A look inside the Windows Kernel

Bruno Pujos

LSE

July 18, 2013

Plan

1. Introduction

Introduction

What this talk is about?

- Security of the Windows Kernel
- Presentation of some exploits
- What changed in the security of the kernel, since Windows NT 5.1 (Windows XP)

Motivation for attacking the kernel

- Sandbox bypassing
- Full access to everything
- The fun

Plan

1. Introduction
2. Basics of Windows Kernel
3. CVE-2011-1237
4. Evolution from XP to 8
5. CVE-2013-3660
6. Conclusion

Plan

2 Basics of Windows Kernel

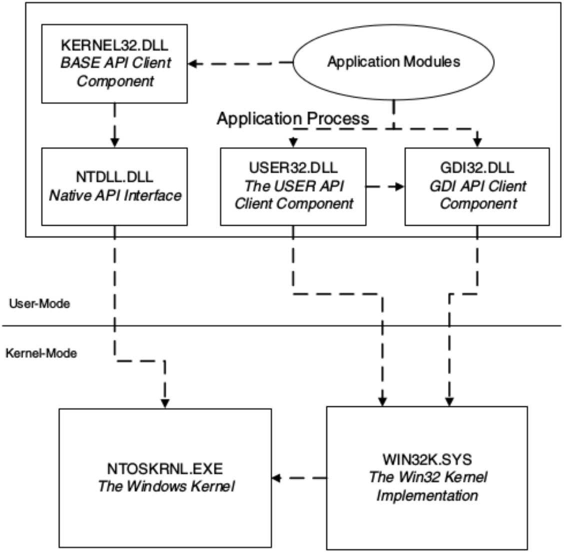


Figure 3.3 The Win32 interface DLLs and their relation to the kernel components.

HAL

- HAL : The hardware abstraction layer (hal.dll)
- ”a layer of software that deals directly with your computer hardware.” (msdn)
- Layer for suporting different hardware with the same software
- HalDispatchTable : holds the addresses of a few HAL routines

Win32k.sys

- Kernel mode driver
- Introduce in NT 4.0 for performance reason
- Two parts :
  - The Graphics Device Interface (GDI)
  - The Window Manager

User objects

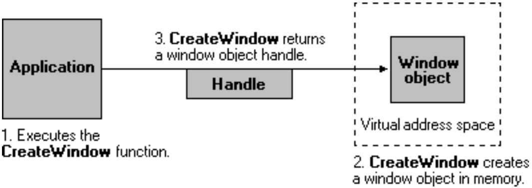
- User entities (Windows, menu, keyboard layout. . .)
- Managed by the Window Manager
- Represented by a handle
- Handle table keeps track of each user object
  - The address of the object
  - The type of the object
  - A flag
  - The owner and a wUniq value

User objects

- User entities (Windows, menu, keyboard layout. . .)
- Managed by the Window Manager
- Represented by a handle
- Handle table keeps track of each user object
  - The address of the object
  - The type of the object
  - A flag
  - The owner and a wUniq value

User objects

- User entities (Windows, menu, keyboard layout. . .)
- Managed by the Window Manager
- Represented by a handle
- Handle table keeps track of each user object
  - The address of the object
  - The type of the object
  - A flag
  - The owner and a wUniq value



User-Mode Callback

- A way to communicate between kernel and user:
  - access to some structures in user mode
  - used to support hooking
  - ...
- CBT-Hook: receive notifications from windows
- WindowProc: callback function wich processes the messages sent to a window

User-Mode Callback

- A way to communicate between kernel and user:
  - access to some structures in user mode
  - used to support hooking
  - ...
- CBT-Hook: receive notifications from windows
- WindowProc: callback function wich processes the messages sent to a window

Plan

3 CVE-2011-1237

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Vulnerability

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Vulnerability

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Vulnerability

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Vulnerability

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Vulnerability

- Vulnerability discovered by Tarjei Mandt (@kernelpool), based on his paper *Kernel Attacks through User-Mode Callbacks*
- Use After Free of a window object (User Object)
  - During the creation of a new window, you can give a parent in a CBT-Hook
  - Using another hook during the creation, you can destroy this window
  - We have a way to allocate a buffer with our content and the size we want with *SetWindowTextW*. We will use it to put what we want at the position of the free window
  - The parent is used at the end of *LinkWindow*, and it has been freed
- We can map the Null page and put our shellcode in it, in userland. Our goal is to call it

