Hacking C#

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

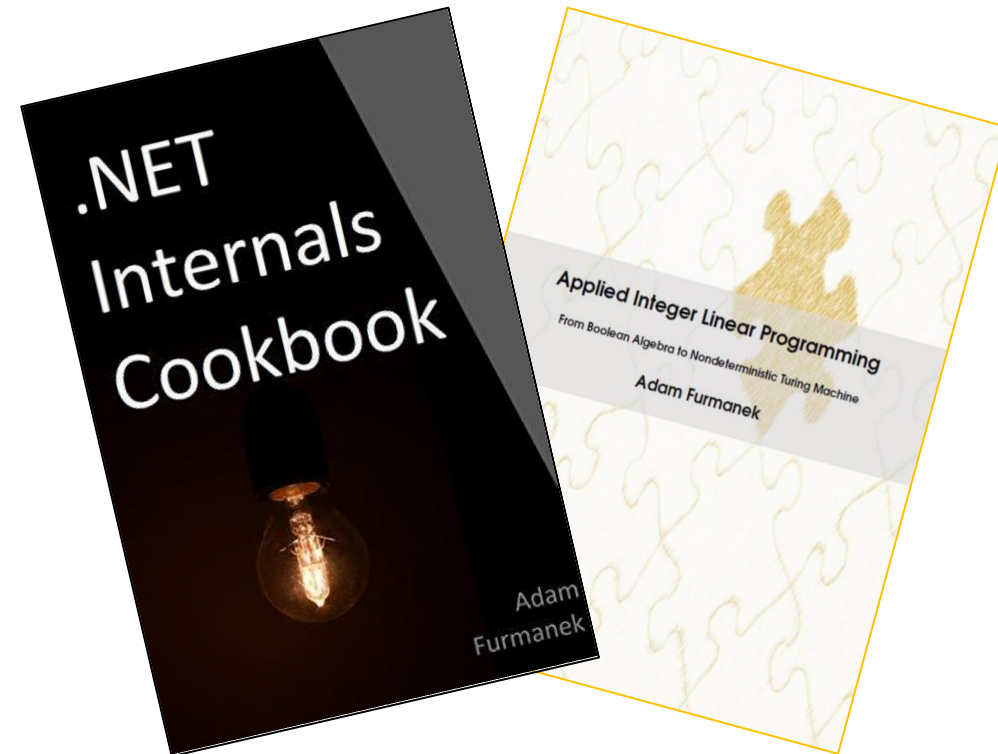
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

You always show how to use machine code and other hacks to (...) but you could just show how all these hacks work.

KEVIN GOSSE

Agenda

Avoiding dynamic dispatch.

Awaiting async void methods.

Running machine code from a byte array.

- Hijacking methods.
- Running things on other desktops.
- Catching unhandled exceptions.
- Catching SOE.

Abusing type system.

- Serializing non-serializable type.
- Implementing multiple inheritance.

Avoiding dynamic dispatch

Dynamic Dispatch

```
public class Base{
    public virtual void Foo(int dummyParameter){
        Console.WriteLine("Base: " + dummyParameter);
    }
}

public class Derived : Base{
    public override void Foo(int dummyParameter){
        Console.WriteLine("Derived: " + dummyParameter);
    }
}

public class Derived2 : Derived{
    public override void Foo(int dummyParameter){
        Console.WriteLine("Derived2 " + dummyParameter);
    }
}
```

```
public class Program
{
    public static void Main()
    {
        Base b = new Derived2();
        b.Foo(123);
    }
}
```

IL

CALL

The call instruction calls the **method indicated by the method descriptor passed with the instruction**. The method descriptor is a metadata token that indicates the method to call and the number, type, and order of the arguments that have been placed on the stack to be passed to that method as well as the calling convention to be used.

CALLVIRT

The callvirt instruction calls a late-bound method on an object. That is, the method is chosen **based on the runtime type** of obj rather than the compile-time class visible in the method pointer.

Callvirt can be used to call both virtual and instance methods.

