SkyRecon®

WINDOWS PRIVILEGE ESCALATION THROUGH LPC AND ALPC INTERFACES

Prepared by:	Thomas Garnier Research engineer Skyrecon.com
Date:	2 June 2008

Summary

Introduction

This paper presents SkyRecon Systems research on LPC and ALPC kernel interfaces. Other papers have already been written on LPC, describing how it works and how it can be used to improve local exploitation reliability. Following on the advantage of this body of work, our research goal was to see if any use of these interfaces could lead to a privilege escalation. It also enabled us to discover the ALPC interface, an evolution of the LPC interface introduced in Windows Vista and to try to understand why this part of the kernel has been changed on Windows Vista.

As this paper relies greatly on undocumented code, it will describe the obvious elements under discussion. Only Microsoft can provide clear documentation on their internal components.

Our research has resulted in two Microsoft security bulletins (MS07-066 and MS08-002) which are described in this paper. Even if there is no single way to achieve higher privileges, it highlights the design and security issues that simplify local exploitation.

LPC Interface

LPC (Local Procedure Call) is a Microsoft Windows kernel component used for communication between processes (IPC). This undocumented interface is used in the background of the known Windows API. Most system components use the LPC interface to communicate with lower-level security programs. A list of services is available only through this communication channel.

For example:

- CSRSS manages threads and processes using an LPC port.
- RPC interface uses LPC transportation for local communication.
- OLE communication is based on LPC generated ports.

This communication component is very important to Windows architecture. It can be seen as a local socket with rights-level support. As an internal component of the Windows kernel, it is undocumented by Microsoft.

Details on the LPC interface

Even thought the LPC interface is undocumented, most functions have been described in various articles [1] [2] [7]. This article does not focus on the details for using the LPC interface, but presents the basics needed for understanding the LPC interface design and common issues.

The communication system uses a named kernel object called a port. This object cannot be opened with classical functions like *CreateFile*. The object path is specified during port creation and used for client connection. Port access can be restrained using a security descriptor with the named object.

Following are some functions used on LPC. These functions are in the kernel but ntdll.dll exports syscall wrappers:

```
NTSTATUS NTAPI NtCreatePort(
                                  OUT PHANDLE PortHandle,
                                  IN POBJECT_ATTRIBUTES ObjectAttributes,
                                  IN ULONG MaxConnectionInfoLength,
                                  IN ULONG MaxMsgSize,
                                  IN ULONG Reserved OPTIONAL
                                  );
```

This creates an LPC port and sets up its handle on *PortHandle* variable. The *ObjectAttributes* argument contains the port object path and properties.

All interactions between the client and server use a message structure. During the connection procedure a message is sent to the server. It sees this message as any other except that the message type field specifies that it is a client connection request. This design implementation allows servers to treat all requests on a single thread. Even if it decreases answer time on a critical LPC port, a classical one connection per thread is still possible.

LPC cannot be understood without looking at message structure and types.

Following are the available message types:

```
typedef enum _LPC_MSG_TYPE {
    LPC_NEW_MSG,
    LPC_REQUEST,
    LPC_REPLY,
    LPC_DATAGRAM,
    LPC_LOST_REPLY,
    LPC_PORT_CLOSED,
    LPC_CLIENT_DIED,
    LPC_EXCEPTION,
    LPC_DEBUG_EVENT,
    LPC_ERROR_EVENT,
    LPC_CONN_REQ,
} LPC_MSG_TYPE;
```

Following is the message structure:

```
typedef struct _LPC_MESSAGE {
    USHORT          DataSize;
    USHORT          TotalSize;
    LPC_MSG_TYPE    MsgType;
    USHORT          VirtRangOff;
    CLIENT_ID       ClientId;
    ULONG           Mid;
    ULONG           CallbackId;
} LPC_MESSAGE, *PLPC_MESSAGE;
```

In this structure, only the first three fields are used. The *ClientId* field indicates the caller processed and threaded; this information is now filled by the kernel function. This field was responsible for a reported spoofing bug. When a user sends a normal message, custom data is stored next to the LPC_MESSAGE structure and data size specified in *DataSize* field. Custom data depends on server request format as data on sockets. The *TotalSize* field is the sum of *DataSize* and LPC_MESSAGE structure sizes (0x18). The *MsgType* field is usually LPC_NEW_MSG, LPC_REQUEST or LPC_REPLY. Depending of the function, wrong message types are corrected or return an error.

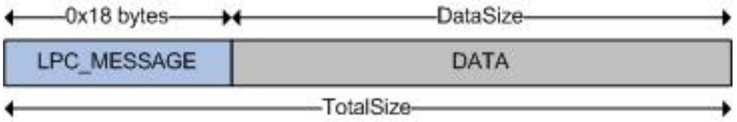


Figure 1 - Message buffer organization

Following is the LPC connection function:

```
NTSTATUS NTAPI          NtConnectPort(
                                OUT PHANDLE PortHandle,
                                IN PUNICODE_STRING PortName,
                                IN PSECURITY_QUALITY_OF_SERVICE QoS OPTIONAL,
                                IN OUT PPORT_SECTION_CLIENT ClientShared OPTIONAL,
                                OUT PPORT_SECTION_SERVER ServerShared OPTIONAL,
                                OUT PULONG MaxMsgSize OPTIONAL,
                                IN OUT PVOID ConnectData OPTIONAL,
                                IN OUT PULONG ConnectDataLength OPTIONAL
                                );
```

This function connects to a submitted port name and retrieves a client port handle. The LPC supports quality of service (*QoS*) despite never being used. The *ClientShared* argument was outlined by Cesar Cerrudo [3] because it allows the mapping of a section on client and server process. This feature returns remote mapped base address and then makes local exploitation easier. Exploiting this advantage is examined more in details later. *ServerShared* argument is the same feature but for mapping on client process requested by the server. *ConnectData* and *ConnectDataLength* arguments specify connection information. ThemeApiPort a LPC port on Windows XP used for theme management waits for particular connection data information. Most of the time, the values needed are static and can be easily found by looking at the assembly code.

Once the server receives a connection request, it can discard it depending on provided information. Some ports are reserved for specific threads, processes, or privileges and discard other messages. LPC allows impersonation of the client user; it is a fast way to check for caller rights. In the LPC interface, there are many different functions for sending and receiving messages: *NtRequestPort*, *NtReplyPort*, *NtReplyWaitReceivePort*, *NtReplyWaitReplyPort* and *NtRequestWaitReplyPort*. As seen in the function names, a message can be sent “wait for a reply” or “ do not wait for a reply”.

Past LPC vulnerabilities

Many vulnerabilities have been found in both LPC kernel interface and userland message management. First, the vulnerabilities allow a total redirection or management of messages. LPC kernel functions blindly trust input information. LPC message structure is somewhat bizarre as it maintains two different sizes. This creates confusion in userland interface but also in kernel message treatment.

