failure: making the host race itself

```
Kernel mode

The Cascade Of Failure: making the host race itself
```

```
Waiting on MSG 0
ack from guest
RNDIS worker thread 1

Waiting on MSG 1
ack from guest
RNDIS worker thread 2
```
Kernel mode

The Cascade Of Failure: making the host race itself
The Cascade Of Failure: making the host race itself

- RNDIS MSG queue
- Written to the receive buffer after a controlled delay
- RNDIS MSG 8
- RNDIS MSG 8 CMPLT
- RNDIS worker thread 1
- Waiting on MSG 1 ack from guest
- RNDIS worker thread 2
- OK MSG 0!
- Channel thread
- vmswitch
- vmbus channel
- Host OS
Winning the race: configuring the delay

• We can delay the event by $N$ time units, but what’s $N$’s value?
  • We have a limited number of tries: need to be smart

• Can we distinguish between race attempt outcomes?
  • If so we could search for the right $N$
GPADL 0

GPADL 1

Update pointer to receive buffer

Update bounds of sub-allocations

Too early

Too late

Just right
Update pointer to receive buffer

Update bounds of sub-allocations
Update pointer to receive buffer

Update bounds of sub-allocations
Update pointer to receive buffer

Update bounds of sub-allocations
Winning the race: configuring the delay

• We can delay the event by $N$ time units, but what’s $N$’s value?
  • We have a limited number of tries: need to be smart

• Can we distinguish between race attempt outcomes?
  • Yes
    • If we’re too early, increase $N$
    • If we’re too late, decrease $N$
    • If we’re just right... celebrate 😊

• In practice we usually converge to the right $N$ in <10 attempts
  • $N$ can vary from machine to machine and session to session
Exploiting the vulnerability

- Controlling what’s written out-of-bounds
- Winning the race
- Finding a reliable corruption target
Finding a target: where’s our buffer?

• GPADL mapping
  • GPADL PAs mapped into an MDL using VmbChannelMapGpadl
  • MDL then mapped to VA space using MmGetSystemAddressForMdlSafe

• Where are MDLs mapped to? The SystemPTE region

• What’s mapped adjacent to our MDL?

  0: kd> !address @@c++(ReceiveBuffer)
  Usage:
  Base Address:  ffffffff`273d5000
  End Address:  ffffffff`27606000
  Region Size:  00000000`00231000
  VA Type:  SystemRange

• ...other MDLs 😑
Finding a target: other MDLs and... stacks???

0: kd> !address

...  
fffdd80`273bb000 ffffdd80`273c1000 0`000006000 SystemRange Stack  Thread: ffffc903f188b080  
fffdd80`273c1000 ffffdd80`273c6000 0`00005000 SystemRange Stack  Thread: ffffc903eed10800  
fffdd80`273c6000 ffffdd80`273cc000 0`00006000 SystemRange Stack  Thread: ffffc903f182b080  
fffdd80`273cc000 ffffdd80`273cf000 0`00003000 SystemRange Stack  Thread: ffffc903f181f080  
fffdd80`273d5000 ffffdd80`27606000 0`00231000 SystemRange Stack  Thread: ffffc903ee878080  
fffdd80`27606000 ffffdd80`2760c000 0`00006000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`2760c000 ffffdd80`2760d000 0`00001000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`2760d000 ffffdd80`27613000 0`00006000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`27613000 ffffdd80`27625000 0`00012000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`27625000 ffffdd80`2762b000 0`00006000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`2762b000 ffffdd80`2762c000 0`00001000 SystemRange Stack  Thread: ffffc903ee981080  
fffdd80`2762c000 ffffdd80`27632000 0`00006000 SystemRange Stack  Thread: ffffc903f1bc64c0  
...
Finding a target: kernel stacks

• Windows kernel stacks
  • Fixed 7 page allocation size
    • 6 pages of stack space
    • 1 guard page at the bottom
  • Allocated in the SystemPTE region
  • Great corruption target if within range – gives instant ROP

