Analyzing local privilege escalations in win32k

10/2008

mxatone
mxatone@gmail.com

Contents

1 Foreword **2**

2 Introduction **2**

3 Win32k design **3**
3.1 General security implementation 3
3.2 KeUsermodeCallback utilization 5

4 Discovery and exploitation **7**
4.1 DDE Kernel pool overflow 7
4.1.1 Vulnerability details 7
4.1.2 Pool overflow exploitation 9
4.1.3 Delayed free pool overflow on Windows Vista 11
4.2 NtUserfnOUTSTRING kernel overwrite vulnerability 13
4.2.1 Evading ProbeForWrite function 13
4.2.2 Vulnerability details 13
4.3 LoadMenu handle table corruption 15
4.3.1 Handle table 15
4.3.2 Vulnerability details 16

5 GUI architecture protection **17**

6 Conclusion **18**

1 Foreword

Abstract: This paper analyzes three vulnerabilities that were found in win32k.sys that allow kernel-mode code execution. The win32k.sys driver is a major component of the GUI subsystem in the Windows operating system. These vulnerabilities have been reported by the author and patched in MS08-025[1]. The first vulnerability is a kernel pool overflow with an old communication mechanism called the *Dynamic Data Exchange* (DDE) protocol. The second vulnerability involves improper use of the *ProbeForWrite* function within string management functions. The third vulnerability concerns how win32k handles system menu functions. Their discovery and exploitation are covered.

2 Introduction

The design of modern operating systems provides a separation of privileges between processes. This design restricts a non-privileged user from directly affecting processes they do not have access to. This enforcement relies on both hardware and software features. The hardware features protect devices against unknown operations. A secure environment provides only necessary rights by filtering program interaction within the overall system. This control increases provided interfaces and then security risks. Abusing operating system design or implementation flaws in order to elevate a program's rights is called a privilege escalation.

During the past few years, userland code and protection had been ameliorated. The amelioration of operating system understanding has made abnormal behaviour detection easier. The exploitation of classical weakness is harder than it was. Nowadays, local exploitation directly targets the kernel. Kernel local privilege escalation brings up new exploitation methods and most of them are certainly still undiscovered. Even if the Windows kernel is highly protected against known attack vectors, the operating system itself has a lot of different drivers that contribute to its overall attack surface.

On Windows, the graphical user interface (GUI) is divided into both kernel-mode and user-mode components. The *win32k.sys* driver handles user-mode requests for graphic rendering and window management. It also redirects DirectX calls on to the appropriate driver. For local privilege escalation, win32k represents an interesting target as it exists on all versions of Windows and some features have existed for years without modifications.

This article presents the author's work on analyzing the win32k driver to find and report vulnerabilities that were addressed in Microsoft bulletin `MS08-025` [1]. Even if the patch adds an overall protection layer, it concerns three reported vulnerabilities on different parts of the driver. The Windows graphics

stack is very complex and this article will focus on describing some of win32k's organization and functionalities. Any reader who is interested in this topic is encouraged to look at MSDN documentation for additional information[2].

The structure of this paper is as follows. In chapter 3, the win32k driver architecture basics will be presented with a focus on vulnerable contexts. Chapter 4 will detail how each of the three vulnerabilities was discovered and exploited. Finally, chapter 5 will discuss possible security improvements for the vulnerable driver.

3 Win32k design

Windows is based on a graphical user interface and cannot work without it. Only Windows Serer 2008 in server core mode uses a minimalist user interface but share the exact same components that typical user interfaces. The win32k driver is a critical component in the graphics stack exporting more than 600 functions. It extends the *System Service Descriptor Table* (SSDT) with another table called (`_W32pServiceTable`). This driver is not as big as the main kernel module (*ntoskrnl.exe*) but its interaction with the user-mode is just as important. The service table for win32k contains less than 300 functions depending on the version of Windows. The win32k driver commonly transfers control to user-mode with a user-mode callback system that will be explained in this part. The interface between user-mode modules and kernel-mode drivers has been built in order to facilitate window creation and management. This is a critical feature of Windows which may explain why exactly the same functions can be seen across multiple operating system versions.