Following is a list of some of the reported vulnerabilities:

MS00-003 - Spoofed LPC Port Request (Impersonate a privileged user)
MS00-070 - Multiple LPC and LPC Ports Vulnerabilities (Privilege escalation and message leaking)
MS03-031 - Cumulative Patch for Microsoft SQL Server (Privilege escalation)
MS04-044 - Vulnerabilities in Windows Kernel and LSASS (Privilege escalation)
MS07-029 - Windows DNS RPC Interface (Remote and local privilege escalation)

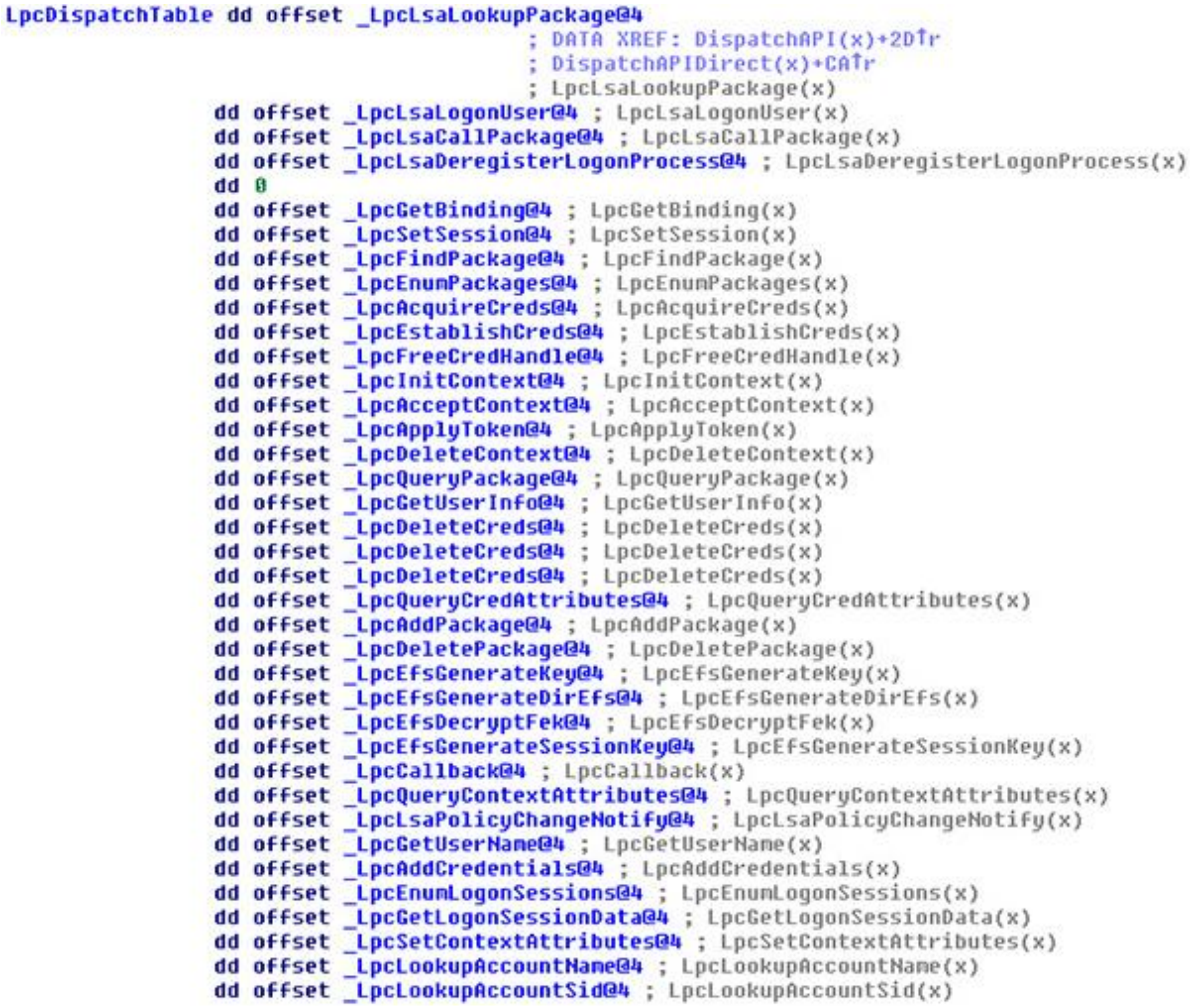
LSASS local privilege escalation – MS08-002

Vulnerable context

During our research on the LPC interface, we looked at many different interfaces to see how they handle requests. We questioned whether or not certain interfaces should be restrained even with Administrator privileges. An LPC interface should be treated like a socket and input should not be trusted. We were unable to find any LPC port that totally trusts input values. Without trusting input, it is not safe to reuse a controlled buffer. An important vulnerability has thus been found as a message buffer was improperly used to store state data.

LSASS (Local Security Authority Subsystem Service) supplies services for local and domain users. The lsasrv.dll creates a public LPC port (*LsaAuthenticationPort*). This port contains a dispatch table that redirects message requests to appropriate functions.

Function names describe perfectly the functionalities:



```
LpcDispatchTable dd offset _LpcLsaLookupPackage@4
                                        ; DATA XREF: DispatchAPI(x)+2D↑r
                                        ; DispatchAPIDirect(x)+CA↑r
                                        ; LpcLsaLookupPackage(x)
                dd offset _LpcLsaLogonUser@4 ; LpcLsaLogonUser(x)
                dd offset _LpcLsaCallPackage@4 ; LpcLsaCallPackage(x)
                dd offset _LpcLsaDeregisterLogonProcess@4 ; LpcLsaDeregisterLogonProcess(x)
                dd 0
                dd offset _LpcGetBinding@4 ; LpcGetBinding(x)
                dd offset _LpcSetSession@4 ; LpcSetSession(x)
                dd offset _LpcFindPackage@4 ; LpcFindPackage(x)
                dd offset _LpcEnumPackages@4 ; LpcEnumPackages(x)
                dd offset _LpcAcquireCreds@4 ; LpcAcquireCreds(x)
                dd offset _LpcEstablishCreds@4 ; LpcEstablishCreds(x)
                dd offset _LpcFreeCredHandle@4 ; LpcFreeCredHandle(x)
                dd offset _LpcInitContext@4 ; LpcInitContext(x)
                dd offset _LpcAcceptContext@4 ; LpcAcceptContext(x)
                dd offset _LpcApplyToken@4 ; LpcApplyToken(x)
                dd offset _LpcDeleteContext@4 ; LpcDeleteContext(x)
                dd offset _LpcQueryPackage@4 ; LpcQueryPackage(x)
                dd offset _LpcGetUserInfo@4 ; LpcGetUserInfo(x)
                dd offset _LpcDeleteCreds@4 ; LpcDeleteCreds(x)
                dd offset _LpcDeleteCreds@4 ; LpcDeleteCreds(x)
                dd offset _LpcDeleteCreds@4 ; LpcDeleteCreds(x)
                dd offset _LpcQueryCredAttributes@4 ; LpcQueryCredAttributes(x)
                dd offset _LpcAddPackage@4 ; LpcAddPackage(x)
                dd offset _LpcDeletePackage@4 ; LpcDeletePackage(x)
                dd offset _LpcEfsGenerateKey@4 ; LpcEfsGenerateKey(x)
                dd offset _LpcEfsGenerateDirEfs@4 ; LpcEfsGenerateDirEfs(x)
                dd offset _LpcEfsDecryptFek@4 ; LpcEfsDecryptFek(x)
                dd offset _LpcEfsGenerateSessionKey@4 ; LpcEfsGenerateSessionKey(x)
                dd offset _LpcCallback@4 ; LpcCallback(x)
                dd offset _LpcQueryContextAttributes@4 ; LpcQueryContextAttributes(x)
                dd offset _LpcLsaPolicyChangeNotify@4 ; LpcLsaPolicyChangeNotify(x)
                dd offset _LpcGetUserName@4 ; LpcGetUserName(x)
                dd offset _LpcAddCredentials@4 ; LpcAddCredentials(x)
                dd offset _LpcEnumLogonSessions@4 ; LpcEnumLogonSessions(x)
                dd offset _LpcGetLogonSessionData@4 ; LpcGetLogonSessionData(x)
                dd offset _LpcSetContextAttributes@4 ; LpcSetContextAttributes(x)
                dd offset _LpcLookupAccountName@4 ; LpcLookupAccountName(x)
                dd offset _LpcLookupAccountSid@4 ; LpcLookupAccountSid(x)
```

Figure 2 – LSASS LPC interface dispatch table

LSASS provides many different functions wrapped by Windows API. Some of these functions are restricted to certain processes or certain rights. This table contains *LpcInitContext* and *LpcAcceptContext* functions which extract data from an LPC message using a capture buffer system. It copies data from a remote process and replaces each remote address with its local copy.

