Статья Тактики Red Team: Использование системных вызовов в C# - Написание кода

xss.is/threads/42197

В моем предыдущем посте "Тактики Red Team: использование системных вызовов на C# - предварительные знания" мы рассмотрели некоторые основные предварительные концепции, которые нам нужно было понять, прежде чем мы сможем использовать системные вызовы на C#.
Мы затронули некоторые подробные темы, такие как внутреннее устройство Windows и, конечно же, системные вызовы. Мы также рассмотрели, как работает .NET Framework и как мы можем использовать неуправляемый код на C# для выполнения наших системных вызовов на ассемблере.

Теперь, если вы еще не читали мой предыдущий пост - настоятельно рекомендую вам это сделать. В противном случае вы можете потеряться и будете совершенно не знакомы с некоторыми из представленных здесь тем. Конечно, я постараюсь объяснить как можно лучше и предоставлю ссылки на внешние ресурсы по некоторым темам - но все (в основном все), о чем здесь пойдет речь, находится в предыдущем посте!

В сегодняшней статье мы сосредоточимся на написании кода для выполнения действительного системного вызова, используя все, что мы узнали. Помимо написания кода, мы также рассмотрим некоторые концепции управления нашим кодом, чтобы мы могли подготовить его для будущей интеграции между другими инструментами. Эта идея интеграции будет похожа на то, как SharpSploit от Ryan Cobb был разработан для интеграции с другими проектами C#, но наши не дойдут до такой степени.

Моя первоначальная идея для этой части статьи заключалась в том, чтобы показать вам, как разработать реальный инструмент, который мы могли бы использовать во время операций, например Dumpert или SysWhispers. Но после некоторого размышления о том, насколько длинным и сложным будет сообщение в статье, я вместо этого решил написать простой PoC (Proof of Concept), демонстрирующий выполнение одного системного вызова.

Я искренне верю, что после прочтения этой статьи в блоге и просмотра примера кода (который я также опубликую на GitHub) вы сможете самостоятельно написать код для инструмента! Я также добавлю несколько ссылок на инструменты, которые используют те же концепции системных вызовов в C#, в конце этого сообщения, если вам нужно больше вдохновения.

Кто знает, может быть, я сделаю стрим, где мы все вместе сможем написать новый крутой инструмент!

Хорошо, разобравшись с этим, давайте откроем Visual Studio или Visual Code и поработаем руками набросав немного кода!

Разработка нашего кода и структуры классов

Если есть одна вещь, которую я узнал при написании пользовательских инструментов для red team - будь то вредоносное ПО или какой-то имплант - это то, что нам нужно организовать наш код и идею и разделить их на классы.

Классы - один из самых фундаментальных типов C#. Проще говоря, класс - это структура данных, которая объединяет поля и методы (а также другие функции-члены) в один блок. Конечно, классы могут использоваться как объекты и поддерживать наследование и полиморфизм, которые являются механизмами, с помощью которых наши производные классы могут расширять и определять другие базовые классы.

После создания эти классы можно затем использовать в нашей кодовой базе, добавив директиву using в другой файл исходного кода. Затем это позволит нам получить доступ к статическим членам наших предыдущих классов и вложенным типам без необходимости уточнять доступ с помощью имени класса.

Например, предположим, что у нас есть новый класс под названием "Syscalls", в котором хранится наша логика системных вызовов. Если бы мы не добавили директиву using в наш код C#, тогда нам нужно было бы квалифицировать нашу функцию с полным именем класса. Итак, если бы наш класс Syscalls содержал ассемблерный код syscall для NtWriteFile, то для доступа к этому методу внутри другого класса мы бы сделали что-то вроде Syscalls.NtWriteFile. Это нормально, но после нескольких раз вызов класса становится утомительным.

Теперь некоторые из вас могут спросить: "Зачем нам это нужно?"

Две причины. Во-первых, это для организационных целей и для того, чтобы наш код был "чистым". Во-вторых, это позволяет нам с легкостью отлаживать и исправлять проблемы в нашем коде, вместо того, чтобы пролистывать массивный кусок текста и пытаться найти точку с запятой.

Помимо этого, давайте попробуем организовать наш код! Для начала давайте создадим новый проект для консольного приложения .NET Framework и настроим его на использование .NET Framework 3.5 - вот так.

Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

SharpCall

XSS.is

Location

C:\Users\User\Source\Repos

Solution name

SharpCall

Place solution and project in the same directory

Framework

.NET Framework 3.5

После завершения у вас должен быть доступ к новому файлу C# под названием Program.cs.
Если мы посмотрим на правую часть Visual Studio, мы заметим, что в нашем Solution Explorer у нас есть следующая структура.

> +SharpCall SLN (Solution)
> |
> +->Properties
> |
> +->References
> |
> +->Program.cs (Main Program)

Наш файл Program.cs будет содержать основную логику нашего приложения. В случае нашего PoC мы захотим вызвать и использовать наши системные вызовы в этом файле. Как было замечено ранее, системные вызовы происходят внутри ЦП, когда

инструкция syscall вызывается вместе с допустимым идентификатором syscall. Эта инструкция заставляет ЦП переключаться из пользовательского режима в режим ядра для выполнения определенных привилегированных операций.

Если бы мы использовали только один системный вызов, мы могли бы просто включить его в файл Program.cs. Но, поступая так, мы могли бы вызвать у себя головную боль, если бы в дальнейшем мы решили построить эту программу либо с большей модульностью, либо с гибкостью, чтобы упростить интеграцию с другими приложениями - будь то дропперы или вредоносное ПО.

Поэтому нам нужно всегда думать о будущем - и для начала было бы неплохо разделить все наши ассемблерные системные вызовы в отдельный файл. Таким образом, если возникнет необходимость в интеграции большего количества системных вызовов, мы можем просто добавить их в один класс и просто вызвать ассемблерный код из нашей программы.

Именно этим мы и займемся! Мы начнем с добавления нового файла в наш солюшн и назовем его Syscalls.cs. Наша структура солюшена теперь должна выглядеть примерно так.

> +SharpCall SLN (Solution)
> |
> +->Properties
> |
> +->References
> |
> +->Program.cs (Main Program)
> |
> +->Syscalls.cs (Class to Hold our Assembly and Syscall Logic)

Отлично, мы можем начать кодить, верно? Ну, не совсем - здесь мы забываем одну важную вещь. Помните, что, поскольку мы будем использовать неуправляемый код, нам также необходимо создать экземпляры функций Windows API, чтобы мы могли вызывать их из нашей программы на C#. А чтобы использовать неуправляемые функции, нам необходимо вызвать платформу (P/Invoke) их структуры и параметры, а также любые другие дополнительные поля флагов.

Опять же, мы можем сделать это в файле Program.cs, но он будет намного чище и организован, если мы будем выполнять всю работу с P/Invoke в отдельном классе. Итак, давайте добавим к нашему солюшену еще один файл и назовем его Native.cs, поскольку он будет содержать наши нативные функции Windows.

Наша структура теперь должна выглядеть примерно так:

> +SharpCall SLN (Solution)
> |
> +->Properties
> |
> +->References
> |
> +->Program.cs (Main Program)
> |
> +->Syscalls.cs (Class to Hold our Assembly and Syscall Logic)
> |
> +->Native.cs (Class to Hold our Native Win32 APIs and Structs)

Теперь, когда у нас есть организованное приложение и мы знаем, что к чему, мы наконец можем приступить к кодингу!

Написание нашего кода системного вызова

Поскольку это POC, я буду использовать системный вызов NtCreateFile для создания временного файла на нашем рабочем столе. Если мы сможем заставить это работать, это подтвердит надежность нашей логики кода. После этого мы сможем сосредоточиться на написании более сложных инструментов и расширении нашего класса системных вызовов дополнительными системными вызовами.

Также небольшое примечание - весь код, написанный ниже, будет работать только в системах x64, но не на x86.

Хорошо, для начала нам нужно получить ассемблерный код для нашего системного вызова NtCreateFile. Как объяснено и подробно описано в моем предыдущем посте, мы можем сделать это, используя WinDBG для дизассемблирования и проверки функции вызова NtCreateFile в ntdll.

Получив адрес памяти функции и проанализировав инструкции по адресу памяти, мы должны теперь увидеть следующий результат.

C:\Windows\System32\notepad.exe - WinDbg (1.0.2001.02001)

```
ModLoad: 00007ffd`75c00000 00007ffd`75c80000   C:\Windows\System32\bcryptPrimitives.dll
ModLoad: 00007ffd`77d20000 00007ffd`77dc9000   C:\Windows\System32\shcore.dll
ModLoad: 00007ffd`77c60000 00007ffd`77d03000   C:\Windows\System32\advapi32.dll
ModLoad: 00007ffd`76920000 00007ffd`769b7000   C:\Windows\System32\sechost.dll
ModLoad: 00007ffd`67100000 00007ffd`67385000   C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.18362
(f8c.f80): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffd`7889121c cc              int     3
0:000> x ntdll!NtCreateFile
00007ffd`7885cb50 ntdll!NtCreateFile (NtCreateFile)
0:000> u 00007ffd`7885cb50
ntdll!NtCreateFile:
00007ffd`7885cb50 4c8bd1          mov     r10,rcx
00007ffd`7885cb53 b855000000      mov     eax,55h
00007ffd`7885cb58 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffd`7885cb60 7503            jne     ntdll!NtCreateFile+0x15 (00007ffd`7885cb65)
00007ffd`7885cb62 0f05            syscall
00007ffd`7885cb64 c3              ret
00007ffd`7885cb65 cd2e            int     2Eh
00007ffd`7885cb67 c3              ret
```

System Call #

System Call CPU Instruction

Посмотрев на дизассемблерный код, мы видим, что наш идентификатор системного вызова — 0x55. А если мы посмотрим слева от инструкций ассемблера, мы увидим шестнадцатеричное представление наших инструкций системного вызова. Поскольку в C# нет встроенного ассемблера, мы собираемся использовать этот шестнадцатеричный код в качестве шелл-кода, который будет добавлен в простой байтовый массив.

Мы сделаем это, перейдя к нашему файлу Syscalls.cs и создадим новый статический массив байтов под названием bNtCreateFile, как показано.

```
using System;

namespace SharpCall
{
    class Syscalls
    {

        static byte[] bNtCreateFile =
        {
            0x4C, 0x8B, 0xD1,               // mov r10, rcx
            0xB8, 0x55, 0x00, 0x00, 0x00,   // mov eax, 0x55 (NtCreateFile Syscall)
            0x0F, 0x05,                     // syscall
            0xC3                            // ret
        };
    }
}
```

XSS.is

Замечательно, наша первый ассемблерный код системного вызова завершен! Но как мы собираемся создать код для выполнения этого? Что ж, если бы вы обратили внимание на мой предыдущий пост, вы бы узнали о чем-то, что называется делегатами.

Делегаты - это просто тип, представляющий ссылки на методы с определенным списком параметров и типом возвращаемого значения. Когда вы создаете экземпляр делегата, вы можете связать его экземпляр с любым методом, имеющим совместимую сигнатуру и тип возвращаемого значения. Затем мы можем вызвать наш делегированный метод через экземпляр делегата.

Это может показаться немного запутанным, но если вы помните, в моем последнем посте мы определили новый делегат под названием EnumWindowsProc, а позже определили реализацию делегатов через OutputWindow. Эта реализация для делегата просто сообщила C#, что мы хотим делать с данными, которые передаются в эту ссылку на функцию - будь то управляемый или неуправляемый код.

Мы можем сделать то же самое здесь, в нашем классе Syscall.cs, определив делегата нашей неуправляемой функции, которой в данном случае будет NtCreateFile. Как только этот делегат определен, мы можем продолжить и реализовать логику, которая будет обрабатывать преобразование нашего ассемблерного кода для системных вызовов в допустимую функцию.

Но не будем забегать вперед. Во-первых, нам нужно определить подпись для нашего делегата NtCreateFile. Для этого мы начнем с создания нового общедоступного типа структуры под названием Delegates в нашем классе Syscall.

Эта структура будет содержать все наши собственные функции (делегат), сигнатуру, чтобы они могли использоваться нашими системными вызовами.

Program.cs | Syscalls.cs* | Native.cs

SharpCall | SharpCall.Syscalls.Delegates

```
1   using System;
2
3   namespace SharpCall
4   {
5       class Syscalls
6       {
7
8           static byte[] bNtCreateFile =
9           {
10              0x4C, 0x8B, 0xD1,               // mov r10, rcx
11              0xB8, 0x55, 0x00, 0x00, 0x00,   // mov eax, 0x55 (NtCreateFile Syscall)
12              0x0F, 0x05,                     // syscall
13              0xC3                            // ret
14          };
15
16          public struct Delegates
17          {
18
19          }
20      }
21  }
```

XSS.is

Прежде чем мы определим нашего делегата, давайте взглянем на синтаксис C NtCreateFile.

> __kernel_entry NTSTATUS NtCreateFile(
> OUT PHANDLE FileHandle,
> IN ACCESS_MASK DesiredAccess,
> IN POBJECT_ATTRIBUTES ObjectAttributes,
> OUT PIO_STATUS_BLOCK IoStatusBlock,
> IN PLARGE_INTEGER AllocationSize,
> IN ULONG FileAttributes,
> IN ULONG ShareAccess,
> IN ULONG CreateDisposition,
> IN ULONG CreateOptions,
> IN PVOID EaBuffer,
> IN ULONG EaLength
>);
>
> Click to expand...

Изучив синтаксис, мы быстро замечаем несколько вещей, которых раньше не видели.

Прежде всего, мы замечаем, что функция NtCreateFile имеет возвращаемый тип NTSTATUS, который представляет собой структуру, содержащую 32-битное целое число без знака для каждого идентификатора сообщения. Мы также видим, что

некоторые из параметров функции принимают набор различных флагов и структур, таких как флаги ACCESS_MASK, структура OBJECT__ATTRIBUTES и структура IO_STATUS_BLOCK.

Если мы посмотрим на другие параметры функции, такие как FileAttributes и CreateOptions, мы увидим, что они также принимают определенные флаги.

`FileAttributes`

The file attributes. Explicitly specified attributes are applied only when the file is created, superseded, or, in some cases, overwritten. By default, this value is a **FILE_ATTRIBUTE_NORMAL**, which can be overridden by an ORed combination of one or more **FILE_ATTRIBUTE_***xxxx* flags, which are defined in Wdm.h and NtDdk.h. For a list of flags that can be used with **NtCreateFile**, see **CreateFile**.

`ShareAccess`

XSS.is

The type of share access that the caller would like to use in the file, as zero, or as one or a combination of the following values.

Value	Meaning
FILE_SHARE_READ	The file can be opened for read access by other threads' calls to **NtCreateFile**.
FILE_SHARE_WRITE	The file can be opened for write access by other threads' calls to **NtCreateFile**.
FILE_SHARE_DELETE	The file can be opened for delete access by other threads' calls to **NtCreateFile**.

`CreateOptions`

The options to be applied when creating or opening the file, as a compatible combination of the following flags.

Value	Meaning
FILE_DIRECTORY_FILE	The file being created or opened is a directory file. With this flag, the *CreateDisposition* parameter must be set to **FILE_CREATE**, **FILE_OPEN**, or **FILE_OPEN_IF**. With this flag, other compatible *CreateOptions* flags include only the following: **FILE_SYNCHRONOUS_IO_ALERT**, **FILE_SYNCHRONOUS_IO _NONALERT**, **FILE_WRITE_THROUGH**, **FILE_OPEN_FOR_BACKUP_INTENT**, and **FILE_OPEN_BY_FILE_ID**.
FILE_NON_DIRECTORY_FILE	The file being opened must not be a directory file or this call fails. The file object being opened can represent a data file, a logical, virtual, or physical device, or a volume.
FILE_WRITE_THROUGH	Applications that write data to the file must actually transfer the data into the file before any requested write operation is considered complete. This flag is automatically set if the *CreateOptions* flag **FILE_NO_INTERMEDIATE _BUFFERING** is set.

XSS.is

Итак, здесь кроется основная проблема использования неуправляемого кода в C# - это тот факт, что нам нужно вручную создать эти перечислители и структуры флагов, которые будут содержать те же коды значений, что и Windows. В противном случае,

если параметры, которые мы передаем в наш системный вызов, содержат неожиданные значения, это приведет к тому, что системный вызов либо прервется, либо вернет ошибки.

К счастью для нас, на помощь приходит вики-страница P/Invoke. Здесь мы можем узнать, как реализовать наши собственные функции, структуры и флаги.

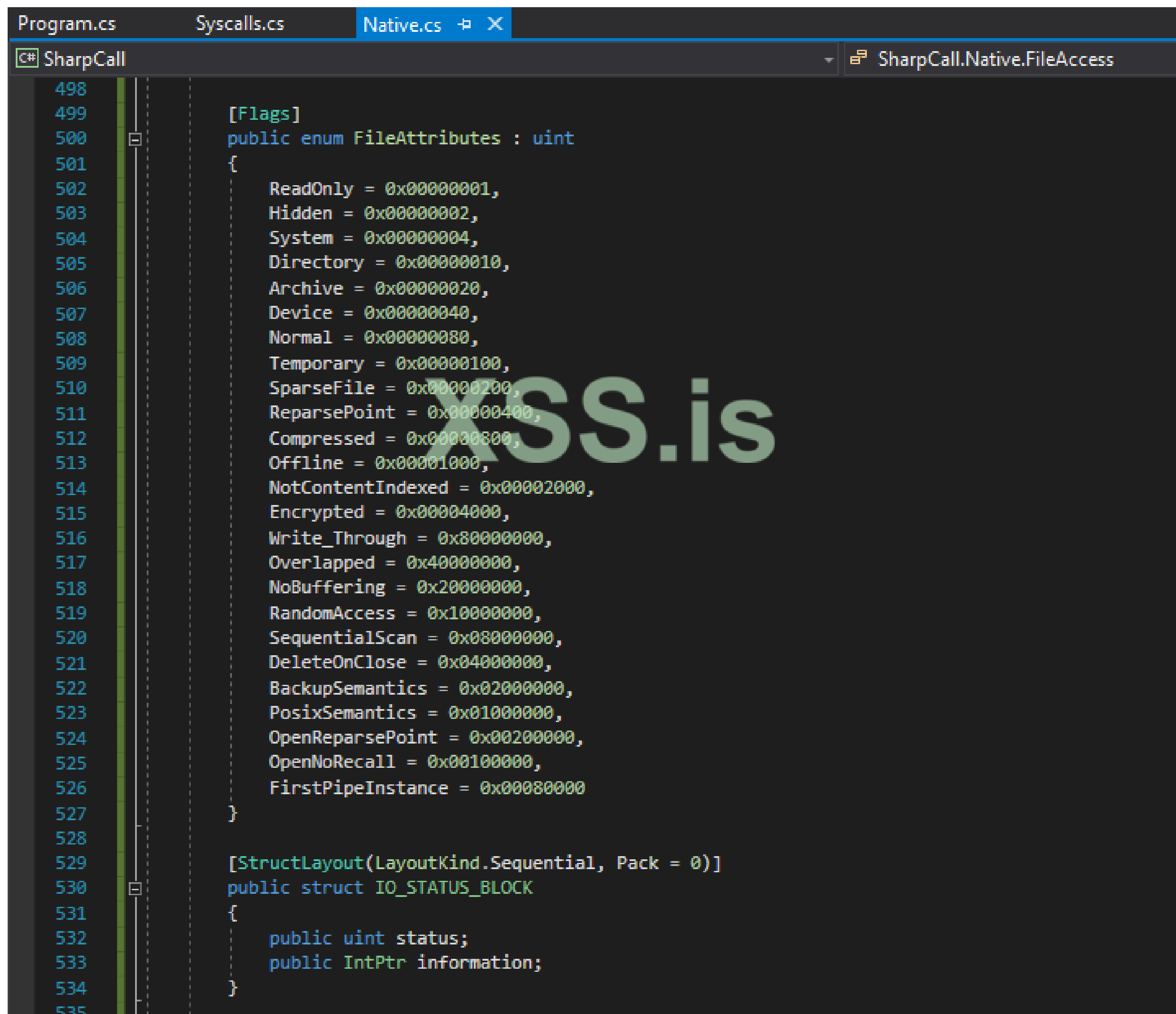
Вы также можете использовать веб-сайт Microsoft Reference Source и искать нужные структуры и флаги доступа. Они будут намного ближе к исходным ссылкам на Windows, чем то, что могло быть у P/Invoke.

Следующие ссылки должны помочь нам реализовать необходимые структуры и флаги, необходимые для выполнения NtCreateFile с правильными значениями параметров:

> ### pinvoke.net: NtStatus (Enums)
>
> Windows NT status codes
>
> www.pinvoke.net
>
> ### pinvoke.net: ACCESS_MASK (Enums)
>
> The [ACCESS_MASK] data type is a double word value that defines standard, specific, and generic rights. These rights are used in access control entries (ACEs) and are the primary means of specifying the requested or granted access to an object.
>
> www.pinvoke.net
>
> ### pinvoke.net: ntcreatefile (ntdll)
>
> Creates a new file or directory, or opens an existing file, device, directory, or volume.
>
> www.pinvoke.net
>
> ### pinvoke.net: CreateFile (kernel32)
>
> The CreateFile API
>
> www.pinvoke.net
>
> Click to expand...

Поскольку все эти значения, структуры и флаги являются "нативными" для Windows, давайте продолжим и добавим их в файл Native.cs под классом Native.

После того, как все будет реализовано и очищено, часть вашего файла Native.cs должна выглядеть примерно так.



В качестве примечания - это лишь небольшая часть реализованных нативных структур и флагов. Если вы хотите увидеть полную реализацию, взгляните на файл Native.cs из проекта SharpCall на моем GitHub.

Также обратите внимание на то, как мы вызываем ключевое слово public перед каждым перечислителем структур и флагов. Это сделано для того, чтобы мы могли получить доступ к объектам из других файлов в нашей программе.

Замечательно, теперь, когда они реализованы, мы можем продолжить преобразование типов данных C++ NtCreateFile в типы данных C#. После преобразования ваш синтаксис C# должен выглядеть так:

> NTSTATUS NtCreateFile(
> out Microsoft.Win32.SafeHandles.SafeFileHandle FileHadle,
> FileAccess DesiredAcces,
> ref OBJECT_ATTRIBUTES ObjectAttributes,
> ref IO_STATUS_BLOCK IoStatusBlock,
> ref long AllocationSize,
> FileAttributes FileAttributes,
> FileShare ShareAccess,
> CreationDisposition CreateDisposition,
> CreateOption CreateOptions,
> IntPtr EaBuffer,
> uint EaLength
>);
>
> Click to expand...

Теперь, прежде чем реализовывать эту структуру в качестве делегата, давайте кратко рассмотрим некоторые преобразованные типы данных.

Как было сказано ранее, обычно любые указатели или дескрипторы в C++ могут быть преобразованы в IntPtr в C#, но в этом случае вы заметите, что я преобразовал PHANDLE (указатель на дескриптор) в тип данных SafeFileHandle. Причина, по которой мы делаем это, заключается в том, что SafeFileHandle представляет собой класс-оболочку для дескриптора файла, который понимает C#.

И поскольку мы имеем дело с созданием файлов и будем передавать эти данные через делегаты из управляемого в неуправляемый код (и наоборот), нам нужно убедиться, что C# может обрабатывать и понимать тип данных, который он маршалирует, иначе мы можем столкнуться с ошибками.

Остальное должно быть самоочевидным, поскольку FileAttributes, FileShare и эти типы данных являются просто представлением переменных и значений внутри структур и перечислителей флагов, которые мы добавили к классу Native. Это просто сообщает C#, что всякий раз, когда данные передаются в эти параметры - будь то значение или дескриптор - тогда на них нужно ссылаться в этом конкретном перечислителе структур/флагов.

Вы могли заметить еще несколько вещей: я добавил ключевые слова ref и out к некоторым параметрам. Просто эти ключевые слова указывают на то, что аргументы могут передаваться по ссылке, а не по значению.

Разница между ref и out заключается в том, что для ключевого слова ref параметр или аргумент должен быть инициализирован перед его передачей, в отличие от out, где нам это не нужно. Другое отличие состоит в том, что для ref данные могут передаваться в двух направлениях, и любые изменения, внесенные в этот аргумент в методе, будут отражены в этой переменной, когда управление вернется к вызывающему методу. Для out данные передаются только однонаправленно, и любое значение, возвращаемое нам вызывающим методом, устанавливается в ссылочную переменную.

Итак, в случае NtCreateFile мы устанавливаем ключевое слово out для FileHandle, поскольку это будет указатель на переменную, которая получает дескриптор файла в случае успешного вызова. Это просто означает, что данные передаются нам только "обратно".

Имеет смысл? Хорошо!

Теперь, когда у нас есть это, мы можем наконец добавить наш синтаксис C# для NtCreateFile в нашу недавно добавленную структуру Delegates в нашем классе Syscalls.

После этого наш класс Syscalls должен выглядеть примерно так.

```
Native.cs    Program.cs    Syscalls.cs
SharpCall                                    SharpCall.Syscalls.Delegates.NtCreateFile
1   using System;
2   using System.Runtime.InteropServices;
3
4   using static SharpCall.Native;
5
6   namespace SharpCall
7   {
8       class Syscalls
9       {
10
11          static byte[] bNtCreateFile =
12          {
13              0x4C, 0x8B, 0xD1,               // mov r10, rcx
14              0xB8, 0x55, 0x00, 0x00, 0x00,   // mov eax, 0x55 (NtCreateFile Syscall)
15              0x0F, 0x05,                     // syscall
16              0xC3                            // ret
17          };
18
19          public struct Delegates
20          {
21              [UnmanagedFunctionPointer(CallingConvention.StdCall)]
22              public delegate NTSTATUS NtCreateFile(
23                  out Microsoft.Win32.SafeHandles.SafeFileHandle FileHandle,
24                  FileAccess DesiredAcces,
25                  ref OBJECT_ATTRIBUTES ObjectAttributes,
26                  ref IO_STATUS_BLOCK IoStatusBlock,
27                  ref long AllocationSize,
28                  FileAttributes FileAttributes,
29                  FileShare ShareAccess,
30                  CreationDisposition CreateDisposition,
31                  CreateOption CreateOptions,
32                  IntPtr EaBuffer,
33                  uint EaLength
34                  );
35          }
36      }
37  }
```

XSS.is

ПРИМЕЧАНИЕ. Вы могли заметить, что я добавил SharpCall.Native вверху файла. Это просто указывает C# использовать статический класс Native. Как объяснялось ранее, мы делаем это, чтобы мы могли напрямую использовать наши собственные функции, импорт структур и флагов.

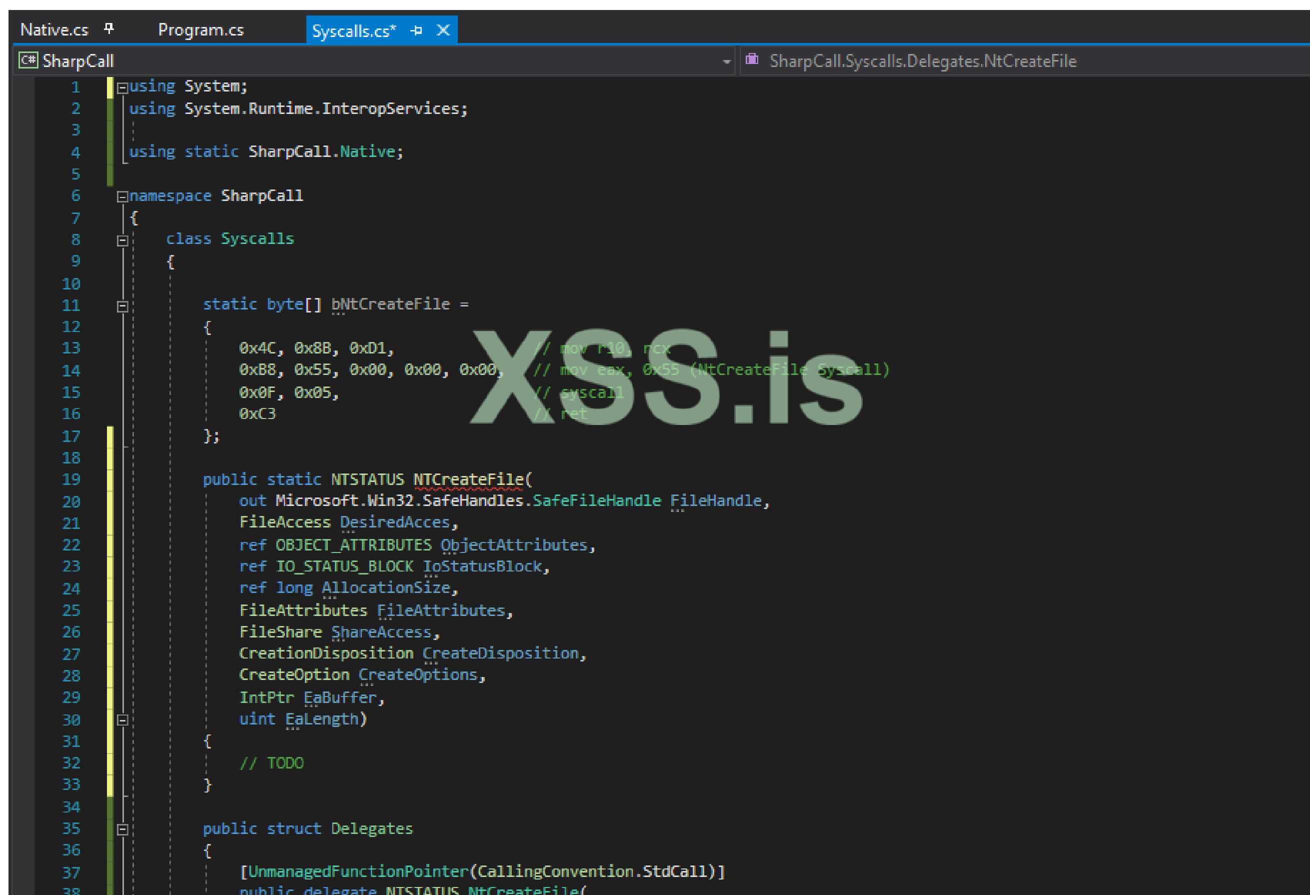
Хорошо, прежде чем мы продолжим, обратите внимание, что в структуре делегатов, прежде чем мы настроим наш делегат NtCreateFile, я вызываю атрибут UnmanagedFunctionPointer. Этот атрибут управляет поведением маршалинга подписи делегата как указателя на неуправляемую функцию, когда он передается в или из неуправляемого кода.

Это важная часть информации, которую нам необходимо включить, поскольку мы будем использовать небезопасный код для маршалинга нашего неуправляемого указателя из ассемблерного кода системных вызовов этим делегатам функций - как объяснялось в моем предыдущем посте.

Отлично, мы добиваемся прогресса! Теперь, когда у нас есть наши структуры, перечислители флагов и наш делегат функции, мы можем продолжить и начать реализацию делегата для обработки любых переданных в него параметров. Затем эти параметры будут обрабатываться нашим ассемблерным кодом системных вызовов.

Давайте продолжим и создадим (или, другими словами, создадим экземпляр) нашего делегата функции NtCreateFile. Мы можем сделать это сразу после ассемблерного кода системного вызова.

После этого ваш файл Syscalls.cs должен выглядеть примерно так, как показано ниже.

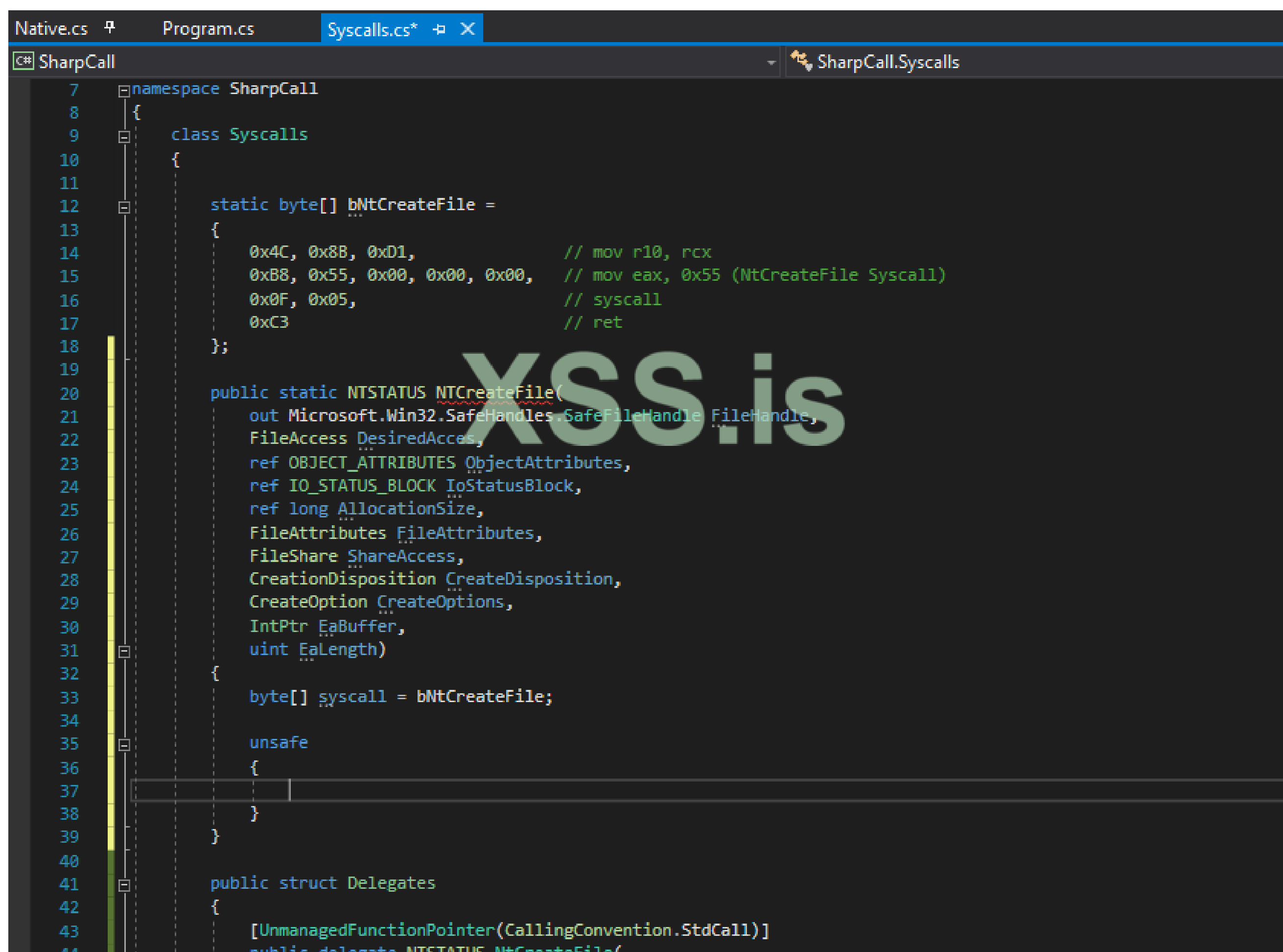


В скобки с комментарием TODO (сразу после созданного экземпляра делегата) мы добавим код для обработки данных, передаваемых в управляемый и неуправляемый код и из него.

Если вы помните из моего последнего сообщения, я объяснил, как Marshal.GetDelegateForFunctionPointer позволяет нам преобразовывать указатель неуправляемой функции в делегат указанного типа. Используя это с небезопасным контекстом, это позволит нам создать указатель на место в памяти, где находится наш шелл-код (который будет нашем ассемблерным кодом системных вызовов), и позволит нам выполнить код из управляемого кода через делегат.

Здесь мы будем делать то же самое. Итак, для начала, давайте убедимся, что мы создаем новый массив байтов с именем syscall и устанавливаем для него то же значение, что и в нашей сборке bNtCreateFile. После этого укажите небезопасный контекст и добавьте несколько скобок, в которые будет помещен наш небезопасный код.

После завершения ваш недавно обновленный файл Syscalls.cs должен выглядеть следующим образом.



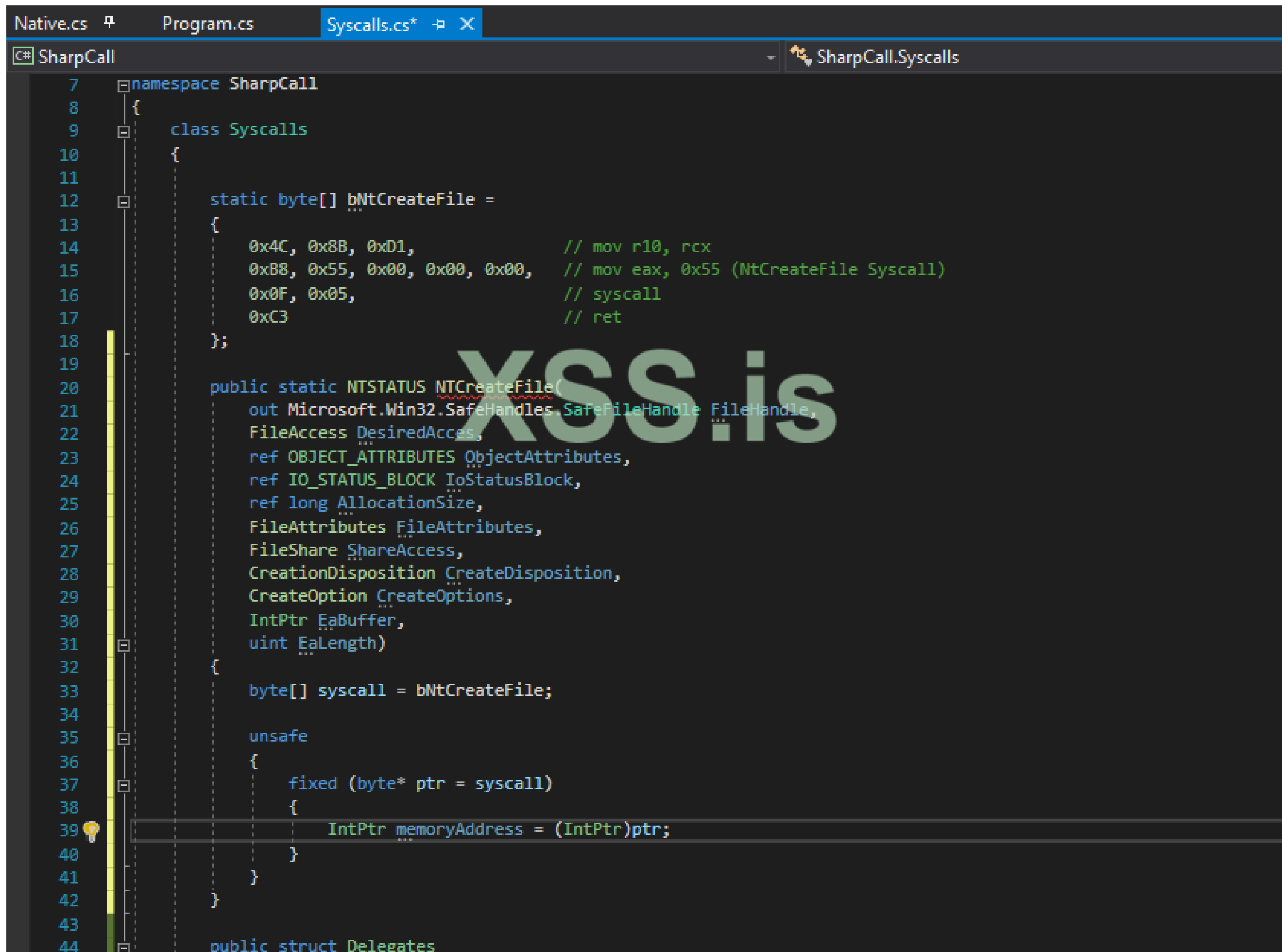
Теперь, как я объяснил в своем предыдущем посте, в этом небезопасном контексте мы инициализируем новый байтовый указатель с именем ptr и установим для него значение syscall, в котором находится наш байтовый массив.

Как вы увидите ниже и как объяснялось ранее, мы используем фиксированный оператор для этого указателя, чтобы мы могли предотвратить перемещение сборщика мусора нашего байтового массива системных вызовов в памяти.

После этого мы просто преобразуем типа указатель байтового массива в IntPtr с именем memoryAddress. Это позволит нам получить адрес памяти, в котором находится массив байтов системных вызовов в нашем приложении во время

выполнения.

После выполнения вышеуказанного наш обновленный файл Syscall.cs должен выглядеть так, как показано ниже.



Теперь, что касается этой части, я предлагаю вам обратить пристальное внимание, поскольку именно здесь происходит волшебство!

Поскольку теперь у нас есть (или будет) адрес памяти, в котором находится наш ассемблерный код системных вызовов во время выполнения приложения, нам нужно что-то сделать, чтобы убедиться, что она будет правильно выполняться в выделенной для нее области памяти.

Если вы знакомы с тем, как работает шелл-код во время разработки эксплойта - всякий раз, когда мы хотим написать, прочитать или даже выполнить шелл-код в нашем целевом процессе или целевых страницах памяти, то нам необходимо убедиться, что эти области памяти имеют надлежащие права доступа. Если вы не знакомы с этим, прочитайте, как модель безопасности Windows позволяет вам контролировать безопасность процессов и права доступа.

Например, давайте посмотрим, какие средства защиты памяти NtCreateFile имеются в блокноте при выполнении.

> 0:000> x ntdll!NtCreateFile
> 00007ffb`f6b9cb50 ntdll!NtCreateFile (NtCreateFile)
> 0:000> !address 00007ffb`f6b9cb50
>
> Usage: Image
> Base Address: 00007ffb`f6b01000
> End Address: 00007ffb`f6c18000
> Region Size: 00000000`00117000 (1.090 MB)
> State: 00001000 MEM_COMMIT
> Protect: 00000020 PAGE_EXECUTE_READ
> Type: 01000000 MEM_IMAGE
> Allocation Base: 00007ffb`f6b00000
> Allocation Protect: 00000080 PAGE_EXECUTE_WRITECOPY
> Image Path: ntdll.dll
> Module Name: ntdll
> Loaded Image Name: C:\Windows\SYSTEM32\ntdll.dll
> Mapped Image Name:
> More info: lmv m ntdll More info: !lmi ntdll More info: ln 0x7ffbf6b9cb50 More info: !dh 0x7ffbf6b00000 Content source: 1 (target), length: 7b4b
> Click to expand...

Как показано выше, блокнот имеет разрешения на чтение и выполнение для NtCreatreFile в виртуальной памяти процессов. Причина этого в том, что блокнот должен быть уверен, что он может выполнять системный вызов, а также должен иметь возможность читать возвращаемые значения.

В моем предыдущем посте я объяснил, как виртуальное адресное пространство каждого приложения является частным и как одно приложение не может изменять данные, принадлежащие другому приложению, если только процесс не сделает доступной часть своего частного адресного пространства.

Теперь, когда мы используем небезопасный контекст в C# и проходим границы между управляемым и неуправляемым кодом, нам нужно управлять доступом к памяти в пространстве виртуальной памяти наших программ, поскольку среда CLR не сделает этого за нас! И нам нужно сделать это, чтобы мы могли записать наши параметры в наш системный вызов, выполнить системный вызов, а также прочитать возвращенные данные для нашего делегата!

Но как это сделать? Итак, позвольте мне представить вам нашего нового маленького друга и прекрасную функцию под названием VirtualProtect.

Что VritualProtect позволяет нам сделать, так это изменить защиту в области зафиксированных страниц в виртуальном адресном пространстве вызывающего процесса. Это означает, что, используя эту встроенную функцию против нашего адреса памяти системных вызовов (который мы только что получили), мы можем убедиться, что виртуальная память процесса настроена на чтение-запись-выполнение!

Итак, давайте реализуем эту встроенную функцию внутри Native.cs. Таким образом, мы можем использовать его в Syscalls.cs для изменения защиты памяти в нашем ассемблерном коде.

Как всегда, давайте взглянем на структуру C для этой функции.

```
BOOL VirtualProtect(
LPVOID lpAddress,
SIZE_T dwSize,
DWORD flNewProtect,
PDWORD lpflOldProtect
);
```

Вроде достаточно просто. Нам просто нужно не забыть добавить флаги flNewProtect вместе с функцией.

Давайте добавим это. После этого наши реализованные флаги защиты памяти внутри класса Native должны выглядеть так.

```
Program.cs    Syscalls.cs    Native.cs
SharpCall                                  SharpCall.Native.UNICODE_STRING
519             RandomAccess = 0x10000000,
520             SequentialScan = 0x08000000,
521             DeleteOnClose = 0x04000000,
522             BackupSemantics = 0x02000000,
523             PosixSemantics = 0x01000000,
524             OpenReparsePoint = 0x00200000,
525             OpenNoRecall = 0x00100000,
526             FirstPipeInstance = 0x00080000
527         }
528
529         public enum AllocationProtect : uint
530         {
531             PAGE_EXECUTE = 0x00000010,
532             PAGE_EXECUTE_READ = 0x00000020,
533             PAGE_EXECUTE_READWRITE = 0x00000040,
534             PAGE_EXECUTE_WRITECOPY = 0x00000080,
535             PAGE_NOACCESS = 0x00000001,
536             PAGE_READONLY = 0x00000002,
537             PAGE_READWRITE = 0x00000004,
538             PAGE_WRITECOPY = 0x00000008,
539             PAGE_GUARD = 0x00000100,
540             PAGE_NOCACHE = 0x00000200,
541             PAGE_WRITECOMBINE = 0x00000400
542         }
543
544         [StructLayout(LayoutKind.Sequential, Pack = 0)]
545         public struct IO_STATUS_BLOCK
546         {
547             public uint status;
548             public IntPtr information;
549         }
550
551         [StructLayout(LayoutKind.Sequential, Pack = 0)]
552         public struct OBJECT_ATTRIBUTES
553         {
554             public Int32 Length;
555             public IntPtr RootDirectory;
556             public IntPtr ObjectName;
```

XSS.is

И функция VirtualProtect будет выглядеть примерно так.

```
Program.cs    Syscalls.cs    Native.cs
SharpCall                                  SharpCall.Native
534         }
535
536         [StructLayout(LayoutKind.Sequential, Pack = 0)]
537         public struct OBJECT_ATTRIBUTES
538         {
539             public Int32 Length;
540             public IntPtr RootDirectory;
541             public IntPtr ObjectName;
542             public uint Attributes;
543             public IntPtr SecurityDescriptor;
544             public IntPtr SecurityQualityOfService;
545
546         }
547
548         [StructLayout(LayoutKind.Sequential, Pack = 0)]
549         public struct UNICODE_STRING
550         {
551             public ushort Length;
552             public ushort MaximumLength;
553             public IntPtr Buffer;
554
555         }
556
557         [DllImport("kernel32.dll")]
558         static extern bool VirtualProtectEx(IntPtr hProcess, IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
559     }
560 }
561
```

XSS.is

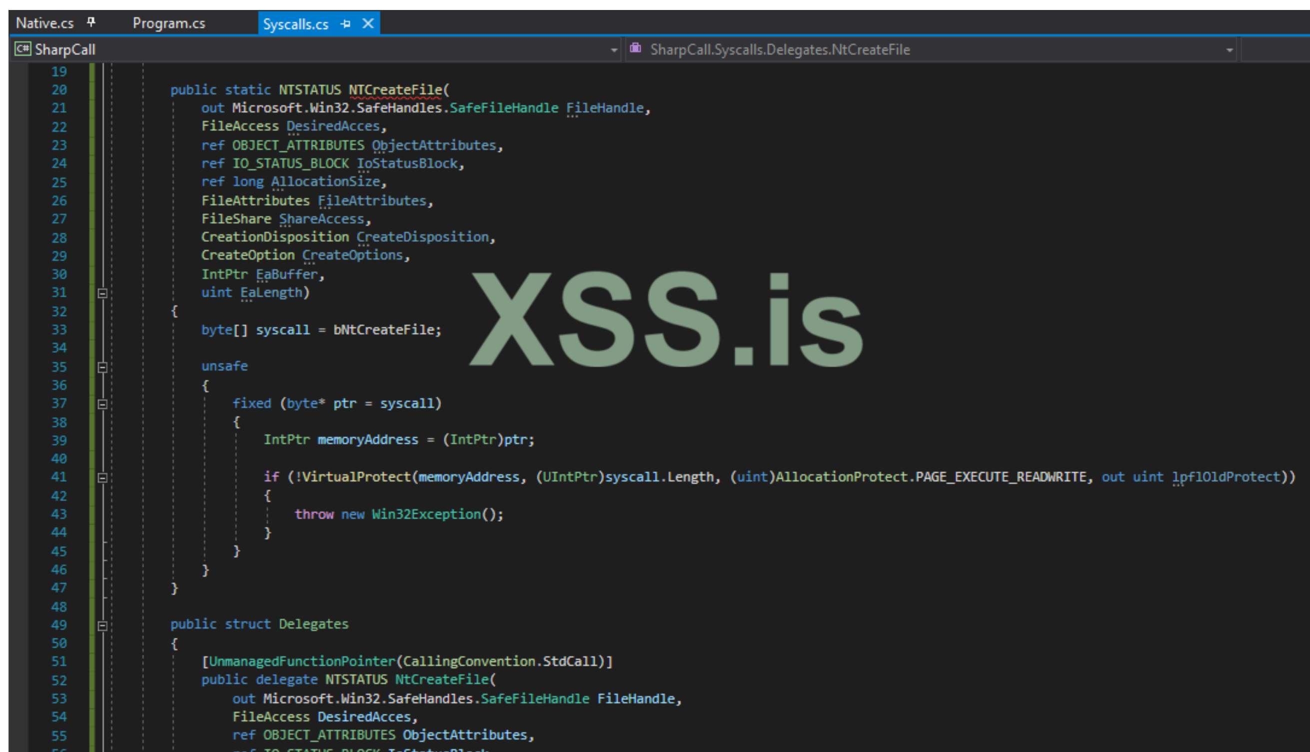
Прекрасно! Мы уже добились огромного прогресса и приближаемся к концу! Ну... вроде того. Есть еще кое-что, что нужно сделать.

Теперь, когда у нас реализована наша функция VirtualProtect, давайте вернемся к нашему файлу Syscall.cs и выполним функцию VirtualProtect для нашего указателя memoryAddress, чтобы дать ему права на чтение-запись-выполнение.

В то же время давайте поместим эту встроенную функцию в оператор IF. Таким образом, в случае сбоя функции мы можем вызвать исключение Win32Exception, чтобы показать нам код ошибки и остановить выполнение.

Кроме того, не забудьте добавить using System.ComponentModel; директива в верхней части вашего кода. Таким образом, вы сможете использовать класс Win32Exception.

После этого наш код должен выглядеть следующим образом:



Хорошо, поэтому, если выполнение VirtualProtect прошло успешно, тогда адрес виртуальной памяти нашей неуправляемой сборки системных вызовов (на которую указывает переменная memoryAddress) теперь должен иметь разрешения на чтение-запись-выполнение.

Это означает, что теперь у нас есть указатель на неуправляемую функцию. Итак, как объяснялось ранее и в моем предыдущем посте - что нам нужно сделать сейчас, это использовать Marshal.GetDelegateForFunctionPointer для преобразования нашего

указателя неуправляемой функции в делегат указанного типа. В этом случае мы будем преобразовывать указатель на функцию в делегат NtCreateFile.

Я знаю, что некоторые из вас могут быть немного сбиты с толку или недоумевать, почему мы это делаем. Вам должно было стать очевидно, что мы пытаемся сделать, когда я объяснил защиту памяти. Но в любом случае позвольте мне объяснить это прежде чем двигаться дальше.

Причина, по которой мы преобразуем наш указатель неуправляемой функции в делегат NtCreateFile, заключается в том, что функция будет вести себя как функция обратного вызова при выполнении нашего ассемблерного кода системных вызовов. Вернитесь к строке 20 нашего файла Syscalls.cs.

Что мы там делаем? Если вы ответили "передали параметры в функцию", то вы правы!

Как только этот делегат примет наши параметры для создания файла, он продолжит работу и обновит место в памяти нашего системного вызова для чтения-записи-выполнения. Затем он возьмет этот указатель на системный вызов и преобразует его в наш делегат NtCreateFile, который преобразует наш системный вызов в фактическое представление функции.

Как только это будет сделано, мы вызовем оператор return для нашего инициализированного делегата вместе с переданными параметрами. По сути, именно в этот момент мы помещаем параметры в стек, выполняем системный вызов и возвращаем результаты обратно вызывающей стороне - которые должны поступать из Program.cs!

Теперь имеет смысл? Отлично! Считайте себя выпускником академии системных вызовов!

Хорошо, со всем этим объясненным, давайте продолжим и реализуем наше преобразование Marshal.GetDelegateForFunctionPointer, сначала создав экземпляр нашего делегата NtCreateFile и вызвав его AssemblydFunction. После этого давайте выполним преобразование нашего неуправляемого указателя в нашего делегата.

После этого мы можем написать простой оператор return, чтобы вернуть все параметры из нашего системного вызова через созданный экземпляр делегата AssemblydFunction.

Наш завершенный код Syscall.cs теперь должен выглядеть следующим образом.

```
Native.cs    Program.cs    Syscalls.cs
SharpCall    SharpCall.Syscalls.Delegates.NtCreateFile
25                  ref long AllocationSize,
26                  FileAttributes FileAttributes,
27                  FileShare ShareAccess,
28                  CreationDisposition CreateDisposition,
29                  CreateOption CreateOptions,
30                  IntPtr EaBuffer,
31                  uint EaLength)
32              {
33                  byte[] syscall = bNtCreateFile;
34
35                  unsafe
36                  {
37                      fixed (byte* ptr = syscall)
38                      {
39                          IntPtr memoryAddress = (IntPtr)ptr;
40
41                          if (!VirtualProtect(memoryAddress, (UIntPtr)syscall.Length, (uint)AllocationProtect.PAGE_EXECUTE_READWRITE, out uint lpflOldProtect))
42                          {
43                              throw new Win32Exception();
44                          }
45
46                          Delegates.NtCreateFile assembledFunction = (Delegates.NtCreateFile)Marshal.GetDelegateForFunctionPointer(memoryAddress, typeof(Delegates.NtCreateFile));
47
48                          return (NTSTATUS)assembledFunction(out FileHandle,
49                              DesiredAcces,
50                              ref ObjectAttributes,
51                              ref IoStatusBlock,
52                              ref AllocationSize,
53                              FileAttributes,
54                              ShareAccess,
55                              CreateDisposition,
56                              CreateOptions,
57                              EaBuffer,
58                              EaLength);
59                      }
60                  }
61              }
```

XSS.is

И вот она, окончательная версия того, как будет выполняться наш системный вызов после вызова функции!

Выполнение нашего системного вызова

Итак, мы реализовали нашу логику системного вызова, теперь все, что осталось сделать, это фактически написать код в нашей программе для использования функции NtCreateFile, которая первоначально будет выполнять наш системный вызов.

Для начала давайте удостоверимся, что мы импортируем наши статические классы, чтобы мы могли использовать все наши собственные функции и наш системный вызов, например.

```
Native.cs    Program.cs    Syscalls.cs
SharpCall    SharpCall.Program
1   using System;
2   using System.Runtime.InteropServices;
3
4   using static SharpCall.Native;
5   using static SharpCall.Syscalls;
6
7   namespace SharpCall
8   {
9       class Program
10      {
11          static void Main(string[] args)
12          {
13              // TODO
14          }
15      }
16  }
17
```

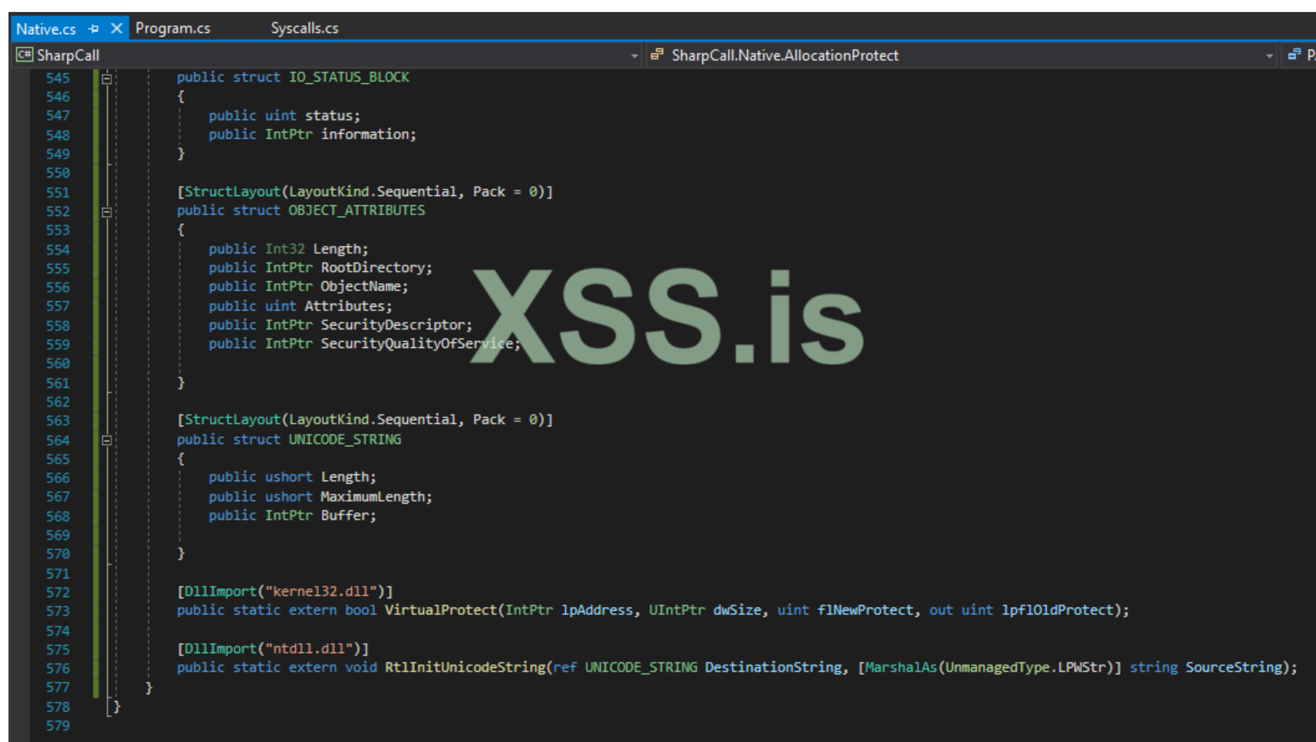
XSS.is

Как только это будет сделано, мы можем начать инициализацию структур и переменных, необходимых для NtCreateFile, таких как дескриптор файла и атрибуты объекта.

Но прежде чем мы это сделаем, позвольте мне заявить об одном. OBJECT_ATTRIBUTES, в частности член ObjectName, требует указателя на UNICODE_STRING, который содержит имя объекта, дескриптор которого должен быть открыт. В частности, это имя файла, который мы хотим создать.

Теперь, для неуправляемого кода, чтобы инициализировать эту структуру, нам нужно вызвать функцию RtlUnicodeStringInit.

Итак, давайте обязательно добавим эту функцию в наш файл Native.cs, чтобы мы могли использовать эту функцию.




```
public struct IO_STATUS_BLOCK
{
    public uint status;
    public IntPtr information;
}

[StructLayout(LayoutKind.Sequential, Pack = 0)]
public struct OBJECT_ATTRIBUTES
{
    public Int32 Length;
    public IntPtr RootDirectory;
    public IntPtr ObjectName;
    public uint Attributes;
    public IntPtr SecurityDescriptor;
    public IntPtr SecurityQualityOfService;

}

[StructLayout(LayoutKind.Sequential, Pack = 0)]
public struct UNICODE_STRING
{
    public ushort Length;
    public ushort MaximumLength;
    public IntPtr Buffer;

}

[DllImport("kernel32.dll")]
public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);

[DllImport("ntdll.dll")]
public static extern void RtlInitUnicodeString(ref UNICODE_STRING DestinationString, [MarshalAs(UnmanagedType.LPWStr)] string SourceString);
    }
}
```

XSS.is

Как только у нас это будет, мы можем продолжить и инициализировать наши первые несколько структур. Мы создадим дескриптор файла, а также нашу строковую структуру в Юникоде.

Мы выберем сохранение нашего тестового файла на рабочий стол, поэтому зададим путь к файлу C: \Users\User\Desktop.test.txt, как показано ниже.

```
Native.cs    Program.cs    Syscalls.cs
SharpCall                                   SharpCall.Program

using System;
using System.Runtime.InteropServices;

using static SharpCall.Native;
using static SharpCall.Syscalls;

namespace SharpCall
{
    class Program
    {
        static void Main(string[] args)
        {
            Microsoft.Win32.SafeHandles.SafeFileHandle fileHandle;
            UNICODE_STRING filename = new UNICODE_STRING();
            RtlInitUnicodeString(ref filename, @"\??\C:\Users\User\Desktop\test.txt");
            IntPtr objectName = Marshal.AllocHGlobal(Marshal.SizeOf(filename));
            Marshal.StructureToPtr(filename, objectName, true);
        }
    }
}
```

XSS.is

После этого мы можем инициализировать нашу структуру OBJECT_ATTRIBUTES.

```
Native.cs    Program.cs    Syscalls.cs
SharpCall                                   SharpCall.Program

using System;
using System.Runtime.InteropServices;

using static SharpCall.Native;
using static SharpCall.Syscalls;

namespace SharpCall
{
    class Program
    {
        static void Main(string[] args)
        {
            Microsoft.Win32.SafeHandles.SafeFileHandle fileHandle;
            UNICODE_STRING filename = new UNICODE_STRING();
            RtlInitUnicodeString(ref filename, @"\??\C:\Users\User\Desktop\test.txt");
            IntPtr objectName = Marshal.AllocHGlobal(Marshal.SizeOf(filename));
            Marshal.StructureToPtr(filename, objectName, true);

            OBJECT_ATTRIBUTES FileObjectAttributes = new OBJECT_ATTRIBUTES
            {
                Length = (int)Marshal.SizeOf(typeof(OBJECT_ATTRIBUTES)),
                RootDirectory = IntPtr.Zero,
                ObjectName = objectName,
                Attributes = 0x00000040, // OBJ_CASE_INSENSITIVE
                SecurityDescriptor = IntPtr.Zero,
                SecurityQualityOfService = IntPtr.Zero
            };
        }
    }
}
```

XSS.is

Наконец, все, что осталось сделать, это инициализировать структуру IO_STATUS_BLOCK и вызвать наш делегат NtCreateFile вместе с его параметрами для выполнения системного вызова!

После всего этого ваш окончательный файл Program.cs должен выглядеть следующим образом.

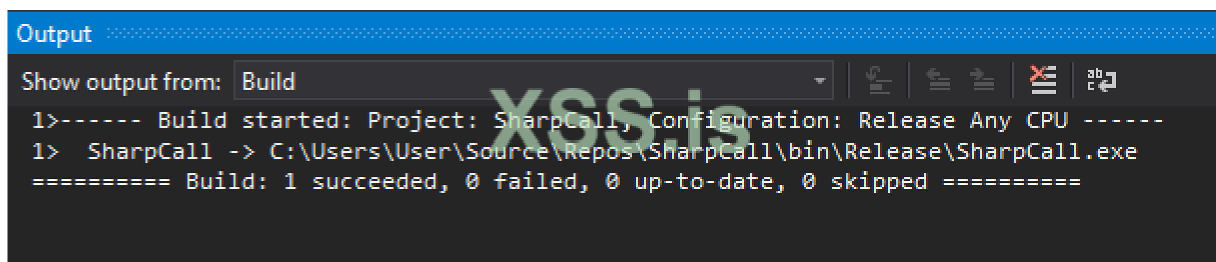
```
Native.cs    Program.cs    Syscalls.cs
SharpCall                                    SharpCall.Program
 9      class Program
10      {
11          static void Main(string[] args)
12          {
13              Microsoft.Win32.SafeHandles.SafeFileHandle fileHandle;
14              UNICODE_STRING filename = new UNICODE_STRING();
15              RtlInitUnicodeString(ref filename, @"\??\C:\Users\User\Desktop\test.txt");
16              IntPtr objectName = Marshal.AllocHGlobal(Marshal.SizeOf(filename));
17              Marshal.StructureToPtr(filename, objectName, true);
18
19              OBJECT_ATTRIBUTES FileObjectAttributes = new OBJECT_ATTRIBUTES
20              {
21                  Length = (int)Marshal.SizeOf(typeof(OBJECT_ATTRIBUTES)),
22                  RootDirectory = IntPtr.Zero,
23                  ObjectName = objectName,
24                  Attributes = 0x00000040, // OBJ_CASE_INSENSITIVE
25                  SecurityDescriptor = IntPtr.Zero,
26                  SecurityQualityOfService = IntPtr.Zero
27              };
28
29              IO_STATUS_BLOCK IoStatusBlock = new IO_STATUS_BLOCK();
30              long allocationSize = 0;
31
32              var status = NTCreateFile(
33                  out fileHandle,
34                  FileAccess.FILE_GENERIC_WRITE,
35                  ref FileObjectAttributes,
36                  ref IoStatusBlock,
37                  ref allocationSize,
38                  FileAttributes.Normal,
39                  FileShare.Write,
40                  CreationDisposition.FILE_OVERWRITE_IF,
41                  CreateOption.FILE_SYNCHRONOUS_IO_NONALERT,
42                  IntPtr.Zero,
43                  0);
44          }
45      }
46  }
```

XSS.is

Отлично, мы наконец-то завершили наш код! Теперь самое важное - компиляция кода!

В Visual Studio убедитесь, что мы изменили конфигурацию решения на «Release». Оттуда на панели инструментов выше нажмите Build -> Build Solution.

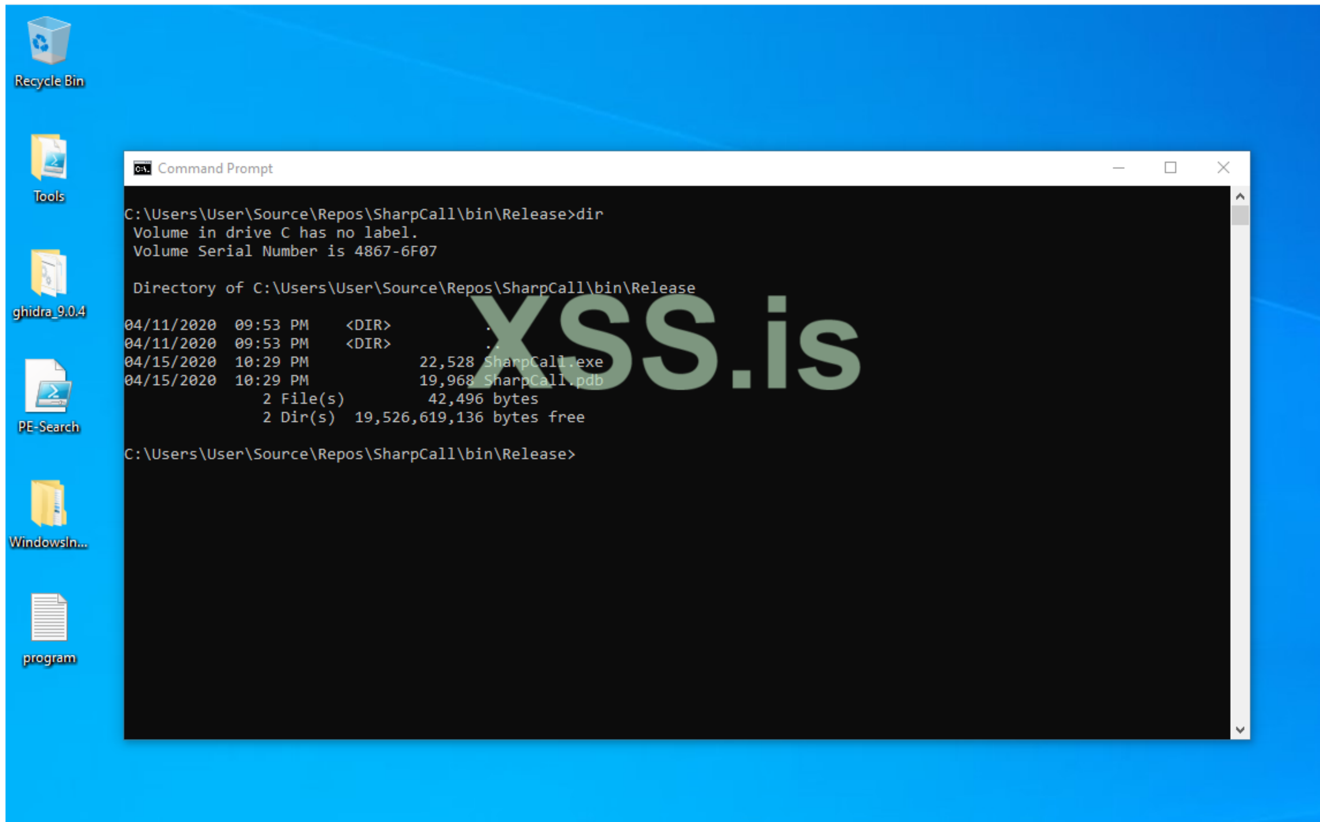
Через несколько секунд вы должны увидеть следующий результат, который показывает нам, что компиляция прошла успешно!



Ладно, не будем слишком волноваться! Код все равно может дать сбой во время тестирования, но я уверен, что это не так!

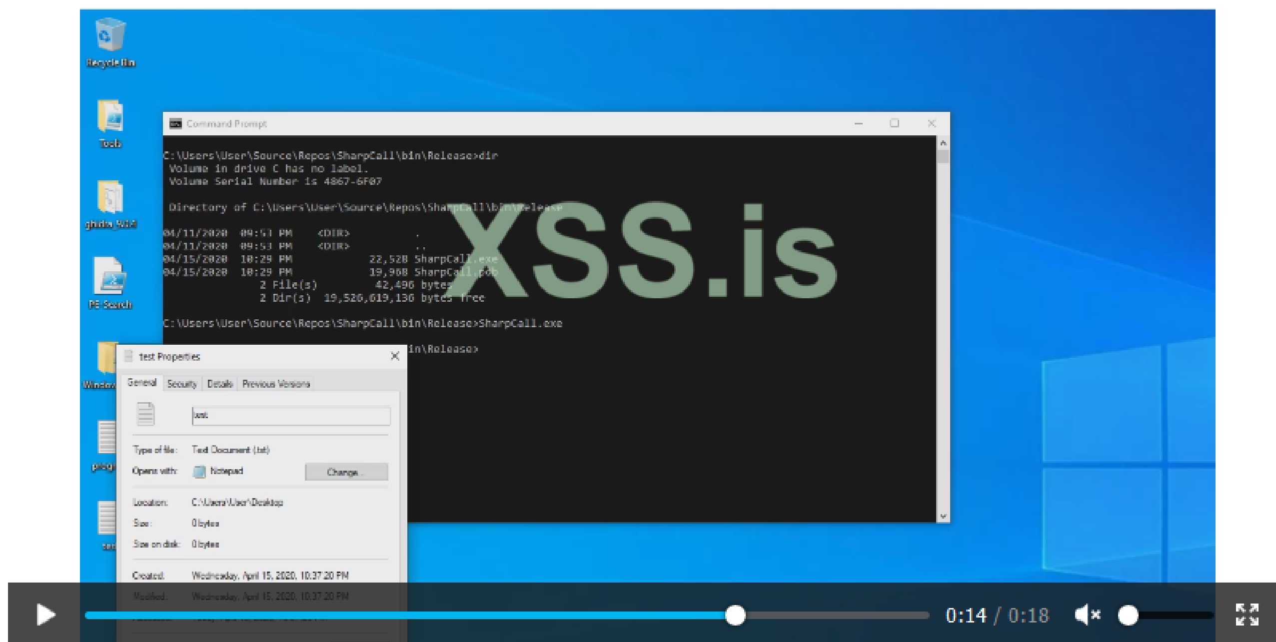
Чтобы протестировать наш недавно скомпилированный код, давайте откроем командную строку и перейдем туда, где скомпилирован наш проект. В моем случае это C:\Users\User\Source\Repos\ SharpCall\bin\Release\.

Как видите, на моем рабочем столе нет файла test.txt, как показано ниже.



Если все пойдет хорошо, то после выполнения нашего исполняемого файла SharpCall.exe должен быть выполнен наш системный вызов, а на рабочем столе должен быть создан новый файл test.txt.

Хорошо, момент истины. Посмотрим на этого плохого парня в действии!

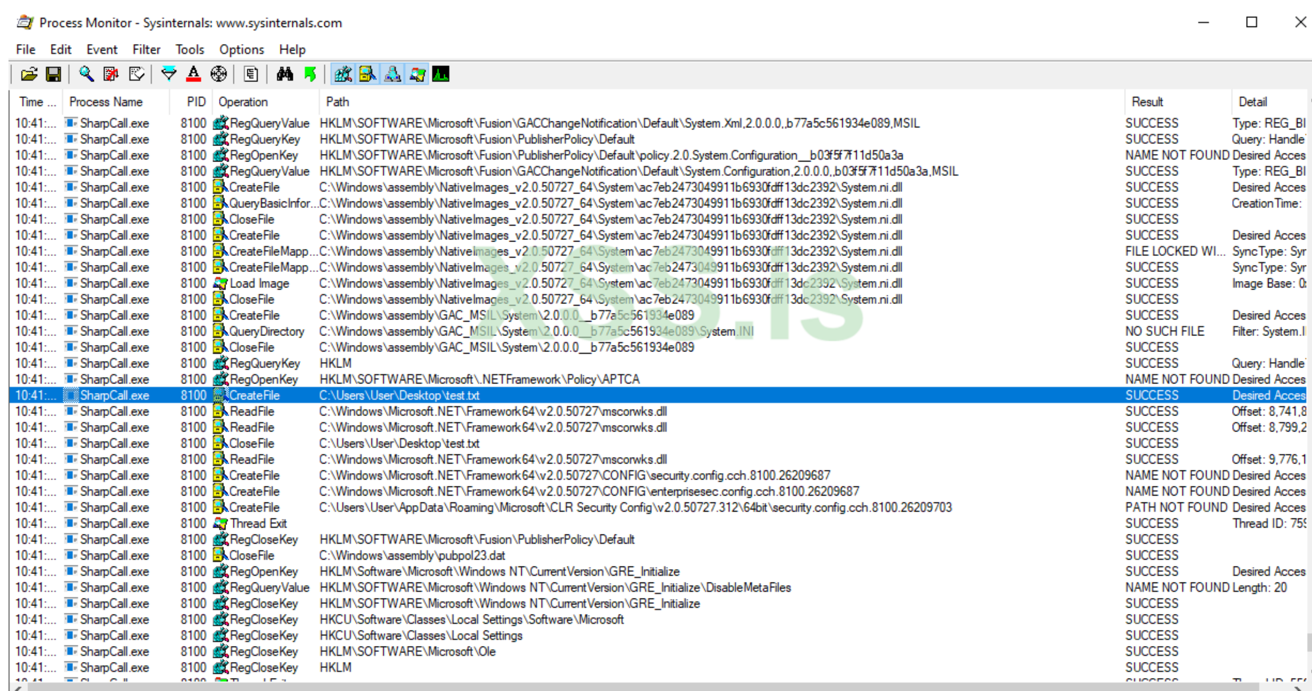


И вот оно! Наш код работает, и мы смогли успешно выполнить наш системный вызов!

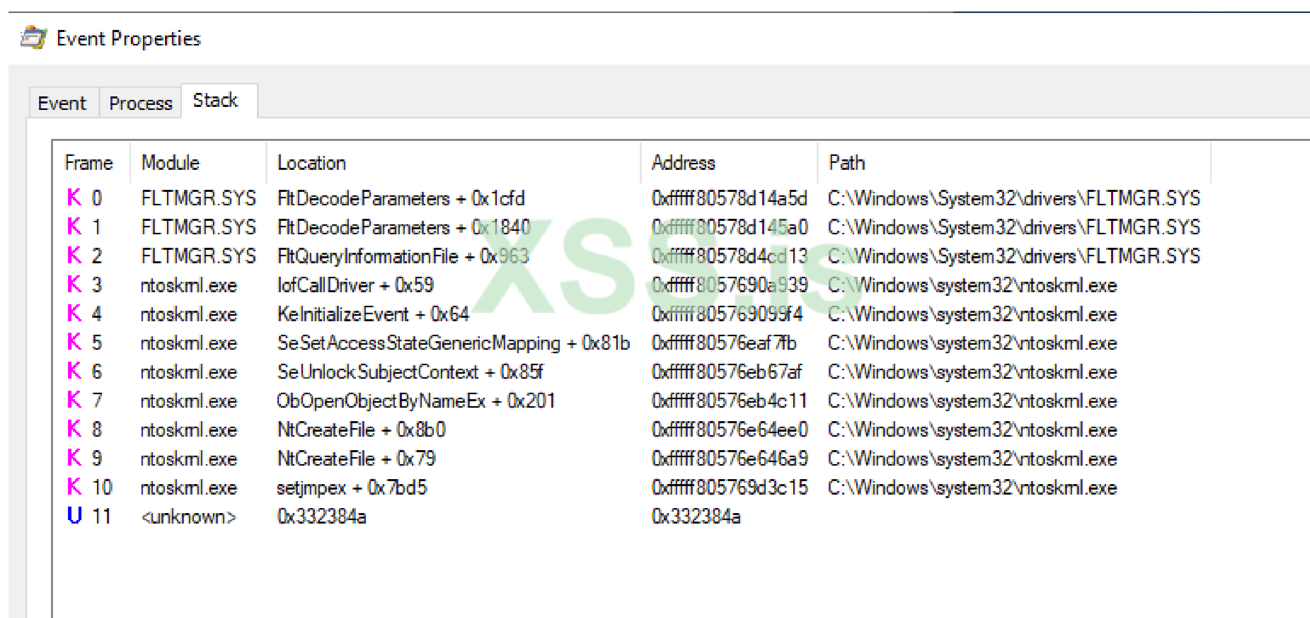
Но как мы можем быть уверены, что это был системный вызов, а не только собственная функция api из ntdll?

Чтобы убедиться, что это был наш системный вызов, мы можем снова использовать Process Monitor для отслеживания нашего исполняемого файла.

Отсюда мы можем просмотреть определенные свойства операции чтения/записи и их стек вызовов.



После наблюдения за процессом во время выполнения мы видим, что для нашего файла test.txt была одна операция CreateFile. Если бы мы просмотрели стек вызовов этой операции, мы бы увидели следующее.


Event Properties

Event | Process | Stack

Frame	Module	Location	Address	Path
K 0	FLTMGR.SYS	FltDecodeParameters + 0x1cfd	0xfffff80578d14a5d	C:\Windows\System32\drivers\FLTMGR.SYS
K 1	FLTMGR.SYS	FltDecodeParameters + 0x1840	0xfffff80578d145a0	C:\Windows\System32\drivers\FLTMGR.SYS
K 2	FLTMGR.SYS	FltQueryInformationFile + 0x963	0xfffff80578d4cd13	C:\Windows\System32\drivers\FLTMGR.SYS
K 3	ntoskrnl.exe	IofCallDriver + 0x59	0xfffff8057690a939	C:\Windows\system32\ntoskrnl.exe
K 4	ntoskrnl.exe	KeInitializeEvent + 0x64	0xfffff805769099f4	C:\Windows\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	SeSetAccessStateGenericMapping + 0x81b	0xfffff80576eaf7fb	C:\Windows\system32\ntoskrnl.exe
K 6	ntoskrnl.exe	SeUnlockSubjectContext + 0x85f	0xfffff80576eb67af	C:\Windows\system32\ntoskrnl.exe
K 7	ntoskrnl.exe	ObOpenObjectByNameEx + 0x201	0xfffff80576eb4c11	C:\Windows\system32\ntoskrnl.exe
K 8	ntoskrnl.exe	NtCreateFile + 0x8b0	0xfffff80576e64ee0	C:\Windows\system32\ntoskrnl.exe
K 9	ntoskrnl.exe	NtCreateFile + 0x79	0xfffff80576e646a9	C:\Windows\system32\ntoskrnl.exe
K 10	ntoskrnl.exe	setjmpex + 0x7bd5	0xfffff805769d3c15	C:\Windows\system32\ntoskrnl.exe
U 11	<unknown>	0x332384a	0x332384a	

Посмотрите на это! Никаких вызовов с или на ntdll не производилось! Просто простой системный вызов из неизвестной области памяти в ntoskrnl.exe! Мы сделали правильный системный вызов!

По сути, это обойдёт любые перехваты API, если бы они были реализованы в NtCreateFile!

Заключение

И вот оно, дамы и господа! Узнав много нового о Windows Internals, Syscalls и C#, вы теперь сможете использовать то, что узнали здесь, для создания ваших собственных системных вызовов на C#!

Окончательный код этого проекта был добавлен в репозиторий Sharp Call на моем Github.

Я упомянул в начале этого сообщения в блоге, что опубликую несколько ссылок на проекты, использующие ту же функциональность. Так что если вы застряли или просто хотите вдохновения, я предлагаю вам взглянуть на следующие проекты.

> https://github.com/b4rtik/SharpMiniDump/

> https://github.com/Kudaes/LOLBITS
> https://github.com/badBounty/directInjectorPOC

https://github.com/badBounty/directInjectorPOC

Ладно, вот и все! Я очень благодарен всем за то, что прочитали эти сообщения в блоге и за то, что первая часть имела такой шокирующий успех! Я не ожидал, что она будет так хорошо принята. Надеюсь, вам понравилась эта часть так же, как и часть 1, и я также надеюсь, что вы узнали что-то новое!

Спасибо всем за чтение! Cheers!

Источник: https://jhalon.github.io/utilizing-syscalls-in-csharp-2/
Автор перевода: yashechka
Переведено специально для портала XSS.is (c)