[WINDOWS SECURITY]

Reserve Objects in Windows 7

By Matthew "j00ru" Jurczyk

Microsoft is continuously improving the Windows operating system, as well as implementing brand new features and functionalities, which obviously make things much easier for both users and software developers. On the other hand, as new code is being introduced to the existing kernel- or user-mode modules, new opportunities might be opened for potential attackers, aiming at using the system's capabilities in favor of subverting its security. Proving the above thesis is one of this paper's objectives – as the reader will find out, there are always two sides of the coin.

As indicated in my previous article – *Windows Objects in Kernel Vulnerability Exploitation*[1] – the Object Manager is a crucial subsystem implemented as a part of the Windows Executive, since it manages access to mostly every kind of system resource utilized by the applications. In this article, I would like to introduce a new type of objects – *Reserve Objects* – which have been shipped together with the *Windows 7* product. As it turns out, the nature of these objects makes it possible to use them as a very handy helper tool, in the context of various, known kernel attacks.

Furthermore, according to the author's observations, the mechanism described in this paper is currently in the initial phase of development, and is very likely to evolve in the future Windows versions – in such case, it might become even more useful for *ring-0 hackers*.

New Windows = new system calls

Because of the fact that Microsoft developers are gaining feedback and overall experience of how well the current system mechanisms are working, the native system-call set as well as official API differ between distinct Windows versions (please note that while the API interface must provide backwards compatibility, there is no such guarantee regarding native calls). As a very good example, one should take a look at a comparison table[2], presenting changes between Windows 7 and Windows Vista SP1, in terms of *ntdll.dll* exported symbols. As can be seen, numerous new functions have been added, while only a couple of them removed.

A majority of the new function set is composed of names beginning with *Rtl** (*Run-time library*), implemented as helper routines, commonly utilized by the official API code (such as *kernel32.dll*). Aside from these, one can also find around fifteen new *Nt** symbols, which represent fresh kernel functions that are exposed to *ring-3*, so that user-defined applications (or more likely, system libraries) can take advantage of what the new system provides. *Listing 1* presents a complete set of new *ntdll* names within our interest.

What shouldn't be a surprise is the fact that most of the new *syscalls* do not implement a completely new feature – instead, they seem to extend the functionalities that have already been there, using additional parameters, and providing extra capabilities which were not present before. For instance, the *NtCreateProfileEx* function adds in options that were not available in older *NtCreateProfile* - the same effect affects *syscalls* like *NtOpenKey(Ex)*, *NtQuerySystemInformation(Ex)* and many others.

To get to the point, the functions that we are mostly interested in, are:

- *NtAllocateReserveObject* – system call responsible for creating an object on the kernel side – performing a memory allocation on the kernel pool, returning an adequate Handle etc,
- *NtQueueApcThreadEx* – system call which can optionally take advantage of the previously allocated *Reserve Object* while inserting an APC (*Asynchronous Procedure Call*) into the specified thread's queue,
- *NtSetIoCompletionEx* – system call incrementing the pending IO counter for an IO Completion Object. As opposed to *NtSetIoCompletion* function, it can utilize the *Reserve Objects*, as well.

As can be seen, all of these three above functions have been introduced in Windows 7 and, at the same time, no accurate information regarding these routines is publicly available. In order to get a good understanding on what this new types of object really are, let's focus on the allocation function, in the first place.

nt!NtAllocateReserveObject

In order to give you the best insight of the underlying mechanisms, I would like to begin with a thorough analysis of the allocation function; you can find its pseudo-code (presented in a C-like form) in *Listing 2*.

Listing 1. Interesting system calls introduced in Windows 7

```
NtAllocateReserveObject
NtQueueApcThreadEx
NtSetIoCompletionEx
```

The system call requires three arguments to be passed – one of which is an output parameter, used to return the object handle to the user's application, while the other two are meant to supply the type and additional information regarding the object to be allocated. Right after entering the function, the *hObject* pointer is compared against *nt!MmUserProbeAddress*, ensuring that the address does not exceed the user memory regions. Moreover, since the number of supported reserve object types is limited (and equals two at the time of writing this paper), every higher number inside *ObjectType* bails out the function execution.

