Статья Исследуем и обходим перехваты/хуки функций на уровне пользователя

xss.is/threads/43097

Привет, друзья. В этой статье мы с вами будем исследовать перехваты функций в режиме пользователя (методом сплайсинга), которые нашей беззащитной малварушке устанавливают антивирусы, сендбоксы, ханипоты и другие наглые и беспардонные программы. Делать мы это будем на практике, так что расчехляйте свои текстовые редакторы или IDEши. Писать мы будет на смеси C и C++, используя в основном WinAPI функции и некоторые COM-интерфейсы. Для компиляции я буду использовать MinGW, но код с некоторыми изменениям можно будет адаптировать для компиляции в Visual Studio. Наверное, только за исключением ассемблерных вставок. Последний раз, когда я компилил что-либо с помощью cl.exe в 64-битном режиме ассемблерные вставки не поддерживались, если эта ситуация поменялась, то напишите пожалуйста об этом в комментариях, интересно об этом узнать.

Отдельное спасибо за вдохновение на написание этой статьи я хотел бы выразить мюсье X-Shar'y, а мюсье modexp'y за идейность и некоторые вещи, которые я у него честно, ну скажем так, «подглядел». Так же давайте сделаем небольшой дисклеймер: код, представленный в этой статье, может иметь баги, может палиться антивирусами, а поскольку этот код — достаточно низкоуровневая сишечка и плюсы, то может даже «вызывать назальных демонов». Целью этой статьи является не дать вам готовый инструмент, а научить вас этот конкретный инструмент писать. И да, это сравнительно базовая статья, в которой я старался достучаться до как можно большего диапазона людей, от нубасов, желающих учится, до разной степени опытности разработчиков. Ну посмотрим, что у нас из этого выйдет.

Для начала неплохо было бы разобраться, а для чего собственно антивирусы, сендбоксы и ханипоты это (перехваты/хуки) делают? Давайте попробуем в вкратце ответить на этот вопрос. Так или иначе наша программа должна каким-то образом общаться с окружающей ее операционной системой и компьютером, на котором она исполняется. В противном случае программа была бы абсолютно бесполезна (как Хаскелл до изобретения монад). Операционная система является абстракцией над компьютером и его оборудованием. Таким образом, чтобы, например, считать файл, нашей программе нужно попросить, мол ну давай операционная система, покажи, что там в этом файле есть интересного. Для этого наша просьба проходит большое количество программных абстракций. Например, если мы вызываем для чтения файла метод из .NET File.ReadAllBytes, он в контексте своей работы вызовет CreateFile из kernel32.dll, он в свою очередь вызовет NtCreateFile из ntdll.dll, который в свою очередь сделает системный вызов, который уже обработает ядро операционной

системы, и результат вернется нам обратно по этой цепочке абстракций. Механизм системных вызовов — это отдельная интересная тема, но в контексте этой статьи вам нужно понимать только то, что с каждым системным вызовом ассоциирован его уникальный номер. Этот номер однозначно говорит ядру операционной системы, что конкретно мы от него хотим. В отличии от Линукса номера системных вызовов в Венде не фиксированы и могут меняться от версии к версии. Поэтому для разработчиков в общем случае рекомендуют использовать тонкий слой абстракции в виде ntdll.dll, либо более толстые слои абстракции в виде kernel32.dll и других библиотек. Но мы же с вами не пальцем деланные, верно?

Антивирусы, сендбоксы и ханипоты — очень коварны, им позарез нужно знать, а что именно делает наша программа непосредственно во время выполнения. Это реализуется для логирования, анализа или детектирования потенциально вредоносных действий. Очевидным решением этой задачи является перехват API-функций, которые использует программа, и анализ их параметров и результатов. Следует заметить, что самым логичным является перехват на самом нижнем уровне абстракции, поскольку в противном случае программа смогла бы обойти перехват, воспользовавшись функцией более низкого уровня абстракции. Для подавляющего большинства действий, которые имеет смысл мониторить с точки зрения антивирусов, сендбоксов и ханипотов, этот самый низкий уровень абстракции является библиотекой ntdll.dll. Ниже нее только уровень системных вызовов и ядро операционной системы. Некоторые антивирусы производят мониторинг на уровне ядра операционной системы, в общем случае, находясь в режиме пользователя, мы с этим ничего поделать не можем. Ну кроме разве что использования программных ошибок в реализации драйвера антивируса, или же эфемерных race condition атак, о которых слагали легенды, но которых никто в живую не видел.

Стоит так же немного рассказать о самом популярном методе перехвата API-функций, который в определенных кругах носит название «сплайсинг». Суть этого метода заключается в том, что первые инструкции оригинальной функции заменяются на безусловный переход (jmp, push+ret, не суть важно) на свой собственный обработчик. Этот обработчик произведет необходимый анализ аргументов функции, если понадобится вызовет оригинальную функцию и проанализирует результаты ее вызова. У метода сплайсинга есть ряд проблем. Некоторые антивирусы учитывают их, а некоторые просто забивают на них хер, так как их возникновение — весьма редкая ситуация. Давайте рассмотрим несколько таких проблем, ради интереса. Во-первых, в тот момент, когда мы будем патчить инструкции один или несколько других потоков могут эти инструкции исполнять. Эту проблему можно попробовать решить за счет остановки всех других потоков процесса и установки патча (модификации инструкций) атомарно. Во-вторых, среди тех инструкций, которые мы будем модифицировать могут находится условные или безусловные переходы. В этом случае

при патчинге нарушится граф потока исполнения функции и эта проблема уже совсем другого уровня. Самое простое решение в этом случае — ничего не решать. Если мы видим условный или безусловный переход в том месте, где нам нужно поставить сплайсинг, просто смирится и не ставить его. Ну хватит теории, давайте переходить к практике.

