Bypassing patchguard on Windows 8.1 and Windows 10

Mark Ermolov, Artem Shishkin

Positive Technologies

What is patchguard?

— “Please don’t patch our kernels” call from MS

— Even if your kernel patch is correct, you’ll catch a BSOD

- 0x109 CRITICAL_STRUCTURE_CORRUPTION

— Protected structures

- System images: ntoskrnl.exe, win32k.sys, hal.dll etc.
- System structures: IDT, GDT, Syscall tables etc.

— Periodic checksums validation for protected stuff

— Doesn’t work on Windows 9

What if we really need to?

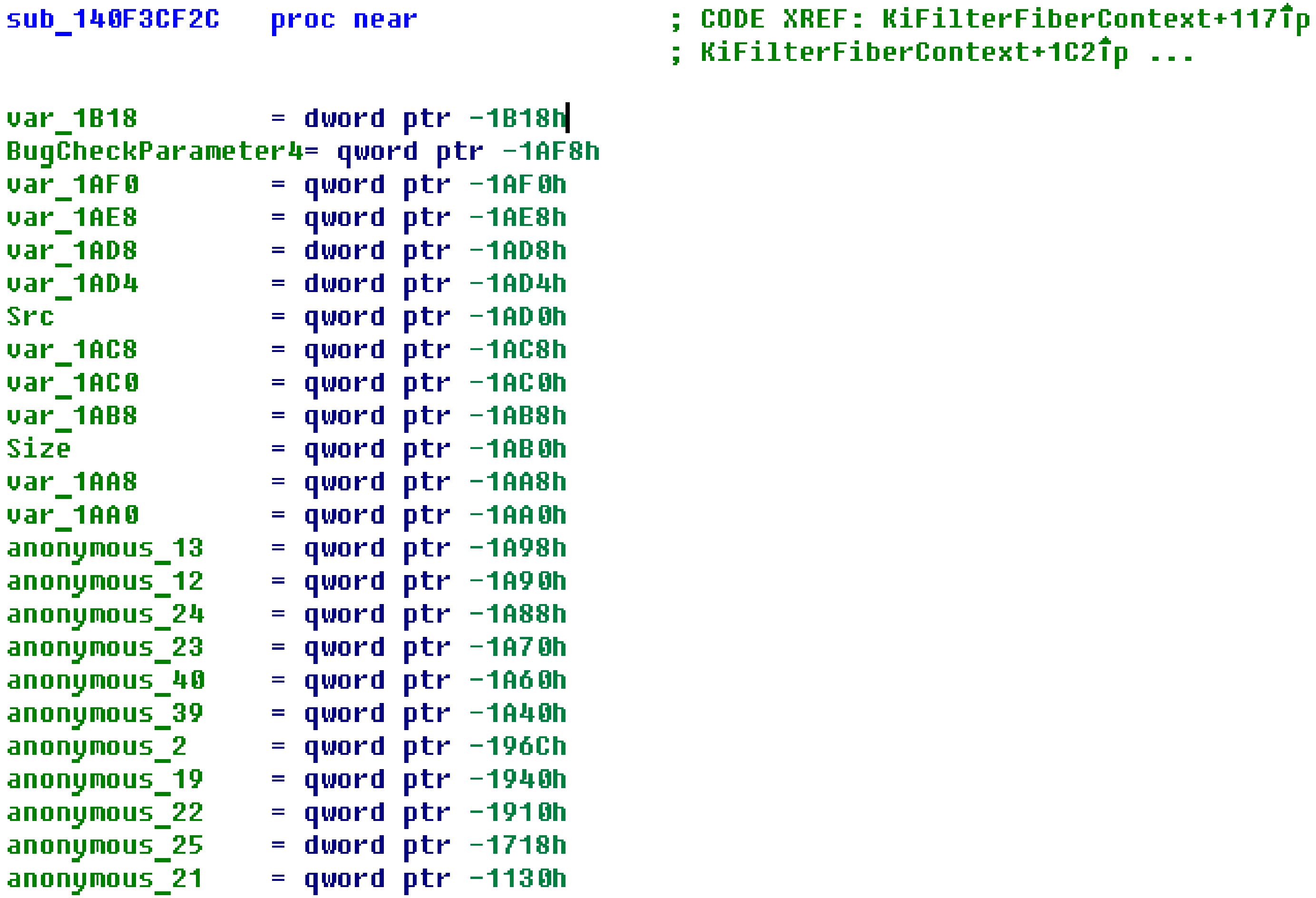
— Go for it!