• Problems
  • How does the SystemPTE region allocator work?
  • Can we reliably place a stack at a known offset from our receive buffer?
  • Can we even “place” a stack? How do we spawn threads?
SystemPTE allocator

• Bitmap based
  • Each bit represents a page
  • Bit 0 means free page, 1 means allocated
• Uses a “hint” for allocation
  • Scans bitmap starting from hint
  • Wraps around bitmap if needed
  • Places hint at tail of successful allocations
• Bitmap is expanded if no space is found
SystemPTE allocator

• Bitmap based
  • Each bit represents a page
  • Bit 0 means free page, 1 means allocated

• Uses a “hint” for allocation
  • Scans bitmap starting from hint
  • Wraps around bitmap if needed
  • Places hint at tail of successful allocations

• Bitmap is expanded if no space is found
• Example 1: allocating 5 pages
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages

![Allocation bitmap](image)
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
- Example 2: allocating 5 pages again

```
Bitmap hint
Free page  Allocated page
```

Allocation bitmap
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
- Example 2: allocating 5 pages again
SystemPTE allocator

• Bitmap based
  • Each bit represents a page
  • Bit 0 means free page, 1 means allocated

• Uses a “hint” for allocation
  • Scans bitmap starting from hint
  • Wraps around bitmap if needed
  • Places hint at tail of successful allocations

• Bitmap is expanded if no space is found
• Example 1: allocating 5 pages
• Example 2: allocating 5 pages again
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
- Example 2: allocating 5 pages again
- Example 3: allocating 17 pages
**SystemPTE allocator**

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
- Example 2: allocating 5 pages again
- Example 3: allocating 17 pages
SystemPTE allocator

• Bitmap based
  • Each bit represents a page
  • Bit 0 means free page, 1 means allocated
• Uses a “hint” for allocation
  • Scans bitmap starting from hint
  • Wraps around bitmap if needed
  • Places hint at tail of successful allocations
• Bitmap is expanded if no space is found
• Example 1: allocating 5 pages
• Example 2: allocating 5 pages again
• Example 3: allocating 17 pages
SystemPTE allocator

• Bitmap based
  • Each bit represents a page
  • Bit 0 means free page, 1 means allocated

• Uses a “hint” for allocation
  • Scans bitmap starting from hint
  • Wraps around bitmap if needed
  • Places hint at tail of successful allocations

• Bitmap is expanded if no space is found

• Example 1: allocating 5 pages

• Example 2: allocating 5 pages again

• Example 3: allocating 17 pages
SystemPTE allocator

- Bitmap based
  - Each bit represents a page
  - Bit 0 means free page, 1 means allocated
- Uses a “hint” for allocation
  - Scans bitmap starting from hint
  - Wraps around bitmap if needed
  - Places hint at tail of successful allocations
- Bitmap is expanded if no space is found
- Example 1: allocating 5 pages
- Example 2: allocating 5 pages again
- Example 3: allocating 17 pages
Finding a target: allocation primitives

• Receive/send buffers: we can map an arbitrary number of arbitrarily sized MDLs
  • (“arbitrary”: still have size/number limits, but they’re pretty high)

• Receive/send buffers: can be revoked
  • NVSP_MSG1_TYPE_REVOKE_RECV_BUF and NVSP_MSG1_TYPE_REVOKE_SEND_BUF
  • Since replacing buffers is a bug, we can only revoke the last one sent for each

• We have pretty good allocation and freeing primitives for manipulating the region

• But we need a way to allocate new stacks if we want to target them...
  • Can we spray host-side threads?
Finding a target: stack allocation primitives

• vmswitch relies on System Worker Threads to perform asynchronous tasks
  • NT-maintained thread pool
  • Additional threads are added to the pool when all others are busy
• Basic idea: trigger an asynchronous task many times in rapid succession
  • If enough tasks are queued quickly enough, threads will be spawned
• Several vmswitch messages rely on System Worker Threads
  • In this exploit we use NVSP_MSG2_TYPE_SEND_NDIS_CONFIG
• Problem
  • This method usually lets us create about 5 threads
  • What if there are already a lot of threads in the system worker pool?
  • Would be nice to be able to terminate them...
Finding a target: stack allocation primitives

