Windows Anti-Debug Reference

Nicolas Falliere 2007-09-12

Intro

This paper classifies and presents several anti-debugging techniques used on Windows NT-based operating systems.
Anti-debugging techniques are ways for a program to detect if it runs under control of a debugger. They are used by commercial executable protectors, packers and malicious software, to prevent or slow-down the process of reverse-engineering.

We'll suppose the program is analyzed under a ring3 debugger, such as OllyDbg on Windows platforms. The paper is aimed towards reverse-engineers and malware analysts.
Note that we will talk purely about generic anti-debugging and anti-tracing techniques. Specific debugger detection, such as window or processes enumeration, registry scanning, etc. will not be addressed here.

Anti-debugging and anti-tracing techniques

Exploiting memory discrepancies

(1) kernel32!IsDebuggerPresent

IsDebuggerPresent returns 1 if the process is being debugged, 0 otherwise. This API simply reads the PEB!BeingDebugged byte-flag (located at offset 2 in the PEB structure).
Circumventing it is as easy as setting PEB!BeingDebugged to 0.
Example:

```
call IsDebuggerPresent
test eax, eax
jne @DebuggerDetected
...
```

(2) PEB!IsDebugged

This field refers to the second byte in the Process Environment Block of the process. It is set by the system when the process is debugged.
This byte can be reset to 0 without consequences for the course of execution of the program (it is an informative flag).

Example:

```
mov eax, fs:[30h]
mov eax, byte [eax+2]
test eax, eax
jne @DebuggerDetected
...
```

(3) PEB!NtGlobalFlags

When a process is created, the system sets some flags that will define how various APIs will behave for this program. Those flags can be read in the PEB, in the DWORD located at offset 0x68 (see the reference).
By default, different flags are set depending if the process is created under a debugger or not. If the

process is debugged, some flags controlling the heap manipulation routines in ntdll will be set: *FLG_HEAP_ENABLE_TAIL_CHECK*, *FLG_HEAP_ENABLE_FREE_CHECK* and *FLG_HEAP_VALIDATE_PARAMETERS*.
This anti-debug can be bypassed by resetting the NtGlobalFlags field.

Example:
```
mov eax, fs:[30h]
mov eax, [eax+68h]
and eax, 0x70
test eax, eax
jne @DebuggerDetected
...
```

(4) Heap flags

As explained previously, NtGlobalFlags informs how the heap routines will behave (among other things). Though it is easy to modify the PEB field, if the heap does not behave the same way as it should when the process is not debugged, this could be problematic. It is a powerful anti-debug, as process heaps are numerous, and their chunks can be individually affected by the FLG_HEAP_* flags (such as chunk tails). Heap headers would be affected as well. For instance, checking the field ForceFlags in a heap header (offset 0x10) can be used to detect the presence of a debugger.

There are two easy ways to circumvent it:

- Create a non-debugged process, and attach the debugger once the process has been created (an easy solution is to create the process suspended, run until the entry-point is reached, patch it to an infinite loop, resume the process, attach the debugger, and restore the original entry-point).

- Force the NtGlobalFlags for the process that we want to debug, via the registry key "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options": Create a subkey (not value) named as your process name, and under this subkey, a String value "GlobalFlags" set to nothing.

Example:
```
mov eax, fs:[30h]
mov eax, [eax+18h] ;process heap
mov eax, [eax+10h] ;heap flags
test eax, eax
jne @DebuggerDetected
...
```

(5) Vista anti-debug (no name)

Here's an anti-debug specific to Windows Vista that I found by comparing memory dumps of a program running with and without control of a debugger. I'm not sure of its realiability, but it's worth mentionning (tested on Windows Vista 32 bits, SP0, English version).

When a process is debugged, its main thread TEB, at offset 0xBFC, contains a pointer to a unicode string referencing a system dll. Moreover, the string follows this pointer (therefore, located at offset 0xC00 in the TEB). If the process is not debugged, the pointer is set to NULL and the string is not present.

