

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.

```
7C910628 68 10040000 PUSH 410 CreateRemoteThread
7C91062B 68 0038217C PUSH kernel32.7C818900
7C91062E 58 501D7F7F CALL kernel32._SH_prolog
7C910630 81 C0365B7C MOV EAX,DWORD PTR DS:[_security_cookie]
7C910633 8445 E4 MOV DWORD PTR SS:[EBP-1C],EAX
7C910636 8440 00 MOV EAX,DWORD PTR SS:[EBP+8]
7C910639 8440 44FCFFFF MOV DWORD PTR SS:[EBP-8C],EAX
7C91063C 8B75 0C MOV EDI,DWORD PTR SS:[EBP+C]
7C91063F 8B5D 14 MOV ESI,DWORD PTR SS:[EBP+14]
7C910642 8B45 18 MOV EAX,DWORD PTR SS:[EBP+18]
7C910645 8B5E 34FCFFFF MOV DWORD PTR SS:[EBP-3C],EAX
7C910648 8B4E 20 MOV EAX,DWORD PTR SS:[EBP+20]
7C91064B 8445 38FCFFFF MOV DWORD PTR SS:[EBP-30],EAX
7C91064E 3302 XOR EDI,EDI
7C910651 8445 48FCFFFF MOV DWORD PTR SS:[EBP-38],EAX
7C910654 3302 XOR EDI,EDI
7C910657 8B50 4CFCFFFF LEA EDI,DWORD PTR SS:[EBP-34]
7C91065A 8B4B STOS DWORD PTR ES:[EDI]
7C91065D 8B5E 28FCFFFF LEA EDI,DWORD PTR SS:[EBP-30]
7C910660 58 PUSH EDI
7C910663 7445 1E 01 TEST BYTE PTR SS:[EBP+1E],1
7C910666 7545 0940200 JNC kernel32.7C818A89
7C910669 52 PUSH EDI
7C91066C 8B45 10 MOV DWORD PTR SS:[EBP+10]
7C91066F 51 PUSH EAX
7C910672 58 50DFFFF CALL kernel32.BaseCreateStack(x,x,x,x)
7C910675 8B50 MOV EAX,EAX
7C910678 7545 0840200 JNC kernel32.7C818A92
7C91067B 38FF XOR EDI,EDI
7C91067E 47 INC EDI
7C910681 57 PUSH EDI
7C910684 57 PUSH EDI
7C910687 8B5E 28FCFFFF MOV DWORD PTR SS:[EBP-30]
```

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**.

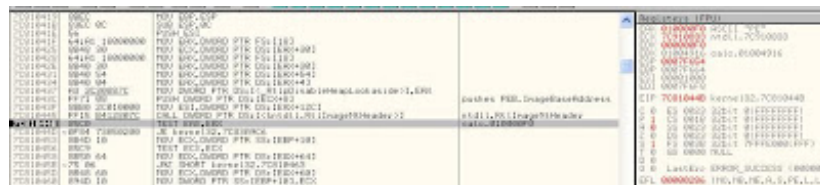
N.B. If you call the "**CreateRemoteThread**" function with the "**dwStackSize**" parameter set to e.g. **0x10000**, then **BaseCreateStack** commits **0x10000** bytes. On the other side, if the "**CreateRemoteThread**" function is called with the "**dwCreationFlags**" parameter having the

"**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);
StackStartAddress-=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.

```

75843F92  F7D1          NOT ECX
75843F94  234D 10      AND ECX, DWORD PTR SS:[EBP+10]
75843F97  51          PUSH ECX
75843F98  F7D8          NEG EBX
75843F9A  1E03          SBB EBX, EBX
75843F9C  2345 10      AND EBX, DWORD PTR SS:[EBP+10]
75843F9F  58          PUSH EBX
75843FA0  53          PUSH EBX
75843FA1  56          PUSH ESI
75843FA2  FFB5 90FFFFFF PUSH DWORD PTR SS:[EBP-160]
75843FA3  FFB5 B0FFFFFF PUSH DWORD PTR SS:[EBP-150]
75843FAE  FFB5 D0FFFFFF PUSH DWORD PTR SS:[EBP-140]
75843FB4  FFB5 94FFFFFF PUSH DWORD PTR SS:[EBP-16C]
75843FB8  68 FFFF1F00  PUSH 1FFFFFFF
75843FBF  0D05 C4FFFFFF LEA EBX, DWORD PTR SS:[EBP-13C]
75843FC5  58          PUSH EBX
75843FC6  FF15 64198375 CALL DWORD PTR DS:[&ntdll.NtCreateThreadEx], &ZwCreateThreadEx