This schema illustrates each step on the LSASS capturing buffer system:

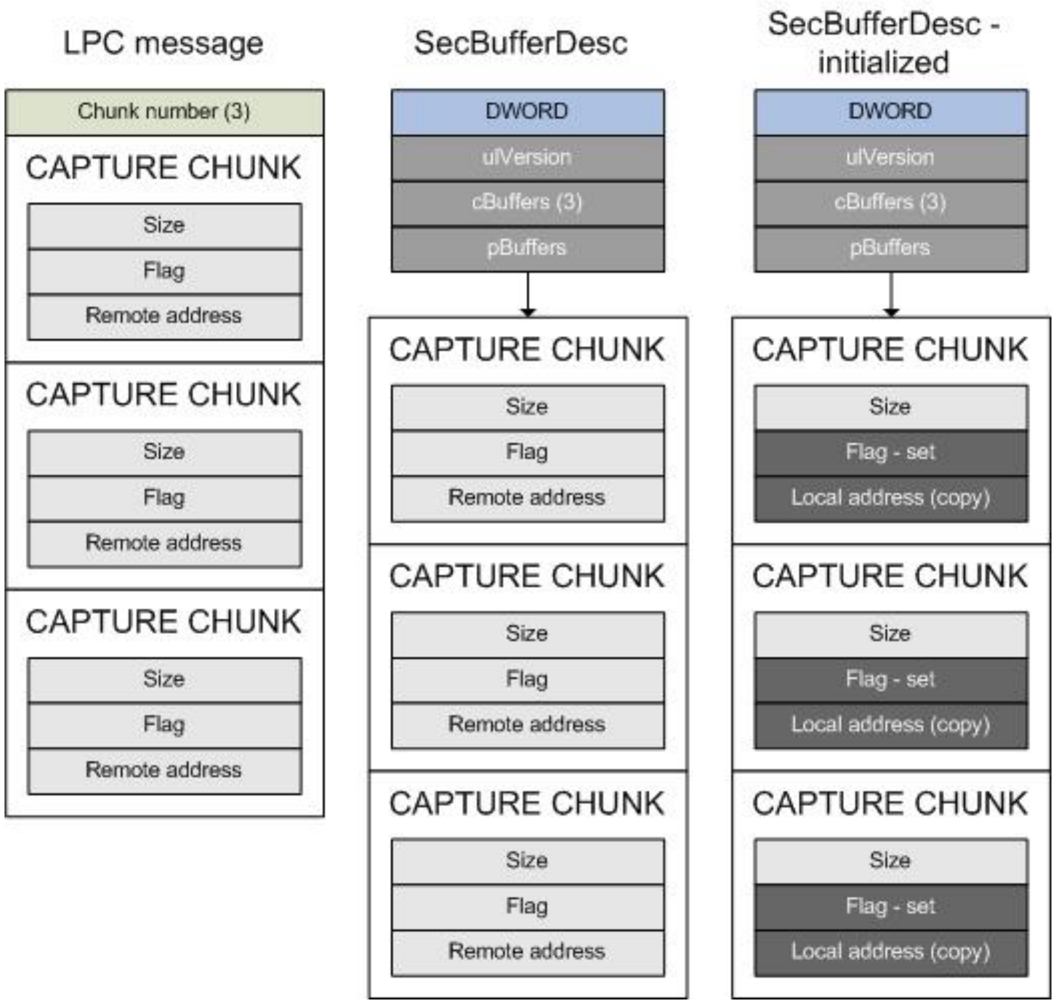


Figure 3 - *SecBufferDesc* transformations

In the first step, LPC message contains all chunks and a count number. Then it is copied into a structure called *SecBufferDesc*. Copied data are then initialized so that the flag field is overwritten with a state (local address is allocated or not). The address field points to a heap buffer which is a copy of the remote address content. The first transformation is made by *LsapCaptureBuffers* function. Depending of a flag argument, it will initialize *SecBufferDesc* or let the caller do it. Initialization is made by the *MapTokenBuffer* function. The *LsapUncaptureBuffers* function liberates allocated data by looking at Flag field state.

An attacker able to control the Flag field value, would also then be able to free any address on LSASS process. This is possible due to a crafted message that discards initialization of a *SecBufferDesc* structure in *LpcInitContext*.

Vulnerable code:

```
lea     ecx, [ebp+var_138]
push    esi                            ; init flag = 0 (FALSE)
mov     [ebp+var_34.pBuffers], ecx
lea     ecx, [ebp+var_1C]
push    ecx
```

```
lea     ecx, [ebp+var_34]
push    ecx                           ; struct _SecBufferDesc *
push    eax
push    ebx
mov     [ebp+var_34.cBuffers], 0Ah
call    LsapCaptureBuffers
cmp     eax, esi                      ; return value < 0?
mov     [ebp+var_8], eax
jl      loc_error                     ; error jump

cmp     [ebx+40h], esi                ; size == 0 (impossible)
jz      short loc_after_init

test    byte ptr [ebx+19h], 1         ; check LPC message flag <===== ISSUE
jnz     short loc_after_init          ; if not zero : discard initialization

push    esi
lea     eax, [ebp+var_34]
push    eax                           ; struct _SecBufferDesc *
call    MapTokenBuffer                ; init SecBufferDesc chunk array
cmp     eax, esi
jl      loc_error                     ; init failed?

loc_after_init:                       ; from this block, all path lead to
cmp     [ebx+0Ch], esi                ; LsapUncaptureBuffers
jz      loc_7573CFF1
```

This assembly code proves that by using a simple submitted flag in the LPC message, it is possible to avoid *SecBufferDesc* chunk array initialization. Any address can be freed by submitting a custom address with a flag that indicates it is locally allocated. Note that this vulnerability does not exist in Windows Vista and Windows 2008 operating systems. It seems that during *Security Development Lifecycle* (SDL) code review, Microsoft tightened this part of the code. Our exploitation technique targets Windows XP SP2 (DEP activated).

Custom free exploitation

RtlFreeHeap function control is an uncommon vulnerability. The classical heap vulnerabilities scenario is an overflow which replaces chunk structure. Many papers exist on that topic and so known exploitation techniques are not presented here; only our modification is explained. In order to completely understand the exploitation technique, please read the lookaside table exploitation technique developed by Matt Conover & Oded Horovitz [4]. Kostya Kortchinsky also wrote a good article on that subject [5]. This exploitation technique also relies on an LPC feature that allows section mapping on remote process [3].

During the *RtlFreeHeap* procedure, the allocated chunk coalesces with previous and next free chunks. Once it is done, depending on heap flags and status, it is pushed onto a lookaside list entry or a freelist table entry. The lookaside table is supported by most heap utilization; this is the case with *lsasrv.dll* custom heap. If the lookaside table entry for the allocated chunk size contains less than 3 entries, it will go into this table. If a buffer is controlled once it is free and in lookaside, the lookaside list can be corrupted by modifying the 4 first bytes (Flink pointer) which points to the next lookaside chunk in the list.