3.1 General security implementation

The most important part of a driver in terms of security is how it validates user-mode inputs. Each argument passed as a pointer must be a valid user-mode address and be unchangeable to avoid race conditions. This validation is often accomplished by comparing a provided address with an address near the base of kernel memory using functions such as `ProbeForRead` and `ProbeForWrite`. Input contained within pointers is also typically cached in local variables (capturing). The Windows kernel design is very strict on this part. When you look deeper into win32k's functions, you will see that they do not follow the same strict integrity verifications made by the kernel. For example, consider the following check made by the Windows kernel (translated to C):

```
NTSTATUS NTAPI  NtQueryInformationPort(
    HANDLE PortHandle,
    PORT_INFORMATION_CLASS PortInformationClass,
```

```
    PVOID PortInformation,
    ULONG PortInformationLength,
    PULONG ReturnLength
    )

[...] // Prepare local variables

if (AccesMode != KernelMode)
{
    try {
        // Check submitted address - if incorrect, raise an exception
        ProbeForWrite( PortInformation, PortInformationLength, 4);

        if (ReturnLength != NULL)
        {
            if (ReturnLength > MmUserProbeAddress)
                *MmUserProbeAddress = 0; // raise exception

            *ReturnLength = 0;
        }

    } except(1) {  // Catch exceptions
        return exception_code;
    }
}

[...] // Perform actions
```

We can see that the arguments are tested in a very simple way before doing anything else. The ReturnLength field implements its own verification which relies directly on `MmUserProbeAddress`. This variable marks the separation between user-mode and kernel-mode address spaces. In case of an invalid address, an exception is raised by writting in this variable which is read-only. The `ProbeForRead` and `ProbeForWrite` functions verifications routines raised an exception if an incorrect address is encounter. However, the win32k driver does not allows follow this pattern:

```
BOOL NtUserSystemParametersInfo(
    UINT uiAction,
    UINT uiParam,
    PVOID pvParam,
    UINT fWinIni)

[...]   // Prepare local variables

switch(uiAction)
{
    case SPI_1:
        // Custom checks
        break;
    case SPI_2:
        size = sizeof(Stuct2);
        goto prob_read;
    case SPI_3:
```

```
        size = sizeof(Stuct3);
        goto prob_read;
    case SPI_4:
        size = sizeof(Stuct4);
        goto prob_read;
    case SPI_5:
        size = sizeof(Stuct5);
        goto prob_read;
    case SPI_6:
        size = sizeof(Struct6);

prob_read:
        ProbeForRead(pvParam, size, 4)

        [...]
}

[...] // Perform actions
```

This function is very complex and this example presents only a small part of the checks. Some parameters need only classic verification while others implement their own. This elaborate code can create confusion which improves the chances of a local privilege escalation. The issues comes from unordinary kernel function that handles multiple features at the same time without implementing a standardized function prototype. The Windows kernel solved this issue on *NtSet** and *NtQuery** functions by using two simple arguments. The first argument is a classical buffer and the second argument is its size. For example, the `NtQueryInformationPort` function will check the buffer in a generic way and then only verify that the size correspond to the specified feature. The win32k design implementation ameliorates GUI development but make code review very difficult.

3.2 KeUsermodeCallback utilization

Typical interaction between user-mode and kernel-mode is done via syscalls. A user-mode module may request that the kernel execute an action and return needed information. The win32k driver has a callback system to do the exact opposite. The `KeUsermodeCallback` function calls a user-mode function from kernel-mode. This function is undocumented and provided by the kernel module in a secure way in order to switch into user-mode properly[4]. The win32k driver uses this functionality for common task such as loading a dll module for event catching or retrieving information. The prototype of this function:

```
NTSTATUS KeUserModeCallback (
    IN ULONG ApiNumber,
    IN PVOID InputBuffer,
    IN ULONG InputLength,
    OUT PVOID *OutputBuffer,
```

```
    IN PULONG OutputLength
    );
```

Microsoft did not make a system to retrieve arbitrary user-mode function addresses from the kernel. Instead, the win32k driver has a set of functions that it needs to call. This list is kept in an undocumented function table in the *Process Environment Block* (PEB) structure for each process. The `ApiNumber` argument refers to an index into this table.