- There’s no by-design way to terminate worker threads from a guest
- But there are bugs we can use! 😊
- NVSP_MSG1_TYPE_REVOKE_SEND/RECV_BUF
  - Revocation done on system worker threads
  - Deadlock bug: when multiple revocation messages handled, all but the last system worker thread would be deadlocked forever
- We can use this to lock out an “arbitrary” number of system worker threads
- We now have a limited thread stack spray!
SystemPTE massaging strategy

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer  
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks

Two possible outcomes, both manageable
SystemPTE massaging strategy
Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy
Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #1

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
Exploiting the vulnerability

- Controlling what’s written out-of-bounds
- Winning the race
- Finding a reliable corruption target
- Bypassing KASLR
nvsp_message struct

• Represents messages sent to/from vmswitch over vmbus

```
struct nvsp_message {
    struct nvsp_message_header hdr;
    union nvsp_all_messages msg;
} __packed;
```
nvsp_message struct

- Represents messages sent to/from vmswitch over vmbus

```c
struct nvsp_message {
    struct nvsp_message_header hdr;
    union nvsp_all_messages msg;
} __packed;
```
### NVSP_MSG1_TYPE_SEND_NDIS_VER

<table>
<thead>
<tr>
<th>UINT32</th>
<th>hdr.msg_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT32</td>
<td>ndis_major_ver</td>
</tr>
<tr>
<td>UINT32</td>
<td>ndis_minor_ver</td>
</tr>
</tbody>
</table>

### NVSP_MSG1_TYPE_SEND_RNDIS_PKT_COMPLETE

<table>
<thead>
<tr>
<th>UINT32</th>
<th>hdr.msg_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT32</td>
<td>status</td>
</tr>
</tbody>
</table>
NVSP_MSG1_TYPE_SEND_NDIS_VER

- UINT32 hdr.msg_type
- UINT32 ndis_major_ver
- UINT32 ndis_minor_ver

msg.send_ndis_ver

NVSP_MSG1_TYPE_SEND_RNDIS_PKT_COMPLETE

- UINT32 hdr.msg_type
- UINT32 status

msg.send_rndis_pkt_complete
NVSP_MSG1_TYPE_SEND_NDIS_VER

<table>
<thead>
<tr>
<th>UINT32</th>
<th>hdr.msg_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT32</td>
<td>ndis_major_ver</td>
</tr>
<tr>
<td>UINT32</td>
<td>ndis_minor_ver</td>
</tr>
</tbody>
</table>

NVSP_MSG1_TYPE_SEND_RNDIS_PKT_COMPLETE

<table>
<thead>
<tr>
<th>UINT32</th>
<th>hdr.msg_type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>status</td>
</tr>
</tbody>
</table>
### NVSP_MSG1_TYPE_SEND_NDIS_VER

- **hdr.msg_type**: UINT32
- **ndis_major_ver**: UINT32
- **ndis_minor_ver**: UINT32

### NVSP_MSG1_TYPE_SEND_RNDIS_PKT_COMPLETE

- **hdr.msg_type**: UINT32
- **status**: UINT32

The size of the `nvsp_message` is `sizeof(nvsp_message)`.
Infoleak

- `nvsp_message` is allocated on the stack
- Only the first 8 bytes are initialized
- `sizeof(nvsp_message)` is returned

⇒ 32 bytes of uninitialized stack memory are sent back to guest
Putting it all together

- We can leak 32 bytes of host stack memory
- We can leak a vmswitch return address
- With a return address we can build a ROP chain 😊
Putting it all together