By corrupting the lookaside heap, the allocation return value can be redirected and a part of memory overwritten. Lookaside heap corruption from a target *RtlFreeHeap* should respect these steps:

- Flood allocation without freeing for the target size (empty lookaside table entry)
- Free a fake chunk of the specified size
- Modify the next lookaside single linked list (replace Flink on the free buffer)
- Allocate the target size without freeing it (push the next lookaside entry in the table)
- Allocate the target size again. The *RtlAllocateHeap* function will return the target address

The LSASS capture buffer system is perfect for this, as the allocated size can be chosen remotely and data copied into it. Note that if a custom flag on a captured chunk is submitted, the previous allocated buffer can be discarded without being freed. In some flag values, an error is reported and the function forgets the allocated buffer liberation.

So which address is freed can be controlled but the process heap chunk addresses are unknown to the client process. If a fake heap chunk is created, this *RtlFreeHeap* exploitation technique can be applied. The hard part is that the buffer must be controlled in the remote process. Using the LPC mapping feature, this is just a formality. A shared section can be mapped on the LPC server in order to get remote address mapping.

Windows XP SP2 introduces a random cookie in the heap chunk header. This is made to block the freeing of an overflowed or unknown chunk. The cookie has a 1 byte value (with 256 possibilities) that consists of the chunk address and static value stored in the heap. The cookie verification algorithm is:

((ChunkAddr >> 3) ^ (ChunkCookie) ^ (HeapCookie)) == 0

In the current context, it is really easy to bruteforce this cookie because the address does not change with each attempt. In the worst case, it would need 256 tries.

Following are samples of fake chunk states during bruteforce:

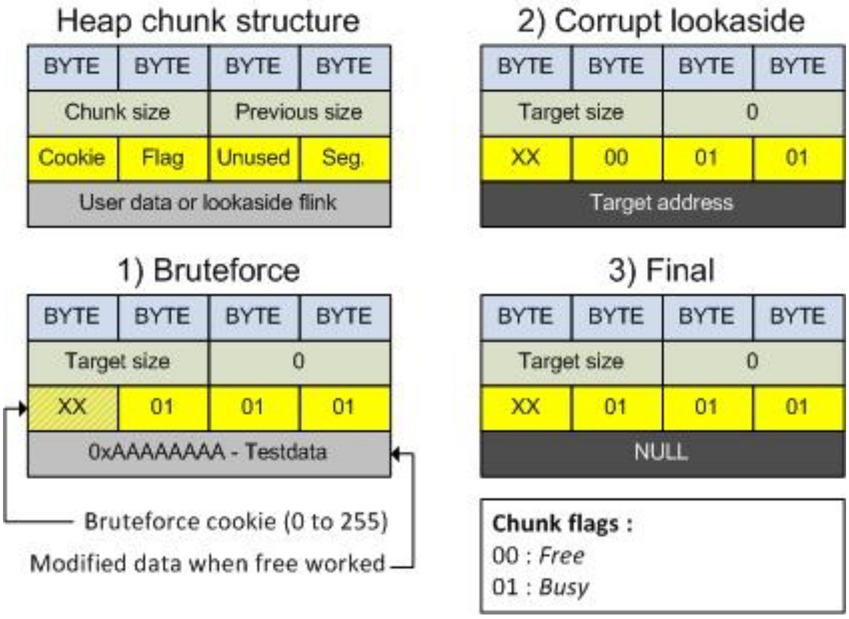


Figure 4 - Fake chunk states

By looking at 0xAAAAAAAA data, we can see if the buffer was pushed into a lookaside list entry. Once this is done, the next lookaside address can be replaced. The address just added to the lookaside list table entry is then the first. In two allocations of target size, the target address is returned by the *RtlAllocateHeap* function. The final context is exactly the same as a normal lookaside exploitation case.

Now, anything in process memory space can be overwritten. That Windows XP SP2 introduces the DEP (Data Execution Prevention) protection must be taken into consideration. This protection prevents execution of memory pages which do not have execution flags [9] on supported hardware. It was created to block buffer overflow attacks. On Windows XP SP2, the LSASS process has DEP activated by default. The shared section for the fake heap chunk can only be mapped with read and write protection flags. Thus we cannot just jump into this section with DEP enabled. Skape and Skywing in “Bypassing Windows Hardware-enforced DEP” [6] demonstrate that DEP protection can be disabled by directly jumping into, within an appropriate context, the *LdrpCheckNXCompatibility* function (in ntdll). It is harder to do this from a heap context, but is still possible as will be described.

In Figure 1, an entry of the LSASS LPC dispatch table is unset. It represents an unsupported request. The LPC dispatch table management function checks the function pointer against the NULL value before it executes it. This is perfect for redirecting the execution path. The LPC dispatch table is overwritten with the same values but replaced the empty entry. With a custom request, control flow is redirected to deactivate DEP protection then jump into the mapped shellcode. Without DEP, it is simply a matter of calling the mapped section.

The context during LPC dispatch call is:

- First argument and EDI register point to the allocated LPC_MESSAGE.
- 0x18 first bytes of this buffer (LPC_MESSAGE structure size) are not fully controlled.

- Execution path must be redirected using *ntdll.dll* module (also needed to look for *LdrpCheckNXCompatibility* function).

Context register can change between module versions (service pack, language pack). This test has been made by looking for the LPC dispatch table using pattern matching. This approach worked on all tested Windows XP SP2 versions.

Getting execution path redirection:

```
In RtlInsertElementGenericTableFull (call from dispatch table overwritted entry):

mov     esi, [ebp+arg_0]     ; our buffer address
lea     eax, [edi+18h]       ; we control data at eax address
push    eax
push    esi
call    dword ptr [esi+1Ch] ; controlled call

In RtlDeleteElementGenericTableAvl:

call    dword ptr [esi+30h] ; add a new call layer

In __chkstk:

xchg    eax, esp             ; eax = &[buffer+0x18]
mov     eax, [eax]           ; return addr (after call [esi+30h])
push    eax                  ; like a jmp eax
retn

In RtlDeleteElementGenericTableAvl (return):

mov     al, 1                ; needed for DEP desactivation
pop     edi                  ; pop [buffer+0x18]
pop     esi                  ; pop [buffer+0x1C]
pop     ebp                  ; pop [buffer+0x20] ==> set EBP value
retn    8                    ; return on [buffer+0x24]

In LdrpCheckNXCompatibility:

cmp     al, 1                ; this check is needed for desactivation
push    2
pop     esi
jz      loc_7C94FEBA

loc_7C94FEBA:
mov     [ebp+var_4], esi
jmp     loc_7C92D403

loc_7C92D403:
cmp     [ebp+var_4], 0       ; is not zero
jnz     loc_7C945D6D

loc_7C945D6D:
push    4
lea     eax, [ebp+var_4]
push    eax
push    22h
push    0FFFFFFFFh
call    _ZwSetInformationProcess@16 ; desactivate DEP protection
jmp     loc_7C92D441

loc_7C92D441:
pop     esi
leave                        ; ESP = EBP (we redirect stack to our mapped section)
retn    4                    ; return to LPC mapped section
```

This assembly samples show execution path that redirects the stack and disables DEP before returning to the shellcode address. DEP evasion is always hard when the stack is not controlled. Getting control is still possible and without relying on more than a single module. The *ntdll.dll* module is used because it is needed to turn off DEP protection. This exploitation technique should be stable as neither the heap free list nor any internal structures are broken. Exploitation could be delayed if the lookaside entry is not empty. Using a double free on the fake chunk, the lookaside entry can be assured (infinite looping linked list) but someone could also allocate the target size before or after. A better way to avoid any issue is to choose an uncommon size and exploits the vulnerability as fast as possible.