In order to return on user-mode, `KeUserModeCallback` retrieves the user-mode stack address from saved user-mode context stored in a thread's `KTRAP_FRAME` structure. It saves current stack level and uses `ProbeForWrite` to check if there is enough room for the input buffer. The `Inputbuffer` argument is then copied into the user stack and an argument list is created for the function being called. The `KiCallUserMode` function prepares the return in user-mode by saving important information in the kernel stack. This callback system works as a normal syscall exit procedure except than stack level and `eip` register has been changed. The callback start in the `KiUserCallbackDispatcher` function.

```
VOID KiUserCallbackDispatcher(
    IN ULONG ApiNumber,
    IN PVOID InputBuffer,
    IN ULONG InputLength
    );
```

The user-mode function `KiUserCallbackDispatcher` receives an argument list which contains `ApiNumber`, `InputBuffer`, and `InputLength`. It does appropriate function dispatching using the PEB dispatch table. When it is finished the routine invokes interrupt 0x2b to transfer control back to kernel-mode. In turn, the kernel inspects three registers:

- `ecx` contains a user-mode pointer for OutputBuffer
- `edx` is for OutputLength
- `eax` contains return status.

The `KiCallbackReturn` kernel-mode function handles the *0x2B* interrupt and passes important registers as argument for the `NtCallbackReturn` function. Everything is cleaned using saved information within the kernel stack and it transfers to previously called `KeUsermodeCallback` function with proper output argument sets.

The reader should notice that nothing is done to check ouput data. Each kernel function that uses the user-mode callback system is responsible for verifying output data. An attacker can simply hook the `KiUserCallbackDispatcher`

function and filter requests to control output pointer, size and data. This user-mode call can represent an important issue if it was not verified as seriously as system call functions.

4 Discovery and exploitation

The win32k driver was patched by the MS08-025 bulletin [1]. This bulletin did not disclose any details about the issues but it did talk about a vulnerability which allows privilege elevation though invalid kernel checks. This patch increases the overall driver security by adding multiple verifications. In fact, this patch was due to three different reported vulnerabilities. The following sections explain how these vulnerabilities were discovered and exploited.

4.1 DDE Kernel pool overflow

The *Dynamic Data Exchange* (DDE) protocol is a GUI integrated message system [5]. Despite Windows operating system has already many different message mechanisms, this one share data across process by sharing GUI handles and memory section. This feature is quite old but still supported by Microsoft application as Internet explorer [6] and used in application firewalls bypass technique. During author's research on win32k driver, he investigated how the `KeUsermodeCallback` function was used. As described previously, this function does not verify directly output data. This lack of validation is what leads to this vulnerability.

4.1.1 Vulnerability details

The vulnerability exists in the `xxxClientCopyDDEIn1` win32k function. It is not called directly but it is used internally in the kernel when messages are exchanged between processes using the DDE protocol. In this context, the `OutputBuffer` verification is analyzed.

```
In xxxClientCopyDDEIn1 function:

lea     eax, [ebp+OutputLength]
push    eax
lea     eax, [ebp+OutputBuffer]
push    eax
push    8                                ; InputLength
lea     eax, [ebp+InputBuffer]
push    eax
push    32h                              ; ApiNumber
call    ds:__imp__KeUserModeCallback@20
mov     esi, eax                         ; return < 0 (error ?)
```

```
call    _EnterCrit@0
cmp     esi, edi
jl      loc_BF92C6D4

cmp     [ebp+OutputLength], 0Ch          ; Check output length
jnz     loc_BF92C6D4

mov     [ebp+ms_exc.disabled], edi       ; = 0
mov     edx, [ebp+OutputBuffer]
mov     eax, _Win32UserProbeAddress
cmp     edx, eax                         ; Check OutputBuffer address
jb      short loc_BF92C5DC

[...]

loc_BF92C5DC:
mov     ecx, [edx]
loc_BF92C5DE:
mov     [ebp+var_func_return_value], ecx
or      [ebp+ms_exc.disabled], 0FFFFFFFFh
push    2
pop     esi
cmp     ecx, esi                         ; first OutputBuffer ULONG must be 2
jnz     loc_BF92C6D4
xor     ebx, ebx
inc     ebx
mov     [ebp+ms_exc.disabled], ebx       ; = 1
mov     [ebp+ms_exc.disabled], esi       ; = 2
mov     ecx, [edx+8]                     ; OutputBuffer - user mode ptr
cmp     ecx, eax                         ; Win32UserProbeAddress - check user mode ptr
jnb     short loc_BF92C602

[...]

loc_BF92C602:
push    9
pop     ecx
mov     esi, eax
lea     edi, [ebp+copy_output_data]
rep movsd
mov     [ebp+ms_exc.disabled], ebx             ; = 1
push    0
push    'EdsU'
mov     ebx, [ebp+copy_output_data.copy1_size] ; we control this
mov     eax, [ebp+copy_output_data.copy2_size] ; and this
lea     eax, [eax+ebx+24h]                     ; integer overflow right here
push    eax                                    ; NumberOfBytes
call    _HeavyAllocPool@12
mov     [ebp+allocated_buffer], eax
test    eax, eax
jz      loc_BF92C6B6

mov     ecx, [ebp+var_2C]
mov     [ecx], eax                             ; save allocation addr
push    9
pop     ecx
lea     esi, [ebp+copy_output_data]
```

```
mov     edi, eax
rep movsd                                       ; Copy output data
test    ebx, ebx
jz      short loc_BF92C65A

mov     ecx, ebx
mov     esi, [ebp+copy_output_data.copy1_ptr]
lea     edi, [eax+24h]
mov     edx, ecx
shr     ecx, 2
rep movsd                                       ; copy copy1_ptr (with copy1_size)
mov     ecx, edx
and     ecx, 3
rep movsb

loc_BF92C65A:
mov     ecx, [ebp+copy_output_data.copy2_size]
test    ecx, ecx
jz      short loc_BF92C676
mov     esi, [ebp+copy_output_data.copy2_ptr]
lea     edi, [ebx+eax+24h]
mov     edx, ecx
shr     ecx, 2
rep movsd movsd                                 ; copy copy2_ptr (with copy2_size)
mov     ecx, edx
and     ecx, 3
rep movsb
```