Dispatch

```
1 reference
public static void InvokeMethod<T>(Expression<Action> caller) where T : class
{
    if (caller.Body.NodeType != ExpressionType.Call)
    {
        throw new ArgumentException("Parameter must call a method", "caller");
    }
    var castedExpression = (MethodCallExpression)caller.Body;
    var evaluatedArguments = castedExpression.Arguments.Select(a => GetLambda(a).Invoke()).ToArray();
    var methodToCall = typeof(T).GetMethod(castedExpression.Method.Name, evaluatedArguments.Select(a => a.GetType()).ToArray());
    var methodInvoker = GetCaller<T>(castedExpression, methodToCall);
    methodInvoker.Invoke((T)GetLambda(castedExpression.Object).Invoke(), evaluatedArguments);
}

1 reference
public static Action<T, object[]> GetCaller<T>(MethodCallExpression expression, MethodInfo methodToCall)
{
    var helperMethod = new DynamicMethod(string.Empty,
                                         typeof(void),
                                         new[] { typeof(T), typeof(object[]) },
                                         typeof(T).Module,
                                         true);
    var ilGenerator = helperMethod.GetILGenerator();
    ilGenerator.Emit(OpCodes.Ldarg_0);
    for (int i = 0; i < expression.Arguments.Count(); ++i)
    {
        var argumentType = expression.Arguments[i].Type;
        ilGenerator.Emit(OpCodes.Ldarg_1);
        ilGenerator.Emit(OpCodes.Ldc_I4, i);
        ilGenerator.Emit(OpCodes.Ldelem, typeof(object));
        if (argumentType.IsValueType)
        {
            ilGenerator.Emit(OpCodes.Unbox_Any, argumentType);
        }
    }
    ilGenerator.Emit(OpCodes.Call, methodToCall);
    ilGenerator.Emit(OpCodes.Ret);
    var methodInvoker = (Action<T, object[]>)helperMethod.CreateDelegate(typeof(Action<T, object[]>));
    return methodInvoker;
}
```

Awaiting async void methods

Awaiting *async void*

We cannot do it directly as method returns nothing.

We need to implement custom synchronization context.

To handle exceptions we need to write custom task scheduler.

await async void

Program.cs — AwaitVoid — AwaitVoid.MyContext — Checkpoint()

```csharp
            }
        }

        public override void OperationStarted()
        {
            _taskCount++;
        }

        public override void OperationCompleted()
        {
            _taskCount--;
            SignalIfDone();
        }
    }

    public static class DelegateHelper
    {
        public static Task AwaitAsynchronousHandlers(this Delegate @delegate)
        {
            var context = new MyContext();
            var thread = new Thread(() => {
                SynchronizationContext.SetSynchronizationContext(context);
                @delegate.DynamicInvoke();
                context.Checkpoint();
            });
            thread.Start();
            return context.Waiter;
        }
    }

    delegate void Worker();

    public class Program
    {
        static event Worker Workers;

        public static void Main()
        {
            Workers += async () => await Task.Delay(100).ContinueWith(t => Console.WriteLine(100));
            Workers += async () => await Task.Delay(1500).ContinueWith(t => Console.WriteLine(1500));
            Workers += () => Console.WriteLine("No delay");
            Workers.AwaitAsynchronousHandlers().Wait();
        }
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
No delay
100
1500
Press any key to continue . . .
```

Catch exceptions in *async void*

Program.cs

CatchAsyncVoid | CatchAsyncVoid.MyContext | Run(Action action)

```
82          public static Task Run(Action action)
83          {
84              return Task.Run(() =>
85              {
86                  var oldContext = SynchronizationContext.Current;
87                  var newContext = new MyContext();
88                  try
89                  {
90                      SynchronizationContext.SetSynchronizationContext(newContext);
91                      var spanningTask = newContext.factory.StartNew(action);
92                      foreach (var task in newContext.scheduler.tasks.GetConsumingEnumerable())
93                      {
94                          newContext.scheduler.TryExecuteTask(task);
95                          task.GetAwaiter().GetResult();
96                      }
97                      spanningTask.GetAwaiter().GetResult();
98                  }
99                  finally
100                 {
101                     SynchronizationContext.SetSynchronizationContext(oldContext);
102                 }
103             });
104         }
105     }
106
107     class Program
108     {
109         static void Main(string[] args)
110         {
111             Console.WriteLine("Preparing job to run");
112             var task = MyContext.Run(() => Throw());
113             Console.WriteLine("Job is scheduled, will run any second. Sleeping main thread");
114             Thread.Sleep(5000);
115             Console.WriteLine("Catching exception");
116             try
117             {
118                 task.GetAwaiter().GetResult(); // Using Wait() here (or in lines 95, 97) instead would return AggregateException instead of original one
119             }
120             catch (Exception e)
121             {
122                 Console.WriteLine("Swallowing exception " + e.GetType() + "\n" + e);
123             }
124
125             Thread.Sleep(1000);
126             Console.WriteLine("Done");
127         }
128
```