```

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

```

PAGE:00658831 pTHREAD      = dword ptr 8
PAGE:00658831 DesiredAccess = dword ptr 0Ch
PAGE:00658831 Obj_Attr     = dword ptr 10h
PAGE:00658831 hProcess    = dword ptr 14h
PAGE:00658831 lpStartAddress = dword ptr 18h
PAGE:00658831 lpParameter  = dword ptr 1Ch
PAGE:00658831 bCreateSuspended = dword ptr 20h
PAGE:00658831 StackZeroBits = dword ptr 24h
PAGE:00658831 SizeOfStackCommit = dword ptr 28h
PAGE:00658831 SizeOfStackReserve = dword ptr 2Ch
PAGE:00658831 lpBytesBuffer = dword ptr 30h
PAGE:00658831              push 30Ch
PAGE:00658836              push offset stru .44EC98
PAGE:0065883B              call ___SEH_prolog@_GS
PAGE:00658840              mov     eax, [ebp+pTHREAD]
PAGE:00658843              mov     [ebp+var_300], eax

```

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

```

PAGE:006589FA              call   _PspCreateUserContext@20 ; PspCreateUserContext(x,x,x,x,x)
PAGE:006589FB              lea   eax, [ebp+var_300]
PAGE:006589FC              push  eax
PAGE:006589FD              push  eax
PAGE:006589FE              push  ebx
PAGE:00658A00              push  [ebp+bCreateSuspended]
PAGE:00658A03              lea   eax, [ebp+var_300]
PAGE:00658A06              push  eax
PAGE:00658A0C              push  [ebp+var_08]
PAGE:00658A14              push  esi
PAGE:00658A15              push  ebx
PAGE:00658A16              push  [ebp+var_30C]
PAGE:00658A1C              push  [ebp+var_304]
PAGE:00658A22              push  [ebp+DesiredAccess]
PAGE:00658A25              push  [ebp+var_308]
PAGE:00658A28              lea   ebx, [ebp+var_30C]
PAGE:00658A31              lea   ecx, [ebp+ppStruct]
PAGE:00658A37              call  _PspCreateThread@58 ; PspCreateThread(x,x,x,x,x,x,x,x,x,x,x,x)
PAGE:00658A3C              mov   esi, eax
PAGE:00658A3E DeleteCreateProc@01          ; CODE REF: NtCreateThreadEx(x,x,x,x,x,x,x,x,x,x)+15A7F]
PAGE:00658A3E              lea   eax, [ebp+ppStruct]
PAGE:00658A44              call  _PspDeleteCreateProcessContext@0 ; PspDeleteCreateProcessContext(x)

```

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

```

PAGE:0061A4DB              lea   eax, [esp+19Ch+var_100]
PAGE:0061A4DF              push  eax
PAGE:0061A4E0              push  [esp+180h+var_168]
PAGE:0061A4E3              push  [ebp+arg_24]
PAGE:0061A4E7              push  [esp+180h+var_16C]
PAGE:0061A4EB              push  [esp+180h+var_15C]
PAGE:0061A4EF              push  [esp+180h+var_17C]
PAGE:0061A4F3              push  [esp+184h+var_14C]
PAGE:0061A4F7              push  ebx
PAGE:0061A4F8              call  _PspAllocateThread@98 ; PspAllocateThread(x,x,x,x,x,x,x,x,x,x)
PAGE:0061A4FD              mov   [esp+190h+var_17C], eax
PAGE:0061A501              test  eax, eax

PAGE:0061A051 loc_61A051          ; CODE REF: PspAllocateThread(x,x,x,x,x,x,x,x,x,x)+2597]
PAGE:0061A051              lea   eax, [ebp+var_0C]
PAGE:0061A053              push  eax
PAGE:0061A055              push  [ebp+arg_0]
PAGE:0061A058              call  _RtlStackAttachProcess@0 ; RtlStackAttachProcess(x,x)
PAGE:0061A05D              push  [ebp+arg_10]
PAGE:0061A058              push  [ebp+arg_1C]
PAGE:0061A063              push  dword ptr [esi+k]
PAGE:0061A066              push  dword ptr [esi+0Ch]
PAGE:0061A069              push  dword ptr [esi+8]
PAGE:0061A06C              call  _RtlCreateUserStack@24 ; RtlCreateUserStack(x,x,x,x,x)
PAGE:0061A06E              mov   esi, eax
PAGE:0061A073              test  esi, esi