After the sanity checks are performed, an internal *nt!ObCreateObject* routine is used to create an object of a certain size and type (you can find the function's definition in *Listing 3*) – the interesting part begins here. As can be seen, both the *ObjectType* and *ObjectSizeToAllocate* parameters are volatile – instead, the *PspMemoryReserveObjectTypes* and *PspMemoryReserveObjectSizes* internal arrays are employed, together with the *ObjectType* parameter used as an index into these.

As mentioned before, only two types of reserve objects are currently available: *UserApcReserve* and *IoCompletionReserve objects*. Each of them has a separate OBJECT_TYPE descriptor structure, containing some of the object characteristics, such as its name, allocation type (paged/non-paged pool), and others. The pointers to these structs are available through the *PspMemoryReserveObjectTypes* array; the object descriptors for both types are presented in *Listing 4*. This observation alone implies that one is able to choose the object type to be used.

Listing 2. NtAllocateReserveObject function pseudo-code

```
#define APC_OBJECT           0
#define IO_COMPLETION_OBJECT 1
#define MAX_OBJECT_ID        1

NTSTATUS STDCALL NtAllocateReserveObject(
   OUT PHANDLE hObject,
   IN  POBJECT_ATTRIBUTES ObjectAttributes,
   IN  DWORD   ObjectType)
{
  PVOID ObjectBuffer;
  HANDLE hOutputHandle;
  NTSTATUS NtStatus;

  if(PreviousMode == UserMode)
  {
    // Validate hObject
  }
  if(ObjectType > MAX_OBJECT_ID)
  {
    /* Bail out: STATUS_INVALID_PARAMETER
     */
  }
  else
  {
    NtStatus = ObCreateObject(PreviousMode,
                              PspMemoryReserveObjectTypes[ObjectType],
                              ObjectAttributes,
                              PreviousMode,
                              0,
                              PspMemoryReserveObjectSizes[ObjectType],
                              0,
                              0,
                              &ObjectBuffer);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */

    memset(ObjectBuffer,0,PspMemoryReserveObjectSizes[ObjectType]);

    if(ObjectType == IO_COMPLETION)
    {
      //
      // Perform some ObjectBuffer initialization
      //
      ObjectBuffer[0x0C] = 3;
      ObjectBuffer[0x20] = PspIoMiniPacketCallbackRoutine;
      ObjectBuffer[0x24] = ObjectBuffer;
      ObjectBuffer[0x28] = 0;
    }

    NtStatus = ObInsertObjectEx(ObjectBuffer,
                                &hOutputHandle,
                                0,
                                0xF0003,
                                0,
                                0,
                                0);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */

    *hObject = hOutputHandle;
  }

  return NtStatus;
}
```

Table 1. ***PspMemoryReserveObjectSizes*** **contents on 32- and 64-bit Windows 7 architecture**

	Windows 7 x86	Windows 7 x86-64
UserApcReserve	0x34	0x60
IoCompletionReserve	0x2C	0x58

The second dynamic argument passed to *ObCreateObject* is the size of a buffer, sufficient to hold the object's internal structure. Considering the differences between the size of a machine word on x86 and x86-64, one shouldn't be surprised that the object sizes stored in the *PspMemoryReserveObjectSizes* array are also distinct. The exact numbers stored in the aforementioned array are presented in *Table 1*.

After the object is successfully allocated, the buffer is zeroed, so that no trash bytes could cause any trouble from this point on. Next then, in case of IoCompletion allocation, *ObjectBuffer* is filled with some initial values, such as a pointer to itself or a callback function address. Please note that no initialization is performed for an UserApc object, which remains empty until some other function references the object's pool buffer.