Access to the LSASS process from a guest account has important consequences. An attacker could retrieve SYSTEM account privileges and access hidden passwords stored on the registry. He could also filter other LPC requests and then access logon requests. It could use the private LPC channel between LSASS and the kernel and certainly gain access into it without loading a driver. Having now seen how to exploit this vulnerability, it is time to let think about how to protect Windows against those attacks.

Protecting LPC interface against privilege escalation

Based on this vulnerability, it is clear that exploitation is a lot easier using a mapped section. After conducting some tests on different versions of Windows system, we did not find public LPC interface server or client using this feature. The kernel uses it on different connections within private LPC like "*\SeLsaCommandPort*" or "*\XactSrvLpcPort*" ports. On the other hand, it has been described and documented for exploitation purposes. The use of this feature can be restricted depending on the right level between client and server. It has to be tested on different configuration to assure it does not block any communication. Another approach would be a blacklist of LPC ports but this could create issues in the future if the interface is redesigned in another operating system version.

This exploitation is also possible using a documented DEP deactivation technique [6]. This protection mechanism is too easy to stop by the process itself. The existing code which allows DEP deactivation was made for compatibility with modules that do not support DEP protection. A solution would be for the compatibility check and protection deactivation to be made directly in the kernel. Recently Microsoft has provided the *SetProcessDEPPolicy* function to disable DEP from a single call with only one argument to 0 [11] [12]. It makes exploitation even easier from a return-to-libc point of view. So a process must be able to disable DEP when it wants, this is a feature. One can understand Microsoft as DEP is not a case by case protection; when something goes wrong the process crash. Microsoft has to support as much configuration as possible. Protection should focus on checking if the DEP deactivation is legitimate and that it is not done by a malicious code.

It is possible to overwrite any part of memory because of Windows heap layout. Microsoft has already improved its heap in Windows Vista. This new heap does not use a lookaside table. It does not mean there is no possible exploitation technique. It just means that exploitation, if possible, could take a lot of time. The Heap chunk header is xored with a random value to avoid modification. A wrong header on some configuration (default on 64 bit platforms, set to on in Vista system components) stops the program. As DEP protection, this heap behavior is secure but too strict.

Crashing a system component that cannot restart and just shutting down the operating systems should not be a final solution. Of course, Windows XP SP2 heap cannot be secured using this algorithm. In this specific case, cookie bruteforce is easy to detect by looking for heap error reporting. Easily guessing the cookie is possible only if the address is not changed. A good protection technique would harden heap verification before any call to *RtlFreeHeap* function or once too many errors have been reported. Exploiting a free directly on the heap is harder. It does not solve double free issues where free buffer can be modified. Allocated chunk addresses returned by *RtlAllocateHeap* could also be filtered. All these solutions can decrease performances if wrongly implemented.

ALPC interface

Windows Vista introduces a new version of LPC called ALPC (Advanced Local Procedure Call). As LPC, the ALPC interface is an undocumented communication feature of the Windows kernel. Previous LPC functions have been kept for compatibility but use internal ALPC functions. Our research on LPC interface was extended to understand how ALPC works internally. A new component in the kernel accessible from any process must hardly be secure especially in Windows Vista security architecture.

This part of the paper shows the work done on the ALPC interface and how we discovered MS07-066.

Details on the ALPC interface

The ALPC interface does not introduce a lot of new features but is a redesign of the whole package. This new version was mainly done for performance. It supports *I/O completion port* [14], a thread resource organization mechanism that ameliorates server fastness. Some userland interfaces were changed in order to improve their reply time and stop dealing with request one by one. The LPC kernel code remained unchanged for so many years that its design could be drastically improved. There were almost no common functions between LPC syscalls. Only global variables were shared across functions even if they were doing approximately the same thing.

This graph shows that a single internal function is used for message send and reply on ALPC interface.

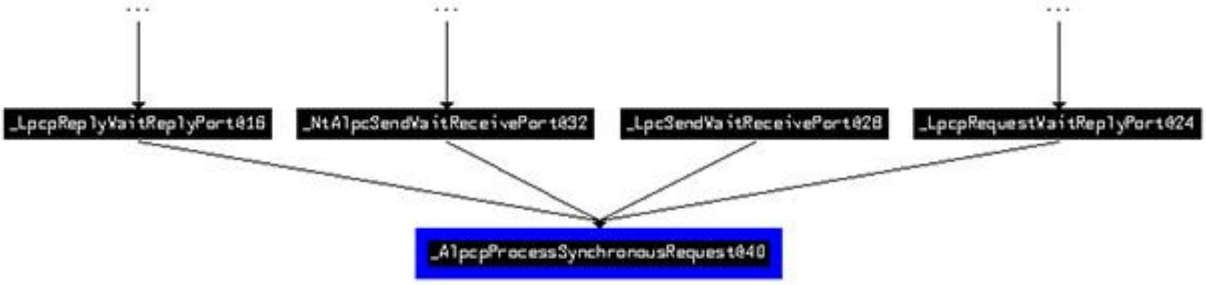
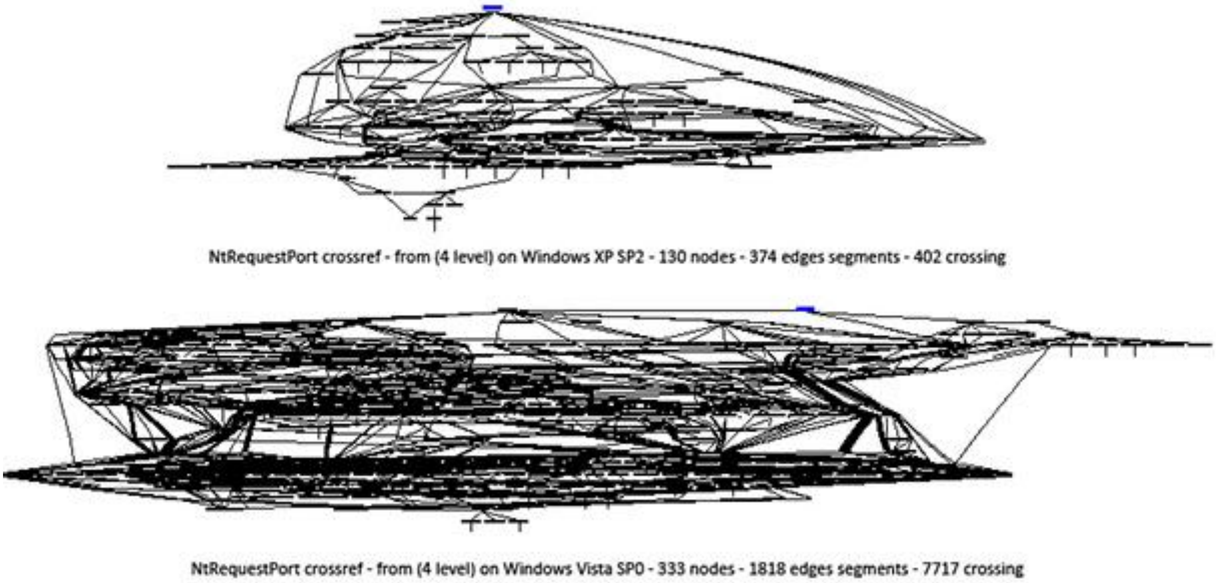

...
...
...
_LpcpReplyWaitReplyPort@16
_NtAlpcSendWaitReceivePort@32
_LpcSendWaitReceivePort@28
_LpcpRequestWaitReplyPort@24
_AlpcpProcessSynchronousRequest@40

Figure 5 - Cross reference from IDA



NtRequestPort crossref - from (4 level) on Windows XP SP2 - 130 nodes - 374 edges segments - 402 crossing

NtRequestPort crossref - from (4 level) on Windows Vista SP0 - 333 nodes - 1818 edges segments - 7717 crossing

Figure 6 - Cross reference (from) on *NtRequestPort* on both XP and Vista from IDA

Internal functions starting with “Lpcp” are used by kernel exported LPC functions. The whole message management algorithm is always the same whether the LPC or ALPC interfaces is used. The LPC interface was constitute by enormous functions. The ALPC functions are smaller and reused in different parts of the interface. It is certainly more complex as it supports both versions but this support was not possible using the previous architecture. This new modularity was very well designed but as it will be shown on next part, it can be exaggerated.