Для начала мы напишем инструмент, предназначенный для поиска и вывода перехваченных или же просто модифицированных API-функций в той или иной динамической библиотеке, а чуть попозже мы обсудим, что мы можем с этими перехватами сделать. Инструмент будет предельно прост, он будет загружать не модифицированный образ динамической библиотеки в память на кучу процесса, проходить по всем именованным экспортируемым функциями, сравнивать пролог этих функций в памяти и в загруженном с диска не модифицированном образе, получать и выводить дизассемблерный листинг изменений в удобном для сравнения виде. Сперва мы напишем несколько вспомогательных функций, который нам потом пригодятся, рассмотрим следующий код:
C++:

```
//---------------------------------------//
// Макросы для вывода сообщений об ошибках //
//---------------------------------------//
#define ERR Log("Error: %s line=%d code=%d", __FILE__, __LINE__, GetLastError())
#define ERR_HRES Log("Error: %s line=%d code=%x", __FILE__, __LINE__, hres)

//-------------------------------//
// Определение режима логирования //
//-------------------------------//
#define LOG_MODE_STDOUT  0
#define LOG_MODE_DBGVIEW 1
#define LOG_MODE_MSGBOX  2
#define LOG_MODE LOG_MODE_STDOUT

//---------------------------//
// Реализация выражения defer //
//---------------------------//
template <typename F>
struct privDefer {
    F f;
    privDefer(F f) : f(f) {}
    ~privDefer() { f(); }
};

//---------------------------//
// Реализация выражения defer //
//---------------------------//
template <typename F>
privDefer<F> defer_func(F f) {
    return privDefer<F>(f);
}

//---------------------------//
// Реализация выражения defer //
//---------------------------//
#define DEFER_1(x, y) x##y
#define DEFER_2(x, y) DEFER_1(x, y)
#define DEFER_3(x)    DEFER_2(x, __COUNTER__)
#define defer(code)   auto DEFER_3(_defer_) = defer_func([&](){code;})

//----------------------//
// Вывести строку в лог //
//----------------------//
VOID Log(LPCSTR frm, ...) {
    va_list args;
    va_start(args, frm);

    CHAR buf[512]; // Форматируем строку
    auto len = vsnprintf_s(buf, 512, frm, args);
    if(len == 0) { return; }
```

```
    for(int i = len - 1; buf[i] == '\n'; i--)
    { buf[i] = '\0'; }

    #if LOG_MODE == LOG_MODE_STDOUT
        puts(buf);
    #elif LOG_MODE == LOG_MODE_DBGVIEW
        OutputDebugStringA(buf);
    #elif LOG_MODE == LOG_MODE_MSGBOX
        MessageBoxA(NULL, buf, NULL, MB_OK);
    #else
        #error "Invalid log mode"
    #endif
}

//--------------------------//
// Выделение памяти на куче //
//--------------------------//
LPVOID Malloc(DWORD size) {
    // Получаем кучу процесса
    HANDLE heap = GetProcessHeap();
    if(heap == NULL) { ERR; return NULL; }

    // Выделяем память на куче процесса
    LPVOID buffer = HeapAlloc(heap, HEAP_ZERO_MEMORY, size);
    if(buffer == NULL) { ERR; return NULL; }
    else { return buffer; }
}

//----------------------------//
// Освобождение памяти на куче //
//----------------------------//
VOID Free(LPVOID buffer) {
    // Получаем кучу процесса
    HANDLE heap = GetProcessHeap();
    if(heap == NULL) { ERR; return; }

    // Освобождаем память на куче
    if(!HeapFree(heap, 0, buffer))
    { ERR; }
}

//-----------------------------------//
// Считать весь файл в буффер на куче //
//-----------------------------------//
LPVOID LoadFile(LPCWSTR path, LPDWORD rsize) {
    // Открываем файл для чтения данных
    HANDLE hfile = CreateFileW(path, GENERIC_READ,
    FILE_SHARE_VALID_FLAGS, NULL, OPEN_EXISTING, 0, NULL);
    if(hfile == INVALID_HANDLE_VALUE) { ERR; return NULL; }
    defer(CloseHandle(hfile));
```

```
    // Получаем размер буффера для файла
    DWORD size = GetFileSize(hfile, NULL);
    if(size == INVALID_FILE_SIZE)
    { ERR; return NULL; }

    // Создаем буффер для файла
    LPVOID buffer = Malloc(size);
    if(buffer == NULL) { ERR; return NULL; }

    DWORD read = 0; // Считываем данные файла в буффер
    if(!ReadFile(hfile, buffer, size, &read, NULL))
    { ERR; Free(buffer); return NULL; }

    // Возвращаем размер, если он нужен
    if(rsize != NULL) { *rsize = size; }

    // Возвращаем буффер
    return buffer;
}

//--------------------------------------//
// Считать файл этого конкретного модуля //
//--------------------------------------//
LPVOID LoadFile(HMODULE hmod, LPDWORD rsize) {
    WCHAR path[MAX_PATH]; // Получаем полный путь
    if(!GetModuleFileNameW(hmod, path, MAX_PATH))
    { ERR; return NULL; }

    // Считываем все данные файла
    LPVOID buffer = LoadFile(path, rsize);
    if(buffer == NULL) { ERR; return NULL; }
    else { return buffer; }
}

//----------------------------------//
// Получить указатель на NT-загловки //
//----------------------------------//
PIMAGE_NT_HEADERS GetNtHeaders(HMODULE hmod) {
    // Получаем DOS-заголовок
    auto dos_head = (PIMAGE_DOS_HEADER)hmod;
    if(dos_head->e_magic != IMAGE_DOS_SIGNATURE)
    { ERR; return NULL; }

    // Получаем NT-заголовки
    auto nt_head = (PIMAGE_NT_HEADERS)((LPBYTE)hmod + dos_head->e_lfanew);
    if(nt_head->Signature != IMAGE_NT_SIGNATURE) { ERR; return NULL; }
    else { return nt_head; }
}

//------------------------------------------------//
// Привести адрес в виртуальной памяти к смещению //
```

```
//-------------------------------------------------//
DWORD VaToFileOffset(HMODULE hmod, LPVOID address) {
    // Получаем NT-заголовки
    PIMAGE_NT_HEADERS nt_head = GetNtHeaders(hmod);
    if(nt_head == NULL) { ERR; return (DWORD)-1; }

    // Высчитываем RVA по базовому адресу и указателю
    auto rva = (DWORD)((LPBYTE)address - (LPBYTE)hmod);

    // Проходим по секциям и находим, в какой из них лежит RVA
    PIMAGE_SECTION_HEADER sec_head = IMAGE_FIRST_SECTION(nt_head);
    for(WORD i = 0; i < nt_head->FileHeader.NumberOfSections; i++) {
        DWORD start = sec_head[i].VirtualAddress;
        DWORD end = start + sec_head[i].SizeOfRawData;

        if(rva >= start && rva <= end) {
            return sec_head[i].PointerToRawData + (rva - start);
        }
    }

    // Не нашли смещение по
    // неведомой причине
    return (DWORD)-1;
}
```

Макросы ERR и ERR_HRES предназначены для вывода в консоль сообщений об ошибках, в том случае, если что-либо с кодом пойдет не так. Далее идет реализация выражения defer для языка C++ (аля Go, Nim и Zip), это своего рода RAII (выражение, написанное внутри defer будет выполнено при выходе из блока, в котором оно объявлено). Его очень удобно использовать в нативных языках программирования для освобождения каких-либо ресурсов. Функции Malloc и Free выделяют и освобождают данные на куче процесса. Функции LoadFile открывают и считывают файл на кучу процесса, возвращая указатель на созданный буфер для файла. Функция GetNtHeaders проверяет корректность сигнатур DOS-овского и NT-заголовков (смотри формат PE-файлов) и возвращает указатель на NT-заголовки. Функция VaToFileOffset по указателю в виртуальной памяти и базовому адресу динамической библиотеки, высчитывает смещение этого указатель внутри незаммапированного PE-файла.

Ранее в статье я сказал, что мы будем использовать дизассемблер и это была чистая правда. Вы в принципе можете использовать для этой задачи любой дизассемблер, который вам по душе, или же даже дизассемблер длин (если вам не нужно выводить дизассемблерный листинг на экран). Но я буду использовать дизассемблер, встроенный в отладчик, встроенный в операционную систему. А если более конкретно, то dbgeng.dll и COM-интерфейсы IDebugClient и IDebugControl3. Давайте рассмотрим следующий фрагмент кода:

C++:

```
//-------------------------------------------//
// Глобальные указатели на интерфейсы отладчика //
//-------------------------------------------//
static IDebugClient*   DebugClient  = NULL;
static IDebugControl3* DebugControl = NULL;

//---------------------------------------//
// Подцепить отладчик к текущему процессу //
//---------------------------------------//
bool DebuggerAttachSelf() {
    // Инициализация COM
    CoInitialize(NULL);

    // Объявление переменных интерфейсов
    auto control = (IDebugControl3*)NULL;
    auto client = (IDebugClient*)NULL;

    // Создание интерфейса клиента отладчика
    auto hres = DebugCreate(IID_IDebugClient, (LPVOID*)&client);
    if(FAILED(hres)) { ERR_HRES; return false; }

    // Создание интерфейса контроллера клиента отладчика
    hres = client->QueryInterface(IID_IDebugControl3, (LPVOID*)&control);
    if(FAILED(hres)) { ERR_HRES; return false; }

    auto pid = GetCurrentProcessId(); // Подключение к текущему процессу
    hres = client->AttachProcess(0, pid, DEBUG_ATTACH_NONINVASIVE |
DEBUG_ATTACH_NONINVASIVE_NO_SUSPEND);
    if(FAILED(hres)) { ERR_HRES; return false; }

    // Запуск ожидание событий отладки
    hres = control->WaitForEvent(DEBUG_WAIT_DEFAULT, INFINITE);
    if(FAILED(hres)) { ERR_HRES; return false; }

    // Сохраняем интерфейсы в
    // глобальные переменные
    DebugControl = control;
    DebugClient = client;
    return true;
}

//-------------------------------------------------//
// Дизассемблировать инструкцию по переданному адресу //
//-------------------------------------------------//
DWORD DebuggerDisassemble(LPVOID address, LPSTR string, DWORD slen) {
    ULONG64 prev = (ULONG64)address;
    ULONG64 next = 0;
    DWORD len = 0;

    // Вызов отладчика для дизассемблирования инструкции
    auto hres = DebugControl->Disassemble((ULONG64)address, 0, string, slen, &len, &next);
```

```
    if(FAILED(hres)) { ERR_HRES; return 0; }

    // Возврат длины инструкции
    return (DWORD)(next - prev);
}

//----------------------//
// Отключение отладчика //
//----------------------//
void DebuggerDetachSelf() {
    // Освобождение контрола
    if(DebugControl != NULL) {
        DebugControl->Release();
        DebugControl = NULL;
    }

    // Освобождение клиента
    if(DebugClient != NULL) {
        DebugClient->DetachProcesses();
        DebugClient->Release();
        DebugClient = NULL;
    }

    // Освобождение COM
    CoUninitialize();
}
```

Сначала мы объявим две глобальные переменные, которые будут содержать в себе указатели на интерфейсы DebugClient и IDebugControl3 класса отладчика. Функция AttachDebugger инициализирует и подключает отладчик к текущему процессу. В начале функции мы инициализируем инфраструктуру COM и создаем клиент для отладчика с помощью функции DebugCreate. Затем у класса отладчика мы запрашиваем интерфейс контроллера отладчика. Далее мы просто подключаем отладчик к текущему процессу и говорим ему ожидать различных отладочных событий. Вообще говоря, вполне вероятно, что эту штуку можно использовать для противодействия отладке, но я не проверял, это так, небольшой вброс. Функция Disassemble дизассемблирует инструкцию по переданному адресу в строковый буфер и возвращает ее длину. Да-да, мы будем ее использовать и в качестве LDE (length disassembly engine или дизассемблер длин инструкций, кому как больше нравится). Функция DetachDebugger отключает отладчик от текущего процесса и освобождает оба интерфейса, затем производит деинициализацию COM инфраструктуры.

Далее мы напишем функции для проведения анализа на наличие изменений в загруженном образе произвольной динамической библиотеки. По базовому адресу динамической библиотеки мы получим полный путь до ее файла, загрузим его и поищем, какие изменения были произведены. Рассмотрим следующий код:

C++:

```
//-------------------------------------------------------//
// Проверка изменений в коде функции на диске и в памяти //
//-------------------------------------------------------//
VOID CheckForChanges(LPCSTR name, LPBYTE ptr1, LPBYTE ptr2) {
    DWORD changes = 0; // Счетчик изменений
    for(DWORD i = 0; i < 32; i++) {
        if(ptr1[i] == ptr2[i]) { break; }
        else { changes = changes + 1; }
    }

    // Изменений нет, так что выходим
    if(changes == 0) { return; }

    // Буффер для листинга
    // дизассемблерного кода
    CHAR buf[256];
    DWORD len = 0;

    // Выводим инструкции на диске
    Log("Original %s:", name);
    for(DWORD i = 0; i < changes; i += len) {
        len = DebuggerDisassemble(&ptr1[i], buf, 256);
        if(len == 0) { ERR; return; }
        Log("%s", buf);
    }

    // Выводим новую строку
    Log(" ");

    // Выводим инструкции в памяти
    Log("Changed %s:", name);
    for(DWORD i = 0; i < changes; i += len) {
        len = DebuggerDisassemble(&ptr2[i], buf, 256);
        if(len == 0) { ERR; return; }
        Log("%s", buf);
    }

    // Выводим новую строку
    Log(" "); Log(" ");
}

//-------------------------------------//
// Провести анализ модуля по указателю //
//-------------------------------------//
VOID Analyze(HMODULE hmod) {
    // Считываем модуль из файла
    LPVOID buffer = LoadFile(hmod);
    if(buffer == NULL) { ERR; return; }
    defer(Free(buffer));

    // Получаем NT-заголовки
```

```
    auto nt_head = GetNtHeaders(hmod);
    if(nt_head == NULL) { ERR; return; }

    // Получаем данные о таблице экспорта
    auto dir_id = IMAGE_DIRECTORY_ENTRY_EXPORT;
    auto exp_va = nt_head->OptionalHeader.DataDirectory[dir_id].VirtualAddress;
    auto exp_tbl = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)hmod + exp_va);
    auto name_rva = (LPDWORD)((LPBYTE)hmod + exp_tbl->AddressOfNames);

    // Проходим по всем записям экспорта по имени
    for(DWORD i = 0; i < exp_tbl->NumberOfNames; i++) {
        auto name = (LPCSTR)((LPBYTE)hmod + name_rva[i]);
        auto address = (LPVOID)GetProcAddress(hmod, name);
        auto offset = VaToFileOffset(hmod, address);
        if(offset == (DWORD)-1) { continue; }

        // Проверяем каждый из них
        auto ptr2 = (LPBYTE)address;
        auto ptr1 = (LPBYTE)buffer + offset;
        CheckForChanges(name, ptr1, ptr2);
    }
}

//---------------------------------//
// Провести анализ модуля по имени //
//---------------------------------//
VOID Analyze(LPCWSTR dllname) {
    auto hmod = GetModuleHandleW(dllname);
    if(hmod == NULL) { ERR; }
    else { Analyze(hmod); }
}
```

Функция Analyze загружает данные файла библиотеки с диска, получает указатель на NT-заголовки PE-файла, находит таблицу экспорта и в цикле проходит по всем именованным экспортам динамической библиотеки. Для каждой экспортируемой по имени функции определяет смещение начала этой функции в файле с диска, а затем вызывает функцию CheckForChanges для имени функции и двух указателей: указатель на оригинальный код, загруженный с диска, и указатель на код функции в памяти. Функция CheckForChanges считает, сколько байт в функции изменилось, а затем выводит дизассемблерный листинг для этих изменений.

Для тестирования я создал два исполняемых файла analyze32.exe и analyze64.exe, которые выводят изменения для двух самых актуальных в контексте статьи динамических библиотек: ntdll.dll и kernel32.dll. Я и X-Shar запускали их на компьютерах с антивирусами, давайте в качестве примера рассмотрим результат выполнения для одного из антивирусов (тот, кто угадает, что это за антивирус — получит лайк от меня):

Code:

```
Original NtCreateFile:
00000000`02539210 4c8bd1          mov     r10,rcx
00000000`02539213 b852000000      mov     eax,52h

Changed NtCreateFile:
00000000`77029dd0 e938fd76fd      jmp     SYSFER+0x19b0d (00000000`74799b0d)


Original NtCreateKey:
00000000`02538e90 4c8bd1          mov     r10,rcx
00000000`02538e93 b81a000000      mov     eax,1Ah

Changed NtCreateKey:
00000000`77029a50 e9f40077fd      jmp     SYSFER+0x19b49 (00000000`74799b49)


Original NtCreateUserProcess:
00000000`02539790 4c8bd1          mov     r10,rcx
00000000`02539793 b8aa000000      mov     eax,0AAh

Changed NtCreateUserProcess:
00000000`7702a350 e930f876fd      jmp     SYSFER+0x19b85 (00000000`74799b85)


Original NtDeleteFile:
00000000`02539810 4c8bd1          mov     r10,rcx
00000000`02539813 b8b2000000      mov     eax,0B2h

Changed NtDeleteFile:
00000000`7702a3d0 e9ecf776fd      jmp     SYSFER+0x19bc1 (00000000`74799bc1)


Original NtDeleteKey:
00000000`02539820 4c8bd1          mov     r10,rcx
00000000`02539823 b8b3000000      mov     eax,0B3h

Changed NtDeleteKey:
00000000`7702a3e0 e944f976fd      jmp     SYSFER+0x19d29 (00000000`74799d29)


Original NtDeleteValueKey:
00000000`02539850 4c8bd1          mov     r10,rcx
00000000`02539853 b8b6000000      mov     eax,0B6h

Changed NtDeleteValueKey:
00000000`7702a410 e9e8f776fd      jmp     SYSFER+0x19bfd (00000000`74799bfd)


Original NtMapViewOfSection:
00000000`02538f40 4c8bd1          mov     r10,rcx
00000000`02538f43 b825000000      mov     eax,25h
```

```
Changed NtMapViewOfSection:
00000000`77029b00 e9340177fd      jmp     SYSFER+0x19c39 (00000000`74799c39)


Original NtOpenFile:
00000000`02538ff0 4c8bd1          mov     r10,rcx
00000000`02538ff3 b830000000      mov     eax,30h

Changed NtOpenFile:
00000000`77029bb0 e9c00077fd      jmp     SYSFER+0x19c75 (00000000`74799c75)


Original NtOpenKey:
00000000`02538de0 4c8bd1          mov     r10,rcx
00000000`02538de3 b80f000000      mov     eax,0Fh

Changed NtOpenKey:
00000000`770299a0 e90c0377fd      jmp     SYSFER+0x19cb1 (00000000`74799cb1)


Original NtOpenKeyEx:
00000000`02539c10 4c8bd1          mov     r10,rcx
00000000`02539c13 b8f2000000      mov     eax,0F2h

Changed NtOpenKeyEx:
00000000`7702a7d0 e918f576fd      jmp     SYSFER+0x19ced (00000000`74799ced)


Original NtRenameKey:
00000000`0253a0a0 4c8bd1          mov     r10,rcx
00000000`0253a0a3 b83b010000      mov     eax,13Bh

Changed NtRenameKey:
00000000`7702ac60 e900f176fd      jmp     SYSFER+0x19d65 (00000000`74799d65)


Original NtSetInformationFile:
00000000`02538f30 4c8bd1          mov     r10,rcx
00000000`02538f33 b824000000      mov     eax,24h

Changed NtSetInformationFile:
00000000`77029af0 e9ac0277fd      jmp     SYSFER+0x19da1 (00000000`74799da1)


Original NtSetValueKey:
00000000`025392c0 4c8bd1          mov     r10,rcx
00000000`025392c3 b85d000000      mov     eax,5Dh

Changed NtSetValueKey:
00000000`77029e80 e958ff76fd      jmp     SYSFER+0x19ddd (00000000`74799ddd)
```

```
Original NtTerminateProcess:
00000000`02538f80 4c8bd1          mov     r10,rcx
00000000`02538f83 b829000000      mov     eax,29h

Changed NtTerminateProcess:
00000000`77029b40 e9d40277fd      jmp     SYSFER+0x19e19 (00000000`74799e19)


Original NtTerminateThread:
00000000`025391f0 4c8bd1          mov     r10,rcx
00000000`025391f3 b850000000      mov     eax,50h

Changed NtTerminateThread:
00000000`77029db0 e9a00077fd      jmp     SYSFER+0x19e55 (00000000`74799e55)


Original ZwCreateFile:
00000000`02539210 4c8bd1          mov     r10,rcx
00000000`02539213 b852000000      mov     eax,52h

Changed ZwCreateFile:
00000000`77029dd0 e938fd76fd      jmp     SYSFER+0x19b0d (00000000`74799b0d)


Original ZwCreateKey:
00000000`02538e90 4c8bd1          mov     r10,rcx
00000000`02538e93 b81a000000      mov     eax,1Ah

Changed ZwCreateKey:
00000000`77029a50 e9f40077fd      jmp     SYSFER+0x19b49 (00000000`74799b49)


Original ZwCreateUserProcess:
00000000`02539790 4c8bd1          mov     r10,rcx
00000000`02539793 b8aa000000      mov     eax,0AAh

Changed ZwCreateUserProcess:
00000000`7702a350 e930f876fd      jmp     SYSFER+0x19b85 (00000000`74799b85)


Original ZwDeleteFile:
00000000`02539810 4c8bd1          mov     r10,rcx
00000000`02539813 b8b2000000      mov     eax,0B2h

Changed ZwDeleteFile:
00000000`7702a3d0 e9ecf776fd      jmp     SYSFER+0x19bc1 (00000000`74799bc1)


Original ZwDeleteKey:
```

```
00000000`02539820 4c8bd1          mov     r10,rcx
00000000`02539823 b8b3000000      mov     eax,0B3h

Changed ZwDeleteKey:
00000000`7702a3e0 e944f976fd      jmp     SYSFER+0x19d29 (00000000`74799d29)


Original ZwDeleteValueKey:
00000000`02539850 4c8bd1          mov     r10,rcx
00000000`02539853 b8b6000000      mov     eax,0B6h

Changed ZwDeleteValueKey:
00000000`7702a410 e9e8f776fd      jmp     SYSFER+0x19bfd (00000000`74799bfd)


Original ZwMapViewOfSection:
00000000`02538f40 4c8bd1          mov     r10,rcx
00000000`02538f43 b825000000      mov     eax,25h

Changed ZwMapViewOfSection:
00000000`77029b00 e9340177fd      jmp     SYSFER+0x19c39 (00000000`74799c39)


Original ZwOpenFile:
00000000`02538ff0 4c8bd1          mov     r10,rcx
00000000`02538ff3 b830000000      mov     eax,30h

Changed ZwOpenFile:
00000000`77029bb0 e9c00077fd      jmp     SYSFER+0x19c75 (00000000`74799c75)


Original ZwOpenKey:
00000000`02538de0 4c8bd1          mov     r10,rcx
00000000`02538de3 b80f000000      mov     eax,0Fh

Changed ZwOpenKey:
00000000`770299a0 e90c0377fd      jmp     SYSFER+0x19cb1 (00000000`74799cb1)


Original ZwOpenKeyEx:
00000000`02539c10 4c8bd1          mov     r10,rcx
00000000`02539c13 b8f2000000      mov     eax,0F2h

Changed ZwOpenKeyEx:
00000000`7702a7d0 e918f576fd      jmp     SYSFER+0x19ced (00000000`74799ced)


Original ZwRenameKey:
00000000`0253a0a0 4c8bd1          mov     r10,rcx
00000000`0253a0a3 b83b010000      mov     eax,13Bh
```

```
Changed ZwRenameKey:
00000000`7702ac60 e900f176fd      jmp     SYSFER+0x19d65 (00000000`74799d65)


Original ZwSetInformationFile:
00000000`02538f30 4c8bd1          mov     r10,rcx
00000000`02538f33 b824000000      mov     eax,24h

Changed ZwSetInformationFile:
00000000`77029af0 e9ac0277fd      jmp     SYSFER+0x19da1 (00000000`74799da1)


Original ZwSetValueKey:
00000000`025392c0 4c8bd1          mov     r10,rcx
00000000`025392c3 b85d000000      mov     eax,5Dh

Changed ZwSetValueKey:
00000000`77029e80 e958ff76fd      jmp     SYSFER+0x19ddd (00000000`74799ddd)


Original ZwTerminateProcess:
00000000`02538f80 4c8bd1          mov     r10,rcx
00000000`02538f83 b829000000      mov     eax,29h

Changed ZwTerminateProcess:
00000000`77029b40 e9d40277fd      jmp     SYSFER+0x19e19 (00000000`74799e19)


Original ZwTerminateThread:
00000000`025391f0 4c8bd1          mov     r10,rcx
00000000`025391f3 b850000000      mov     eax,50h

Changed ZwTerminateThread:
00000000`77029db0 e9a00077fd      jmp     SYSFER+0x19e55 (00000000`74799e55)
```

Как мы видим в этих логах, антивирус перехватывает большое количество функций из библиотеки ntdll.dll. В том числе анализу подвергаются манипуляции с файлами, ключами реестра, процессами, потоками и виртуальной памятью. По этим перехватам можно судить о возможностях проактивной защиты антивируса, а именно, какие конкретные функции и действия приложения могут быть подвергнуты анализу. Теперь, когда у вас есть готовые утилиты для вывода перехватов системных функции, если вам будет интересно, попробуйте позапускать эти утилиты на своих тестовых компьютерах с антивирусами и сендбоксами, и поделитесь получившимися логами. Интересно было бы собрать и проанализировать статистику по этому вопросу.

Теперь давайте подумаем о том, а как, собственно, мы можем обойти такие перехваты. В известной малвари применялись различные методики. Некоторые копировали ntdll.dll в файл с другим именем, загружали в свой процесс с помощью LoadLibrary и вызывали нужные им функции. Некоторые загружали второй образ ntdll.dll в память процесса с помощью собственного загрузчика. Я предложу вам еще несколько вариантов, которые мы рассмотрим ниже. Но хочу вас предупредить, реализация, представленная в статье, разрабатывалась и тестировалась для 64-битных приложений, запущенных на 64-битной операционной системе. Сделать тоже самое для 32-битных приложений, запущенных на 32-битной операционной системе, не позволило время, текущий и так уже большой объем текста в статье, отсутствие у меня 32-битных операционных систем в виртуалках и еще куча других отмазок, которые я мог бы придумать. Для 32-битных приложений, запущенных на 64-битной операционной системе можно воспользоваться техникой Heavens Gate. Она весьма и весьма сложная и тянет на отдельную статью, поэтому в этой статье я не буду ее рассматривать, но дам вам ссылку на оригинальную статью по ней на русском: http://hex.pp.ua/heavens-gate.php и несколько ссылок на библиотеки, с разной степенью успеха реализующих эту технику: https://github.com/dadas190/Heavens-Gate-2.0 и https://github.com/JustasMasiulis/wow64pp

Для начала мы напишем простой тест, чтобы в последствии убедиться, что наши хитросделанные (ну вы поняли) вызовы сисколлов отрабатывают так же, как и оригинальные. Тестирование мы будем производить на компьютере без антивирусов, так как для нас сейчас важнее корректность алгоритма, нежели обходы конкретного антивируса. В качестве примера мы будем использовать системный вызов NtOpenProcess и с его помощью попробуем открыть системный процесс с идентификатором 4, при этом само собой ожидая в ответ получить NTSTATUS 0xC0000005 или 0xC0000022, то есть access denied. Напишем такую функцию (подробное объяснение ее функционала, я думаю излишне):
C++:

```
//-----------------------------//
// Тест оригинального сисколла //
//-----------------------------//
HANDLE TestOriginal(DWORD pid) {
    // Получаем базовый адрес ntdll.dll
    auto hmod = GetModuleHandleW(L"ntdll.dll");
    if(hmod == NULL) { ERR; return NULL; }

    // Получаем адрес необходимого нам сискола
    auto func = (PNtOpenProcess)GetProcAddress(hmod, "NtOpenProcess");
    if(func == NULL) { ERR; return NULL; }

    OBJECT_ATTRIBUTES attrs; // Инициализируем аттрибуты
    InitializeObjectAttributes(&attrs, NULL, 0, NULL, NULL);

    // Инициализируем pid
    CLIENT_ID clid = { 0 };
    clid.UniqueProcess = (PHANDLE)(ULONGLONG)pid;

    auto han = (HANDLE)NULL; // Пытаемся открыть хендл процесса
    auto res = func(&han, PROCESS_ALL_ACCESS, &attrs, &clid);
    Log("Original result is 0x%X", res);
    return han;
}
```

В качестве первого примера обхода создадим код, который будет копировать инструкции оригинального системного вызова в исполняемый буфер из оригинальной динамической библиотеки на диске. Рассмотрим следующий код:
C++:

```
//----------------------------------------//
// Тест с копировнием кода сисколла с диска //
//----------------------------------------//
HANDLE TestDuplicated(DWORD pid) {
    // Получаем базовый адрес ntdll.dll
    auto hmod = GetModuleHandleW(L"ntdll.dll");
    if(hmod == NULL) { ERR; return NULL; }

    // Получаем адрес необходимого нам сискола
    auto func = (LPVOID)GetProcAddress(hmod, "NtOpenProcess");
    if(func == NULL) { ERR; return NULL; }

    // Считываем модуль из файла
    LPVOID buffer = LoadFile(hmod);
    if(buffer == NULL) { ERR; return NULL; }
    defer(Free(buffer));

    // Получаем смещение сисколла в файле
    auto offset = VaToFileOffset(hmod, func);
    if(offset == (DWORD)-1) { ERR; return NULL; }

    // Выделяем буффер исполняемой памяти
    auto exec = VirtualAlloc(NULL, 4096, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if(exec == NULL) { ERR; return NULL; }
    defer(VirtualFree(exec, 4096, MEM_RELEASE));

    DWORD position = 0;
    CHAR string[256];

    // Копируем 16 инструкции, чтобы наверняка
    for(int i = 0; i < 16; i++) {
        auto pt1 = (LPBYTE)exec + position;
        auto pt2 = (LPBYTE)func + position;
        auto pt3 = (LPBYTE)buffer + offset + position;
        auto len = DebuggerDisassemble(pt3, string, 256);
        if(len == 0) { ERR; return NULL; }

        memcpy(pt1, pt3, len);
        position += len;
    }

    // Преводим адрес к функции
    auto func2 = (PNtOpenProcess)exec;

    OBJECT_ATTRIBUTES attrs; // Инициализируем аттрибуты
    InitializeObjectAttributes(&attrs, NULL, 0, NULL, NULL);

    // Инициализируем pid
    CLIENT_ID clid = { 0 };
    clid.UniqueProcess = (PHANDLE)(ULONGLONG)pid;
```

```
    auto han = (HANDLE)NULL; // Пытаемся открыть хендл процесса
    auto res = func2(&han, PROCESS_ALL_ACCESS, &attrs, &clid);
    Log("Duplicated result is 0x%X", res);
    return han;
}
```

Этим кодом мы получаем адрес системного вызова в памяти процесса, загружаем оригинальную динамическую библиотеку ntdll.dll, определяем смещение кода системного вызова, выделяем буфер исполняемой памяти, с помощью дизассемблера длин копируем первые 16 инструкций (системный вызов точно меньше 16 инструкций, копируем 16 инструкций, чтобы уж наверняка угадать) в исполняемый буфер и запускаем их. При тестировании мы получаем тот же результат, что и с эталонным тестом, который мы сделали ранее. Следует заметить, что скорее всего ровно тот же алгоритм будет работать для 32-битных приложений, запущенных на 32-битной операционной система (как сказано ранее, я не тестил, но пока не вижу причин, чтобы это не работало). Очевидно, что для 32-битных приложений, запущенных на 64-битной системе, это работать не будет, так как в оригинальной динамической библиотеке в системных вызовах не прописан KiFastSystemCall. В принципе, при желании это можно реализовать, но для этого понадобится чуток модифицировать копируемый код системного вызова, прописав в нужное место нужную ссылку.

Вторым кодом, мы попробуем написать шаблонную функцию, которая запустит системный вызов по известному на этапе компиляции номеру. Как я сказал ранее, номера системных вызовов могут меняться от версии к версии, но все эти номера известны и опубликованы в https://github.com/j00ru/windows-syscalls так что в принципе мы можем допустить хардкод этих номеров в своей программе. Рассмотрим код:
C++:

```
//-------------------------------------------------------//
// Шаблон для вызова сисколла с захардкоженным номером //
//-------------------------------------------------------//
template <DWORD Num, typename RetType, typename ... Args>
__attribute__((naked)) RetType Syscall(Args ... args) {
    asm volatile("mov r10, rcx\n"
                 "mov eax, %0\n"
                 "syscall\n"
                 "ret" :: "r" (Num));
}

//---------------------------------//
// Тест с хардкодом номера сисколла //
//---------------------------------//
HANDLE TestSyscallHardcode(DWORD pid) {
    OBJECT_ATTRIBUTES attrs; // Инициализируем аттрибуты
    InitializeObjectAttributes(&attrs, NULL, 0, NULL, NULL);

    // Инициализируем pid
    CLIENT_ID clid = { 0 };
    clid.UniqueProcess = (PHANDLE)(ULONGLONG)pid;

    auto han = (HANDLE)NULL;
    auto res = Syscall<0x26, NTSTATUS>(&han, PROCESS_ALL_ACCESS, &attrs, &clid);
    Log("Hardcoded syscall result is 0x%X", res);
    return han;
}
```

Шаблонная функция Syscall принимает в качестве аргумента шаблона номер системного вызова. Я тестировал ее на операционной системе Windows 10, где номер системного вызова NtOpenProcess равен 0x26. Шаблонная функция объявлена как naked, то есть у нее не будет типичного для всех сишных функций пролога. В теле функции объявлена ассемблерная вставка, которая и произведет системный вызов. Нам осталось только запустить ее и удостовериться, что мы получим тот же результат, что и в эталонном тесте. Этот вариант, за исключением того, что номера системных вызовов придется хардкодить и надо следить за тем, что эти номера могут измениться в более новых версиях операционных систем семейства Виндовс, является достаточно простым в реализации. А в том случае, если антивирусы начнут критично относится к чтению данных ntdll.dll, других вариантов вероятно и не будет.