Going further into the function's body, a call into nt!*ObInsertObjectEx* is issued, in order to put the object into the local process' handle table (i.e. retrieve a numeric ID number, representing the *resource* in *ring-3*). The handle is put into the local *hOutputHandle* variable, and respectively copied into the *hObject* pointer, specified by the application (and already verified). If everything goes fine up to this point, the system call handler returns with the ERROR_SUCCESS status.

In short, *NtAllocateReserveObject* makes it possible for any system user to allocate a buffer on the non-paged kernel pool, and obtain a HANDLE representation of this buffer in *user-mode*. As it will turn out later in this paper, the above can give us pretty much control over the kernel memory, when exploiting custom vulnerabilities.

nt!NtQueueApcThreadEx

The first user-controlled function (i.e. system call handler) being able to operate on the *Reserve Objects* is responsible for queuing *Asynchronous Procedure Calls*[3,4] in the context of a specified thread. Once again, Listing 5 presents a C-like pseudo-code of the function's real implementation.

First of all, the KTHREAD address assigned to the input *hThread* parameter is retrieved using *ObReferenceObjectByHandle*. If the call succeeds, and the thread doesn't have a SYSTEM_THREAD flag set, the execution can go two ways:

- If *hApcReserve* is a non-zero value, the object's memory block address is obtained, and stored in *ApcBuffer*. Next then, an atomic compare-exchange operation is performed, in order to mark the reserve object as "busy" – the first DWORD of the buffer is used for this purpose. *ApcBuffer* is increased by sizeof(DWORD), pointing to the beginning of the _KAPC structure. Eventually, the *Kernel*-and *RundownRoutine* function pointers are set to adequate addresses, so that the reserve object is correctly freed after the APC finishes its execution.
- If *hApcReserve* equals zero, a straightforward allocation of 0x30 (Windows 7 x86) or 0x58 (Windows 7 x86-64) bytes is performed on the *Non-Paged Pool*, and the resulting pointer is assigned to *ApcBuffer*. The *Kernel-Routine* and *RundownRoutine* pointers are set to *IopDeallocateApc* and *ExFreePool*, respectively.

After the *if* statement, a *KeInitializeApc* call is made, specifying the *ApcBuffer* pointer as destination KAPC address, and passing the rest of the previously initialized arguments (*KernelRoutine*, *RundownRoutine*, *ApcRoutine*, *ApcArgument1*). Finally, a call to *KeInsertQueueApc* is issued, which results in having the KAPC structure (pointed to by *ApcBuffer*) inserted into the APC queue of the thread in consideration.

On Microsoft Windows versions prior to 7, the user was unable to get the kernel to make use of a specific memory block of a known address. Instead, the latter execution path of the above *if* statement was always taken. If the application really wanted to queue an APC, the required space was allocated right before queuing the structure – both these operations used to happen inside one routine (system call). Therefore, no kernel memory address was revealed to the user, thus making it impossible to utilize the KAPC structures (on the kernel pool) in stable attacks against the kernel. Fortunately for us, times have apparently changed ;-)

Listing 3. Kernel object-management functions' definitions

```
NTSTATUS ObCreateObject (
  IN KPROCESSOR_MODE ObjectAttributesAccessMode OPTIONAL,
  IN POBJECT_TYPE ObjectType,
  IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
  IN KPROCESSOR_MODE AccessMode,
  IN PVOID Reserved,
  IN ULONG ObjectSizeToAllocate,
  IN ULONG PagedPoolCharge OPTIONAL,
  IN ULONG NonPagedPoolCharge OPTIONAL,
  OUT PVOID *Object );

NTSTATUS ObInsertObject (
  IN PVOID Object,
  IN PACCESS_STATE PassedAccessState OPTIONAL,
  IN ACCESS_MASK DesiredAccess,
  IN ULONG AdditionalReferences,
  OUT PVOID *ReferencedObject OPTIONAL,
  OUT PHANDLE Handle );
```