— But…

- Patchguard developers are prepared for reverse engineers
- Hyper-inlined obfuscation © Alex Ionescu
- Anti-debugging tricks
- Several ways of checks invocation

Code obfuscation

— Symbol stripping

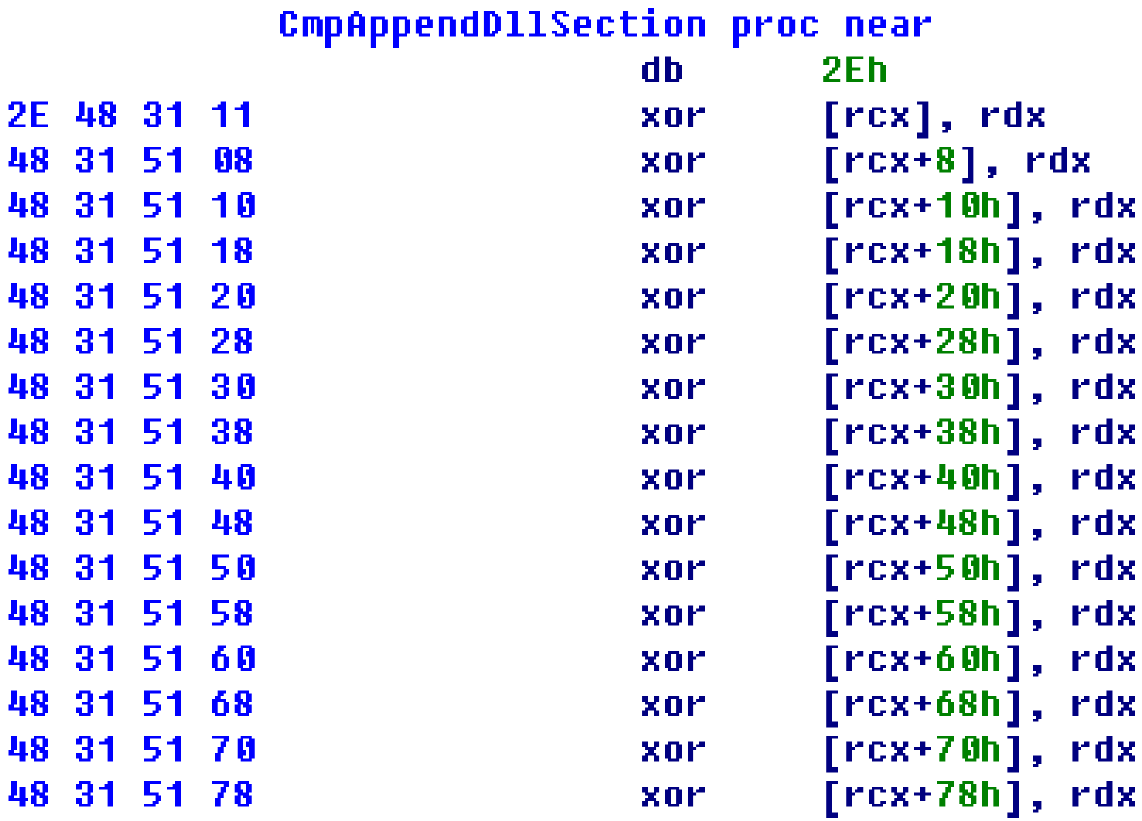


```
sub_140F3CF2C   proc near               ; CODE XREF: KiFilterFiberContext+117↑p
                                        ; KiFilterFiberContext+1C2↑p ...

var_1B18        = dword ptr -1B18h
BugCheckParameter4= qword ptr -1AF8h
var_1AF0        = qword ptr -1AF0h
var_1AE8        = qword ptr -1AE8h
var_1AD8        = dword ptr -1AD8h
var_1AD4        = dword ptr -1AD4h
Src             = qword ptr -1AD0h
var_1AC8        = qword ptr -1AC8h
var_1AC0        = qword ptr -1AC0h
var_1AB8        = qword ptr -1AB8h
Size            = qword ptr -1AB0h
var_1AA8        = qword ptr -1AA8h
var_1AA0        = qword ptr -1AA0h
anonymous_13    = qword ptr -1A98h
anonymous_12    = qword ptr -1A90h
anonymous_24    = qword ptr -1A88h
anonymous_23    = qword ptr -1A70h
anonymous_40    = qword ptr -1A60h
anonymous_39    = qword ptr -1A40h
anonymous_2     = qword ptr -196Ch
anonymous_19    = qword ptr -1940h
anonymous_22    = qword ptr -1910h
anonymous_25    = dword ptr -1718h
anonymous_21    = qword ptr -1130h
```