The DDE `copydata` buffer contains two different buffers with their respective sizes. These sizes are used to calculate the size of a buffer that is allocated. However, appropriate checks are not made to detect if an integer overflow occurs. An integer overflow exists when an arithmetic operation is done between different integers that would go behind maximum integer value and then create a lower integer[7]. As such, the allocated buffer may be smaller than each buffer size which leads to a kernel pool overflow. The pool is the name used to designated the Windows kernel heap.

4.1.2 Pool overflow exploitation

The key to exploiting this issue is more about how to exploit a kernel pool overflow. Previous work has described the kernel pool system and exploitation[8] [9]. This paper will focus on the exploiting the vulnerability being described.

The kernel pool can be thought of as a heap. Memory is allocated by the `ExAllocatePoolWithTag` function and then freed using the `ExFreePoolWithTag` function. Depending of memory size, a header chunk precedes memory data. Exploiting a pool overflow involves replacing the next chunk header with a crafted version. This header is available though ntoskrnl module symbols as:

```
typedef struct _POOL_HEADER
```

```
{
    union
    {
        struct
        {
            USHORT PreviousSize : 9;
            USHORT PoolIndex : 7;
            USHORT BlockSize : 9;
            USHORT PoolType : 7;
        }
        ULONG32 Ulong1;
    }
    union
    {
        struct _EPROCESS* ProcessBilled;
        ULONG PoolTag;
        struct
        {
            USHORT AllocatorBackTraceIndex;
            USHORT PoolTagHash;
        }
    }
} POOL_HEADER, *POOL_HEADER; // sizeof(POOL_HEADER) == 8
```

Size fields are a multiple of 8 bytes as an allocated block will always be 8 byte aligned[1]. The `PoolIndex` field is not important for our overflow and can be set to 0. The `PoolType` field contains chunk state with multiple possible flags. The busy flag changes between operating system version but free chunk always got the `PoolType` field to zero.

During a pool overflow, the next chunk header is overwritten with malicious values. When the allocated block is freed, the `ExFreePoolWithTag` function will look at the next block type. If the next block is free it is coalesced by unlinking and merging it with current block. The `LIST_ENTRY` structure links blocks together and is adjacent to the `POOL_HEADER` structure if current chunk is free. The unlinking procedure is exactly the same as the behavior of the user-mode heap except that no safe unlinking check is done. This procedure is repeated for previous block. Many papers already explained unlinking exploitation which allows writing 4 bytes to a controlled address. However, this attack breaks a pool's internal linked list and exploitation must take this into consideration. As such, it is necessary to restore the pool's list integrity to prevent the system from crashing.

There are a number of different addresses that may be overwritten such as directly modifying code or overwriting the contents of a function pointer. In local kernel exploitation, the target address should be uncommonly unused by the kernel to prevent operating system instability. In his paper, Rubén Santamarta used a function pointer accessible though an exported kernel variable named

[1]Windows 2000 pool architecture is different. Memory blocks are aligned on 16 bytes and flags type is a simple UCHAR (no bitfields).

