Arancino techniques

# Code Cache Artifacts

### EIP Detection

*Problem*: The value of the EIP is not the one expected by the program   
*Cause*: Due to the nature of a DBI the executed code is stored in a memory region called code cache

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| *int 2e* instruction (now *sysenter*) has the side effect of storing the EIP in EDX to be able to restore it after the return from kernel mode. This let the malware know the effective value of EIP | Insertion of a call to a function that patches the EDX with the real EIP right after the *int 2e* instruction in the code |

### Self-Modifying Code

*Problem*: A DBI executes the cache where the code is not modified, following a wrong path   
*Cause*: Due to the nature of a DBI the executed code is stored in a memory region called code cache

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Modifying itself a malware can change its behavior at runtime in very colorful manners, and since the code executed is not the real one but a cache in a different memory region from the one where the real program is held, this will lead the DBI to misinterpret the malware | Every time a new *Trace* is collected the boundaries are retrieved, then for each write instruction the target address is checked, and if it falls in one of the *Trace* boundaries the memory is marked as written. If the EIP points to a memory marked as written the DBI is forced to discard the trace and build a new one |

# Environment Artifacts

### Parent Detection

*Problem*: The malware parent is not cmd.exe  
*Cause*: The malware is launched by the DBI itself, so it’s also the parent process of the malware

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Calling *NtQuerySystemInformation* the malware can analyze the returned structure *SYSTEM\_PROCESS\_INFO* | Hooking the function, it’s possible to return a modified *SYSTEM PROCESS INFO* where each process named pin.exe is replace with cmd.exe |
| Open the process *CSRSS.exe* which is responsible to maintain its own process list in user-space | Hooking *NtOpenProcess*, and defining a list of *PID* of process that the process shouldn’t be able to see, everytime the target of this function is in the list *NTSTATUS\_ACCESS\_DENIED* is returned |

### Memory Fingerprinting

*Problem*: The malware can find artifacts in the memory such as string and code patterns   
*Cause*: Sharing the same memory with the malware there is unavoidably artifacts in the memory

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Using *VirtualQuery* and *NtQueryVirtualMemory* a malware can determine the state of a particular memory page | Intercepting and controlling the result of these functions it’s possible to hide the content of the memory returning a *MEM\_FREE* value  The white list of memory region is created and updated at run-time when:   * A library is loaded * An allocation is carried out with *NtAllocateVirtualMemory, RtlAllocateheap* and *RtlReallocateHeap* * For the stack the ESP register is recovered and the memory in range [ESP, ESP + Default stack size] is whitelisted * PEB and TEB are whitelisted as well * Mapped files are also whitelisted |

*Issues*: Although the malware is unaware of the presence of artifacts it could try to allocate that space of memory crashing. It’s highly inefficient to allocate a large part of memory just to find out the presence of a DBI and this behavior can be detected fairly easy

# JIT Compiler Detection

### DLL Hook

*Problem*: Initial instructions of some functions are not the ones expected  
*Cause*: DBI tools need to hook functions to intercept the execution of a not instrumented trace inserting jumps at the beginning of some functions

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Malware can check if the initial instructions of common functions have been modified | To hide these differences at the beginning of the instrumentation addresses in memory of the hooked functions are retrieved (for instance with *GetModuleHandle*) and are stored and labelled as protected. When the malware tries to access these memory regions are redirected to the original ones saved in a different memory region.  Using this method *address Space Layout Randomization (ASLR)* is not a problem |

### Memory Page Permissions

*Problem*: There’s a plethora of pages marked as RWX  
*Cause*: A JIT compiler needs to write and then execute a significant amount of code

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| A malware could scan the entire process address space counting the occurrences of RWX marked memory pages, if it’s considerably high the malware could detect the presence of a DBI | The solution comes for free with the one implemented in the [DLL Hook](#_DLL_Hook) problem which reveals only whitelisted memory regions |

### Memory Allocations

*Problem*: Intense allocation of memory  
*Cause*: A JIT compiler needs to store the instruction before executing it

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Counting the number of times *ZwAllocateVirtualMemory* is called and comparing the results with a previous defined threshold the malware could detect the presence of a DBI.  This is usually carried out inserting a simple *assembly* routine at the beginning of the *ZwAllocateVirtualMemory* which increments a counter | The approach to defeat this technique is:   1. Redirecting every write attempt of the malware inside .text segment of libraries and 2. Faking the results of suspicious read that have their target address inside the .text segment of libraries   This will deny both the insertion and the check of such a tool |

*Issues(maybe)*: This is going to block every kind of attempt to overwrite the original code of functions and to insert hooks inside protected memory regions

# Overhead Detection

### Windows Time

*Problem*: Bad speed performance   
*Cause*: Since the DBI interprets the instructions it’s inevitably slower than a native execution

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| The retrieval of some fields of a structure called *KUSER\_SHARED\_DATA* through *GetTickCount* and *timeGetTime*. However, the former simply access the fields *TickCountMultiplier* and *TickCountQuad* and the latter the field *interrupt\_time* represented by the struct *\_KSYSTEM\_TIME* | Hooking these Win API is not sufficient, since as said before they just read memory, so an intercept of reading from those memore region is needed.  When such a behavior is revealed, the time represented in the structures is divided by a user defined *TICK DIVISOR* |

### CPU Time

*Problem*: Bad speed performance   
*Cause*: Since the DBI interprets the instructions it’s inevitably slower than a native execution

|  |  |
| --- | --- |
| **Evasion** | **Solution** |
| Through the usage of the assembly instruction *rdtsc* the malware can retrieve the number of clock cycles elapsed since the last reset.  This is a 64-bit integer so in i386 architectures it’s composed of high (EDX) and low (EAX) part | If *rdtsc* is found in a trace, a function that divide the return value by a user defined factor is inserted, patching the value of the two registers |