```

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

```

PAGE:0065B078 mov     eax, large fs:[12Ah]; ETHREAD
PAGE:0065B07B mov     eax, [eax+50h]; EPROCESS
PAGE:0065B083 mov     eax, [eax+12Ch]; SectionBasedAddress
PAGE:0065B089 and     [ebp+var_2A], 0
PAGE:0065B08B and     [ebp+ms_exc.registration.TrypLow], 0
PAGE:0065B091 push   eax; SectionBasedAddress
PAGE:0065B092 call   _RtlImageNtHeader@4; RtlImageNtHeader(x)
PAGE:0065B097 test   eax, eax
PAGE:0065B099 jz     short BadExecFormat
PAGE:0065B099 mov     ecx, [eax+40h]; Ecx is now SizeOfStackCommit
PAGE:0065B09E mov     [ebp+loc_SizeOfStackCommit], ecx
PAGE:0065B0A1 mov     ebx, [eax+60h]; Ebx is now SizeOfStackReserve
PAGE:0065B0A4 mov     [ebp+loc_SizeOfStackReserve], ebx
PAGE:0065B0A7 jmp     short NEXT

```

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 **DbgUilssueRemoteBreakin** 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.

```

PAGE:0065B0E0 test   edi, edi; EDI holds ArgCommitSz
PAGE:0065B0E2 jnz   short CommitSzPassed; ESI holds ArgReserveSz
PAGE:0065B0E2 mov   edi, [ebp+loc_SizeOfStackCommit]; Take from PE header
PAGE:0065B0E2 CommitSzPassed:; CODE BRIF: RtlCreateUserStack(x,x,x,x,x,x)*C7F
PAGE:0065B0E2 test   esi, esi; ESI holds ArgReserveSz
PAGE:0065B0E4 jnz   short ReserveSzPassed
PAGE:0065B0E4 mov   esi, [ebp+loc_SizeOfStackReserve]; Take from PE header
PAGE:0065B0E4 ReserveSzPassed:; CODE BRIF: RtlCreateUserStack(x,x,x,x,x,x)*52F
PAGE:0065B0E4; RtlCreateUserStack(x,x,x,x,x,x)*CE7
PAGE:0065B0E9 test   edi, edi
PAGE:0065B0EB jnz   short CommitSzPassed2
PAGE:0065B0EB mov   edi, 4000h; 0x4000 is now the default Commit Size
PAGE:0065B102

```

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**).

```

PAGE:0065B102 CommitSzPassed2:; CODE BRIF: RtlCreateUserStack(x,x,x,x,x,x)*95F
PAGE:0065B102 cmp   edi, esi
PAGE:0065B104 jb   short CommitLessThanReserve
PAGE:0065B106 lea   esi, [edi+00177h]
PAGE:0065B10C and   esi, 0FFF0000h
PAGE:0065B112