HalDispatchTable[10]. This function pointer is used by KeQueryIntervalProfile which is called by the system call NtQueryIntervalProfile. Overwriting the function pointer at HalDispatchTable+4 does not break system behavior as this function is unsupported[2] in default configuration. For our exploitation, this is the best choice as it is easy to launch and target.

The exploitation code for this this particular vulnerability should produce this fake chunk:

```
Fake next pool chunk header for Windows XP / 2003:

PreviousSize = (copy1_size + sizeof(POOL_HEADER)) / 8
PoolIndex    = 0
BlockSize    = (sizeof(POOL_HEADER) + 8) / 8
PoolType     = 0 // Free chunk

Flink = Execute address - 4    // in userland - call +4 address
Blink = HalDispatchTable + 4   // in kernelland

Modification for Windows 2000 support:

PreviousSize = (copy1_size + sizeof(POOL_HEADER)) / 16
BlockSize    = (sizeof(POOL_HEADER) + 15) / 16
```

The Flink field points on a user-mode address less 4 that will be called from the kernel address space once the Blink function pointer would be replaced. When called by the kernel, the user-mode address will execute at ring0 and is able to modify operating system behavior.

In this specific vulnerability, to avoid a crash and control copied data in target memory buffer, copy2_ptr should point to a NO_ACCESS memory page. When the copy occurs, an exception will be raised which will be caught by a try/except block in the function. For this exception, the allocated buffers are freed. Copied memory size would be controlled by copy1_size field and integer overflow will be done by copy2_size field. This configuration allows to overflow only the necessary part.

4.1.3 Delayed free pool overflow on Windows Vista

The allocation pool type in win32k on Windows Vista uses an undocumented DELAY_FREE flag. With this flag, the ExFreePoolWithTag function does not liberate a memory block but instead pushes it into a deferred free list. If the kernel needs more memory or the deferred free list is full it will pop an entry off the list and liberate it through normal means. This can cause problems because the actual free may not occur until many minutes later in a potentially different

[2] A clean privilege escalation code should consider restoring overwritten data.

process context. Due to this problem, both `Flink` and `Blink` pointers must be in the kernel mode address space.

The `HalDispatchTable` overwrite technique can be reused to support this configuration. The `KeQueryIntervalProfile` function disassembly shows how the function pointer is used. This context is always the same across Windows versions.

```
mov     [ebp+var_C], eax
lea     eax, [ebp+arg_0]
push    eax
lea     eax, [ebp+var_C]
push    eax
push    0Ch
push    1
call    off_47503C       ; xHalQuerySystemInformation(x,x,x,x)
```

The first and the second arguments points into user-mode in the NULL page. This page can be allocated using the `NtAllocateVirtualMemory` function with an unaligned address in NULL page. The kernel function will realign this pointer on lower page and allocate this page[3]. In order to exploit this context, a stub of machine code must be found which returns on first argument and where next 4 bytes can be overwritten. This is the case of function epilogues as for `wcslen` function:

```
.text:00463B4C                sub     eax, [ebp+arg_0]
.text:00463B4F                sar     eax, 1
.text:00463B51                dec     eax
.text:00463B52                pop     ebp
.text:00463B53                retn
.text:00463B54                db 0CCh ; alignement padding
.text:00463B55                db 0CCh
.text:00463B56                db 0CCh
.text:00463B57                db 0CCh
.text:00463B58                db 0CCh
```

In this example, the `00463B51h` address fits our needs. The *pop* instruction pass the return address and the *retn* instruction return in 1. The alert reader noticed that the selected address start at *dec* instruction. The unlinking procedure unlinks the next 4 bytes and the `00463B54h` address has 5 padding bytes. Without this padding, overwriting unknown assembly could lead to a crash compromising the exploitation. The location of this target address changes depending on operating system version but this type of context can be found using pattern matching. On Windows Vista, the vulnerability exploitation loops calling the `NtQueryIntervalProfile` function until deferred free occurs and exploitation is successful. This loop is mandatory as pool internal structure must be corrected.

[3]This page is also used in kernel NULL dereference vulnerabilities.

4.2 NtUserfnOUTSTRING kernel overwrite vulnerability

The `NtUserfnOUTSTRING` function is accessible through an internal table used by `NtUserMessageCall` exported function. Functions starting by "*NtUserfn*" can be called with SendMessage function exported by *user32.dll* module. For this function the `WM_GETTEXT` window message is necessary. Notice that in some cases a direct call is needed for successful exploitation. Verifications made by SendMessage function are trivial as it is used for different functions but should be considered. The MSDN website describes `SendMessage` utilization [3].

