

Антиотладка. Теория и практика защиты приложений от дебага

Nik Zentel, 17.01.2018 2 комментария 37 949 Добавить в избранное



Содержание статьи

- 01: **IsDebuggerPresent()** и структура **PEB**
 - 01.1: **Process Environment Block**
 - 01.2: **NtGlobalFlag**
 - 01.3: **Flags и ForceFlags**
- 02: **CheckRemoteDebuggerPresent()** и **NtQueryInformationProcess**
 - 02.1: **Тонкости NtQueryInfoProcess**
 - 02.2: **DebugObject**
 - 02.3: **ProcessDebugFlags**
- 03: **Проверка родительского процесса**
- 04: **TLS Callbacks**
- 05: **Отладочные регистры**
- 06: **NtSetInformationThread**
 - 06.1: **NtCreateThreadEx**
- 07: **SeDebugPrivilege**
- 08: **SetHandleInformation**
- 09: **Заключение**

К методам детектирования отладки прибегают многие программисты: одни хотели бы уберечь свои продукты от конкурентов, другие противостоят вирусным аналитикам или автоматическим системам распознавания малвари. Мы в подробностях рассмотрим разные методы борьбы с дебагом — от простых до довольно нетривиальных.

Поскольку сейчас популярна не только архитектура x86, но и x86-64, многие старые средства обнаружения отладчиков устарели. Другие требуют корректировки, потому что жестко завязаны на смещения в архитектуре x86. В этой статье я расскажу о нескольких методах детекта отладчика и покажу код, который будет работать и на x64, и на x86.

IsDebuggerPresent() и структура PEB

Начинать говорить об антиотладке и не упомянуть о функции IsDebuggerPresent() было бы неправильно. Она универсальна, работает на разных архитектурах и очень проста в использовании. Чтобы определить отладку, достаточно одной строки кода: if (IsDebuggerPresent()) .

Что представляет собой WinAPI IsDebuggerPresent? Эта функция обращается к структуре PEB.

Process Environment Block

Блок окружения процесса (PEB) заполняется загрузчиком операционной системы. находится в адресном пространстве процесса и может быть модифицирован из режима usermode. Он содержит много полей: например, отсюда можно узнать информацию о текущем ядре, окружении и загруженных модулях. Получить структуру PEB можно, обратившись к ней напрямую по адресу fs:[30h] для x86 и gs:[60h] для x64.

Соответственно, если загрузить в отладчик функцию IsDebuggerPresent(), на x86-системе мы увидим:

```
mov     eax,dword ptr fs:[30h]
movzx   eax,byte ptr [eax+2]
ret
```

А на x64 код будет таким:

```
mov     rax,qword ptr gs:[60h]
movzx   rax,byte ptr [rax+2]
ret
```

Что значит byte ptr [rax+2]? По этому смещению находится поле BeingDebugged в структуре PEB, которое и сигнализирует нам о факте отладки. Как еще можно использовать PEB для обнаружения отладки?

NtGlobalFlag

Во время отладки система выставляет флаги FLG_HEAP_VALIDATE_PARAMETERS, FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK, в поле NtGlobalFlag, которое находится в структуре PEB. Отладчик использует эти флаги для контроля разрушения кучи посредством переполнения. Битовая маска флагов — 0x70. Смещение NtGlobalFlag в PEB для x86 составляет 0x68, для x64 — 0x6C. Чтобы показать пример кода детекта отладчика по NtGlobalFlag, воспользуемся функциями intrinsics, а чтобы код был более универсальным, используем директивы препроцессора:

```
#ifdef _WIN64
DWORD pNtGlobalFlag = NULL;
PPEB pPeb = (PPEB)_readqword(0x60);
pNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0x6C);
#else
DWORD pNtGlobalFlag = NULL;
PPEB pPeb = (PPEB)_readfdword(0x30);
pNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0x68);
#endif

if ((pNtGlobalFlag & 0x70) != 0) std::cout << "Debugger detected!\n";
```