• We can leak 32 bytes of host stack memory
• We can leak a vmswitch return address
• With a return address we can build a ROP chain 😊
• Final exploit:
  • Use infoleak to locate vmswitch
  • Use information to build a ROP chain
    • We don’t know for sure which stack we’re corrupting, so we prepend a ROP NOP-sled
    • (that just means a bunch of pointers to a RET instructions in a row)
  • Perform host SystemPTE massaging
  • Use race condition to overwrite host kernel thread stack with ROP chain
Bypassing KASLR without an infoleak

• Our infoleak applied to Windows Server 2012 R2, but not Windows 10
  • Oops 😞

• How do we deal with KASLR without an infoleak?
  • KASLR only aligns most modules up to a 0x10000 byte boundary
  • As a result, partial overwrites are an option

• Example:
  • Return address is: 0xfffffffff808e059f3be (RndisDevHostDeviceCompleteSetEx+0x10a)
  • Corrupt it to: 0xfffffffff808e04b8705 (ROP gadget: pop r15; ret;)

• Can only do a single partial overwrite though... is that useful?
  • Only one partial overwrite because our OOB write is contiguous
Replaceable receive buffer

Thread stack

Free page
Allocated page

SystemPTE massaging
Partial overwrite

• What if we use it to get RSP into our send buffer?
  • Target return address: 0xFFFFF808E059F3BE
  • We corrupt it to: 0xFFFFF808E059DA32
    
    ```
    lea r11, [rsp+0E50h]
    mov rbx, [r11+38h]
    mov rbp, [r11+40h]
    mov rsp, r11
    ...
    retn
    ```
  • We end up doing RSP += 0xE78
Partial overwrite

• What if we use it to get RSP into our send buffer?
  • Target return address: 0xFFFFF808E059F3BE
  • We corrupt it to: 0xFFFFF808E059DA32

```assembly
    lea r11, [rsp+0E50h]
    mov rbx, [r11+38h]
    mov rbp, [r11+40h]
    mov rsp, r11
    ...
    retn
```

• We end up doing RSP += 0xE78

Target kernel thread stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF808E059DA32</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Send buffer

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC500F6000000</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Partial overwrite

- What if we use it to get RSP into our send buffer?
  - Target return address: 0xFFFFF808E059F3BE
  - We corrupt it to: 0xFFFFF808E059DA32

```assembly
lea r11, [rsp+0E50h]
mov rbx, [r11+38h]
mov rbp, [r11+40h]
mov rsp, r11
...
retn
```

- We end up doing RSP += 0xE78
Partial overwrite

- What if we use it to get RSP into our send buffer?
  - Target return address: 0xFFFFF808E059F3BE
  - We corrupt it to: 0xFFFFF808E059DA32

```assembly
lea r11, [rsp+0E50h]
mov rbx, [r11+38h]
mov rbp, [r11+40h]
mov rsp, r11
...
retn
```

- We end up doing RSP += 0xE78

- This moves RSP into our send buffer... ... which is shared with the guest
Host kernel stack in shared memory: what now?

1. The host CPU core throws a General Protection Fault (GPF)
   • No KASLR bypass means the RET instruction will necessarily cause a fault
2. The address where the GPF happened is dumped to the stack
   • In shared memory! We can read it, and that’s our KASLR bypass
3. Windows executes its GPF handler, still with the stack in shared memory
4. As attackers, we can:
   1. Locate valid ROP gadget thanks to addresses being dumped to the stack
   2. Manipulate the stack as the exception handler is being executed
      • Includes exception records and of course other return addresses
5. As a result, we get ROP execution in host 😊
Demo time
Hardening Hyper-V
1. Targeted, continuous internal code review effort