Listing 4. The OBJECT_TYPE structures associated with the Reserve Objects

```
kd> dt _OBJECT_TYPE fffffa800093ff30
ntdll!_OBJECT_TYPE
   +0x000 TypeList          : _LIST_ENTRY
   +0x010 Name              : _UNICODE_STRING "UserApcReserve"
   +0x020 DefaultObject     : (null)
   +0x028 Index             : 0x9 ''
   +0x02c TotalNumberOfObjects : 0
   +0x030 TotalNumberOfHandles : 0
   +0x034 HighWaterNumberOfObjects : 0
   +0x038 HighWaterNumberOfHandles : 0
   +0x040 TypeInfo          : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock          : _EX_PUSH_LOCK
   +0x0b8 Key               : 0x72657355
   +0x0c0 CallbackList      : _LIST_ENTRY

kd> dt _OBJECT_TYPE fffffa800093fde0
ntdll!_OBJECT_TYPE
   +0x000 TypeList          : _LIST_ENTRY
   +0x010 Name              : _UNICODE_STRING "IoCompletionReserve"
   +0x020 DefaultObject     : (null)
   +0x028 Index             : 0xa ''
   +0x02c TotalNumberOfObjects : 1
   +0x030 TotalNumberOfHandles : 1
   +0x034 HighWaterNumberOfObjects : 1
   +0x038 HighWaterNumberOfHandles : 1
   +0x040 TypeInfo          : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock          : _EX_PUSH_LOCK
   +0x0b8 Key               : 0x6f436f49
   +0x0c0 CallbackList      : _LIST_ENTRY
```

nt!NtSetIoCompletionEx

The third, and last function within our interest operates on the *IoCompletion* object, previously created or opened using *NtCreateIoCompletion*/*NtOpenIoCompletion* functions. Let's take a look at the pseudo-code (presented in *Listing 6*) and find out what we can expect.

At the very beginning of the function's body, both the *hIoCompletion* and *hReserveObject* handles are referenced – if any of these fails, the execution is aborted. Next then, the *InterlockedCompareExchange* function is called, for the same reason as it was before – in order to synchronize the access to the object by concurrent threads running on the system. An internal *IoSetIoCompletionEx* function is called, and in case it fails for any reason, the object is restored to its previous state (i.e. with the first DWORD set to zero), and the function bails out. Otherwise, the ERROR_SUCESS status is returned.

Malicious utilization

Now, as the *Reserve Object* term is clear, we can finally find out some practical examples of how a potential attacker can take advantage of the new object types.

UserApcReserve as a write-what-where target

Because of the fact that Windows kernel make it possible for a user-mode process to retrieve information regarding all active objects present in the system (including information like the owner's PID, numeric handle value, the object's descriptor address and others), one is able to find the address associated to a given object, very easily. More information on how to extract this kind of information from the operating system can be found in the *NtQuerySystemInformation* documentation[5,6] (together with the *SystemHandleInformation* parameter).

In general, when a kernel module decides to manually allocate memory using kernel pools, the resulting address (returned by *ExAllocatePool* or equivalent) never leaves kernel mode, and therefore is never revealed to the user-mode caller. Due to this "limitation", and because of the fact that it is very unlikely to successfully foresee or guess the allocation address – such memory areas cannot be used as a reasonable *write-what-where* attack target. For instance, the *NtQueueApcThread* system call has always used a dynamic buffer to store the required KAPC structure on every Windows NT-family version previous to Windows 7 – and so, it never appeared to become targeted by a stable code-execution exploit.

Nowadays, since the users can choose between *safe NtQueueThreadApc* and *NtQueueThreadApcEx* (which uses a memory region with known address), things are getting more interesting. The attacker could allocate and initialize the *UserApcReserve* object, find its precise address and overwrite the KAPC structure contents (using a custom *ring-0* vulnerability), and finally flush the APC queue, thus performing a successful Elevation of Privileges attack. A pseudo-code of an exemplary exploit is presented in *Listing 7*.