Flags и ForceFlags

PEB также содержит указатель на структуру _HEAP, в которой есть поля Flags и ForceFlags. Когда отладчик подсоединен к приложению, поля Flags и ForceFlags содержат признаки отладки. ForceFlags при отладке не должно быть равно нулю, поле Flags не должно быть равно 0x00000002:

```
#ifdef _WIN64
PINT64 pProcHeap = (PINT64)(_readqword(0x60) + 0x30); // Получаем структуру _HEAP
PUINT32 pFlags = (PUINT32)(pProcHeap + 0x70); // Получаем Flags внутри _HEAP
PUINT32 pForceFlags = (PUINT32)(pProcHeap + 0x74); // Получаем ForceFlags внутри _HEAP
#else
PPEB pPeb = (PPEB)(_readfdword(0x30) + 0x18);
PUINT32 pFlags = (PUINT32)(pProcHeap + 0x40);
PUINT32 pForceFlags = (PUINT32)(pProcHeap + 0x44);
#endif

if (pFlags & _HEAP_GROWABLE || pForceFlags != 0)
std::cout << "Debugger detected!\n";
}
```

CheckRemoteDebuggerPresent() и NtQueryInformationProcess


Функция CheckRemoteDebuggerPresent, как и IsDebuggerPresent, кросс-платформенная и проверяет наличие отладчика. Ее отличие от IsDebuggerPresent в том, что она умеет проверять не только свой процесс, но и другие по их хендлу. Прототип функции выглядит следующим образом:

```
BOOL WINAPI CheckRemoteDebuggerPresent(
    _In_ HANDLE hProcess,
    _Inout_ PBOOL pbDebuggerPresent
);
```

где hProcess — хендл процесса, который проверяем на предмет подключения отладчика, pbDebuggerPresent — результат выполнения функции (соответственно, TRUE или FALSE). Но самое важное отличие в работе этой функции заключается в том, что она не берет информацию из PEB, как IsDebuggerPresent, а использует функцию WinAPI NtQueryInformationProcess. Прототип функции выглядит так:

```
NTSTATUS WINAPI NtQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_ PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
    _Out_opt_ PULONG ReturnLength
);
```

Поле, которое поможет нам понять, как работает CheckRemoteDebuggerPresent, — это ProcessInformationClass, который представляет собой большую структуру (enum). PROCESSINFOCLASS с параметрами. Функция CheckRemoteDebuggerPresent передает в это поле значение 7, которое указывает на ProcessDebugPort. Дело в том, что при подключении отладчика к процессу в структуре EPROCESS заполняется поле ProcessInformation, которое в коде названо DebugPort.



INFO

Структура EPROCESS, или блок процесса, содержит много информации о процессе, указатели на несколько структур данных, в том числе и на PEB. Заполняется исполнительной системой ОС, находится в системном адресном пространстве (kernelmode), как и все связанные структуры, кроме PEB. Все процессы имеют эту структуру.

Если поле заполнено и порт отладки назначен, то принимается решение о том, что идет отладка. Код для CheckRemoteDebuggerPresent:

```
BOOL IsDbgPresent = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &IsDbgPresent);
if (IsDbgPresent) std::cout << "Debugger detected!\n";
```

Код передачи параметра ProcessDebugPort напрямую в функцию NtQueryInformationProcess:

```
Status = NtQueryInfoProcess(GetCurrentProcess(),
7, // ProcessDbgPort
&DbgPort,
dProcessInformationLength,
NULL);

if (Status == 0x00000000 && DbgPort != 0) std::cout << "Debugger detected!\n";
```

Переменная Status имеет тип NTSTATUS и сигнализирует нам об успехе или неуспехе выполнения функции; в DbgPort проверяем, назначен порт или поле нулевое. Если функция отработала без ошибок и вернула статус 0 и DbgPort имеет ненулевое значение, то порт назначен и идет отладка.

Тонкости NtQueryInfoProcess

Документация MSDN говорит нам, что использовать NtQueryInfoProcess следует при помощи динамической линковки, получая ее адрес из ntdll.dll напрямую, через функции LoadLibrary и GetProcAddress, и определяя прототип функции вручную при помощи typedef:

```
typedef NTSTATUS(WINAPI "NtQueryInformationProcess)(HANDLE, ULONG, PVOID,
PULONG);

NtQueryInfoProcess =
(pNtQueryInformationProcess)GetProcAddress(LoadLibrary("ntdll.dll"),
"NtQueryInformationProcess");
```

Но функция NtQueryInformationProcess может показать несколько признаков отладки, и ProcessDebugPort — только один из них.

