SizeOfStackReserve As Anti-Attaching Trick

waleedassar.blogspot.com/2012/11/sizeofstackreserve-as-anti-attaching.html

In this post i will show you a new ***anti-attaching*** trick that has been tested on ***Windows 7***. It does not work on ***Windows XP*** due to the changes ***Microsoft*** introduced in the way threads are created.

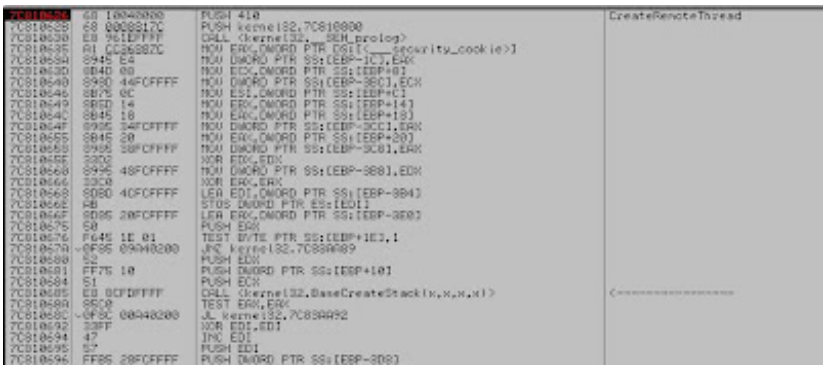
Let's first see how thread creation in ***Windows 7*** is different from that of ***Windows XP***.

In **Windows XP**, whenever you call the ***kernel32*** "***CreateRemoteThread***" or the ***ntdll*** "***RtlCreateUserThread***" function to create a new thread, the following occurs underneath:

The ***kernel32*** "***BaseCreateStack***" or ***ntdll*** "***RtlpCreateStack***" function is called in case of "***CreateRemoteThread***" or "***RtlCreateUserThread***" successively ***to allocate space for the new thread's stack in the address space of the target process.***

N.B. The ***kernel32*** "***CreateThread***" function is only a call to the ***kernel32*** "***CreateRemoteThread***" function with the "***hProcess***" parameter set to ***-1.***

Since there is no big difference between the "***BaseCreateStack***" and "***RtlpCreateStack***" functions, it is enough for us to take the "***BaseCreateStack***" function in disassembly in this post.



The "***BaseCreateStack***" function takes four parameters, only three of them are of interest. The ***first*** parameter is the ***handle*** to the process in which we are about to allocate user stack memory. The ***second*** parameter is the size in bytes of user stack memory to ***COMMIT*** into the target process's address space. The ***third*** parameter is the size in bytes of user stack memory to ***RESERVE*** into the target process's address space. Hereafter, i will refer to them as ***hProcess***, ***CommitSize***, and ***ReserveSize***.

N.B. If you call the "***CreateRemoteThread***" function with the "***dwStackSize***" parameter set to e.g. ***0x10000***, then ***BaseCreateStack*** commits ***0x10000*** bytes. On the other side, if the "***CreateRemoteThread***" function is called with the "***dwCreationFlags***" parameter having the

"**STACK_SIZE_PARAM_IS_A_RESERVATION"** flagset, then ***BaseCreateStack*** Reserves ***0x10000***.

```
//Part of CreateRemoteThread (XP)
if(dwCreationFlags&STACK_SIZE_PARAM_IS_A_RESERVATION)
{
      BaseCreateStack(hProcess,0,dwStackSize);
}
else
{
      BaseCreateStack(hProcess,dwStackSize,0);
}
```

Now, let's dive into the "***BaseCreateStack***" function and see what is going on inside.

1) It extracts the value of ***ImageBase*** from the ***PEB*** of the process in which it is called, the value is then passed to the "***RtlImageNtHeader***" function. If the "***RtlImageNtHeader***" function fails an error ***ERROR_BAD_EXE_FORMAT*** is returned.

If the "***ReserveSize***" parameter passed to it is zero, it uses the value of the "**SizeOfStackReserve**" field of the ***IMAGE_OPTIONAL_HEADER*** structure.

3) Similarly, If the "***CommitSize***" parameter passed to it is zero, it uses the value of the "**SizeOfStackCommit**" field of the ***IMAGE_OPTIONAL_HEADER*** structure. Please remember that the values are extracted from the PE header of the main executable of the process that is calling the "***CreateRemoteThread***" function, not the target process.

4) It then makes some sanitization checks on the ***ReserveSize*** and ***CommitSize***, for example to ensure that the commit size is never greater than the reserve size. It also checks to ensure that the commit size is never lower than the value of the "***MinimumStackCommit***" field of ***PEB***.