2. Break exploit techniques

3. Make components less attractive targets, invest in detection

Breaking the chain
Hardening: kernel stack isolation

To prevent overflowing into kernel stacks, we’ve moved them to their own region

```
0: kd> !address
...
ffffffae8f`050a8000  ffffae8f`050a9000  0`00001000  SystemRange
ffffffae8f`050a9000  ffffae8f`050b0000  0`00007000  SystemRange  Stack  Thread: ffffbc8934d51700
ffffffae8f`050b0000  ffffae8f`050b1000  0`00001000  SystemRange
ffffffae8f`050b1000  ffffae8f`050b8000  0`00007000  SystemRange  Stack  Thread: ffffbc8934d55700
ffffffae8f`050b8000  ffffae8f`050b9000  0`00001000  SystemRange
ffffffae8f`050b9000  ffffae8f`050c0000  0`00007000  SystemRange  Stack  Thread: ffffbc8934d59700
ffffffae8f`050c0000  ffffae8f`050c1000  0`00001000  SystemRange
ffffffae8f`050c1000  ffffae8f`050c8000  0`00007000  SystemRange  Stack  Thread: ffffbc8934d5d700
...
```
Hardening: other kernel mitigations

• Hypervisor-enforced Code Integrity (HVCI)
  • Attackers can’t inject arbitrary code into Host kernel

• Kernel-mode Control Flow Guard (KCFG)
  • Attackers can’t achieve kernel ROP by hijacking function pointers

• Work is being done to enable these features by default

• Future hardware security features: CET
  • Hardware shadow stacks to protect return addresses and prevent ROP
Hardening: VM Worker Process

• Improved sandbox
  • Removed SeImpersonatePrivilege

• Improved RCE mitigations
  • Enabled CFG export suppression
    • Large reduction in number of valid CFG targets
  • Enabled “Force CFG”
    • Only CFG-enabled modules modules can be loaded into VMWP

• Several Hyper-V components being put in VMWP rather than kernel
The Hyper-V bounty program

• Up to $250,000 payout
  • Looking for code execution, infoleaks and denial of service issues

• Getting started
  • Joe Bialek and Nicolas Joly’s talk: “A Dive in to Hyper-V Architecture & Vulnerabilities”
  • Hyper-V Linux integration services
    • Open source, well-commented code available on Github
    • Good way to understand VSP interfaces and experiment!
  • Public symbols for some Hyper-V components
Thank you for your time

Special thanks to Matt Miller, David Weston, the Hyper-V team, the vmswitch team, the MSRC team and all my OSR buddies
Appendix
Hyper-V architecture: VMWP compromise

Host technically compromised, but limited to VMWP user-mode

Malicious guest

Hypercall
Address manager
MSRs
Hyper-V architecture: VMWP compromise
Hyper-V architecture: VMWP to host kernel compromise

Attacker escapes user-mode through local kernel, driver exploit...
Hyper-V architecture: VMWP to host kernel compromise

Attacker goes for host kernel directly through VSP surface
Attacker compromises hypervisor, either directly from guest or through the host.

Hyper-V architecture: hypervisor compromise
vmswitch initialization: NVSP_MSG_TYPE_INIT
vmswitch initialization: NVSP_MSG1_TYPE_SEND_NDIS_VER
vmswitch initialization: NVSP_MSG1_TYPE_SEND_RECV_BUF
vmswitch initialization: NVSP_MSG1_TYPE_SEND_SEND_BUF
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch: how are RNDIS messages handled?
vmswitch state machine

**vmswitch messages**

<table>
<thead>
<tr>
<th>NVSP Message Type</th>
<th>State #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVSP_MSGTYPE_INIT</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_NDIS_VER</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RECV_BUF</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_RECV_BUF</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_SEND_BUF</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_SEND_BUF</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RNDIS_PKT</td>
<td></td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG5_TYPE_SUBCHANNEL</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>RNDIS_INITIALIZE_MSG</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>RNDIS_HALT_MSG</td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>RNDIS_HALT_MSG</td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>RNDIS_INITIALIZE_MSG</td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
vmswitch takeaways

• Send/receive buffers are used to transfer many messages at a time
• Opposite end needs to be prompted over vmbus to read from them
• vmswitch relies on different threads for different tasks
  • vmbus dispatch threads
    • Setup send/receive buffers, subchannels...
    • Read RNDIS messages from send buffer
  • The system worker threads
    • Process RNDIS messages
    • Write responses to receive buffer
• Subchannels only increase bandwidth in that they allow us to alert the opposite end more often
### vmswitch state machine