4.2.1 Evading ProbeForWrite function

The `ProbeForWrite` function verifies that an address range resides in the user-mode address space and is writable. If not, it will raise an exception that can be caught by a try / except code block. This function is used by a lot by drivers which deal with user-mode inputs. THe following is the start of the `ProbeForWrite` function assembly:

```
void __stdcall ProbeForWrite(PVOID Address, SIZE_T Length, ULONG Alignment)

mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+Length]
test    eax, eax
jz      short loc_exit                ; Length == 0

[...]

loc_exit:
pop     ebp
retn    0Ch
```

This short assembly dump underlines one way to evade `ProbeForWrite` function. If Length argument is zero, no verification is done on Address argument. It means that Microsoft considers that a zero length input do not require address to point in userland. Microsoft made a blog post on `MS08-025`[12] and why ProbeForWrite was not modified as expected. Microsoft compatibility concern is understandable but at least ProbeForWrite documentation should be updated for this case.

4.2.2 Vulnerability details

This vulnerability touches not only this function but a whole class of string management functions. Some functions make sure that length argument is not

zero before its modification. Others do not even check the length argument. A proof of concept has been made on this vulnerability by Rubén Santamarta [11].

The `NtUserfnOUTSTRING` function vulnerability evades the `ProbeForWrite` function and overwrites 1 or 2 bytes of kernel memory. This function disassembly is below:

```
In NtUserfnOUTSTRING (WM_GETTEXT)

xor     ebx, ebx
inc     ebx
push    ebx                     ; Alignment = 1
and     eax, ecx                ; eax = our size | ecx = 0x7FFFFFFF
push    eax                     ; If our size is 0x80000000 then
                                ; Length is zero (avoid any check)
push    esi                     ; Our kernel address
call    ds:__imp__ProbeForWrite@12
or      [ebp+var_4], 0FFFFFFFFh
mov     eax, [ebp+arg_14]
add     eax, 6
and     eax, 1Fh
push    [ebp+arg_10]
lea     ecx, [ebp+var_24]
push    ecx
push    [ebp+arg_8]
push    [ebp+arg_4]
push    [ebp+arg_0]
mov     ecx, _gpsi
call    dword ptr [ecx+eax*4+0Ch] ; Call appropriate sub function
mov     edi, eax
test    edi, edi
jz      loc_BF86A623              ; Something goes wrong

[...]

loc_BF86A623:
cmp     [ebp+arg_8], eax          ; Submit size was 0 ? (no)
jz      loc_BF86A6D1

[...]

push    [ebp+arg_18]              ; Wide or Multibyte mode
push    esi                       ; Our address
call    _NullTerminateString@8    ; <== 0 byte or short overwriting
```

In this function, a high size (`0x80000000`) can bypass `ProbeForWrite` function verification. After this verification, it calls a function based on win32k internal function pointer table. This function depends of the calling context. If it is in the same thread that submitted handle it will go directly on retrieval function, otherwise it can be cached by another function waiting for proprietary thread handling this request. This assembly sample highlights null byte overwriting if other functions failed. The null byte assures that a valid string is returned. This is not the only way to overwrite memory. By using an edit box, we could

overwrite kernel memory with a custom string but the first way fit the need.

The exploitation is trivial and will not be detailed in this part. The first vulnerability already exposed a target address and the way to allocate the NULL page which were used to demonstrate this vulnerability.

4.3 LoadMenu handle table corruption

The win32k driver implements its own handle mechanism. This system shares a handle table between user-mode and kernel-mode. This table is mapped into the user mode address space as read-only and is modified in kernel mode address space. The *MS07-017* bulletin found by Cesar Cerrudo during *Month of Kernel Bugs* (MOKB) [13] describes this table and how its modification could permit kernel code execution. This chapter addresses another vulnerability based on GDI handle shared table entry misuse.

4.3.1 Handle table

In the GUI architecture, an handle contains different information as an index in the shared handle table and the object type. The handle table is an array of the undocumented `HANDLE_TABLE_ENTRY` structure.

```
typedef struct _HANDLE_TABLE_ENTRY
{
    union
    {
        PVOID pKernelObject;
        ULONG NextFreeEntryIndex; // Used on free state
    };
    WORD  ProcessID;
    WORD  nCount;
    WORD  nHandleUpper;
    BYTE  nType;
    BYTE  nFlag;
    PVOID pUserInfo;
} HANDLE_TABLE_ENTRY; // sizeof(HANDLE_TABLE_ENTRY) == 12
```

The `nType` field defines the table entry type. A free entry has the type zero and `nFlag` field which defines if it is destroyed or currently in destroy procedure. Normal handle verification routines check this value before getting `pKernelInfo` field which points to the associated kernel handle. In a free entry, the `NextFreeEntryIndex` field contains the next free entry index which is not a pointer but a simple unsigned long value.