Теперь давайте рассмотрим третий вариант. В нем мы считаем оригинальную динамическую библиотеку ntdll.dll с диска, найдем в ней номер системного вызова и примерно такой же шаблонной функцией произведем его запуск. Рассмотрим следующий код:
C++:

```
//-----------------------------------------------//
// Шаблон для вызова сисколла с полученным номером //
//-----------------------------------------------//
template <typename RetType, typename ... Args>
__attribute__((naked)) RetType Syscall(Args ... args, DWORD num) {
    asm volatile("mov r10, rcx\n"
                 "mov eax, %0\n"
                 "syscall\n"
                 "ret" :: "r" (num));
}

//---------------------------------------//
// Получение номера сисколла из функции //
//---------------------------------------//
DWORD GetSyscallNumber(LPVOID func) {
    auto ptr = (LPBYTE)func;
    for(int i = 0; i < 16; i++) {
        if(ptr[0] == 0xB8) { return *(LPDWORD)&ptr[1]; }
        auto len = DebuggerDisassemble(ptr, NULL, 0);
        ptr = ptr + len;
    }

    ERR; return 0;
}

//-----------------------------------//
// Тест с получением номера сисколла //
//-----------------------------------//
HANDLE TestSyscallDynamic(DWORD pid) {
    // Получаем базовый адрес ntdll.dll
    auto hmod = GetModuleHandleW(L"ntdll.dll");
    if(hmod == NULL) { ERR; return NULL; }

    // Получаем адрес необходимого нам сискола
    auto func = (LPVOID)GetProcAddress(hmod, "NtOpenProcess");
    if(func == NULL) { ERR; return NULL; }

    // Считываем модуль из файла
    LPVOID buffer = LoadFile(hmod);
    if(buffer == NULL) { ERR; return NULL; }
    defer(Free(buffer));

    // Получаем смещение сисколла в файле
    auto offset = VaToFileOffset(hmod, func);
    if(offset == (DWORD)-1) { ERR; return NULL; }

    // Получаем номер сисколла из файла
    auto sysptr = (LPBYTE)buffer + offset;
    auto sysnum = GetSyscallNumber(sysptr);
    if(sysnum == 0) { ERR; return NULL; }
```

```
    OBJECT_ATTRIBUTES attrs; // Инициализируем аттрибуты
    InitializeObjectAttributes(&attrs, NULL, 0, NULL, NULL);

    // Инициализируем pid
    CLIENT_ID clid = { 0 };
    clid.UniqueProcess = (PHANDLE)(ULONGLONG)pid;

    auto han = (HANDLE)NULL;
    auto res = Syscall<NTSTATUS, PHANDLE, DWORD, POBJECT_ATTRIBUTES, PCLIENT_ID>(&han,
PROCESS_ALL_ACCESS, &attrs, &clid, sysnum);
    Log("Dynamic syscall (0x%X) result is 0x%X", sysnum, res);
    return han;
}
```

Шаблонная функция Syscall в этот раз принимает номер системного вызова последним параметром. Это сделано для того, чтобы не нарушить расположение параметров системного вызова. В принципе ничего не мешает корректно обработать его передачу через, допустим, первый параметр шаблонной функции, но для простоты я не стал этим заморачиваться. Функция GetSyscallNumber с помощью дизассемблера длин находит инструкцию mov eax, <dword> по первому ее байту (0xB8) и возвращает его. Как и раньше в функции для тестов мы получаем адрес системного вызова в памяти, считываем оригинальную динамическую библиотеку, высчитываем смещение системного вызова в данных оригинальной библиотеки, получаем номер системного вызова с помощью дизассемблера длин и, собственно, запускаем его. Ну и проверим, что такой вызов системного вызова нормально отрабатывает.

В качестве заключения я хочу вкратце коснуться такой ситуации, когда антивирус, сендбокс или ханипот перехватывает не системный вызов, а произвольную функцию, которую вам позарез нужно вызвать. Пытаться снимать хуки не стоит, так как перехватывающая функцию библиотека может хуки восстанавливать, а этим противодействием вы повышаете вероятность того, что процесс упадет, когда один из потоков попытается исполнить изменяемые инструкции. Алгоритм действий в этой ситуации уже должен быть вам интуитивно понятен. Загружаем данные оригинальной динамической библиотеки, находим в этих данных нужную нам функцию. Далее определяем количество измененных байт. Выделяем исполняемый буфер и с помощью дизассемблера длин копируем оригинальные инструкции, которые в виртуальной памяти были изменены. В конец скопированных инструкций добавляем безусловный переход (jmp или push+ret) на первую неизмененную инструкцию функции библиотеки в виртуальной памяти. Вызываем функцию через указатель на исполняемый буфер. Таким образом поток исполнения нашей программы будет просто как бы «обходит» установленный перехват.

Ну что ж, спасибо за внимание! Прошу меня извинить за какие-то потенциальные баги, которые могут присутствовать в представленном мной коде. Эта тема достаточно новая для меня, изучил и накодил ее буквально за несколько последних дней, поскольку обсуждение в теме X-Shar`а пробудило интерес к данному вопросу. Ну и конечно, я же не могу держать язык за зубами, и некоторым участникам нашего уютненького комьюнити я уже успел пообещать статью на эту тему. Надеюсь, что вам было интересно, пишите свои вопросы, критикуйте, обсудим все это.