C:\WINDOWS\system32\cmd.exe

```
Preparing job to run
Job is scheduled, will run any second. Sleeping main thread
        Waiting in async void
        Throwing in async void
Catching exception
Swallowing exception System.Exception
System.Exception: Hahaha from async void
   at CatchAsyncVoid.Program.<Throw>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Program.cs:line 134
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_0(Object state)
   at CatchAsyncVoid.MyContext.<>c__DisplayClass4_0.<Post>b__0() in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Program.cs:line 59
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at CatchAsyncVoid.MyContext.<>c__DisplayClass8_0.<Run>b__0() in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Program.cs:line 95
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at CatchAsyncVoid.Program.Main(String[] args) in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Program.cs:line 118
Done
Press any key to continue . . .
```

Solution Explorer

Wątki

Zadania

Running machine code from a byte array

Function

Typically is JIT-compiled but can be pregenerated (ngen or ready-to-run).

Must have a machine code (or multiple of them).

Has specific calling convention (parameters via registers + via stack).

Has metadata represented via Method Descriptor (or handle).

Machine code

Is NOT an assembly code — one assembly mnemonic represents many machine instructions.

Cannot be easily read backwards in x86 — instructions have different lengths.

Needs to adhere to the endianess.

Is not different from data — it's always a bunch of bytes.

Must adhere to memory page permissions (page must be executable).

Machine code

https://defuse.ca/online-x86-assembler.htm

Useful page for disassembling the code.

Assemble

Enter your assembly code using Intel syntax below.

```
mov eax, 123
```

Architecture: ◉ x86 ○ x64 Assemble

Assembly

Raw Hex (zero bytes in bold):

B87B**000000**

String Literal:

"\xB8\x7B\x00\x00\x00"

Array Literal:

{ 0xB8, 0x7B, 0x00, 0x00, 0x00 }

Disassembly:

```
0:  b8 7b 00 00 00          mov    eax,0x7b
```

Marshal.GetDelegateForFunctionPointer

Converts an unmanaged function pointer to a delegate.

ptr is converted to a delegate that invokes the unmanaged method using the __stdcall calling convention on Windows, or the __cdecl calling convention on Linux and macOS.

Jump

We can take existing machine code and modify in place.

Then we can jump anywhere.

Jumps are relative — they jump „by” offset. Jump 123 means jump 123 bytes forward.

To do an absolute jump we can use push + ret trick.

ByteToFunc

```
namespace ByteToFunc_Marshal
{
    // Flags for VirtualProtect method
    1 reference
    public enum Protection
    {
        PAGE_NOACCESS = 0x01,
        PAGE_READONLY = 0x02,
        PAGE_READWRITE = 0x04,
        PAGE_WRITECOPY = 0x08,
        PAGE_EXECUTE = 0x10,
        PAGE_EXECUTE_READ = 0x20,
        PAGE_EXECUTE_READWRITE = 0x40,
        PAGE_EXECUTE_WRITECOPY = 0x80,
        PAGE_GUARD = 0x100,
        PAGE_NOCACHE = 0x200,
        PAGE_WRITECOMBINE = 0x400
    }

    2 references
    public class FuncGenerator
    {
        // Method to unlock page for executing
        [DllImport("kernel32.dll", SetLastError = true)]
        1 reference
        static extern bool VirtualProtect(IntPtr lpAddress, uint dwSize, uint flNewProtect, out uint lpflOldProtect);

        // Unlocks page for executing
        1 reference
        private static void UnlockPage(int address)
        {
            uint old;
            VirtualProtect((IntPtr)address, 6, (uint)Protection.PAGE_EXECUTE_READWRITE, out old);
        }

        // Some internal storage for pinning
        private static IList<object> memory = new List<object>();
        private static IList<GCHandle> handles = new List<GCHandle>();

        // Pins array with code and returns address to the beginning of the array
        1 reference
        private static IntPtr Pin(object data)
        {
            memory.Add(data);
            var handle = GCHandle.Alloc(data);
            handles.Add(handle);

            return Marshal.ReadIntPtr(GCHandle.ToIntPtr(handle));
        }

        // Returns delegate of type T using class U for stubbing
        2 references
        public static T Generate<T>(byte[] data)
        {
            // Address of machine code in array
            // We omit first 8 bytes (array type and size)
            var arrayCodeAddress = ((int)Pin(data)) + 8;
```