Link Window

- Basically, it just adds an element in a double chained list of windows
- clockObj: part of each User Object, reference counter
- Since we control one of the objects we can decrement an arbitrary a word by one
- If the clockObj is null, it calls the function *HMDestroyUnlockedObject*

Link Window

- Basically, it just adds an element in a double chained list of windows
- clockObj: part of each User Object, reference counter
- Since we control one of the objects we can decrement an arbitrary a word by one
- If the clockObj is null, it calls the function *HMDestroyUnlockedObject*

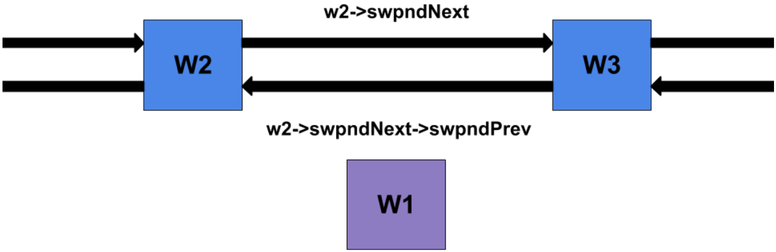
Link Window

- Basically, it just adds an element in a double chained list of windows
- clockObj: part of each User Object, reference counter
- Since we control one of the objects we can decrement an arbitrary a word by one
- If the clockObj is null, it calls the function *HMDestroyUnlockedObject*

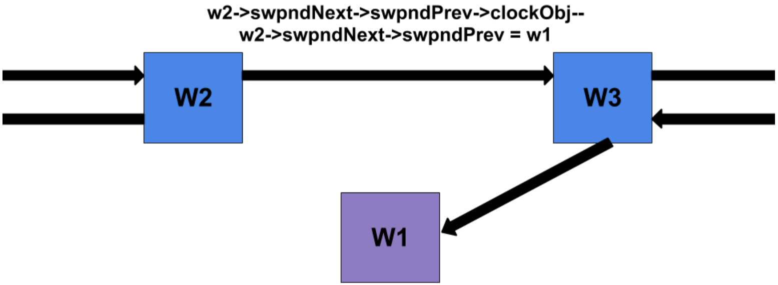
Link Window

- Basically, it just adds an element in a double chained list of windows
- clockObj: part of each User Object, reference counter
- Since we control one of the objects we can decrement an arbitrary a word by one
- If the clockObj is null, it calls the function *HMDestroyUnlockedObject*

Link Window

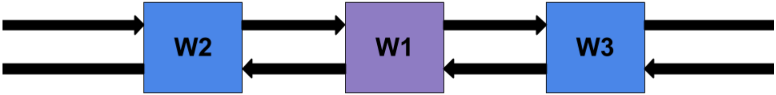


Link Window



w2->swpndNext->swpndPrev->clockObj--
w2->swpndNext->swpndPrev = w1
W2
W3
W1

Link Window



Link Window


w2->swpndNext->swpndPrev->clockObj--
w2->swpndNext->swpndPrev = w1
W2
W3
W1

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (DestroyWindow), realloc with a fake object (SetWindowTextW on B)
  - LinkWindow: decrement by one where we want

Exploitation - Decrement by one

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (*DestroyWindow*), realloc with a fake object (*SetWindowTextW* on B)
  - *LinkWindow*: decrement by one where we want

Exploitation - Decrement by one

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (*DestroyWindow*), realloc with a fake object (*SetWindowTextW* on B)
  - *LinkWindow*: decrement by one where we want

Exploitation - Decrement by one

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (*DestroyWindow*), realloc with a fake object (*SetWindowTextW* on B)
  - *LinkWindow*: decrement by one where we want

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (*DestroyWindow*), realloc with a fake object (*SetWindowTextW* on B)
  - *LinkWindow*: decrement by one where we want

Exploitation - Decrement by one

- Create two windows (A & B)
- Activate the hook
- Create a third window (E)
  - HCBT_CREATEWND: link with the window A
  - WM_NCCREATE: destroy A (*DestroyWindow*), realloc with a fake object (*SetWindowTextW* on B)
  - *LinkWindow*: decrement by one where we want

HMDestroyUnlockedObject

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

A look inside the Windows Kernel

Bruno Pujos

Introduction

Basics of Windows Kernel

CVE-2011-1237

Evolution from XP to 8

CVE-2013-3660

Conclusion

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

HMDestroyUnlockedObject

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

HMDestroyUnlockedObject

HMDestroyUnlockedObject

- *HMDestroyUnlockedObject*: takes the handle from the user object given as argument
- check this condition: `(flag & 1) && !(flag & 2)`
- if it is true, calls the destroying function for the object depending on his type
- If the type is 0 (already free): calls the null page