Listing 5. The NtQueueApcThreadEx routine pseudo-code

```
NTSTATUS STDCALL NtQueueApcThreadEx(
    IN HANDLE hThread,
    IN HANDLE hApcReserve,
    IN PVOID  ApcRoutine,
    IN PVOID  ApcArgument1,
    IN PVOID  ApcArgument2,
    IN PVOID  ApcArgument3)
{
  NTSTATUS NtStatus;
  PVOID ThreadObject;
  PVOID ApcBuffer;
  PVOID KernelRoutine;
  PVOID RundownRoutine;

  NtStatus = ObReferenceObjectByHandle(hThread,
                                      THREAD_SET_CONTEXT,
                                      PsThreadType,
                                      PreviousMode,
                                      &ThreadObject,
                                      0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */

  if(SystemThread(ThreadObject))
  /* Bail out: STATUS_INVALID_HANDLE
   */

  if(hApcReserve != NULL)
  {    NtStatus = ObReferenceObjectByHandle(hApcReserve,
                                        2,
                                        UserApcType,
                                        PreviousMode,
                                        &ApcBuffer,
                                        0);
    if(!NT_SUCCESS(NtStatus))
    /* Bail out: NtStatus
     */

    InterlockedCompareExchange(ApcBuffer,1,0);

    ApcBuffer += 4;

    KernelRoutine  = PspUserApcReserveKernelRoutine;
    RundownRoutine = PspUserApcReserveRundownRoutine;
  }
  else
  {
    ApcBuffer = ExAllocatePoolWithQuotaTag(NonPagedPool,0x30,"Psap");
    if(ApcBuffer == NULL)
    /* Bail out: STATUS_NO_MEMORY
     */

    KernelRoutine  = IopDeallocateApc;
    RundownRoutine = ExFreePool;
  }

  KeInitializeApc(ApcBuffer,
                  ThreadObject,
                  0,
                  KernelRoutine,
                  RundownRoutine,
                  ApcRoutine,
                  1,
                  ApcArgument1);
  if(!KeInsertQueueApc(ApcBuffer,ApcArgument2,ApcArgument3,0))
  {
    RundownRoutine(ApcBuffer);
    /* Bail out: STATUS_UNSUCCESSFUL
     */
  }
  return STATUS_SUCCESS;
}
```

Payload inside kernel memory

Across various security vulnerabilities related to the system core, the specific conditions in which code execution is triggered, are always different. As a consequence of numerous background mechanisms keeping the machine alive, a potential attacker can never predict every single part of the system state, at the time of performing the attack. In some cases, there is no guarantee that the payload code is even executed in the same context as the process that issued the vulnerability. This, in turn, could pose a serious problem in terms of creating a reliable exploit, which should launch the shellcode no matter what's currently happening on the machine.

One possible solution could rely on setting-up the necessary code somewhere inside a known address in *process-independent* kernel memory; and

Listing 6. The NtSetIoCompletionEx routine pseudo-code

```
NTSTATUS STDCALL NtSetIoCompletionEx(
    IN HANDLE hIoCompletion,
    IN HANDLE hReserveObject,
    IN PVOID  KeyContext,
    IN PVOID  ApcContext,
    IN NTSTATUS IoStatus,
    ULONG_PTR IoStatusInformation)
{
  NTSTATUS NtStatus;
  PVOID CompletionObject;
  PVOID ReserveObject;

  NtStatus = ObReferenceObjectByHandle(hIoCompletion,
                                       2,
                                       IoCompletionObjectType,
                                       PreviousMode,
                                       &CompletionObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  NtStatus = ObReferenceObjectByHandle(hReserveObject,
                                       2,
                                       IoCompletionReserveType,
                                       PreviousMode,
                                       &ReserveObject,
                                       0);
  if(!NT_SUCCESS(NtStatus))
  /* Bail out: NtStatus
   */
  InterlockedCompareExchange(ReserveObject,1,0);
  NtStatus = IoSetIoCompletionEx(CompletionObject,
                                 KeyContext,
                                 ApcContext,
                                 IoStatus,
                                 IoStatusInformation,
                                 0,
                                 ReserveObject+4);
  if(!NT_SUCCESS(NtStatus))
  {
    *(DWORD*)ReserveObject = 0;
    /* Bail out: NtStatus
     */
  }
  return STATUS_SUCCESS;
}
```