DebugObject

При отладке приложения создается DebugObject, объект отладки. Если NtQueryInformationProcess в поле ProcessInformationClass передать значение 0x1E, то оно укажет на элемент ProcessDebugObjectHandle и при обработке функции нам будет возвращен хендл объекта отладки. Код похож на предыдущий с тем отличием, что вместо 7 в поле ProcessInformationClass передается значение 0x1E и меняется условие проверки:

```
if (Status == 0x00000000 && hDbgObj) std::cout << "Debugger detected!\n";
```

где hDbgObj — поле ProcessInformation с результатом. Здесь все так же: функция отработала правильно и вернула 0, hDbgObj ненулевой. Значит, отладчик создан.

ProcessDebugFlags

Следующий признак отладки, который может показать функция NtQueryInfoProcess, — это поле ProcessDebugFlags, имеющее номер 0x1F. Передавая значение 0x1E, мы заставляем функцию NtQueryInfoProcess показать нам поле NoDebugInherit, которое находится в структуре EPROCESS. Если поле равно нулю, это значит, что в данный момент приложение отлаживается. Код вызова NtQueryInfoProcess идентичен, меняем только номер ProcessInformationClass и проверку:

```
if (Status == 0x00000000 && NoDebugInherit == 0) std::cout << "Debugger detected!\n";
```

Проверка родительского процесса

Суть этого антиотладочного метода заключается в том, что мы должны проверить, кем именно было запущено приложение, которое мы защищаем: пользователем или отладчиком. Этот способ можно реализовать разными путями — проверить, является ли рапел-процессом explorer.exe либо не выступает ли в этой роли oludbg.exe, x64dbg.exe, x32dbg и так далее. Если попытаться реализовать логику этого метода обнаружения отладки, то приходит в голову еще один простой метод — получить снапшот всех процессов в системе и сравнить название каждого со списком известных отладчиков.

Проверить родительский процесс мы будем при помощи уже известной нам функции NtQueryInformationProcess и структуры PROCESS_BASIC_INFORMATION (поле InheritedFromUniqueProcessId), а получать список всех запущенных процессов в системе можно при помощи CreateToolhelp32Snapshot/Process32First/Process32Next. Чтобы не писать не относящийся к делу код парсинга всех процессов в системе, напишем только основной код получения ID родительского процесса и основную проверку:

```
PROCESS_BASIC_INFORMATION baseInf;

NtQueryInformationProcess(NtCurrentProcess(), ProcessBasicInformation, &baseInf, sizeof
4
```

Итак, в baseInf.InheritedFromUniqueProcessId находится ID процесса, который порождает наш. Его можно использовать как угодно: например, получить из него имя файла, название процесса и сравнить с именами отладчиков или проверить, не explorer.exe ли это.

TLS Callbacks

Этот нетривиальный метод антиотладки заключается в том, что мы встраиваем антиотладочные приемы в TLS Callbacks, которые выполняются, когда входящая точка программы. Внутри самого приложения могут быть установлены точки останова, да и внимание будет сконцентрировано на основном коде приложения, но этот прием завершит отладку, даже толком ее не начав. Кто-то считает этот способ весьма могучим, но сейчас при правильной настройке отладчика процесс отладки может весьма легко пройти, если входе в TLS Callbacks. То есть против матерых реверсеров это не спасет, зато отсечет много школьников, которые не будут понимать, что происходит 😊 Чтобы реализовать этот метод обнаружения, необходимо сказать компилятору создать секцию TLS таким кодом:

```
#pragma comment(linker, "/include:_tls_used")
```

Секция должна иметь имя CRT\$XLY:

```
#pragma section("CRT$XLY", long, read)
```

Сам код имплементации:

```
void WINAPI TlsCallback(PVOID pMod, DWORD Reason, PVOID Con)
{
    if (IsDebuggerPresent()) std::cout << "Debugger detected!\n";
}

__declspec(allocate("CRT$XLYB")) PIMAGE_TLS_CALLBACK CallTSL[] = {CallTSL, NULL};
```

Отладочные регистры

Если в отладочные регистры есть какие-то данные, то это еще один признак. Но дело в том, что отладочные регистры — привилегированный ресурс и получить к ним доступ напрямую можно только в режиме ядра. Но мы попробуем получить контекст потока при помощи функции GetThreadContext и таким образом прочитать данные отладочных регистров. Всего отладочных регистров восемь, DR0–DR7. Первые четыре регистра DR0–DR3 содержат информацию о локальных отладочных регистрах DR4–DR5 — зарезервированные, регистр DR6 заполняется, когда работает брейк-поинт отладчика, и содержит информацию об этом событии. Регистр DR7 содержит биты управления отладкой. Итак, нам интересно, какая информация содержится в первых четырех регистрах.

```
CONTEXT context = {};
context.ContextFlags = CONTEXT_DEBUG_REGISTERS;

GetThreadContext(GetCurrentThread(), context);

if (ctx.Dr0 != 0 ||
ctx.Dr1 != 0 ||
ctx.Dr2 != 0 ||
ctx.Dr3 != 0)
std::cout << "Debugger detected!\n";
```