#### State Machine Diagram

- **None** → **Initializing**
- **Initializing** → **Operational** or **Halted**
- **Operational** → **Halted**
- **Halted** → **None**

---

#### NVSP Message Type

<table>
<thead>
<tr>
<th>NVSP Message Type</th>
<th>State #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVSP_MSG_TYPE_INIT</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_NDIS_VER</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RECV_BUF</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_RECV_BUF</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_SEND_BUF</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_SEND_BUF</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RNDIS_PKT</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG5_TYPE_SUBCHANNEL</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
**vmswitch state machine**

- **None**
  - Transition to **Initializing**

- **Initializing**
  - Transition to **Operational**
  - Transition to **Halted**

- **Operational**
  - Transition to **Halted**

- **Halted**
  - Transition to **Operational**

**NVSP Message Type**

<table>
<thead>
<tr>
<th>NVSP Message Type</th>
<th>State #</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG_TYPE_INIT</td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_NDIS_VER</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RECV_BUF</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_RECV_BUF</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_SEND_BUF</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_SEND_BUF</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RNDIS_PKT</td>
<td>✗</td>
</tr>
<tr>
<td>NVSP_MSG5_TYPE_SUBCHANNEL</td>
<td>✗</td>
</tr>
</tbody>
</table>
Winning the race: continuous writing?

• Easy way to win the race: queue up RNDIS messages and keep having them write to receive buffer continuously
  • Doesn’t work: RNDIS threads blocked until ack from guest
  • Ack and buffer replacement happen on same channel: can’t happen simultaneously...

• ...unless we use subchannels!
  • Multiple channels = simultaneity
Winning the race: continuous writing?

- Easy way to win the race: queue up RNDIS messages and keep having them write to receive buffer continuously
  - Doesn’t work: RNDIS threads blocked until ack from guest
  - Ack and buffer replacement happen on same channel: can’t happen simultaneously...
- ...unless we use subchannels!
  - Multiple channels = simultaneity
- ...but we can’t because of the state machine

<table>
<thead>
<tr>
<th>NVSP Message Type</th>
<th>State #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVSP_MSG_TYPE_INIT</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_NDIS_VER</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RECV_BUF</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_RECV_BUF</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_SEND_BUF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_REVOKE_SEND_BUF</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG1_TYPE_SEND_RNDIS_PKT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSP_MSG5_TYPE_SUBCHANNEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

vmswitch state machine
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks

Allocation bitmap:
- Free page
- Allocated page

Bitmap hint
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy
Outcome #2

1. Spray 1MB buffers

2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)

3. Allocate a 1MB buffer

4. Allocate a 1MB - 7 pages buffer

5. Spray stacks
SystemPTE massaging strategy
Outcome #2

1. Spray 1MB buffers

2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)

3. Allocate a 1MB buffer

4. Allocate a 1MB - 7 pages buffer

5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks

Allocation bitmap

- Free page
- Allocated page

Bitmap hint
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
**SystemPTE massaging strategy**

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   - (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy

Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
SystemPTE massaging strategy
Outcome #2

1. Spray 1MB buffers
2. Allocate a 2MB - 1 page buffer
   • (SystemPTE expansions are done in 2MB steps)
3. Allocate a 1MB buffer
4. Allocate a 1MB - 7 pages buffer
5. Spray stacks
Finding a target: SystemPTE massaging

• After massaging, we know a stack is at one of two offsets from the receive buffer
  • Either 3MB - 6 pages away or 4MB - 6 pages away

• Since we can perform the race reliably, we can just try both possible offsets
  • Note: doing the race requires revoking and re-mapping the receive buffer
  • We can do this because the SystemPTE bitmap will free our 2MB block and reuse it for next 2MB block allocation
  • As a result, we’re almost guaranteed to fall back into the same slot if we’re fast enough

• We can overwrite a stack, but what do we write?
  • Overwriting return addresses requires a host KASLR bypass
  • Easiest way to do this: find an infoleak vulnerability