Method hijacking

```
namespace MethodHijacker2
{
    // This doesn't work with ngened methods - they don't have address stored in method descriptor
    3 references
    public static class MethodHijacker
    {
        3 references
        public static void HijackMethod(MethodBase source, MethodBase target)
        {
            RuntimeHelpers.PrepareMethod(source.MethodHandle);
            RuntimeHelpers.PrepareMethod(target.MethodHandle);

            var sourceAddress = source.MethodHandle.Value;
            var targetAddress = (long)target.MethodHandle.GetFunctionPointer();

            Marshal.WriteInt32(sourceAddress, 8, (int)targetAddress);
        }
    }

    18 references
    class TestClass
    {
        3 references
        public static string ReturnString()
        {
            return "Original string";
        }

        2 references
        public static string ReturnStringHijacked()
        {
            return "Modified string";
        }

        4 references
        public static string StringProperty { get; set; }

        1 reference
        public string NonStaticReturnStringHijacked()
        {
            return "Nonstatic modified string";
        }

        4 references
        public string NonStaticStringProperty { get; set; }
    }
```

Desktop hack

```
namespace DesktopApiExtender
{
    1 reference
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Don't forget to run Sysinternals Desktop application and switch to second desktop
            RunProcess("notepad.exe", "", "");
            Console.ReadLine();
            RunProcess("notepad.exe", "", "Sysinternals Desktop 1");
        }

        [ThreadStatic]
        private static GCHandle? currentDesktopNameHandle;

        2 references
        private static Process RunProcess(string fileName, string arguments, string desktopName)
        {
            var process = new Process();
            var startInfo = new ProcessStartInfo
            {
                WindowStyle = ProcessWindowStyle.Hidden,
                FileName = fileName,
                Arguments = arguments,
                RedirectStandardOutput = true,
                UseShellExecute = false,
                StandardOutputEncoding = Encoding.UTF8
            };
            process.StartInfo = startInfo;

            HijackDesktopLogic(startInfo, desktopName);

            process.Start();
            return process;
        }

        1 reference
        public static void HijackDesktopLogic(ProcessStartInfo startInfo, string desktopName)
        {
            if (!string.IsNullOrEmpty(desktopName))
            {
                byte[] tempDesktopName = Encoding.UTF8.GetBytes(desktopName);
                byte[] desktopNameArray = tempDesktopName.SelectMany(b => new byte[] { b, 0 }).Concat(new byte[] { 0, 0 }).ToArray();
                currentDesktopNameHandle = GCHandle.Alloc(desktopNameArray, GCHandleType.Pinned);
            }

            var matchingType = AppDomain.CurrentDomain.GetAssemblies().SelectMany(a => a.GetTypes()).First(t => t.Name.Contains("STARTUPINFO"));
            var constructor = matchingType.GetConstructor(new Type[0]);
            var newConstructor = typeof(Program).GetMethod(nameof(NewConstructor), BindingFlags.Static | BindingFlags.Public);

            MethodHijacker.MethodHijacker.HijackMethod(constructor, newConstructor);
        }
```

Exceptions in other threads

Unhandled exception kills the application in most cases.