Code obfuscation

— Misleading names

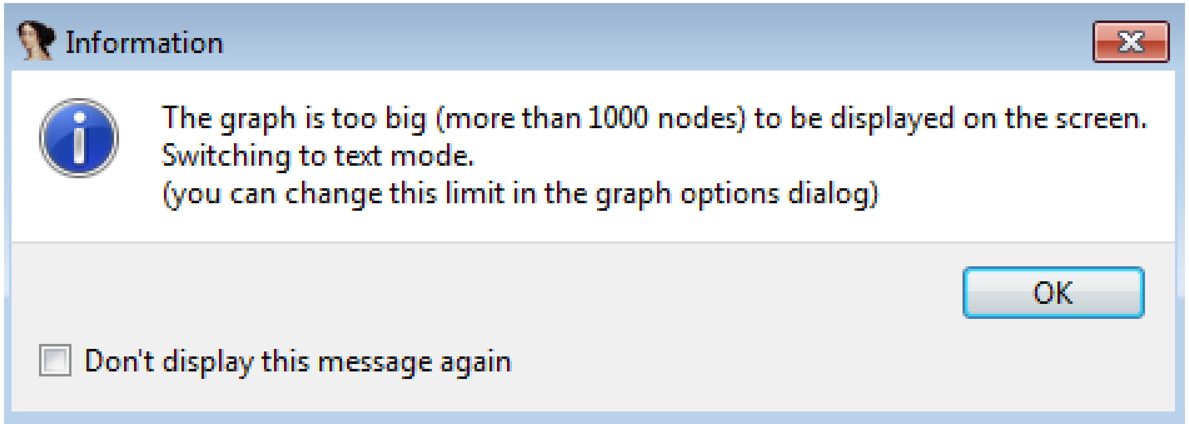


```
            CmpAppendDllSection proc near
                                db      2Eh
2E 48 31 11                     xor     [rcx], rdx
48 31 51 08                     xor     [rcx+8], rdx
48 31 51 10                     xor     [rcx+10h], rdx
48 31 51 18                     xor     [rcx+18h], rdx
48 31 51 20                     xor     [rcx+20h], rdx
48 31 51 28                     xor     [rcx+28h], rdx
48 31 51 30                     xor     [rcx+30h], rdx
48 31 51 38                     xor     [rcx+38h], rdx
48 31 51 40                     xor     [rcx+40h], rdx
48 31 51 48                     xor     [rcx+48h], rdx
48 31 51 50                     xor     [rcx+50h], rdx
48 31 51 58                     xor     [rcx+58h], rdx
48 31 51 60                     xor     [rcx+60h], rdx
48 31 51 68                     xor     [rcx+68h], rdx
48 31 51 70                     xor     [rcx+70h], rdx
48 31 51 78                     xor     [rcx+78h], rdx
```