Standard

- the type for a window is 1
- in a standard moment the flag is 00

Exploitation - Calling the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free

Exploitation - Calling the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free
  - In *LinkWindow* we put a clockObj to 1, and the handler of the window U
  - when clockObj is decremented, call to *HMDestroyUnlockedObject* is done, that passes the test and calls the null page

Exploitation - Calling the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free
  - In *LinkWindow* we put a clockObj to 1, and the handler of the window U
  - when clockObj is decremented, call to *HMDestroyUnlockedObject* is done, that passes the test and calls the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free
  - In *LinkWindow* we put a clockObj to 1, and the handler of the window U
  - when clockObj is decremented, call to *HMDestroyUnlockedObject* is done, that passes the test and calls the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free
  - In *LinkWindow* we put a clockObj to 1, and the handler of the window U
  - when clockObj is decremented, call to *HMDestroyUnlockedObject* is done, that passes the test and calls the null page

Exploitation - Calling the null page

- We create a first window (U)
- We decremant the flag of the handle of U by 3 using the use-after-free (0xFD)
- We decrement the type of the handle of U by 1 (0)
- We trigger once again the use-after-free
  - In *LinkWindow* we put a clockObj to 1, and the handler of the window U
  - when clockObj is decremented, call to *HMDestroyUnlockedObject* is done, that passes the test and calls the null page

Plan

4 Evolution from XP to 8

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

- Kernel ASLR
  - Kernel Address = User Address - Local module base + Kernel module base
- Enhanced /GS
- Guard pages
- DEP improvements
- NULL dereference protection
- Kernel pool integrity checks
- SMEP/PXN

SMEP/PXN

- Supervisor Mode Execution Protection / Privileged Execute Never
- Depends on the processor
- Prevents a kernel thread to execute code in userland
- SMEP is enabled or disabled via CR4 control register
- Possible to bypass
  - ROP
  - Store the shellcode into kernel space

SMEP/PXN

- Supervisor Mode Execution Protection / Privileged Execute Never
- Depends on the processor
- Prevents a kernel thread to execute code in userland
- SMEP is enabled or disabled via CR4 control register
- Possible to bypass
  - ROP
  - Store the shellcode into kernel space

SMEP/PXN

- Supervisor Mode Execution Protection / Privileged Execute Never
- Depends on the processor
- Prevents a kernel thread to execute code in userland
- SMEP is enabled or disabled via CR4 control register
- Possible to bypass
  - ROP
  - Store the shellcode into kernel space

SMEP/PXN

- Supervisor Mode Execution Protection / Privileged Execute Never
- Depends on the processor
- Prevents a kernel thread to execute code in userland
- SMEP is enabled or disabled via CR4 control register
- Possible to bypass
  - ROP
  - Store the shellcode into kernel space

SMEP/PXN

- Supervisor Mode Execution Protection / Privileged Execute Never
- Depends on the processor
- Prevents a kernel thread to execute code in userland
- SMEP is enabled or disabled via CR4 control register
- Possible to bypass
  - ROP
  - Store the shellcode into kernel space

Plan

5 CVE-2013-3660

Vulnerability

- Vulnerability discovered by Tavis Ormandy (@taviso)
- Exploit by Tavis Ormandy and progmboy
- In *win32k!EPATHOBJ::pprFlattenRec*
- Uninitialized pointer for the next in a double linked list (part of a Path object in the GDI in win32k)
- To-userspace dereferences vulnerability
- We want to trigger a write-what-where vulnerability

Vulnerability

- Vulnerability discovered by Tavis Ormandy (@taviso)
- Exploit by Tavis Ormandy and progmboy
- In *win32k!EPATHOBJ::pprFlattenRec*
- Uninitialized pointer for the next in a double linked list (part of a Path object in the GDI in win32k)
- To-userspace dereferences vulnerability
- We want to trigger a write-what-where vulnerability

Vulnerability

- Vulnerability discovered by Tavis Ormandy (@taviso)
- Exploit by Tavis Ormandy and progmboy
- In *win32k!EPATHOBJ::pprFlattenRec*
- Uninitialized pointer for the next in a double linked list (part of a Path object in the GDI in win32k)
- To-userspace dereferences vulnerability
- We want to trigger a write-what-where vulnerability