The GUI object structure depends of object type but starts with the same structure which contains corresponding index in the shared handle table. This

architecture lies on both elements. It switches between each table entry and kernel object depending of needs. A security issue exists if the handle table is not used as it should.

4.3.2 Vulnerability details

The vulnerability itself exists in win32k's `xxxClientLoadMenu` function which does not correctly validate a handle index. This function is called by the `GetSystemMenu` function and returns to user-mode using the `KeUsermodeCallback` function to retrieve a handle index. The following assembly shows how this value is used.

```
and     eax, 0FFFFh         ; eax is controlled
lea     eax, [eax+eax*2]    ; index * 3
mov     ecx, gSharedTable
mov     edi, [ecx+eax*4]    ; base + (index * 12)
```

This assembly sample uses an unchecked handle index and return `pKernelObject` field value of target entry. This pointer is returned by the `xxxClientLoadMenu` function. Proper verification are not made which permit deleted handle manipulation. A deleted handle has its `NextFreeEntryIndex` field set between 0x1 and 0x3FFF. The return value will be in first memory pages.

A system menu is linked to a window object. This window object is designated by an handle passed as an argument of the `GetSystemMenu` function. The `spmenuSys` field of the window object is set with the returned value of the `xxxClientLoadMenu` function. In this specific context, the `spmenuSys` value is hardly predictable inside the NULL page. During thread exit, the Window liberation will look at `spmenuSys` object and using its index in the shared table, toggle `nFlag` field state to destroyed and `nType` as free. In the case the NULL page is filled with zero value, it will destroy the first entry in the GDI shared handle table.

Exploitation is achieved by reusing vulnerable functions once the first entry has been destroyed. The `GetSystemMenu` function locks and unlocks the GDI shared handle table entry linked with kernel object returned by the `xxxClientLoadMenu` function. If the entry flag is destroyed the unlock function calls the type destroy callback. For the first entry, the flag has been set to destroyed. There is no callback for this type as it is not supposed to be unlocked. The unlock function will call zero which allows kernel code execution. This specific handle management architecture stay undocumented. The purpose of liberation callback inside the thread unlocking procedure is unusual.

Exploitation steps:

- Allocate NULL address

- Exploitation loop - second iteration trigger call zero:
  - Create a dialog
  - Set NULL page data to zero
  - Set a relative jmp at zero address
  - Create a menu graphic handle (or another type).
  - Destroy this menu handle
  - Call GetSystemMenu
  - Intercept user callback and return destroyed menu handle index (mask 0x3fff of the handle)
  - Exit this thread - set zero handle entry as free and destroyed.

There are multiple ways to exploit this vulnerability. The author truly believes that exploiting the locking procedure could be used on handle leak vulnerabilities as it was for this vulnerability. Indeed this vulnerability exploitation stays complex and unusual. This specific context made exploitation even more interesting.

5 GUI architecture protection

Create a safe software is a hard task that is definitely harder than find vulnerabilities. This work is even harder when it concerns old components which must respect compatibility rules. This article does not blame Microsoft for those vulnerabilities; it presents global issues on Windows architecture. In Windows Vista, Microsoft starts securing its operating system environment. The Windows Vista base code is definitely safer than it was. Some kernel components as the win32k driver are not safe enough and should be considered as a priority in local operating system security.

The GUI architecture does not respect security basics. Starting from scratch would certainly be a good option if it was possible. The global organization of this driver make security audits a mess. In the other hand, the Windows API shows it responses developer needs. There is a big abstraction layer between userland API and kernel functions. It can be use to rebuild the win32k driver without breaking compatibility. The API must follow user needs and be as easy as it can be. There is no reason that kernel driver exported function could not be changed in a secure way. It represents an enormous work which would be achieved only across operating system version. Nevertheless this is necessary. This modification could also increase performance by reducing unneeded context switching. There is no clever reason going in the kernel to ask userland a value that will be returned to userland. The user-mode callback system does not fit in a consistent GUI architecture.