If it happens on a thread pool it is held until awaiting and then propagated if possible (thrown out of band for *async void*).

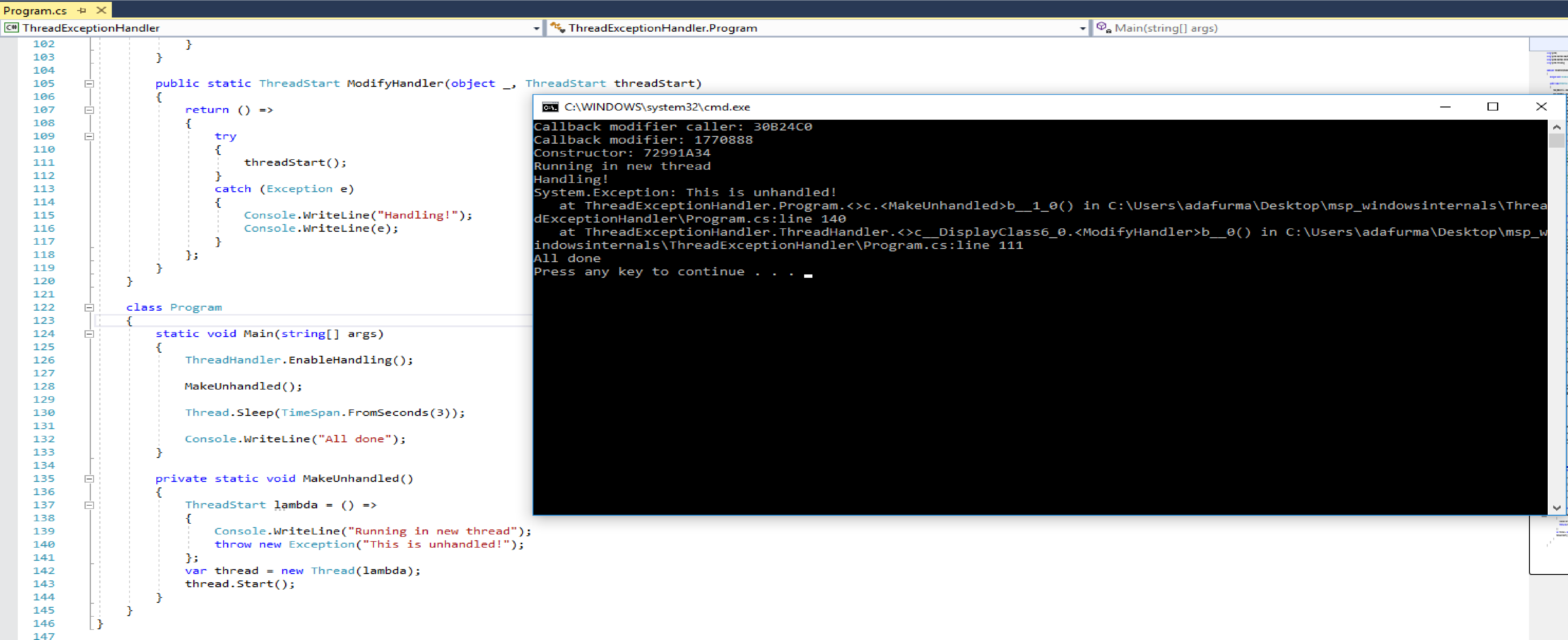
Catching unhandled exception with *AppDomain.CurrentDomain.UnhandledException* doesn’t stop the application from terminating.

ThreadAbortException or *AppDomainUnloadedException* **do not** kill the application.

In .NET 1 it was different:

- Exception on a thread pool was printed to the console and the thread was returned to the pool.
- Exception in other thread was printed to the console and the thread was terminated.
- Exception on the finalizer was printed to the console and finalizer was still working.
- Exception on the main thread resulted in application termination.