5) It calls the "***ZwAllocateVirtualMemory***" function to reserve memory of size ***ReserveSize*** into the address space of the target process with the ***PAGE_READWRITE*** protection attribute.

6) It calls the "***ZwAllocateVirtualMemory***" function to commit ***CommitSize+0x1000*** of the memory reserved in the previous step.

7) The extra page committed in the previous step is then given the ***PAGE_GUARD*** protection attribute.

```
unsigned long StackStartAddress=0;
int ret=ZwAllocateVirtualMemory(hProcess,&StackStartAddress,
                                0,&ReserveSize,MEM_RESERVE,PAGE_READWRITE);
if(ret<0) return ret;
//Here goes some code that Writes to output structure
unsigned long StackStartAddress+=(ReserveSize-COmmitSize);
StartStartAddress-=Page_Size; //Space for the PAGE_GUARD page
CommitSize+=Page_Size;
ret=ZwAllocateVirtualMemory(hProcess,&StackStartAddress,
                                0,&CommitSize,MEM_RESERVE,PAGE_READWRITE);
if(ret<0) return ret;
unsigned long old_prot;
ret=ZwProtectVirtualMemory(hProcess,&StackStartAddress,
                                &Page_Size,PAGE_READWRITE|PAGE_GUARD,&old_prot);
if(ret<0) return ret;
return 0;}
```

Here is a similar reversed code of the "***BaseCreateStack***" function. From here.

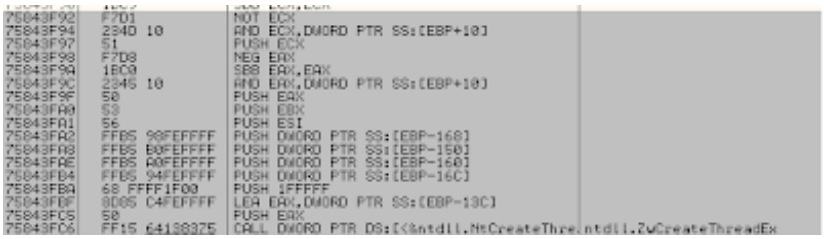
The reason why a ***PAGE_GUARD*** page always exists at the end of committed stack is for the kernel to be notified each time the stack needs to be expanded. For example, if a thread tries to touch its stack's ***PAGE_GUARD*** page, an ***STATUS_GUARD_PAGE_VIOLATION*** exception is raised and swallowed by the kernel and it automatically commits one more page.

N.B. If a thread tries to touch the ***PAGE_GUARD*** page of another thread's stack, the exception is passed to the application or the debugger.

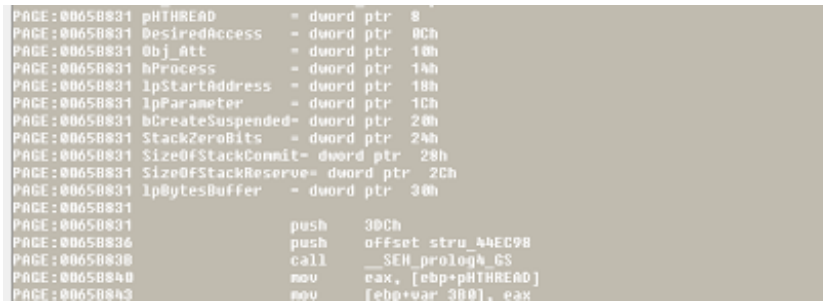
After the stack has been allocated in the target process's address space, the "***CreateRemoteThread***" function formulates a ***CONTEXT*** structure for the new thread. After the previous steps have completed successfully, the "**ZwCreateThread**" function is called to initiate the new remote thread.

Now let's see how threads are created in ***Windows 7***.

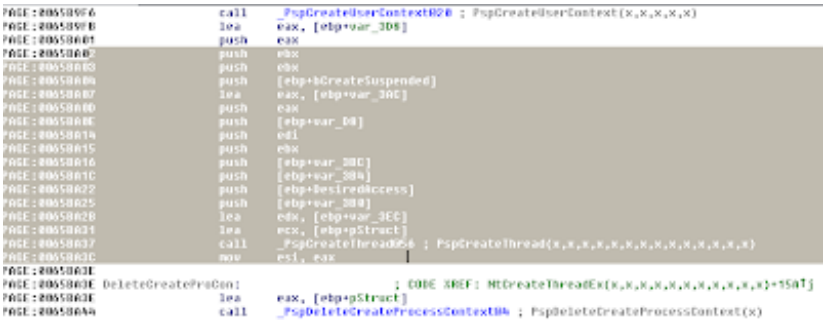
In ***Windows 7***, if we take the "***CreateRemoteThread***" or "***RtlCreateUserThread***" function into disassembly, we will see that the "***dwStackSize***" is directly passed to the "***ZwCreateThreadEx***" function.
So, our first assumption here is that stack allocation is now forwarded to the kernel. Also, we can note that now in later versions of ***Windows*** than ***XP***, the "***ZwCreateThreadEx***" function is by default used for thread creation instead of the "***ZwCreateThread***" function.