Local exploitation techniques also highlight unsecure components as kernel pool and how overwriting some function pointers allow kernel code execution. In the past, the userland has been hardened as exploitation was too easy and third parties software could permit compromising a computer. The kernel performance is critical and adds verification routines and security measure could break this advantage. The solution should be in operating system evolution which does not restrict user experience. The hardware improvement does not forgive that modern operating system requires more resources than before.

Software development follows fastest way except when a specific result is expected. A company does not search the better way but something that cost less for almost the same result. Microsoft did not choose readiness by starting *Security Development Lifecycle* (SDL) [14] and should continue in this way.

6 Conclusion

The Windows kernel components have unequal security verification level. The main kernel module (*ntoskrnl.exe*) respects a standard verification dealing with userland data. The win32k driver does not follow the same rules which creates messy verification algorithms. This driver has an important interaction with userland by different mechanism from usual syscall to userland callback system. This architecture increase attack surface. The vulnerable parts do not concern usual vulnerabilities but also internal mechanism as GUI handle system.

Chapter 4 exposed vulnerabilities discovery and exploitation. Local exploitation has many different attack vectors. Nowadays, the exploitation is fast and sure, it works at any attempts. The kernel exploitation is possible though different techniques.

The win32k driver was not built with a secure design and now it becomes so huge, with so many compatibility restrictions, that every release just implements new features without changing anything else. Windows Vista introduces many modifications but most of them are just automatic integer overflow checks. It will solve many unknown issues but interaction between user-mode and kernel-mode is hardly predictable. Vulnerabilities are not always a matter of proper checks but also system interaction and custom context.

Implementing usual userland protections is not a good solution as kernel exploitation is larger than overflows. The win32k driver could change by using userland abstract layer in order to keep compatibility. This choice is not the easier as it asks more time and work. The patch evoked in this paper ameliorates a little bit win32k security as it goes deeper than reported vulnerabilities. However the Windows Vista version of the win32k driver was concerned by two vulnerabilities even if it was already more secure. Minor modifications do not solve security issues. The overall kernel security has been discussed on different

paper about vulnerabilities but also rootkits. Everyone agree that operating systems must evolve. Windows Seven could introduce a new right architecture which secure critical component or just improve win32k driver security.

References

[1] Microsoft Corporation. *Microsoft Security Bulletin MS08-025*
http://www.microsoft.com/technet/security/Bulletin/MS08-025.mspx

[2] Microsoft Corporation. *Windows User Interface.*
http://msdn.microsoft.com/en-us/library/ms632587(VS.85).aspx

[3] Microsoft Corporation. *SendMessage function.*
http://msdn.microsoft.com/en-us/library/ms644950.aspx

[4] ivanlef0u. *You failed (blog entry about KeUsermodeCallback function in French).*
http://www.ivanlef0u.tuxfamily.org/?p=68

[5] Microsoft Corporation. *About Dynamic Data Exchange.*
http://msdn.microsoft.com/en-us/library/ms648774.aspx

[6] Microsoft Corporation. *DDE Support in Internet Explorer Versions (still supported in ie7).*
http://support.microsoft.com/kb/160957

[7] Wikipedia. *Integer overflow.*
http://en.wikipedia.org/wiki/Integer_overflow

[8] mxatone and ivanlef0u. *Stealth hooking : Another way to subvert the Windows kernel.*
http://www.phrack.org/issues.html?issue=65&id=4#article

[9] Kostya Kortchinsky. *Kernel pool exploitation (Syscan Hong Kong 2008).*
http://www.syscan.org/hk/indexhk.html

[10] Rubén Santamarta. *Exploiting common flaws in drivers.*
http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=fileinfo&id=51

[11] Rubén Santamarta. *Exploit for win32k!ntUserFnOUTSTRING (MS08-25/n).*
http://www.reversemode.com/index.php?option=com_content&task=view&id=50&Itemid=1

[12] Microsoft Corporation. *MS08-025: Win32k vulnerabilities.*
http://blogs.technet.com/swi/archive/2008/04/09/ms08-025-win32k-vulnerabilities.aspx

[13] Cesar Cerrudo. *Microsoft Windows kernel GDI local privilege escalation.* http://projects.info-pull.com/mokb/MOKB-06-11-2006.html

[14] Microsoft Corporation. Steve Lipner and Michael Howard. *The Trustworthy Computing Security Development Lifecycle* http://msdn.microsoft.com/en-us/library/ms995349.aspx