Hijacking thread creation



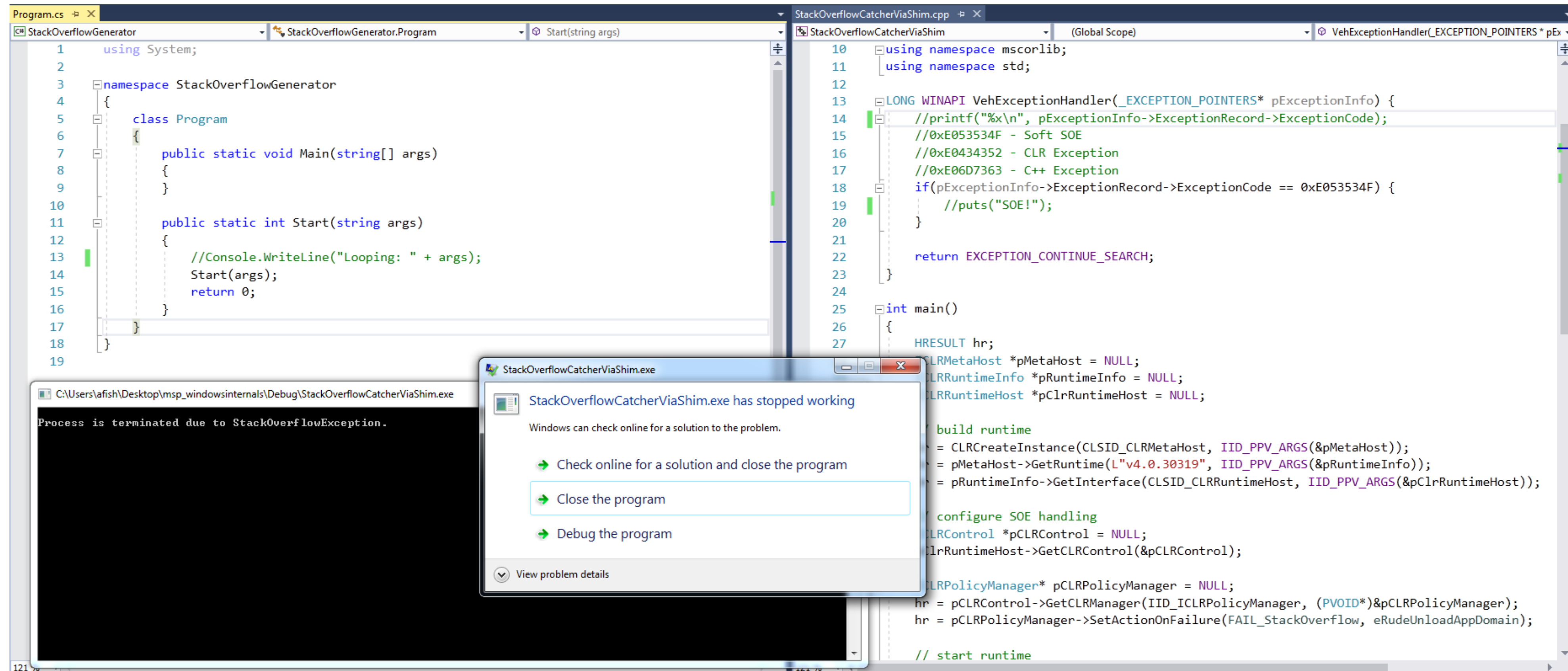
Catching with shim

```
Program.cs
1  using System;
2
3  namespace StackOverflowGenerator
4  {
5      class Program
6      {
7          public static void Main(string[] args)
8          {
9          }
10
11         public static int Start(string args)
12         {
13             Console.WriteLine("Looping: " + args);
14             Start(args);
15             return 0;
16         }
17     }
18 }
19
```

```
Microsoft Visual Studio Debug Console
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
Looping: Argument
I'm done
```

```
StackOverflowCatcherViaShim.cpp
10 using namespace mscorlib;
11 using namespace std;
12
13 LONG WINAPI VehExceptionHandler(_EXCEPTION_POINTERS* pExceptionInfo) {
14     printf("%x\n", pExceptionInfo->ExceptionRecord->ExceptionCode);
15     //0xE053534F - Soft SOE
16     //0xE0434352 - CLR Exception
17     //0xE06D7363 - C++ Exception
18     if(pExceptionInfo->ExceptionRecord->ExceptionCode == 0xE053534F) {
19         puts("SOE!");
20     }
21
22     return EXCEPTION_CONTINUE_SEARCH;
23 }
24
25 int main()
26 {
27     HRESULT hr;
28     ICLRMetaHost *pMetaHost = NULL;
29     ICLRRuntimeInfo *pRuntimeInfo = NULL;
30     ICLRRuntimeHost *pClrRuntimeHost = NULL;
31
32     // build runtime
33     hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_PPV_ARGS(&pMetaHost));
34     hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
35     hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_PPV_ARGS(&pClrRuntimeHost));
36
37     // configure SOE handling
38     ICLRControl *pCLRControl = NULL;
39     pClrRuntimeHost->GetCLRControl(&pCLRControl);
40
41     ICLRPolicyManager* pCLRPolicyManager = NULL;
42     hr = pCLRControl->GetCLRManager(IID_ICLRPolicyManager, (PVOID*)&pCLRPolicyManager);
43     hr = pCLRPolicyManager->SetActionOnFailure(FAIL_StackOverflow, eRudeUnloadAppDomain);
44
45     // start runtime
```