Listing 7. An exemplary write-what-where exploitation scheme

```
VOID Payload()
{
  /* Execute the ring-0 payload
   */
}
VOID Exploit()
{
  /* Allocate the UserApcReserve object
   */
  hObject = NtAllocateReserveObject(UserApcReserve);
  /* Initialize the KAPC structure, using reserve object's memory
   */
  NtQueueApcThreadEx(CurrentThread(),hObject,Payload);
  /* Find the object address [in kernel]
   */
  KAPCAddr = FindObjectAddress(CurrentProcess(),hObject);
  /* Overwrite the APC type with KernelMode, so that the Payload
   * function is called with ring-0 privileges
   */
  OverwriteMemory(KAPCAddr->ApcMode,KernelMode);
  /* Enter alerted state to flush the APC queue, e.g. using SleepEx
   *
   */
  EnterAlertedState();
}
```

then use this address to redirect the vulnerable module's execution path. The question is – how a plain, restricted user can put a fair amount (sufficient to store the payload) of data at a known address in KM? As expected – the *Reserve Objects* can lend us a helping hand here.

If we take a closer look at the KAPC structure definition from the x86-64 architecture OS (presented in *Listing 8*), we can observe that starting with offset +0x030, there are four user-controlled values – all of them defined through the NtQueueThreadApcEx parameters (3^{rd}, 4^{th}, 5^{th}, 6^{th}):

- NormalRoutine – a pointer to the user-specified callback function, called when flushing the APC queue,
- NormalContext – first routine argument, internally used as the *KelnitializeApc* function parameter,
- SystemArgument1, SystemArgument2 – second and third arguments, passed to the *KeInsertQueueApc* function

Being able to control roughly four variables in a row, each of which has the machine word's size (32 bits on x86, 64 bits on x86-64), one can insert 16 or 32 bytes of continuous data (depending on the system architecture), at a known address! Furthermore, because of the fact that one can create any number of such objects, it is possible to create *long chains* of 16/32-byte long code chunks, each connected to the successive one using a simple JMP (or any other, shorter) instruction. The overall idea is presented in *Image 1*.

DEP in Windows x64 kernel

One important issue regarding the idea presented in this section is the uncertainty whether it is possible to execute the code placed inside a pool allocation safely, i.e. avoid problems with some kind of DEP-like protections, that are continuously extended and improved by Microsoft. As MSDN states, however, the hardware-enforced *Data Execution Prevention* aims to protect only one (32-bit platforms) or three (64-bit) crucial parts of the non-executable kernel memory, leaving the rest on its own[7].

DEP is also applied to drivers in kernel mode. DEP for memory regions in kernel mode cannot be selectively enabled or disabled. On 32-bit versions of Windows, DEP is applied to the stack by default. This differs from kernel-mode DEP on 64-bit versions of Windows, where the *stack*, *paged pool*, and *session pool* have DEP applied.

As can be seen, both the stack and all types of kernel pools except the *non-paged* one are protected against code execution. Let's take a look at the *OBJECT_TYPE* structure contents associated to *UserApcReserve* and *IoCompletionReserve* objects (*Listing 9*). Fortunately for us, both objects are allocated on non-paged pool, which means that one can execute the code within a custom KAPC without any real trouble.

Heap spraying-like techniques

If one realizes that the reserve objects are actually small pieces of memory controlled by the user, in terms of content and virtual address, a variety of possible ways of utilization arises. For instance, according to the author's research, it is likely that a user-mode process might be able to partially control the kernel pools memory layout, by properly manipulating the *Reserve Objects* present in the system, i.e. by allocating and freeing appropriate chunks of memory. Due to the fact that any process is able to queue new KAPCs using *NtAllocateReserveObject* + *NtQueueApcThreadEx*, and free them using *SleepEx* (resulting in emptying the queue for a given thread), one could try to use this ability to control the memory allocations performed by other, uncontrolled kernel modules. In practice, there are several internal mechanisms, such as *Safe Pool Unlinking*[8] introduced in Windows 7, purposed to stop hackers from executing arbitrary code through *ring-0* vulnerabilities; since they highly rely on the secrecy of pool allocation addresses, steadily controlling the memory pools layout could result in breaking the latest security measure taken in *kernel-mode*.