Example:

```
call GetVersion
cmp al, 6
jne @NotVista
push offset _seh
push dword fs:[0]
mov fs:[0], esp
mov eax, fs:[18h] ; teb
add eax, 0BFCh
mov ebx, [eax] ; pointer to a unicode string
test ebx, ebx ; (ntdll.dll, gdi32.dll,...)
je @DebuggerNotFound
sub ebx, eax ; the unicode string follows the
sub ebx, 4 ; pointer
jne @DebuggerNotFound
;debugger detected if it reaches this point
;...
```

Exploiting system discrepancies

(1) NtQueryInformationProcess

ntdll!NtQueryInformationProcess is a wrapper around the ZwQueryInformationProcess syscall. Its prototype is the following:

```
NTSYSAPI NTSTATUS NTAPI NtQueryInformationProcess(
IN HANDLE ProcessHandle,
IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
OUT PVOID ProcessInformation,
IN ULONG ProcessInformationLength,
OUT PULONG ReturnLength
);
```

When called with ProcessInformationClass set to 7 (ProcessDebugPort constant), the system will set ProcessInformation to -1 if the process is debugged.
It is a powerful anti-debug, and there is no easy way to circumvent it. However, if the program is traced, ProcessInformation can be modified when the syscall returns.

Another solution is to use a system driver that would hook the ZwNtQueryInformationProcess syscall.
Circumventing NtQueryInformationProcess will bypass many anti-debug techniques (such as CheckRemoteDebuggerPresent or UnhandledExceptionFilter).

Example:

```
push 0
push 4
push offset isdebugged
push 7 ;ProcessDebugPort
push -1
call NtQueryInformationProcess
test eax, eax
jne @ExitError
```

```
cmp isdebugged, 0
jne @DebuggerDetected
...
```

(2) kernel32!CheckRemoteDebuggerPresent

This API takes two parameters: a process handle, and a pointer to a DWORD. If the call is successful, the DWORD value will be set to 1 if the process is being debugged.
Internally, this API calls ntdll!NtQueryInformationProcess with ProcessInformationClass set to ProcessDebugPort (7).

Example:

```
push offset isdebugged
push -1
call CheckRemoteDebuggerPresent
test eax, eax
jne @DebuggerDetected
...
```

(3) UnhandledExceptionFilter

When an exception occurs, with Windows XP SP>=2, Windows 2003, and Windows Vista, the usual way the OS processes the exception is:

- If any, pass control to the per-process Vectored Exception Handlers.
- If the exception is not processed, pass the control to the per-thread top SEH handler, pointed by FS:[0] in the thread that generated the exception. SEH are chained and called in turn if the exception is not processed by the previous in the chain.
- If the exception has not been processed by any of the previous handlers, the final SEH handler (set by the system), will call kernel32!UnhandledExceptionFilter. This function will decide what it should do depending if the process is debugged or not.
- If it is not debugged, it will call the user-defined filter function (set via kernel32!SetUnhandledExceptionFilter).
- If it debugged, the program will be terminated.

The debugger detection in UnhandledExceptionFilter is made with ntdll!NtQueryInformationProcess.

Example:

```
push @not_debugged
call SetUnhandledExceptionFilter
xor eax, eax
mov eax, dword [eax] ; trigger exception
;program terminated if debugged
;...
@not_debugged:
;process the exception
;continue the execution
;...
```

(4) NtSetInformationThread

ntdll!NtSetInformationThread is a wrapper around the ZwSetInformationThread syscall. Its prototype is the following:

```
NTSYSAPI NTSTATUS NTAPI NtSetInformationThread(
IN HANDLE ThreadHandle,
IN THREAD_INFORMATION_CLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);
```

When called with ThreadInformationClass set to 0x11 (ThreadHideFromDebugger constant), the thread will be detached from the debugger.

Similarly to ZwQueryInformationProcess, circumventing this anti-debug requires either modifying ZwSetInformationThread parameters before it's called, or hooking the syscall directly with the use of a kernel driver.

Example:

```
push 0
push 0
push 11h ;ThreadHideFromDebugger
push -2
call NtSetInformationThread
;thread detached if debugged
;...
```

(5) kernel32!CloseHandle and NtClose

APIs making user of the ZwClose syscall (such as CloseHandle, indirectly) can be used to detect a debugger. When a process is debugged, calling ZwClose with an invalid handle will generate a STATUS_INVALID_HANDLE (0xC0000008) exception.

As with all anti-debugs that rely on information made directly available from the kernel (therefore involving a syscall), the only proper way to bypass the "CloseHandle" anti-debug is to either modify the syscall data from ring3, before it is called, or set up a kernel hook.

This anti-debug, though extremely powerful, does not seem to be widely used by malicious programs.

Example:

```
push offset @not_debugged
push dword fs:[0]
mov fs:[0], esp
push 1234h ;invalid handle
call CloseHandle
; if fall here, process is debugged
;...
@not_debugged:
;...
```

(6) Self-debugging

A process can detect it is being debugged by trying to debug itself, for instance by creating a new process, and calling kernel32!DebugActiveProcess(pid) on the parent process.

In turn, this API calls ntdll!DbgUiDebugActiveProcess which will call the syscall ZwDebugActiveProcess. If the process is already debugged, the syscall fails. Note that retrieving the parent process PID can be done with the toolhelp32 APIs (field th32ParentProcessID in the PROCESSENTRY32 structure.

(7) Kernel-mode timers

kernel32!QueryPerformanceCounter is an efficent anti-debug. This API calls ntdll!NtQueryPerformanceCounter which wraps the ZwQueryPerformanceCounter syscall.

Again, there is no easy way to circumvent this anti-tracing trick.

(8) User-mode timers

An API such as kernel32!GetTickCount returns the number of milliseconds ellapsed since the system started. The interesting thing is that it does not make use of kernel-related service to perform its duties. A user-mode process has this counter mapped in its address space. For 8Gb user-mode spaces, the value returned would be:

d[0x7FFE0000] * d[0x7FFE0004] / (2^24)

(9) kernel32!OutputDebugStringA

This anti-debug is quite original, I have encountered it only once, in files packed with ReCrypt v0.80. The trick consists of calling OutputDebugStringA, with a valid ASCII string. If the program is run under control of a debugger, the return value will be the address of the string passed as a parameter. In normal conditions, the return value should be 1.

Example:

```
xor eax, eax
push offset szHello
call OutputDebugStringA
cmp eax, 1
jne @DebuggerDetected
...
```

(10) Ctrl-C

When a console program is debugged, a Ctrl-C signal will throw a EXCEPTION_CTL_C exception, whereas the signal handler would be called directly is the program is not debugged.

Example:

```
push offset exhandler
push 1
call RtlAddVectoredExceptionHandler
push 1
```

```
push sighandler
call SetConsoleCtrlHandler
push 0
push CTRL_C_EVENT
call GenerateConsoleCtrlEvent
push 10000
call Sleep
push 0
call ExitProcess
exhandler:
;check if EXCEPTION_CTL_C, if it is,
;debugger detected, should exit process
;...
sighandler:
;continue
;...
```

CPU anti-debug

(1) Rogue Int3

This is a classic anti-debug to fool weak debuggers. It consists of inserting an INT3 opcode in the middle of a valid sequence of instructions. When the INT3 is executed, if the program is not debugged, control will be given to the exception handler of the protection and execution will continue.
As INT3 instructions are used by debuggers to set software breakpoints, inserting INT3 opcodes can be used to trick the debugger into believing that it is one his breakpoints. Therefore, the control would not be given to the exception handler, and the course of the program would be modified. Debuggers should track where they set software breakpoints to avoid falling for this one.

Similarly, note that INT3 may be encoded as 0xCD, 0x03.

Example:

```
push offset @handler
push dword fs:[0]
mov fs:[0], esp
;...
db 0CCh
;if fall here, debugged
;...
@handler:
;continue execution
;...
```

(2) "Ice" Breakpoint

The so-called "Ice breakpoint" is one of Intel's undocumented instruction, opcode 0xF1. It is used to detect tracing programs.

Executing this instruction will generate a SINGLE_STEP exception. Therefore, if the program is already traced, the debugger will think it is the normal exception generated by executing the instruction with the SingleStep bit set in the Flags registers. The associated exception handler won't

be executed, and execution will not continue as expected.
Bypassing this trick is easy: one can run over the instruction, instead and single-stepping on it. The exception will be generated, but since the program is not traced, the debugger should understand that it has to pass control to the exception handler.

Example:

```
push offset @handler
push dword fs:[0]
mov fs:[0], esp
;...
db 0F1h
;if fall here, traced
;...
@handler:
;continue execution
;...
```

(3) Interrupt 2Dh

Executing this interrupt if the program is not debugged will raise a breakpoint exception. If the program is debugged, and the instruction is not executed with the trace flag, no exception will be generated, and execution will carry on normally. If the program is debugged and the instruction traced, the following byte will be skipped, and execution will continue. Therefore, using INT 2Dh can be used as a powerful anti-debug and anti-tracer mechanism.

Example:

```
push offset @handler
push dword fs:[0]
mov fs:[0], esp
;...
db 02Dh
mov eax, 1 ;anti-tracing
;...
@handler:
;continue execution
;...
```

(4) Timestamp counters

High precision counters, storing the current number of CPU cycles executed since the machine started, can be queried with the RDTSC instruction. Classic anti-debugs consist of measuring time deltas at key points in the program, usually around exception handlers. If the delta is too large, that would mean the program runs under control of a debugger (processing the exception in the debugger, and giving control back to the debuggee is a lengthy task).

Example:

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
rdtsc
push eax
xor eax, eax
```

```
div eax ;trigger exception
rdtsc
sub eax, [esp] ;ticks delta
add esp, 4
pop fs:[0]
add esp, 4
cmp eax, 10000h ;threshold
jb @not_debugged
@debugged:
...
@not_debugged:
...
handler:
mov ecx, [esp+0Ch]
add dword ptr [ecx+0B8h], 2 ;skip div
xor eax, eax
ret
```

(5) Popf and the trap flag

The trap flag, located in the Flags register, controls the tracing of a program. If this flag is set, executing an instruction will also raise a SINGLE_STEP exception. The trap flag can be manipulated in order to thwart tracers. For instance, this sequence of instructions will set the trap flag:

```
pushf
mov dword [esp], 0x100
popf
```

If the program is being traced, this will have no real effect on the flags register, and the debugger will process the exception, believing it comes from regular tracing. The exception handler won't be executed. Circumventing this anti-tracer trick simply require to run over the pushf instruction.

(6) Stack Segment register

Here's a very original anti-tracer. I encountered it in a packer called MarCrypt. I believe it is not widely known, not to mention, used.
It consists of tracing over this sequence of instructions:

```
push ss
pop ss
pushf
nop
```

When tracing over pop ss, the next instruction will be executed but the debugger will not break on it, therefore stopping on the following instruction (NOP in this case).
Marcrypt uses this anti-debug the following way:

```
push ss
; junk
pop ss
pushf
```

```
; junk
pop eax
and eax, 0x100
or eax, eax
jnz @debugged
; carry on normal execution
```

The trick here is that, if the debugger is tracing over that sequence of instructions, popf will be excuted implicitly, and the debugger will not be able to unset the trapflag in the pushed value on the stack. The protection checks for the trap flag and terminates the program if it's found.
One simple way to circumvent this anti-tracing is to breakpoint on popf and run the program (to avoid using the TF flag).

(7) Debug registers manipulation

Debug registers (DR0 through DR7) are used to set hardware breakpoints. A protection can manipulate them to either detect that hardware breakpoints have been set (and therefore, that it is being debugged), reset them or set them to particular values used to perform code checks later. A packer such as tElock makes use of the debug registers to prevent reverse-engineers from using them.
From a user-mode perspective, debug registers cannot be set using the privileged 'mov drx, ...' instruction. Other ways exist:

- An exception can be generated, the thread context modified (it contains the CPU registers at the time the exception was thrown), and then resumed to normal execution with the new context.

- The other way is to use the NtGetContextThread and NtSetContextThread syscalls (available in kernel32 with GetThreadContext and SetThreadContext).

Most protectors use the first, "unofficial" way.

Example:

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
xor eax, eax
div eax ;generate exception
pop fs:[0]
add esp, 4
;continue execution
;...
handler:
mov ecx, [esp+0Ch] ;skip div
add dword ptr [ecx+0B8h], 2 ;skip div
mov dword ptr [ecx+04h], 0 ;clean dr0
mov dword ptr [ecx+08h], 0 ;clean dr1
mov dword ptr [ecx+0Ch], 0 ;clean dr2
mov dword ptr [ecx+10h], 0 ;clean dr3
mov dword ptr [ecx+14h], 0 ;clean dr6
mov dword ptr [ecx+18h], 0 ;clean dr7
xor eax, eax
ret
```

(8) Context modification

As with debug registers manipulation, the context can also be used to modify in an unconventionnal way the execution stream of a program. Debuggers can get easily confused!
Note that another syscall, NtContinue, can be used to load a new context in the current thread (for instance, this syscall is used by the exception handler manager).

Uncategorized anti-debug

(1) TLS-callback

This anti-debug was not so well-known a few years ago. It consists to instruct the PE loader that the first entry point of the program is referenced in a Thread Local Storage entry (10th directory entry number in the PE optional header). By doing so, the program entry-point won't be executed first. The TLS entry can then perform anti-debug checks in a stealthy way.
Note that in practice, this technique is not widely used.
Though older debuggers (including OllyDbg) are not TLS-aware, counter-measures are quite easy to take, by the means of plugins of custom patcher tools.

(2) CC scanning

A common protection feature used by packers is the CC-scanning loop, aimed at detecting software breakpoints set by a debugger. If you want to avoid that kind of troubles, you may want to use either hardware breakpoints or a custom type of software breakpoint. CLI (0xFA) is a good candidate to replace the classic INT3 opcode. This instruction does have the requirements for the job: it raises a privileged instruction exception if executed by a ring3 program, and occupies only 1 byte of space.

(3) EntryPoint RVA set to 0

Some packed files have their entry point RVA set to 0, which means they will start executing 'MZ...' which corresponds to 'dec ebx / pop edx ...'.

This is not an anti-debug trick in itself, but can be annoying if you want to break on the entry-point by using a software breakpoint.

If you create a suspended process, then set an INT3 at RVA 0, you will erase part of the magic MZ value ('M'). The magic was checked when the process was created, but it will get checked again by ntdll when the process is resumed (in the hope of reaching the entry-point). In that case, an INVALID_IMAGE_FORMAT exception will be raised.

If you create your own tracing or debugging tool, you will want to use hardware breakpoint to avoid this problem.

Conclusion

Knowing anti-debugging and anti-tracing techniques (un)commonly used by malware or protectors is useful knowledge for a reverse-engineer. A program will always have ways to find it is run in a debugger - the same applies for virtual or emulated environments, but since ring3 debuggers are some of the most common analysis tools used, knowing common tricks, and how to bypass them, will always prove useful.

Links

MSDN
Portable Executable Tutorial, Matt Pietrek
Syscall Reference, The Metasploit Project
Undocumented Functions for MS Windows NT/2K
Intel Manuals

- Common exception codes - Microsoft Windows SDK, ntdll.h
- Status codes list (including common exception codes) - Microsoft Windows DDK, ntstatus.h
- Context Structures documentation - Microsoft Windows SDK, ntdll.h

Data reference

CONTEXT structure for IA32 processors

```
struct CONTEXT_IA32
{
// ContextFlags must be set to the appropriate CONTEXT_* flag
// before calling (Set|Get)ThreadContext
DWORD ContextFlags;

// CONTEXT_DEBUG_REGISTERS (not included in CONTEXT_FULL)
DWORD Dr0; // 04h
DWORD Dr1; // 08h
DWORD Dr2; // 0Ch
DWORD Dr3; // 10h
DWORD Dr6; // 14h
DWORD Dr7; // 18h

// CONTEXT_FLOATING_POINT
FLOATING_SAVE_AREA FloatSave;

// CONTEXT_SEGMENTS
DWORD SegGs; // 88h
DWORD SegFs; // 90h
DWORD SegEs; // 94h
DWORD SegDs; // 98h

// CONTEXT_INTEGER
DWORD Edi; // 9Ch
DWORD Esi; // A0h
DWORD Ebx; // A4h
DWORD Edx; // A8h
DWORD Ecx; // ACh
DWORD Eax; // B0h

// CONTEXT_CONTROL
DWORD Ebp; // B4h
DWORD Eip; // B8h
DWORD SegCs; // BCh (must be sanitized)
DWORD EFlags; // C0h
```

```
DWORD Esp; // C4h
DWORD SegSs; // C8h

// CONTEXT_EXTENDED_REGISTERS (processor-specific)
BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

Process Environment Block structure (from The Wine Project)

```
struct PEB
{
BOOLEAN InheritedAddressSpace; // 00
BOOLEAN ReadImageFileExecOptions; // 01
BOOLEAN BeingDebugged; // 02
BOOLEAN SpareBool; // 03
HANDLE Mutant; // 04
HMODULE ImageBaseAddress; // 08
PPEB_LDR_DATA LdrData; // 0c
RTL_UPROCESS_PARAMETERS *ProcessParameters; // 10
PVOID SubSystemData; // 14
HANDLE ProcessHeap; // 18
PRTL_CRITICAL_SECTION FastPebLock; // 1c
PVOID /*PPEBLOCKROUTI*/ FastPebLockRoutine; // 20
PVOID /*PPEBLOCKROUTI*/ FastPebUnlockRoutine; // 24
ULONG EnvironmentUpdateCount; // 28
PVOID KernelCallbackTable; // 2c
PVOID EventLogSection; // 30
PVOID EventLog; // 34
PVOID /*PPEB_FREE_BLO*/ FreeList; // 38
ULONG TlsExpansionCounter; // 3c
PRTL_BITMAP TlsBitmap; // 40
ULONG TlsBitmapBits[2]; // 44
PVOID ReadOnlySharedMemoryBase; // 4c
PVOID ReadOnlySharedMemoryHeap; // 50
PVOID *ReadOnlyStaticServerData; // 54
PVOID AnsiCodePageData; // 58
PVOID OemCodePageData; // 5c
PVOID UnicodeCaseTableData; // 60
ULONG NumberOfProcessors; // 64
ULONG NtGlobalFlag; // 68
BYTE Spare2[4]; // 6c
LARGE_INTEGER CriticalSectionTimeout; // 70
ULONG HeapSegmentReserve; // 78
ULONG HeapSegmentCommit; // 7c
ULONG HeapDeCommitTotalFreeTh; // 80
ULONG HeapDeCommitFreeBlockTh; // 84
ULONG NumberOfHeaps; // 88
ULONG MaximumNumberOfHeaps; // 8c
PVOID *ProcessHeaps; // 90
PVOID GdiSharedHandleTable; // 94
PVOID ProcessStarterHelper; // 98
PVOID GdiDCAttributeList; // 9c
PVOID LoaderLock; // a0
```

```
ULONG OSMajorVersion; // a4
ULONG OSMinorVersion; // a8
ULONG OSBuildNumber; // ac
ULONG OSPlatformId; // b0
ULONG ImageSubSystem; // b4
ULONG ImageSubSystemMajorVersion; // b8
ULONG ImageSubSystemMinorVersion; // bc
ULONG ImageProcessAffinityMask; // c0
ULONG GdiHandleBuffer[34]; // c4
ULONG PostProcessInitRoutine; // 14c
PRTL_BITMAP TlsExpansionBitmap; // 150
ULONG TlsExpansionBitmapBits[32]; // 154
ULONG SessionId; // 1d4
};
```

Thread Environment Block structure (from The Wine Project)

```
struct TEB
{
NT_TIB Tib; // 000 Info block
PVOID EnvironmentPointer; // 01c
CLIENT_ID ClientId; // 020 PID,TID
PVOID ActiveRpcHandle; // 028
PVOID ThreadLocalStoragePointer; // 02c
PEB *Peb; // 030
DWORD LastErrorValue; // 034
ULONG CountOfOwnedCriticalSections; // 038
PVOID CsrClientThread; // 03c
PVOID Win32ThreadInfo; // 040
ULONG Win32ClientInfo[0x1f]; // 044
PVOID WOW32Reserved; // 0c0
ULONG CurrentLocale; // 0c4
ULONG FpSoftwareStatusRegister; // 0c8
PVOID SystemReserved1[54]; // 0cc
PVOID Spare1; // 1a4
LONG ExceptionCode; // 1a8
BYTE SpareBytes1[40]; // 1ac
PVOID SystemReserved2[10]; // 1d4
DWORD num_async_io; // 1fc
ULONG_PTR dpmi_vif; // 200
DWORD vm86_pending; // 204
DWORD pad6[309]; // 208
ULONG gdiRgn; // 6dc
ULONG gdiPen; // 6e0
ULONG gdiBrush; // 6e4
CLIENT_ID RealClientId; // 6e8
HANDLE GdiCachedProcessHandle; // 6f0
ULONG GdiClientPID; // 6f4
ULONG GdiClientTID; // 6f8
PVOID GdiThreadLocaleInfo; // 6fc
PVOID UserReserved[5]; // 700
PVOID glDispachTable[280]; // 714
ULONG glReserved1[26]; // b74
```

```
PVOID glReserved2; // bdc
PVOID glSectionInfo; // be0
PVOID glSection; // be4
PVOID glTable; // be8
PVOID glCurrentRC; // bec
PVOID glContext; // bf0
ULONG LastStatusValue; // bf4
UNICODE_STRING StaticUnicodeString; // bf8
WCHAR StaticUnicodeBuffer[261]; // c00
PVOID DeallocationStack; // e0c
PVOID TlsSlots[64]; // e10
LIST_ENTRY TlsLinks; // f10
PVOID Vdm; // f18
PVOID ReservedForNtRpc; // f1c
PVOID DbgSsReserved[2]; // f20
ULONG HardErrorDisabled; // f28
PVOID Instrumentation[16]; // f2c
PVOID WinSockData; // f6c
ULONG GdiBatchCount; // f70
ULONG Spare2; // f74
ULONG Spare3; // f78
ULONG Spare4; // f7c
PVOID ReservedForOle; // f80
ULONG WaitingOnLoaderLock; // f84
PVOID Reserved5[3]; // f88
PVOID *TlsExpansionSlots; // f94
};
```

NtGlobalFlags

```
FLG_STOP_ON_EXCEPTION 0x00000001
FLG_SHOW_LDR_SNAPS 0x00000002
FLG_DEBUG_INITIAL_COMMAND 0x00000004
FLG_STOP_ON_HUNG_GUI 0x00000008
FLG_HEAP_ENABLE_TAIL_CHECK 0x00000010
FLG_HEAP_ENABLE_FREE_CHECK 0x00000020
FLG_HEAP_VALIDATE_PARAMETERS 0x00000040
FLG_HEAP_VALIDATE_ALL 0x00000080
FLG_POOL_ENABLE_TAIL_CHECK 0x00000100
FLG_POOL_ENABLE_FREE_CHECK 0x00000200
FLG_POOL_ENABLE_TAGGING 0x00000400
FLG_HEAP_ENABLE_TAGGING 0x00000800
FLG_USER_STACK_TRACE_DB 0x00001000
FLG_KERNEL_STACK_TRACE_DB 0x00002000
FLG_MAINTAIN_OBJECT_TYPELIST 0x00004000
FLG_HEAP_ENABLE_TAG_BY_DLL 0x00008000
FLG_IGNORE_DEBUG_PRIV 0x00010000
FLG_ENABLE_CSRDEBUG 0x00020000
FLG_ENABLE_KDEBUG_SYMBOL_LOAD 0x00040000
FLG_DISABLE_PAGE_KERNEL_STACKS 0x00080000
FLG_HEAP_ENABLE_CALL_TRACING 0x00100000
FLG_HEAP_DISABLE_COALESCING 0x00200000
FLG_VALID_BITS 0x003FFFFF
```

FLG_ENABLE_CLOSE_EXCEPTION 0x00400000
FLG_ENABLE_EXCEPTION_LOGGING 0x00800000
FLG_ENABLE_HANDLE_TYPE_TAGGING 0x01000000
FLG_HEAP_PAGE_ALLOCS 0x02000000
FLG_DEBUG_WINLOGON 0x04000000
FLG_ENABLE_DBGPRINT_BUFFERING 0x08000000
FLG_EARLY_CRITICAL_SECTION_EVT 0x10000000
FLG_DISABLE_DLL_VERIFICATION 0x80000000