```

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.

```
PAGE:0050127 mov [ebp+ms_exc.registration.trylevel], 1
PAGE:0050130 mov ecx, [ebp+PEB_base]
PAGE:0050133 mov ecx, [ecx+200h]
PAGE:0050139 mov [ebp+MinimumStackCommit], ecx
PAGE:0050140 mov [ebp+ms_exc.registration.trylevel], #FFFFFFh
PAGE:0050143 xor edx, edx
PAGE:0050145 cmp ecx, edx ; ECK is PEB->MinimumStackCommit
PAGE:0050147 jz short ThatSOkay
PAGE:0050149 cmp edi, ecx ; ECK is PEB->MinimumStackCommit
PAGE:0050149 ; EDI is SizeOfStackCommit
PAGE:005014B jnb short ThatSOkay
PAGE:005014D lea edi, [ecx+ebx-1]
PAGE:0050151 and edi, eax
PAGE:0050153 lea esi, [edi+#FFFFFFh]
PAGE:0050159 and esi, #FFF0000h
PAGE:0050165
```

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.

```
PAGE:005015F ; RtlCreateUserStack(x,x,x,x,x)+125f]
PAGE:0050161 mov eax, [ebp+var_19]
PAGE:0050163 mov [ebp+HEH1], eax
PAGE:0050166 mov [ebp+HEH1], eax
PAGE:0050169 mov [ebp+HEH2], eax
PAGE:005016C mov [ebp+HEH2], eax
PAGE:005016F mov [ebp+HEH3], esi ; Here the reserve size is passed
PAGE:0050172 mov [ebp+var_9], eax
PAGE:0050175 mov [ebp+HEH5], eax
PAGE:0050178 push 1Ch
PAGE:005017A lea eax, [ebp+HEH6]
PAGE:005017D push eax
PAGE:005017E push 20h ; ProcessThreadStackAllocation
PAGE:0050181 push #FFFFFFh
PAGE:0050182 call ZwSetInformationProcess@16 ; ZwSetInformationProcess(x,x,x,x)
PAGE:0050187 mov eax, eax
```

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.

```
PAGE:0050E027 or eax, [esi+0]
PAGE:0050E028 or eax, [esi+0]
PAGE:0050E02B jnz NotSelfError
PAGE:0050E02E add esi, 10h
PAGE:0050E16 lea_400F16: ; CODE XREF: NtSetInformationProcess(x,x,x,x)+102Dj]
PAGE:0050E16 ; NtSetInformationProcess(x,x,x,x)+1050j]
PAGE:0050E18 cmp dword ptr [esi], 0
PAGE:0050E19 jz NotSelfError
PAGE:0050E1F lea eax, [ebp+var_1C0]
PAGE:0050E25 push eax
PAGE:0050E26 call _RtlQuerySystemTime@4 ; RtlQuerySystemTime(x)
PAGE:0050E28 rdtsc
PAGE:0050E2D mov ecx, [ebp+var_1C0]
PAGE:0050E30 add ecx, eax
PAGE:0050E35 mov eax, [ebp+var_1BC]
PAGE:0050E38 adc ecx, edx
PAGE:0050E3D and ecx, 1Fh
PAGE:0050E40 inc ecx
PAGE:0050E41 mov [ebp+var_1C], ecx
PAGE:0050E44 mov edx, [esi] ; Read Reserve Size from
PAGE:0050E46 mov [ebp+reserveSize], edx
PAGE:0050E75 lea eax, [esi+0]
PAGE:0050E78 push eax
PAGE:0050E79 push ecx
PAGE:0050E7A push [ebp+var_1C]
PAGE:0050E7B mov eax, eax
PAGE:0050E7E mov ecx, edx
PAGE:0050E81 call _NtScanUserAddressSpace@20 ; NtScanUserAddressSpace(x,x,x,x)
PAGE:0050E86 test eax, eax
PAGE:0050E88 jl loc_400F81
PAGE:0050E8C push 4
PAGE:0050E8F mov eax, edi
PAGE:0050E92 or eax, 2000h
PAGE:0050E95 push eax
PAGE:0050E98 push esi ; ESI points at ReserveSize
PAGE:0050E9B push dword ptr [esi+4]
PAGE:0050E9E lea eax, [esi+0]
PAGE:0050EA1 push eax ; EAX points at BaseAddress
PAGE:0050EA4 push #FFFFFFh
PAGE:0050EA7 call _ZwAllocateVirtualMemory@24 ; ZwAllocateVirtualMemory(x,x,x,x,x)
PAGE:0050EAB jmp loc_400F8D
```

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](#)