Vulnerability

- Vulnerability discovered by Tavis Ormandy (@taviso)
- Exploit by Tavis Ormandy and progmboy
- In *win32k!EPATHOBJ::pprFlattenRec*
- Uninitialized pointer for the next in a double linked list (part of a Path object in the GDI in win32k)
- To-userspace dereferences vulnerability
- We want to trigger a write-what-where vulnerability

Vulnerability

- Vulnerability discovered by Tavis Ormandy (@taviso)
- Exploit by Tavis Ormandy and progmboy
- In *win32k!EPATHOBJ::pprFlattenRec*
- Uninitialized pointer for the next in a double linked list (part of a Path object in the GDI in win32k)
- To-userspace dereferences vulnerability
- We want to trigger a write-what-where vulnerability

Pathrec struct

```
struct _PATHRECORD {
    struct _PATHRECORD *next;
    struct _PATHRECORD *prev;
    ULONG flags;
    ULONG count;
    POINTFIX points[x];
}
```

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

Go to userspace

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

Go to userspace

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
    - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
    - After allocating from *HeavyAllocPool*, it memsets to 0
    - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

Go to userspace

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

Go to userspace

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

Go to userspace

- We need to make a specific AllocObject fail to trigger the exploitable condition: we need memory pressure.
- Allocation of the struct of a PATHREC is done of two possible ways
  - The PATHALLOC system use *HeavyAllocPool* for allocating object but have is own implementation of the free list
  - After allocating from *HeavyAllocPool*, it memsets to 0
  - But in the case of taking an element of the freelist it's not set to 0
- If we can spam the freelist with what we want we have big chances to have the next pointer where we want (in userspace)
- We can do that easily by flattening path with a lot of points we control
- We put a structure we created in userspace and we force the kernel to consider that is the next of his list

bFlatten and pprFlatten

- *EPATHOBJ::bFlatten* just goes through a list and calls *pprFlattenRec* if a flag is set on the element
- *EPATHOBJ::pprFlattenRec*
  - allocates a new pathrec
  - initialises the new (but not the next at this point)
  - sets the next of previous of the new to himself

    ```
    new->previous->next = new;
    ```

  - ...
- if we control the struct we can write the position of the new struct created by *pprFlattenRec*

bFlatten and pprFlatten

- *EPATHOBJ::bFlatten* just goes through a list and calls *pprFlattenRec* if a flag is set on the element
- *EPATHOBJ::pprFlattenRec*
  - allocates a new pathrec
  - initialises the new (but not the next at this point)
  - sets the next of previous of the new to himself

    ```
    new->previous->next = new;
    ```

  - ...
- if we control the struct we can write the position of the new struct created by *pprFlattenRec*

bFlatten and pprFlatten

- *EPATHOBJ::bFlatten* just goes through a list and calls *pprFlattenRec* if a flag is set on the element
- *EPATHOBJ::pprFlattenRec*
  - allocates a new pathrec
  - initialises the new (but not the next at this point)
  - sets the next of previous of the new to himself

    ```
    new->previous->next = new;
    ```

  - ...
- if we control the struct we can write the position of the new struct created by *pprFlattenRec*

bFlatten and pprFlatten

- *EPATHOBJ::bFlatten* just goes through a list and calls *pprFlattenRec* if a flag is set on the element
- *EPATHOBJ::pprFlattenRec*
  - allocates a new pathrec
  - initialises the new (but not the next at this point)
  - sets the next of previous of the new to himself

    ```
    new->previous->next = new;
    ```

  - ...
- if we control the struct we can write the position of the new struct created by *pprFlattenRec*

bFlatten and pprFlatten

- *EPATHOBJ::bFlatten* just goes through a list and calls *pprFlattenRec* if a flag is set on the element
- *EPATHOBJ::pprFlattenRec*
  - allocates a new pathrec
  - initialises the new (but not the next at this point)
  - sets the next of previous of the new to himself

    ```
    new->previous->next = new;
    ```

  - ...
- if we control the struct we can write the position of the new struct created by *pprFlattenRec*

Getting execution

- We can write the address of something we don't control but we control the contents of the first pointer in it: it's the address of our next element in the list
- We can write in the *HalDispatchTable* our pointer on the next will be considered as code when calling the function.
- So we need an address which is a valid pointer for the *bFlatten* loop and a valid code for execution like

```
inc eax ; 0x40
jmp dword ptr [ebp+0x40] ; 0xff6540
```

- We will rewrite the *HALDispatchTable[1]*, called by *NtQueryIntervalProfile* and not used for a lot of other things
- The `ebp+0x40` corresponds to the second argument of the *NtQueryIntervalProfile* where we put the address of our shellcode

Getting execution

- We can write the address of something we don't control but we control the contents of the first pointer in it: it's the address of our next element in the list
- We can write in the *HalDispatchTable* our pointer on the next will be considered as code when calling the function.
- So we need an address which is a valid pointer for the *bFlatten* loop and a valid code for execution like

```
inc eax ; 0x40
jmp dword ptr [ebp+0x40] ; 0xff6540
```

- We will rewrite the *HALDispatchTable[1]*, called by *NtQueryIntervalProfile* and not used for a lot of other things
- The `ebp+0x40` corresponds to the second argument of the *NtQueryIntervalProfile* where we put the address of our shellcode

Getting execution

- We can write the address of something we don't control but we control the contents of the first pointer in it: it's the address of our next element in the list
- We can write in the *HalDispatchTable* our pointer on the next will be considered as code when calling the function.
- So we need an address which is a valid pointer for the *bFlatten* loop and a valid code for execution like

```
inc eax ; 0x40
jmp dword ptr [ebp+0x40] ; 0xff6540
```

- We will rewrite the *HALDispatchTable[1]*, called by *NtQueryIntervalProfile* and not used for a lot of other things
- The `ebp+0x40` corresponds to the second argument of the *NtQueryIntervalProfile* where we put the address of our shellcode

Getting execution

- We can write the address of something we don't control but we control the contents of the first pointer in it: it's the address of our next element in the list
- We can write in the *HalDispatchTable* our pointer on the next will be considered as code when calling the function.
- So we need an address which is a valid pointer for the *bFlatten* loop and a valid code for execution like

```
inc eax ; 0x40
jmp dword ptr [ebp+0x40] ; 0xff6540
```

- We will rewrite the *HALDispatchTable[1]*, called by *NtQueryIntervalProfile* and not used for a lot of other things
- The `ebp+0x40` corresponds to the second argument of the *NtQueryIntervalProfile* where we put the address of our shellcode

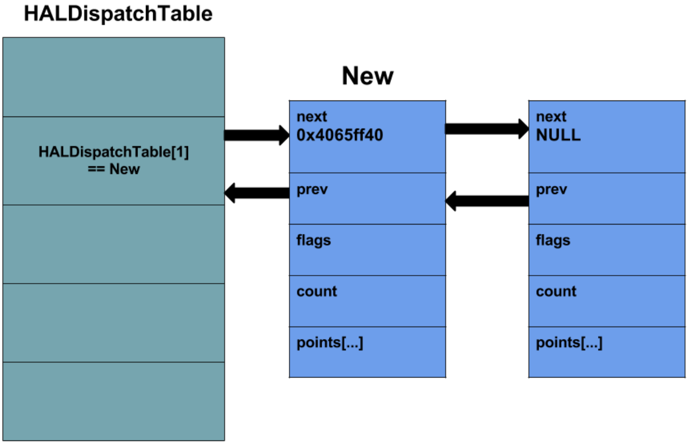
Getting execution

- We can write the address of something we don't control but we control the contents of the first pointer in it: it's the address of our next element in the list
- We can write in the *HalDispatchTable* our pointer on the next will be considered as code when calling the function.
- So we need an address which is a valid pointer for the *bFlatten* loop and a valid code for execution like

```
inc eax ; 0x40
jmp dword ptr [ebp+0x40] ; 0xff6540
```

- We will rewrite the *HALDispatchTable[1]*, called by *NtQueryIntervalProfile* and not used for a lot of other things
- The `ebp+0x40` corresponds to the second argument of the *NtQueryIntervalProfile* where we put the address of our shellcode

Getting execution



Chronology

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, ...)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, . . .)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, . . .)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

Chronology

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, . . .)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

Chronology

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, . . .)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

- Get the addresses in the kernel we need for the exploit (*HALDispatchTable*, ...)
- Allocate three structs PATHRECORD that we need, in particular the one at a precise address (0x4065ff40)
- Put memory pressure
- Put the address of our first PATHRECORD that we want into the freelist
- Flatten the path => write in the *HALDispatchTable*
- Call *NtQueryIntervalProfile* => get shellcode executed

Plan

6 Conclusion

Conclusion

- A lot of improvements between XP and Windows 8
- Lot of checks so exploits are really harder
- Still doable

Questions ?

- Tarjei Mandt (@kernelpool)
- Tavis Ormandy (@taviso)
- Mateusz Jurvczyk (@j00ru)
- Alex Ionescu (@aionescu)
- Ivanlefou (@Ivanlef0u)