NtSetInformationThread

Еще один нетривиальный метод антиотладки основан на передаче флага HideFromDebugger (находится в структуре _ETHREAD за номером 0x11) в функцию NtSetInformationThread. Вот как выглядит прототип функции:

```
NTSTATUS KSetInformationThread(
    _In_ HANDLE ThreadHandle,
    _In_ THREADINFOCLASS ThreadInformationClass,
    _In_ PVOID ThreadInformation,
    _In_ ULONG ThreadInformationLength
);
```

Этот прием спрячет наш поток от отладчика, переставая отправлять ему отладочные события, например такие, как срабатывание точек останова. Особенность этого метода в том, что он универсален и работает благодаря штатным возможностям ОС. Вот код, который реализует отсоединение главного потока программы от отладчика:

```
NTSTATUS stat = NtSetInformationThread(GetCurrentThread(), 0x11, NULL, 0);

NtCreateThreadEx

NTSYSCALLAPI
NTSTATUS
NTAPI
NtCreateThreadEx (
    _Out_ PHANDLE ThreadHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ProcessHandle,
    _In_ PVOID StartRoutine,
    _In_opt_ PVOID Argument,
    _In_ ULONG CreateFlags,
    _In_opt_ ULONG_PTR ZeroBits,
    _In_opt_ SIZE_T StackSize,
    _In_opt_ SIZE_T MaximumStackSize,
    _In_opt_ PVOID AttributeList
);

Код отключения отладчика:

HANDLE hthr = 0;

NTSTATUS status = NtCreateThreadEx(&hthr,
THREAD_ALL_ACCESS, 0, NtCurrentProcess,
(PTHREAD_START_ROUTINE)next, 0,
THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER, 0, 0, 0, 0);

После этого начинает работать функция next() из WinAPI, которая находится в отдельном невидимом для отладчика треде.
```

SeDebugPrivilege

Один из признаков отладки приложения — получение приложением привилегии SeDebugPrivilege. Чтобы понять, есть ли такая привилегия у нашего процесса, можно, например, попытаться открыть какой-нибудь системный процесс. По традиции пробуем открыть csrss.exe. Для этого используем функцию WinAPI OpenProcess с параметром PROCESS_ALL_ACCESS. Вот как реализуется этот метод (в переменной Id_From_csrss находится ID csrss.exe):

```
HANDLE hDebug = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Id_From_csrss);
if (hDebug != NULL) std::cout << "Debugger detected!\n";
```

SetHandleInformation

Функция SetHandleInformation применяется для установки свойств дескриптора объектов, на который указывает hObject. Прототип функции выглядит следующим образом:

```
BOOL SetHandleInformation(
    _In_ HANDLE Handle,
    DWORD dwMask,
    DWORD dwFlags
);
```

Типы объектов различны — например, это может быть задание, отображение файла или мьютекс. Мы можем этим воспользоваться: создадим мьютекс с флагом HANDLE_FLAG_PROTECT_FROM_CLOSE и попробуем его закрыть, попытаемся поймать исключение. Если исключение будет поймано, то процесс отлаживается.

```
HANDLE hMyMutex = CreateMutex(NULL, FALSE, _T("MyMutex"));

SetHandleInformation(hMyMutex, HANDLE_FLAG_PROTECT_FROM_CLOSE, HANDLE_FLAG_PROTECT_FROM_CLOSE);

try {
    CloseHandle(hMyMutex);
}
except (HANDLE_FLAG_PROTECT_FROM_CLOSE) {
    std::cout << "Debugger detected!\n";
}
4
```

Заключение

Мы рассмотрели несколько способов защиты приложения от отладки. Я сталелс показывать разные методы отладки и рассказывать, как они работают на неким уровне. Чтобы лучше разобраться в том, что происходит, ты должен понимать, как работает ОС, как приложение взаимодействует с разными структурами окружения потока и процесса. Надеюсь, моя статья поможет тебе в этом научиться более эффективно защищать приложения от любопытных отладчиков. Если же ты хочешь узнать больше о защите приложений, то тебе стоит подписаться на рассылку SuperSliv.Biz.