New kernel functions, available through ntdll (syscall wrapper), begin with “NtAlpc”:

- NtAlpcAcceptConnectPort
- NtAlpcCancelMessage
- NtAlpcConnectPort
- NtAlpcCreatePort
- NtAlpcCreatePortSection
- NtAlpcCreateResourceReserve
- NtAlpcCreateSectionView
- NtAlpcCreateSecurityContext
- NtAlpcDeletePortSection
- NtAlpcDeleteResourceReserve
- NtAlpcDeleteSectionView
- NtAlpcDeleteSecurityContext
- NtAlpcDisconnectPort
- NtAlpcImpersonateClientOfPort
- NtAlpcOpenSenderProcess
- NtAlpcOpenSenderThread
- NtAlpcQueryInformation
- NtAlpcQueryInformationMessage
- NtAlpcRevokeSecurityContext

- NtAlpcSendWaitReceivePort
- NtAlpcSetInformation

In this new interface, there is only one function to send and receive messages: *NtAlpcSendWaitReceivePort*. It avoids confusion between all the function used on previous version. Advanced explanation on ALPC interface should be available on next Windows Internal edition [8]. Remember that all previous LPC functions are still available and for most actions the ALPC interface is not needed. The ALPC interface inherits LPC message structure and global mechanism.

Let see some important ALPC functions details:

```
NTSTATUS NTAPI        NtAlpcCreatePort(
                                      OUT PHANDLE PortHandle,
                                      IN POBJECT_ATTRIBUTES ObjectAttributes,
                                      IN OUT PALPC_ INFO PortInformation OPTIONAL
                                      );
```

Port creation does not change that much except that options are grouped into a structure submitted in the last argument. This structure is copied during port creation in the ALPC object and access later. The structure should look like this:

```
typedef struct _ALPC_INFO
{
#define PORT_INFO_LPCMODE                         0x001000  // Behave like an LPC port
#define PORT_INFO_CANIMPERSONATE                  0x010000  // Accept impersonation
#define PORT_INFO_REQUEST_ALLOWED                 0x020000  // Allow messages
#define PORT_INFO_SEMAPHORE                       0x040000  // Synchronization system
#define PORT_INFO_HANDLE_EXPOSE                   0x080000  // Accept handle expose
#define PORT_INFO_PARENT_SYSTEM_PROCESS           0x100000  // Kernel ALPC interface
        ULONG Flags;
        SECURITY_QUALITY_OF_SERVICE PortQos;
        ULONG MaxMessageSize;
        ULONG unknown1;
        CHAR cReserved1[8];
        ULONG MaxViewSize;
        CHAR cReserved2[8];
} ALPC_INFO, *PALPC_INFO;
```

The first field enables some optional features in the ALPC created interface. The *Quality of Service* (QoS) support has been moved on this structure, as *MaxMessageSize* field. The *MaxViewSize* field gives max size that can be mapped in associate process in one message. The new mapping section system will be described later on this part. The *unkown1* field is only replied on demand by ALPC query function, its purpose stay unknown. The *cReserved* fields are copied in the ALPC object but there is neither integrity verification nor access in reversed functions.

The ALPC connection function:

```
NTSTATUS NTAPI          NtAlpcConnectPort(
                                                OUT PHANDLE PortHandle,
                                                IN PUNICODE_STRING PortName,
                                                IN POBJECT_ATTRIBUTES ObjectAttributes,
                                                IN PALPC_ INFO PortInformation OPTIONAL,
                                                IN DWORD ConnectionFlags,
                                                IN PSID pSid OPTIONAL,
                                                IN PLPC_MESSAGE ConnectionMessage OPTIONAL,
                                                IN OUT PULONG ConnectMessageSize OPTIONAL,
                                                IN PVOID InMessageBuffer OPTIONAL,
                                                IN PVOID OutMessageBuffer OPTIONAL,
                                                IN PLARGE_INTEGER Timeout OPTIONAL
                                                );
```

The connection function is a lot larger than before and contains questionable features as an *ObjectAttributes* argument for the client object. It seems only used for its security descriptor as an ALPC client should not export its interface by a named object. Another structure would work too; it is certainly implemented for future feature. The *PortInformation* argument is also present on this connection function and strangely only “in” and not “out”. On LPC interface, the *MaxMessageSize* internal object information was available during connection which seems not possible in the ALPC interface. Most arguments are optional and even *ConnectionFlag* can be discarded using a 0 value. However, default connection is asynchronous and does not wait connection acceptation before returning a handle. If target server did not treat your request before your first message, you will get an error msg. Following some *ConnectionFlag* value:

```
#define ALPC_SYNC_CONNECTION        0x020000        // Synchronous connection request
#define ALPC_USER_WAIT_MODE         0x100000        // Wait in user mode
#define ALPC_WAIT_IS_ALERTABLE      0x200000        // Wait in alertable mode
```

Those flags are shared with the send and receive function:

```
NTSTATUS NTAPI NtAlpcSendWaitReceivePort(
                                                   HANDLE PortHandle,
                                                   DWORD SendFlags,
                                                   PLPC_MESSAGE SendMessage OPTIONAL,
                                                   PVOID InMessageBuffer OPTIONAL,
                                                   PLPC_MESSAGE ReceiveBuffer OPTIONAL,
                                                   PULONG ReceiveBufferSize OPTIONAL,
                                                   PVOID OutMessageBuffer OPTIONAL,
                                                   PLARGE_INTEGER Timeout OPTIONAL
                                                   );
```

Only this function shows architecture amelioration as it replaces 4 LPC functions. The send and reception arguments are optional depending if you want to send a message or receive it or do both things at the same time. The *InMessageBuffer* and *OutMessageBuffer* arguments refer to action request send or receive with a message. For example, the section mapping feature of LPC is still present though those type of buffer. This buffer system integrates structures only if there are needed. It creates a complex system which separates the real structure with its interaction. The *ntdll.dll* module provides some functions which handle this separation.

```
NTSTATUS NTAPI AlpcInitializeMessageAttribute(
                                    ULONG TypeFlag,
                                    PVOID pMessageBuffer OPTIONAL,
                                    ULONG BuffSize,
                                    PULONG RequireSize
                                    );

ULONG NTAPI AlpcGetHeaderSize(ULONG TypeFlag);

PVOID NTAPI AlpcGetMessageAttribute(
                              PVOID pMessageBuffer,
                              ULONG TypeFlag
                              );
```

The message initialization requires that the buffer size correspond to the needed flags. This size is available with *AlpcGetHeadersize* function or when *AlpcInitializeMessageAttribute* function returns a STATUS_BUFFER_TOO_SMALL error. The *AlpcGetMessageAttribute* function uses the request type to return a pointer on dedicated memory space in the allocated buffer. Once the message buffer is ready, the first ULONG must be set with setup types. The initialization function just registers allocated type and check routine will look at which type was informed.