Code obfuscation

— Code junk generation

- Loop unrolling
- Dead code insertion
- Indirect calls and variable accesses

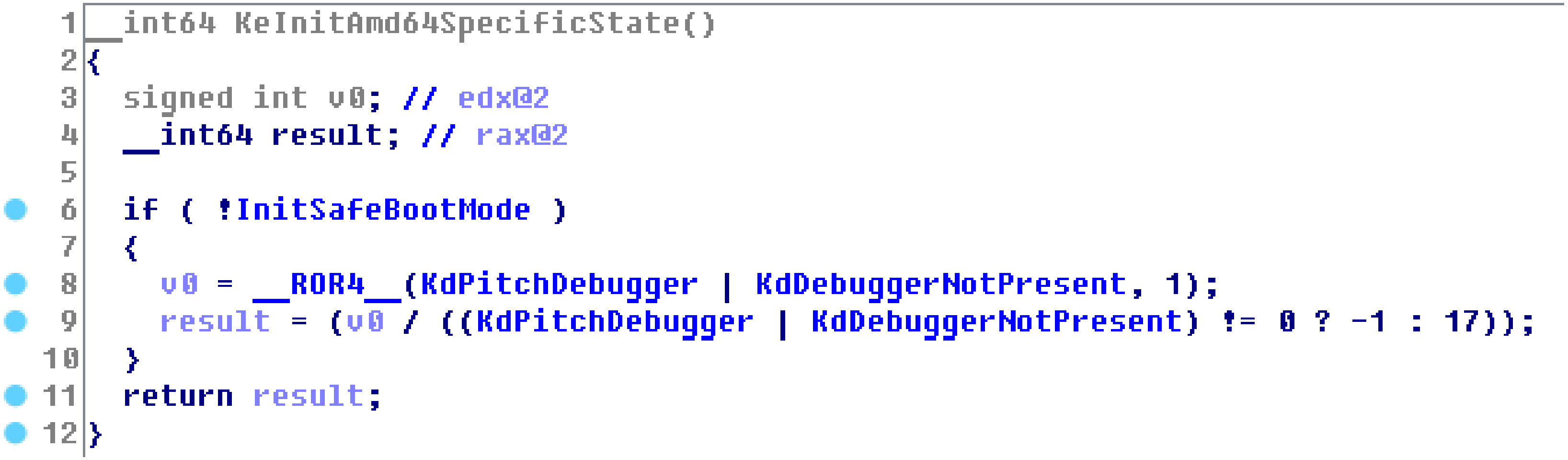


The graph is too big (more than 1000 nodes) to be displayed on the screen.
Switching to text mode.
(you can change this limit in the graph options dialog)

□ Don't display this message again

Anti-debugging

— Works only on free builds without kernel debugger!

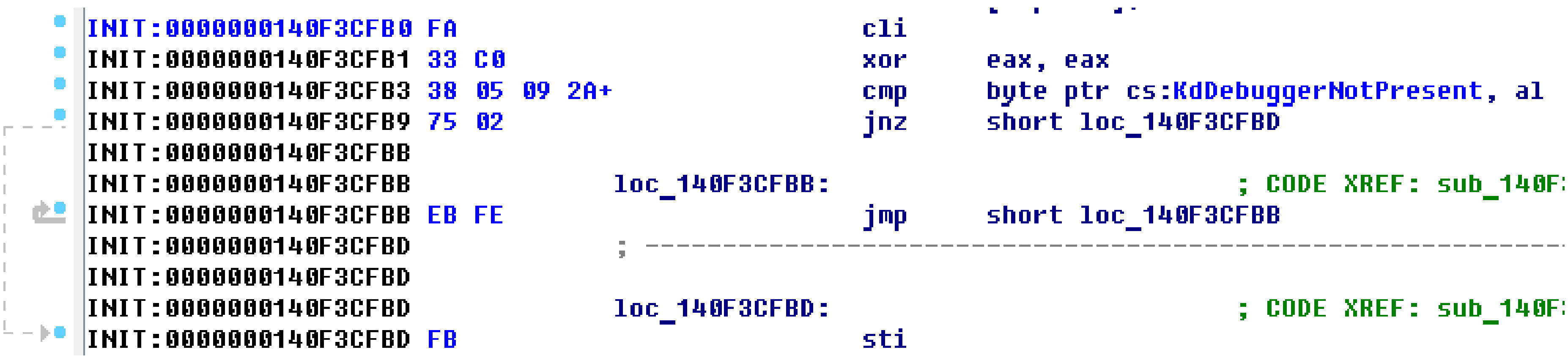


```
__int64 KeInitAmd64SpecificState()
{
  signed int v0; // edx@2
  __int64 result; // rax@2

  if ( !InitSafeBootMode )
  {
    v0 = __ROR4__(KdPitchDebugger | KdDebuggerNotPresent, 1);
    result = (v0 / ((KdPitchDebugger | KdDebuggerNotPresent) != 0 ? -1 : 17));
  }
  return result;
}
```