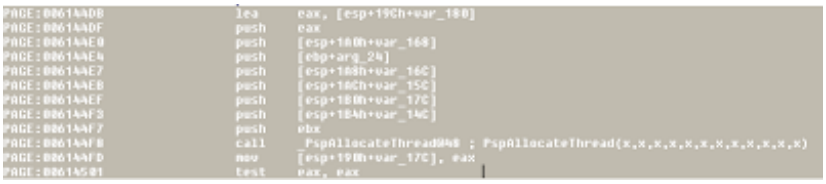
Now let's check the "***NtCreateThreadEx***" function in ***ntoskrnl.exe***.

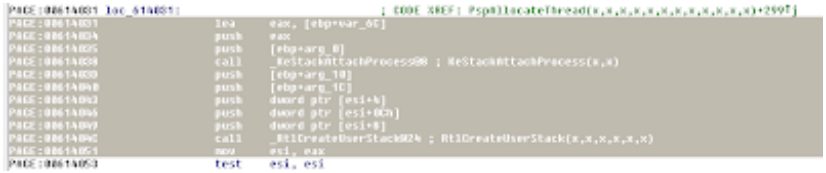


We can easily see in "***NtCreateThreadEx***" a call to the "***PspCreateThread***" function.

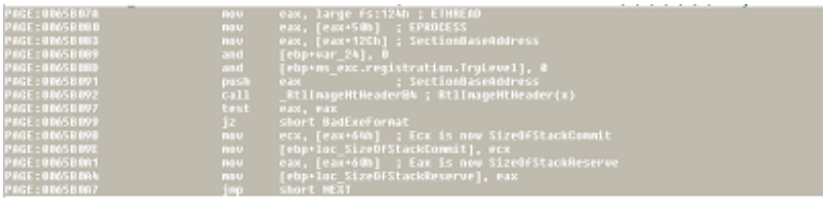


The ***"PspCreateThread"*** function calls the "***PspAllocateThread***" function which calls "***RtlCreateUserStack***" function.





The "***RtlCreateUserStack***" function is called after attaching to the target process's address space. Now let's look at the "***RtlCreateUserStack***" function in disassembly.



Now it is easy to see that it reads the ***PE header*** from the main executable of the process in which the remote thread is being created unlike ***XP*** where information was extracted from the main executable of the process that creates the thread. Yeah, it seems ***Microsoft*** fixed a very minor issue.

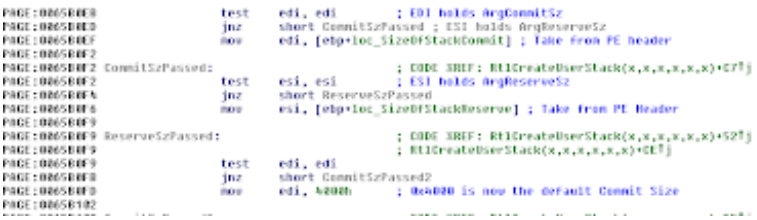
From the image above, it is also easy to conclude that if we forced the "***RtlImageNtHeader***" function to fail, we can prevent any foreign process including the debugger from attaching to our process. The easiest way to accomplish that is by erasing the ***PE header*** at runtime. Any call to ***ZwCreateThreadEx*** as part of calling the "***DebugActiveprocess***" function (Used for attaching to a running process) would fail. For more information and examples, please refer to my previous post.

N.B.DebugActiveProcess calls ***DbgUiIssueRemoteBreakin*** which calls ***~RtlCreateUserThread*** which calls "***ZwCreateThreadEx***".

One may say, "Erasing the whole ***PE header*** may render many ***APIs*** which read from the ***PE header*** useless e.g. ***FindResource*** or ***GetProcAddress***". My answer will be "Yes, you are right".

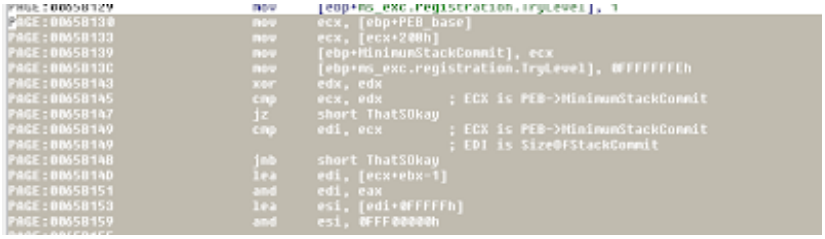
So, we should find a smarter way to do it.

Okay, let's continue disassembling the "***RtlCreateUserStack***" function.

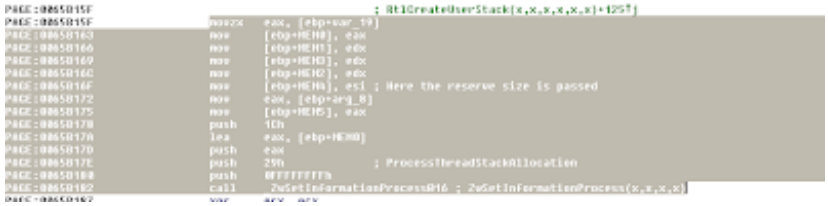


As you can see in the image above if the size of stack commit argument passed to it is zero, it takes the value of the "***SizeOfStackCommit***" field from the PE header. The same measure is taken if the size of stack reserve passed is zero. It is also noteworthy that if both the size of stack commit argument passed and "***SizeOfStackCommit"*** of the PE header are zero, the commit size becomes ***0x4000*** (The default commit size is ***0x4000***).

The function then checks the size of stack commit against the size of stack reserve. If the size of stack commit happens to be greater, then the size of stack reserve is adjusted to be greater.

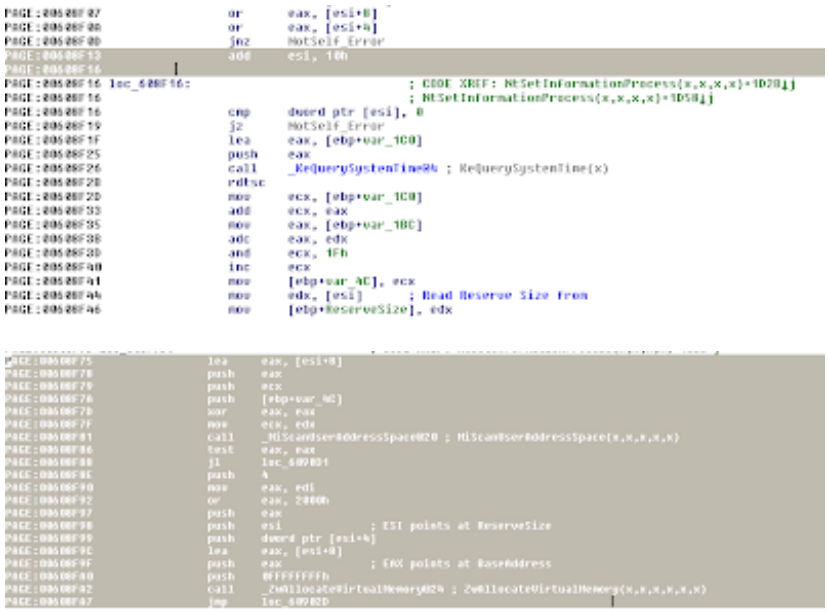


The function then ensures that the size to be committed is not less than the "***MinimumStackCommit***" field of the process's ***PEB***. If it is less, the size to be committed is adjusted.



The function then calls the "***ZwSetInformationProcess***" function with the "***ProcessInformationClass***" parameter set to ***0x29*** (***ProcessThreadStackAllocation***). The size to be reserved is passed in the ***4th*** member of the structure passed in the "***ProcessInformation***" parameter.

Now let's quickly have a look at the "***NtSetInformationProcess***" function.



As you can see in the two images above, the value of the ***4th*** member of the structure passed to the "***ZwSetInformationProcess***" function is used as the "***RegionSize***" parameter

passed to the "***ZwAllocateVirtualMemory***" function.
Given this knowledge, if we at runtime change the value of the "***SizeOfStackReserve***" field of the PE header to a huge value, then we can cause the ***"ZwAllocateVirtualMemory", "ZwSetInformationProcess", "RtlCreateUserThread", "PspAllocateThread", "PspCreateThread", and "NtCreateThreadEx"*** functions to successively fail preventing any foreign processes including debuggers from creating any thread in our process.
A demo can be found here and its source code from here.
Any comments or ideas are more than welcome.
You can follow me on Twitter @waleedassar