In this paper, we do not see in detail each message buffer type, but look at how section mapping has moved on in this new architecture. The old mechanism was quite simple as it was just two parameters in the connection function. In this new version, we prepare internal ALPC representation though message buffer system.

```
// Message buffer type for section view
#define ALPC_MESSAGE_FLAG_VIEW 0x40000000

// Message buffer structure for section view
typedef struct _ALPC_MESSAGE_VIEW {
        ULONG  Flag;
        HANDLE  AlpcSectionHandle;
        PVOID  MapBase;
        SIZE_T  MapSize;
} ALPC_MESSAGE_VIEW, *PALPC_MESSAGE_VIEW;

// With this flag, next function creates its own section (SectionHandle argument must be zero)
#define ALPC_SECTION_NOSECTIONHANDLE 0x40000

NTSTATUS NTAPI NtAlpcCreatePortSection(
                                        HANDLE PortHandle,
                                        ULONG AlpcSectionFlag,
                                        HANDLE SectionHandle OPTIONAL,
                                        ULONG SectionSize,
                                        PHANDLE AlpcSectionHandle,
                                        PULONG ResSize
                                        );

NTSTATUS NTAPI NtAlpcCreateSectionView(
                                        HANDLE PortHandle,
                                        ULONG FlagUnusedMustbeZero,
                                        PALPC_MESSAGE_VIEW pMessageBuffer
                                        );
```

The *NtAlpcCreatePortSection* function adds or creates a section handle in its internal ALPC handle mechanism. The generated handle is set in *AlpcSectionHandle* argument. This handle is an internal ALPC representation that must be set in *AlpcSectionHandle* field of the ALPC_MESSAGE_VIEW structure in the target message buffer. The last step in message buffer preparation is calling the *NtAlpcCreateSectionView* function with the message buffer. It will fill appropriate fields in the view structure. The remote mapping feature cannot be used as before because remote process must submit a receive message buffer with ALPC_MESSAGE_FLAG_VIEW set. The LPC architecture allows deny remote mapping but few bother doing it despite only some private LPC interface used this feature. In ALPC, the remote mapping feature does not reply remote address which limit previous attack vector. The mapping state is not given, if remote mapping is not supported, the client will not be notified. Strangely, the disconnection does not unmap sections and then a spray attack is possible.

The internal mechanism which administrates view section is called ALPC blob. There are many different types of blob: *ConnectionInfo*, *PortSection*, *ResourceReserve*, *SectionView, HandleContext* or *SecurityContext*. They do not rely only on the message buffer but also support custom handles. Next vulnerability is based on the misuse of one of these new elements.

ALPC kernel code execution – MS07-066

Kernel code execution vulnerabilities are not new but surprise by their difference from classical exploitation. In the kernel, a simple mistake can be transformed into a security hole. The vulnerability described here was discovered by looking at message management function.

Vulnerable context

AlpcpLookupMessage is an internal function of the ALPC interface. It takes an ALPC object and a message id and finds the corresponding message in a handle table. This function verifies link between the ALPC object and the target message. When replying, this function identifies the previously exchanged message and returns its kernel representation.

This function starts by looking at message id signedness:

```
mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]              ; <= object
sub     esp, 14h
push    ebx
mov     ebx, [ebp+arg_4]              ; <= messageId
test    ebx, ebx                      ; messageId is not signed ?
push    esi
push    edi
jns     loc_5C0F47                    ; typical message is not signed

test    eax, eax                      ; check object pointer is not NULL
jnz     short loc_5C0E9C

loc_5C0E9C:
mov     ecx, [eax+8]                  ; object internal blob table
test    ecx, ecx
jz      loc_5C0F40

mov     eax, ebx
mov     edi, 7FFFFFFFh
push    offset _AlpcReserveType
and     eax, edi                      ; & 0x7FFFFFFF (discard sign)
add     ecx, 14h
call    @AlpcReferenceBlobByHandle@12 ; (ecx=blobbase, eax=id)
mov     ecx, eax
test    ecx, ecx
jz      short loc_5C0F40              ; not found reference ?

[…]                                   ; Retrieve blob message, use some lock

mov     eax, [ebp+arg_C]
mov     [eax], esi                    ; esi is blob resource reserve message
xor     eax, eax                      ; return STATUS_SUCCESS
pop     edi
pop     esi
pop     ebx
leave
retn    10h
```

A signed handle allows resource reserve blob retrieval. A resource reserve can be registered with the *NtAlpcCreateResourceReserve* function. A resource reserve is linked with a message kept in the kernel for performance improvement. This message is not like any others as it is not used. When a message is created and used by the kernel, it is initialized and so linked to both client and server ALPC port objects. In a resource reserve, the message does not refer to an ALPC server object. The issue arises from a simple mistake in a synchronous reply where the ALPC server object pointer is not checked against the NULL value.

Following is the vulnerable code:

```
lea     eax, [ebp+var_20]       ; will contain kernel message pointer
push    eax
push    [ebp+var_30]
push    [ebp+var_34]            ; <== messageid
push    ebx
call    @AlpcpLookupMessage@16 ; retrieve our kernel message
mov     [ebp+arg_8], eax
test    eax, eax
jl      loc_5C5A6B

mov     esi, [ebp+var_20]       ; take kernel message pointer
lea     edi, [esi+18h]
mov     eax, [edi]
test    al, 40h                 ; check a flag (always pass)
jz      short loc_5C55D9

loc_5C55D9:
test    ax, 100h                ; check another flag (always pass)
jz      loc_5C5668

loc_5C5668:
lea     eax, [esi+38h]
mov     [ebp+var_28], eax
mov     ecx, [eax]
cmp     ecx, ebx                ; check we own this message by looking at client
jz      loc_5C5731              ; object pointer

mov     ecx, [eax]
test    ecx, ecx                ; and is not NULL
jnz     loc_5C5714

mov     eax, [ebp+var_20]
mov     ebx, [eax+3Ch]          ; <=== no NULL check for ALPC server object
mov     esi, [ebx+8]            ; acces violation /!\ (control ESI value)
mov     byte ptr [ebp+arg_8+3], cl
lea     eax, [esi-10h]
mov     [ebp+var_24], eax
push    11h
pop     ecx
mov     edx, eax
xor     eax, eax
lock cmpxchg [edx], ecx         ; temporary DWORD overwrite with 0
test    eax, eax
jz      short loc_5C56AA        ; old value was 0 ?

mov     ecx, edx
call    @ExfAcquirePushLockShared@4 ; made overwritting permanent
```

This code does not properly verify that an ALPC server object is set on the selected message. It will use a NULL pointer as a valid address. The NULL pointer deference can be controlled on the Windows operating system as explained in the next part. On the assembly, you can see that an address is retrieved from ALPC server object (at +8) which is used in *ExfAcquirePushLockShared* function. The lock will be released later using *ExfReleasePushLock* function. The next part demonstrates that the control of a shared lock pointer is enough to overwrite a little part of kernel memory and gain control over the kernel.

NULL deference exploitation

In the kernel, a NULL deference vulnerability exists when a NULL pointer is used without being checked. It allows the userland process to control data from this pointer. NULL deference vulnerability is exploitable in Windows because a memory page starting at 0 can be allocated. This is possible by calling *NtAllocateVirtualMemory* with a base address between 1 and 0xFFF. The allocator will round to the lower page and so allocate the NULL page. In vulnerability described here, albeit will be possible to control the lock pointer and redirect its address to the appropriate kernel space address.

The *ExfAcquirePushLockShared* function is an exported but undocumented function of the Windows kernel. Its behavior on the submitted lock pointer has changed between operating system versions. Its modification of lock structure data defines if this issue can be used to overwrite a part of kernel memory. This function is very complex and some tests have been done instead of understanding its complex lock procedure.