The author is aware of the fact that numerous obstacles are related to the

Listing 8. The pool allocation types assigned to Reserve Objects

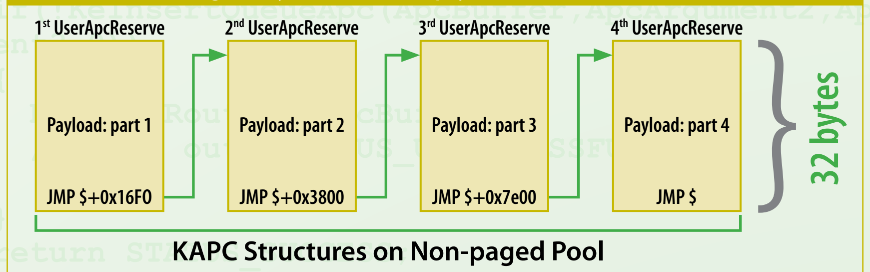
```
UserApcReserve:

      +0x01c ValidAccessMask   : 0xf0003
      +0x020 RetainAccess      : 0
      +0x024 PoolType          : 0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0
      +0x02c DefaultNonPagedPoolCharge : 0xb8

IoCompletionReserve:

      +0x01c ValidAccessMask   : 0xf0003
      +0x020 RetainAccess      : 0
      +0x024 PoolType          : 0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0
      +0x02c DefaultNonPagedPoolCharge : 0xb0
```

Image 1. Exemplary KAPC structure chain, storing 128 bytes of the user's payload in four chunks of data



above ideas – such as fixed memory allocation size (~0x30-0x60 bytes), only one (*non-paged*) type of pool being used and so on – as for now, this subject is left open to be researched by any willing individual. Overall, what should be remarked is that there are still countless ways of evading the generic protections ceaselessly introduced by the operating system vendors. The game is not over, yet ;)

Conclusion

In this paper, the author wanted to present a new, interesting mechanism introduced in the latest Windows version; show some possible ways of turning this functionality against the system and make it work in the attacker's favor; and finally present how fresh, legitimate features created by the OS devs should be analyzed in the context of exploitation usability. As old ideas and methods already have their countermeasures implemented in the system core, new ones have to be developed – the best source for these, in my opinion, is the mechanisms such as the one described in this paper.

It is believed that many interesting, sophisticated attacks against the kernel can be carried out using functionalities like *Reserve Objects*, therefore the author wants to highly encourage every individual interested in *ring-0* hacking, to investigate the subject on one's own and possibly contribute to the narrow kernel exploitation field in some way. Good luck! •

>>REFERENCES

1. Matthew "j00ru" Jurczyk, Windows Objects in Kernel Vulnerability Exploitation, *http://www.hackinthebox.org/misc/HITB-Ezine-Issue-002.pdf*
2. Gynvael Coldwind, Changes in Microsoft Windows 7 vs Microsoft Vista SP1: ntdll.dll, *http://gynvael.coldwind.pl/?id=134*
3. MSDN, Asynchronous Procedure Calls, *http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx*
4. Albert Almeida, Inside NT's Asynchronous Procedure Call, *http://www.drdobbs.com/184416590*
5. MSDN, NtQuerySystemInformation Function, *http://msdn.microsoft.com/en-us/library/ms724509(VS.85).aspx*
6. Sven B. Schreiber, Tomasz Nowak, NtQuerySystemInformation, *http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/System%20Information/NtQuerySystemInformation.html*
7. MSDN, Data Execution Prevention, *http://technet.microsoft.com/en-us/library/cc738483(WS.10).aspx*
8. Swiblog @ Technet, Safe Unlinking in the Kernel Pool, *http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx*