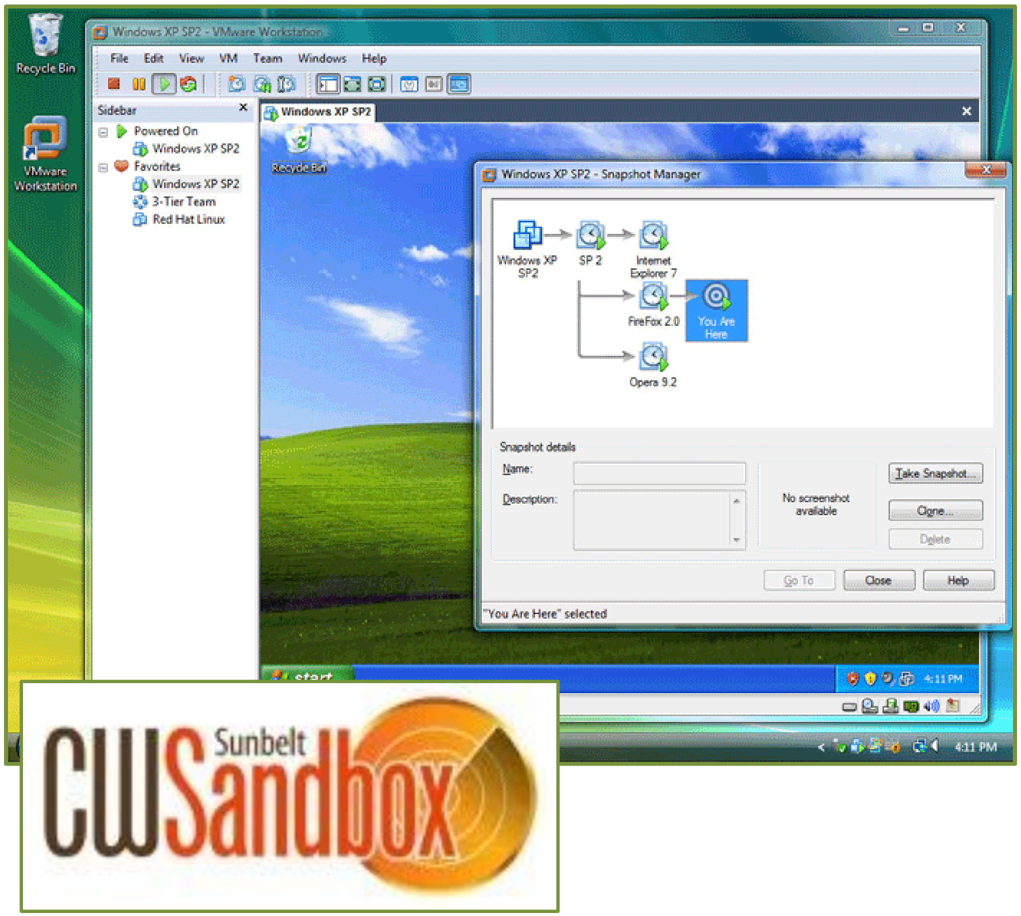
dirtbox, a x86/Windows Emulator

Georg Wicherski
Virus Analyst, Global Research and Analysis Team

Motivation & System Overview

Why not just use CWSandbox, Anubis, Norman‘s, JoeBox, ...

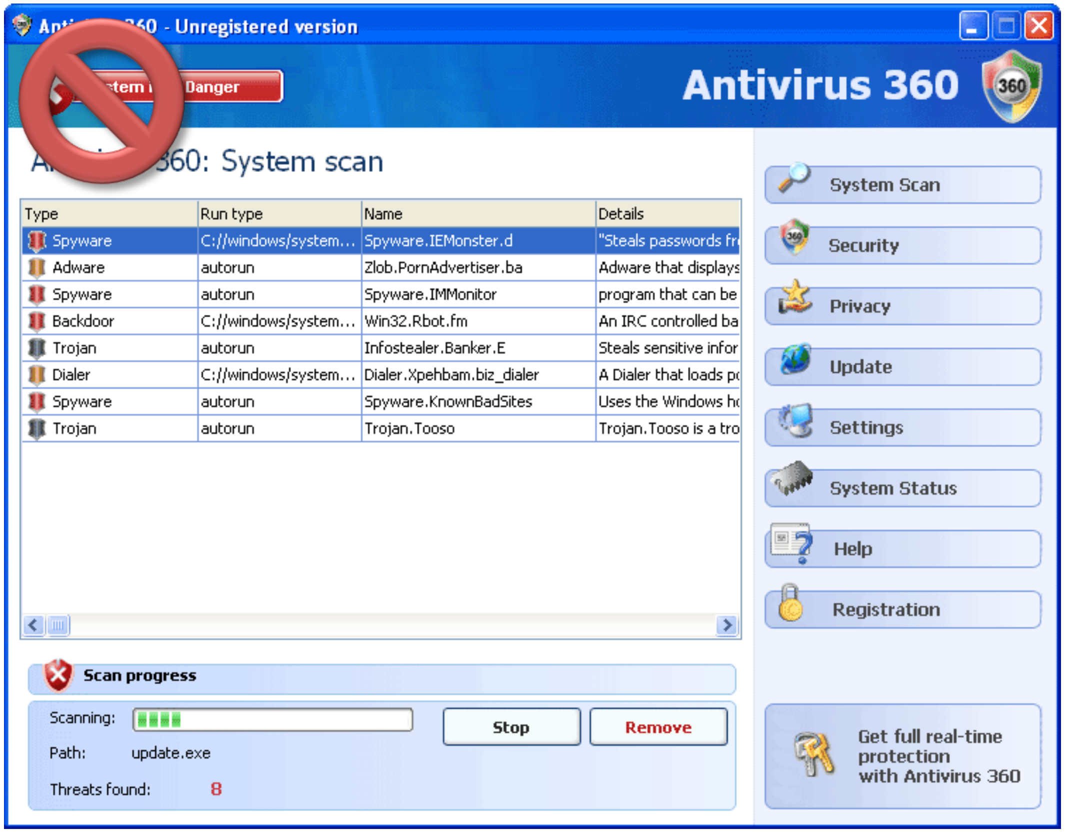
Malware *Analysis* Sandbox Solutions



- VMWare „Rootkits“
  - CWSandbox
  - JoeBox
  - ThreatExpert
  - zBox
  - …
- Norman Sandbox
- Anubis

Malware *Detection* Emulators (A/V)

KASPERSKY lab



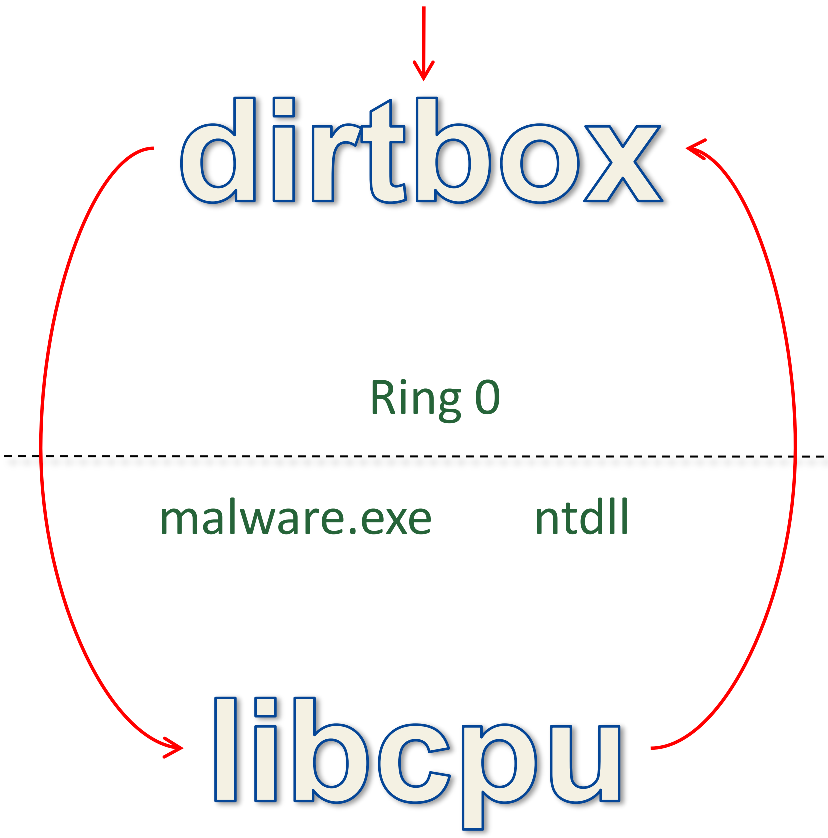
- Most *serious* A/V solutions have one
- API level emulation
- Often pure software emulators
- Detection by
  - Unimplented APIs
  - Heap Layout, SEH handling, ...
  - ...

Detection by API Side-Effects

- Functions containing `try {` in VS C++ share code
  - Epilogue is always the same
  - Uses sequence `push ecx / ret` to return to caller
  - The `ecx` register belongs to the called function by definition, so it is undefinde upon API return
  - The `ecx` value can be predicted because it will point to the API's `ret`
- This breaks *a lot* of A/V emulators right away
  - There are some funny but trivially detected workarounds
  - Could be used for generic anti-emulation detection (use of undefined registers after SEH protected API calls)
- Relies on the fact that the API's bytecode is not emulated

System Overview or „A cat pooped into my sandbox and now I have a dirtbox!“

- System Call Layer Emulation of Windows
- ntdll‘s native code is run inside virtual CPU
  - Other libraries wrap around kernel32 which wraps around ntdll
- Malware issuing system calls directly supported



libcpu

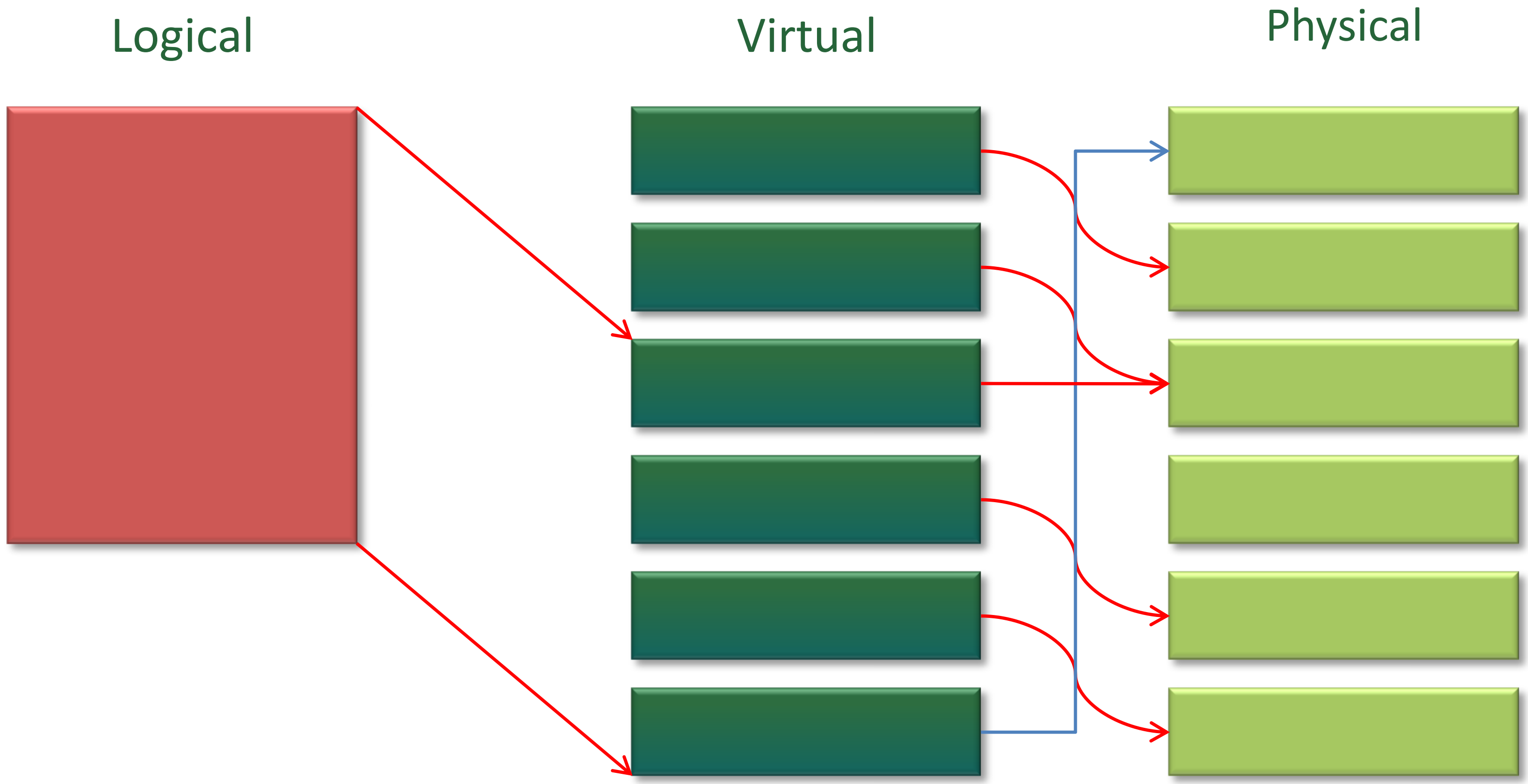
Custom x86 Basic Block Level Virtualization

libcpu Overview

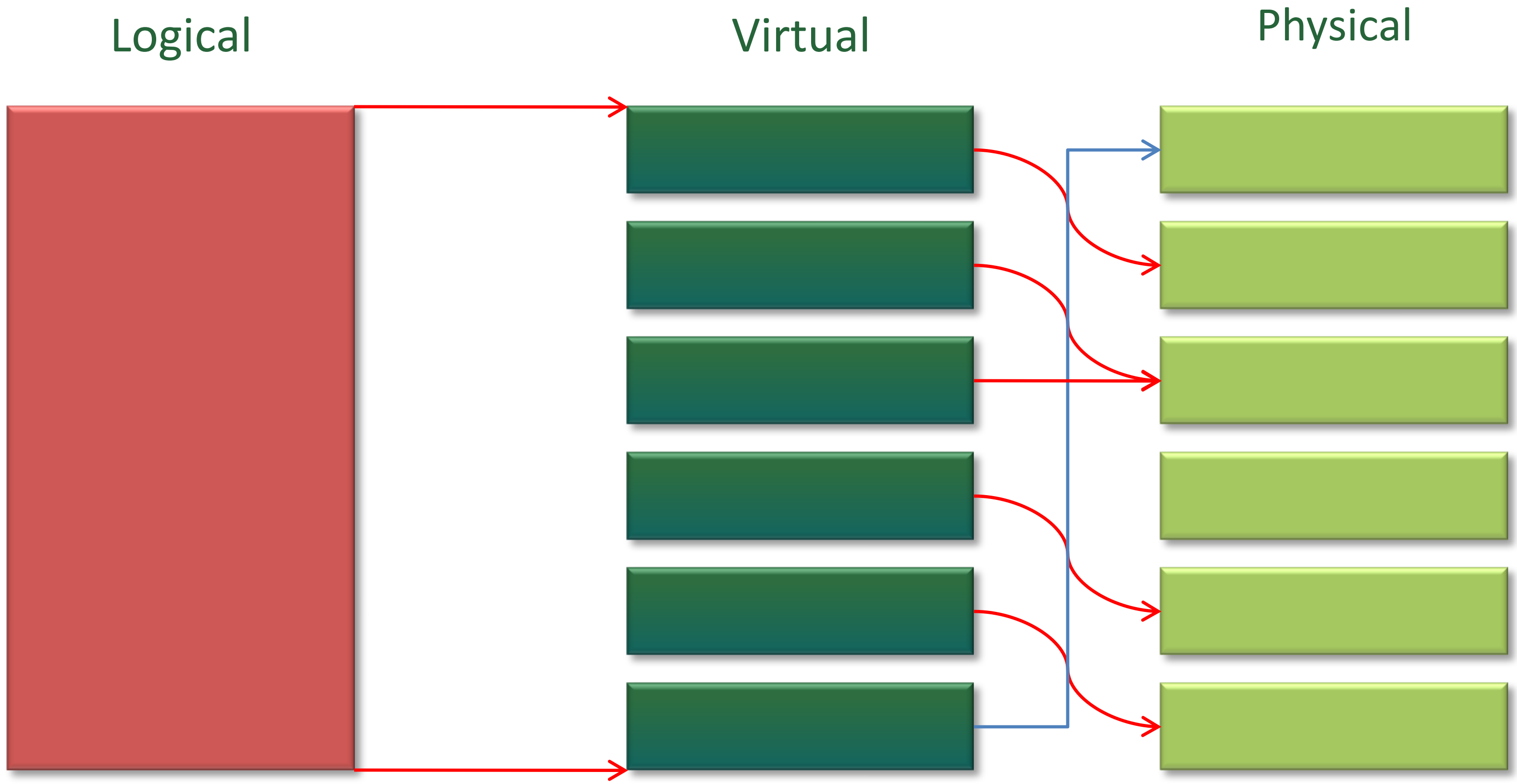
- Software emulation of x86 bytecode is too slow
  - A lot of additional code, such as ntdll & kernel32
- Existing Virtualization solutions are too powerful
  - Implementing their own MMU, support for privileged instructions
- We want instruction level introspection

➢ Homebrew x86 virtualization based on LDT

x86 Memory Views



x86 Memory View on Current OS



x86 Segmentation

- Global Descriptor Table
  - Allocated by Operating System
  - Shared among processes
- Local Descriptor Table
  - Has to be allocated by the OS, too
    - SYS_modify_ldt
    - NtSetLdtEntries
  - Process specific, usually not present

➢ Define 2 GB guest „userland“ LDT segment

Rogue Code Execution

- Basic block level execution on host CPU
  - No instruction rewriting required (thanks to host MMU)
- Basic block is terminated by
  - Control flow modifying instruction
  - Privileged instructions
- Exception: Backward pointing jumps
  - Directly copy if points into same basic block
  - Enhanced loop execution speeds
- Currently no code cache, could cache disassembly results (length of basic block)

Self-Modifying Code

~PROT_WRITE

libcpu Demo

```
edi: 00000000
esi: 00000000
ebp: 00000000
esp: 00400000
ebx: 00000000
edx: 00000000
ecx: 00000000
eax: 00000000
eflags: 00000000
eip: 00400000
00400000 > mov ecx,0x1000000
00400005 > xor eax,eax
00400007 > inc eax
00400008 > dec ecx
00400009 > jnz 0x400007
  0040000b <
edi: 00000000
esi: 00000000
ebp: 00000000
esp: 00400000
ebx: 00000000
edx: 00000000
ecx: 00000000
eax: 01000000
eflags: 00000246
eip: 0040000b
---
edi: 00000000
esi: 00000000
ebp: 00000000
esp: 00400000
ebx: 00000000
edx: 00000000
ecx: 00000000
eax: 01000000
eflags: 00000246
eip: 0040000b
Emulating 0040000b: int3
```

dirtbox

Or „The System Call Implementor‘s Sysiphus Tale“

Why System Call Layer Emulation

- System Calls mostly undocumented
  - Wine, ReactOS, ...
- We get a lot of genuine environment for free!
- There is a fixed number of system calls but an unbound number of APIs (think third party DLLs)
- Some malware uses system calls directly anyway
- Less detectability by API side effects (because we run original bytecode)

Things for Free: PE Parsing & Loading (!)

- Process startup handled mostly by new process
  - Creating process allocates new process: NtCreateProcess
  - Creates „Section“ of new image & ntdll and maps into process, this requires kernel to parse section headers
  - Creates new Thread on Entry Point with APC in ntdll
  - ntdll!LdrInitializeThunk will relocate images if necessary, resolve imports recursively, invoke TLS and DLL startup routines and do magic (see demo).
- All we have to implement is NtCreateSection & NtMapViewOfSection for SEC_IMAGE → we only need to parse PE‘s section headers!

Things for free: Accurate Heap Implementation

- A lot of A/V emulators naturally come with their own guest heap allocator implementations
  - Some even do not put heap headers before blocks
  - Let alone arena structures, …
- The Windows heap is implemented in ntdll
  - Interfacing the kernel with NtVirtualAlloc & NtVirtualFree
  - All protections like heap cookies are present
- Fingerprinting other emulators:
  - Look at **`malloc(0)-8`**, look for proper block header
  - Or overflow until the heap cookie and free

Things for free: Proper SEH Handling

- Generate **`CONTEXT`** record from current CPU state
- Jump to ntdll!KiUserExceptionDispatcher
- ntdll will do proper SEH handling for us
  - Lookup current top of SEH chain in TEB
  - Walk list, invoke exception handlers with correct flags
  - Checking for SafeSEH structures etc.
- Trivial detection for other emulators:
  - Link with SafeSEH header
  - Trigger exception with invalid handler registered
  - Check in UnhandledExceptionHandler

dirtbox Demo

```
src/win32/Process.cpp:0917 systemCall: Invoking system call #170: NtQuerySymbolicLinkObject
src/win32/Process.cpp:0592 sysc_NtQuerySymbolicLinkObject: <7c980280, 0>, 0 -> 'C:\WINDOWS\system32'
src/win32/Process.cpp:0917 systemCall: Invoking system call #25: NtClose
src/win32/Process.cpp:0121 sysc_NtClose: Handle #c
 DEBUG LDR: NEW PROCESS
 DEBUG      Image Path: C:\WINDOWS\system32\simple-time.exe (simple-time.exe)
 DEBUG      Current Directory: C:\WINDOWS\system32\
 DEBUG      Search Path: C:\WINDOWS\system32;C:\WINDOWS\system;C:\WINDOWS;.
src/win32/Process.cpp:0917 systemCall: Invoking system call #116: NtOpenFile
src/win32/Process.cpp:1042 resolveObjectAttributes: (18, 0, 9d84<'\??\C:\WINDOWS\system32\'>, 42)
src/win32/Process.cpp:0374 sysc_NtOpenObject: Allocated handle 10 for '\??\C:\WINDOWS\system32\'
src/win32/Process.cpp:0917 systemCall: Invoking system call #179: NtQueryVolumeInformationFile
src/win32/Process.cpp:0917 systemCall: Invoking system call #83: NtFreeVirtualMemory
src/win32/Process.cpp:0917 systemCall: Invoking system call #17: NtAllocateVirtualMemory
src/win32/Process.cpp:0045 sysc_NtAllocateVirtualMemory: ffffffff, 23000, -, 1000, 1000, 4
src/win32/Process.cpp:0917 systemCall: Invoking system call #84: NtFsControlFile
src/win32/Process.cpp:0446 sysc_NtFsControlFile: <10, 0, 0, 0, 99d8, 90028, (nil), 0, (nil), 0>
src/win32/Process.cpp:0917 systemCall: Invoking system call #139: NtQueryAttributesFile
src/win32/Process.cpp:1042 resolveObjectAttributes: (18, 0, 9db8<'\??\C:\WINDOWS\system32\simple-time.exe.Local'>, 40)
src/win32/Process.cpp:1055 resolveObjectAttributes: Object name could not be resolved: '\??\C:\WINDOWS\system32\simple-time.exe.Local'
src/win32/Process.cpp:0917 systemCall: Invoking system call #83: NtFreeVirtualMemory
 DEBUG LDR: LdrLoadDll, loading kernel32.dll from
src/win32/Process.cpp:0917 systemCall: Invoking system call #125: NtOpenSection
src/win32/Process.cpp:1042 resolveObjectAttributes: (18, 8, 9a24<'kernel32.dll'>, 40)
src/win32/Process.cpp:0374 sysc_NtOpenObject: Allocated handle 14 for 'kernel32.dll'
src/win32/Process.cpp:0917 systemCall: Invoking system call #108: NtMapViewOfSection
src/win32/Process.cpp:0322 sysc_NtMapViewOfSection: (14, ffffffff, 0, 0, 0,     (nil), * 0x77445aec = 0, 1, 0, 4)
src/win32/SectionObject.cpp:0109 mapView: Successful PE loading: 7c800000, f6000
src/win32/Process.cpp:0917 systemCall: Invoking system call #25: NtClose
src/win32/Process.cpp:0121 sysc_NtClose: Handle #14
 DEBUG LDR: ntdll.dll used by kernel32.dll
 DEBUG LDR: Snapping imports for kernel32.dll from ntdll.dll
src/win32/Process.cpp:0917 systemCall: Invoking system call #137: NtProtectVirtualMemory
src/win32/Process.cpp:0238 sysc_NtProtectVirtualMemory: ffffffff, 7c801000, 624, 4 -> 3, 97bc
src/win32/Process.cpp:0917 systemCall: Invoking system call #137: NtProtectVirtualMemory
src/win32/Process.cpp:0238 sysc_NtProtectVirtualMemory: ffffffff, 7c801000, 1000, 20 -> 5, 20
src/win32/Process.cpp:0907 systemCall: Unsupported system call #78: NtFlushInstructionCache!
 DEBUG LDR: LdrGetProcedureAddress by
```

Conclusion & Future Work

Let‘s use this for exploit development!

Detecting dirtbox / Anti-Emulation

- No leaked registers in Ring 0 transition except for **`eax`**
  - Need to provide proper return codes, esp. error codes
  - ntdll just cares about $\geq$ **`0xc0000000`**; malware might look for specific error codes
- Side effects on buffers etc., especially in error cases
  - Fill out **`IN OUT PDWORD Length`** in case of error?
  - Roll back system calls performing multiple things?
- Tradeoff between detectability and performance

Future Work: Adding Tainting & SAT Checking

KASPERSKY lab

- Already did Proof-of-Concept based on STP
- Interleave static analysis into dynamic emulation
  - Look for interesting values (e.g. reads from network, date)
  - Do static forward data-flow analysis on usage
  - If used in conditional jumps, identify interesting values with a SAT Checker (there are better domain specific ways, but I'm lazy)
- ➢ Automatic reconstruction of network protocols (e.g. commands in IRC bots)
- ➢ Identify specific trigger based behaviour
- ➢ Identify Anti-Emulation behaviour

Questions? Thank You!

georg.wicherski@kaspersky.com
blog.oxff.net & securelist.com

KASPERSKY lab