Anti-debugging

— Randomly inserted checks for debugger presence

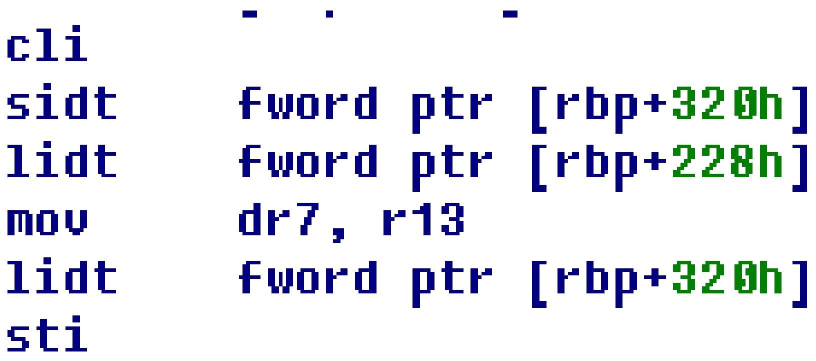


```
INIT:0000000140F3CFB0 FA                                cli
INIT:0000000140F3CFB1 33 C0                             xor     eax, eax
INIT:0000000140F3CFB3 38 05 09 2A+                      cmp     byte ptr cs:KdDebuggerNotPresent, al
INIT:0000000140F3CFB9 75 02                             jnz     short loc_140F3CFBD
INIT:0000000140F3CFBB
INIT:0000000140F3CFBB                       loc_140F3CFBB:                          ; CODE XREF: sub_140F:
INIT:0000000140F3CFBB EB FE                             jmp     short loc_140F3CFBB
INIT:0000000140F3CFBD                       ; ------------------------------------------------------------
INIT:0000000140F3CFBD
INIT:0000000140F3CFBD                       loc_140F3CFBD:                          ; CODE XREF: sub_140F:
INIT:0000000140F3CFBD FB                                sti
```

Anti-debugging

— If you use breakpoints, they will be included to a patchguard checksum, leading to a 0x109 bugcheck