On Windows Vista, there is a modification of lock structure data when the high byte is set:

Before acquire:	XX000000
During lock:	XX000011
After release:	XX000001

The “XX” part of the number is from 0x01 to 0xFF. This context seems uncommon and hard to use to overwrite important data but as the pointer alignment is unchecked on lock procedures, a NULL function pointer can be replaced if it is placed near none zero data. On Windows XP, this function does not behave like this at all. Then this way to modify a fake lock pointer is not reliable between operating systems.

An internal component of the ALPC interface was overwritten but common function pointers can also be targeted as done by Rubén Santamarta [10]. In ALPC each blob type has its own structure as for the resource reserve:

```
0: kd> dds nt!AlpcReserveType
818f3a84  00000007 ; blob type id (7 stands for resource reserve)
818f3a88  72526c41 ; blob tag ‘AlRr’ (ALPC Resource Reserve ?)
818f3a8c  00000000
818f3a90  00000000
818f3a94  00000000
818f3a98  00000000 ; delete callback function (NULL means no callback function)
818f3a9c  819c97ed nt!AlpcpReserveDestroyProcedure ; destroy callback function

0: kd> dd 818f3a98+1 L1
818f3a99  ed000000 ; delete callback unaligned has high byte set
```

If the resource reserve blob type callback function pointer plus one is submitted, it will set unaligned low byte to 1. Once aligned, the callback function will be: 0x00000100. NULL page is still allocated it remains to set the shellcode at this address and delete the resource reserve used for exploitation (call *NtAlpcDeleteResourceReserve*). If the *AlpcReserveType* address is correctly guessed, it will not have any concurrency as the resource reserve feature is not used by any shipped components of Windows Vista.

With a simple int3 instruction, the call stack which goes in the kernel and returns directly in userland can be seen:

```
Break instruction exception - code 80000003 (first chance)
00000100 cc              int     3
1: kd> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
8f09fd28 819cebf3 86bd1dd8 8b90ba28 86bd1dd8 0x100
8f09fd3c 819c9e84 00000038 0012fb20 8b90ba28 nt!AlpcpDeleteBlob+0x68
8f09fd50 8188c96a 00000038 00000000 80000010 nt!NtAlpcDeleteResourceReserve+0x8a
8f09fd50 771d0f34 00000038 00000000 80000010 nt!KiFastCallEntry+0x12a
0012fb08 771cf220 00401506 00000038 00000000 ntdll!KiFastSystemCallRet
0012fb0c 00401506 00000038 00000000 80000010 ntdll!ZwAlpcDeleteResourceReserve+0xc
0012ffac 771aa9bd 7ffdb000 0012d608 00000000 0x401506
```

Kernel code execution gives the highest level of rights available on any operating system. As a driver, it can behave like a rootkit and subvert the Windows kernel.

Protecting the Windows kernel against kernel code execution

In local privilege escalation, Windows kernel exploitation is definitely the next exploitation landscape. Userland protection becomes harder to defeat and the kernel does not contain any. Pool exploitation goes back to old heap 4 bytes unlinking. A NULL deference is as important as a typical overflow and most of the time more stable. Creating protection is harder because kernel performance is really important and a single mistake can directly crash the whole system.

Protecting the Windows kernel can be divided into two different approaches. The first approach is to look for common exploitation methods and modify operating system behavior for each of them. For NULL deference, NULL page allocation can be denied. The system should not use it anyway. Once a little verification is made, you see that the system actually uses it a lot. For example, it allocates NULL page during video initialization and first SYSTEM process creation. Kernel and ntdll module *RtlCreateUserProcess* function allocates NULL page for unknown actions. So it could be denied once the system is correctly started but it must be done with care. For kernel pool overflow, pool verification could be hardened. It would slow down the system a little bit but would increase protection against overflow. It would be a hard task as it is undocumented. In userland, access heap management memory is easy but on the kernel the unexported variables from the kernel have to be found. This way of protecting the kernel is more about using a small trick than real protection. It protects only from known attack vectors. Some kernel vulnerabilities are unique and do not match any known types.

The other approach could rely on new hardware technology. Virtualization is often talked about as a way to create rootkits, but it could be used in a protection mechanism. Hardware monitoring is a good method to see if something goes wrong. In fact, it is easier to describe than to create. The more a system is monitored, the more it will slow down as it calls on each operation. It is certainly the best choice but it needs a lot of research.

It can seem strange than when protecting an operating system one begins by relying more on hardware than software. Some protections between userland and kernelland have existed for years in protection like PaX [13] which uses hardware features never went as far as virtualization. In

Windows, one is caught between what is documented and undocumented stuff. One cannot go too far with the undocumented part as it could break compatibility and make implementation even harder. If Windows provided a documented interface to symbol information during runtime, it would increase protection systems reliabilities and possibilities. Some part of the kernel could also be designed differently as access in userland which cannot be filtered by any means. Each syscall could provide a function pointer called only after arguments were verified and cached on local variables. Of course, this requires a new generation of kernel. It is not the only solution but the Windows kernel should change in order to improve protection against new kernel threats.

Conclusion

The communication mechanism in Windows still contains vulnerabilities, as demonstrated in this paper, even in LSASS which has existed for many years in Windows sub-system architecture. However, these issues are not easy to find and need some experience to be properly exploited. At first sight, they seemed unreliable or uninteresting. Crashing the system can be achieved without reversing any component. In this particular aspect, Windows Vista has drastically improved its code base and robustness. Hence, LSASS vulnerability does not concern Windows Vista, not only because they have hardened their heap component but also because they have improved the code between versions. ALPC kernel vulnerability is not a classical vulnerability and requires a good understanding of the component. The kernel remains much secure than other drivers or operating systems even after this major upgrade.

References

[1] Undocumented Ntinternals
http://undocumented.ntinternals.net
[2] Windows IT Library
http://www.windowsitlibrary.com/Content/356/08/1.html
[3] Blackhat 2006 - WLSI – Windows Local Shellcode Injection by Cesar Cerrudo
http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Cerrudo/bh-eu-06-Cerrudo-up.pdf
[4] Windows Heap exploitation by Matt Conover & Oded Horovitz
http://ivanlef0u.free.fr/repo/windoz/heap/XPSP2%20Heap%20Exploitation.ppt
[5] Heap de Windows : structure, fonctionnement et exploitation by Kostya Kortchinsky (French)
https://www.securinfos.info/jerome/DOC/heap-windows-exploitation.html
[6] Uninformed - Bypassing Windows Hardware-enforced DEP by skape & Skywing
http://www.uninformed.org/?v=2&a=4&t=sumry
[7] Nt debugging blog posts on LPC interface
http://blogs.msdn.com/ntdebugging/archive/tags/lpc/default.aspx
[8] Windows Internal – Fifth edition by Mark Russinovich and David Solomon
http://www.microsoft.com/MSPress/books/12069.aspx
[9] DEP (Data Execution Prevention) - Wikipedia
http://en.wikipedia.org/wiki/Data_Execution_Prevention
[10] Exploiting drivers by Rubén Santamarta
http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=fileinfo&id=51
[11] New NX APIs added to Windows Vista SP1, Windows XP SP3 and Windows Server 2008 - Michael Howard's Web Log
http://blogs.msdn.com/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1-windows-xp-sp3-and-windows-server-2008.aspx
[12] *SetProcessDEPPolicy* - MSDN
http://msdn2.microsoft.com/en-us/library/bb736299(VS.85).aspx
[13] PaX team home page
http://pax.grsecurity.net
[14] Inside I/O Completion Ports – Microsoft technet
http://technet.microsoft.com/en-us/sysinternals/bb963891.aspx