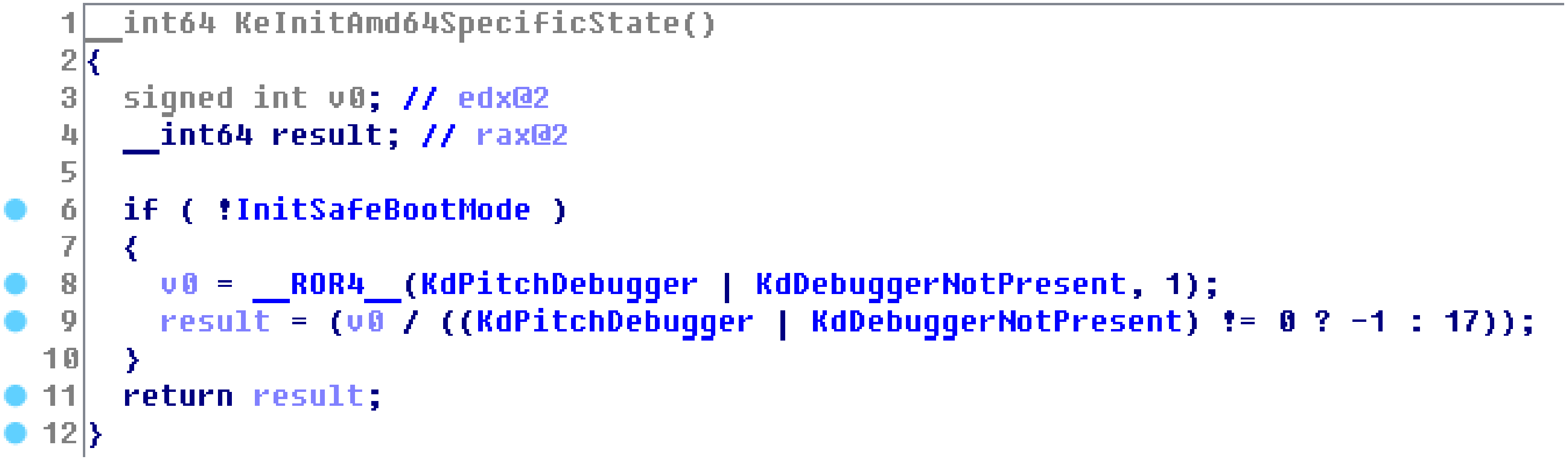
— If you use hardware breakpoints, well…



```
cli
sidt    fword ptr [rbp+320h]
lidt    fword ptr [rbp+228h]
mov     dr7, r13
lidt    fword ptr [rbp+320h]
sti
```

Non-linear code flow

— Active usage of Vectored Exception Handling



```
__int64 KeInitAmd64SpecificState()
{
  signed int v0; // edx@2
  __int64 result; // rax@2

  if ( !InitSafeBootMode )
  {
    v0 = __ROR4__(KdPitchDebugger | KdDebuggerNotPresent, 1);
    result = (v0 / ((KdPitchDebugger | KdDebuggerNotPresent) != 0 ? -1 : 17));
  }
  return result;
}
```

Reverse-engineering

- For dynamic analysis with KD (with windbg f.e.)
  - Remove all kd presence checks manually
    - Look them up with IDA scripting
    - Apply patches in KD with pykd
    - Do it before “Phase1InitializationDiscard“
- For static analysis with IDA
  - Try not to give up waiting for patchguard initialization function decompilation
- Use something else, like hypervisor-based debugger ;)

Reverse-engineering

— Since patchguard is developed incrementally, the key functions in reversing it are

- KiFilterFiberContext – chooses the way for invoking patchguard checks
- Unnamed sub inside KiFilterFiberContext – creates a structure aka patchguard context and schedules it’s verification
- Other functions (like context checkers) can be misleadingly named, but you can look them up around KiFilterFiberContext since they are located in a single compilation unit

Bypassing patchguard

— There are different approaches

  - patch kernel image so that patchguard will just not start
  - hook KeBugCheckEx and restore the state of a system
  - modify checkers so that they would be always valid

  - de-schedule contexts verification
    - This is what we’ve implemented

Contexts verification scheduling

— Context verification might be launched with

- KeSetCoalescableTimer
  - A timer that periodically launches context verification
- Prcb.AcpiReserved
  - A certain ACPI event (f.e. Idle transition)
- Prcb.HalReserved
  - A hal timer clock
- PsCreateSystemThread
  - A queued system thread that sleeps a random amount of time
- KeInsertQueueApc
  - A queued regular kernel APC
- KiBalanceSetManagerPeriodicDpc
  - A periodic event which happens every "KiBalanceSetManagerPeriod" ticks

Contexts verification descheduling

— So we've got to deschedule context verification once and for all

- KeSetCoalescableTimer
  - Timer? Disable!
- Prcb.AcpiReserved
  - Zero out this field
- Prcb.HalReserved
  - Same here
- PsCreateSystemThread
  - Scan sleeping worker threads and set wait time to infinite for suitable
- KeInsertQueueApc
  - Same here
- KiBalanceSetManagerPeriodicDpc
  - Revert to KiBalanceSetManagerDeferredRoutine

Thank you!

- ashiskin [at] ptsecurity [dot] com
- mermolov [at] ptsecurity [dot] com
- www.ptsecurity.com
- blog.ptsecurity.com