Catching with shim



Catching StackOverflowException in C#

1. Generate machine code on the fly.
2. Register VEH handler with P/Invoke.
3. Use „SetJump LongJump” like approach:
   1. Store registers.
   2. Call method generating SOE.
   3. Restore registers in VEH handler.
   4. Rely on VEH mechanism to perform the jump.
4. Continue.

```
56      // Exception info is passed in esp+8 which is fourth parameter according to fastcall calling convention
57      // Having signature correctly like (int, int, int, IntPtr) spils stack because of calling convention - .NET will remove 2 integers from it
58      // We need to fix ret 8 to ret 4 at the end
59      public static int CatchingVeh(int a, int b, int c, IntPtr pExceptionInfo)
60      {
61          //#define EXCEPTION_EXECUTE_HANDLER      1
62          //#define EXCEPTION_CONTINUE_SEARCH      0
63          //#define EXCEPTION_CONTINUE_EXECUTION (-1)
64
65          Console.WriteLine("Catching VEH!");
66
67          uint exceptionCode = (uint)Marshal.ReadInt32(Marshal.ReadIntPtr(pExceptionInfo));
68          Console.WriteLine("Exception code: " + exceptionCode.ToString("X"));
69          if(exceptionCode != 0xC00000FD)
70          {
71              // Not Stack Overflow Exception
72              // Return CONTINUE_SEARCH
73              return 0;
74          }
75
76          // We restore registers here using ExceptionInfo structure
77
78          Console.WriteLine("Info: " + pExceptionInfo.ToString("X"));
79          IntPtr pContextRecord = Marshal.ReadIntPtr(pExceptionInfo + 4);
80          Console.WriteLine("Context: " + pContextRecord.ToString("X"));
81
82          Marshal.WriteIntPtr(pContextRecord + 0xB8, (IntPtr)EipEspEbpEsiEdiEbxHolder[0]);
83          Marshal.WriteIntPtr(pContextRecord + 0xC4, (IntPtr)EipEspEbpEsiEdiEbxHolder[1]);
84          Marshal.WriteIntPtr(pContextRecord + 0xB4, (IntPtr)EipEspEbpEsiEdiEbxHolder[2]);
85          Marshal.WriteIntPtr(pContextRecord + 0xA0, (IntPtr)EipEspEbpEsiEdiEbxHolder[3]);
86          Marshal.WriteIntPtr(pContextRecord + 0x9C, (IntPtr)EipEspEbpEsiEdiEbxHolder[4]);
87          Marshal.WriteIntPtr(pContextRecord + 0xA4, (IntPtr)EipEspEbpEsiEdiEbxHolder[5]);
88
89          Console.WriteLine("Going back to Eip: " + EipEspEbpEsiEdiEbxHolder[0].ToString("X"));
90
91          // Return CONTINUE_EXECUTION
92          return -1;
93      }
94
95      // Method for registering VEH handler
96      [DllImport("Kernel32.dll", SetLastError = true)]
97      static extern int AddVectoredExceptionHandler(uint first, IntPtr addres);
```

C:\WINDOWS\system32\cmd.exe

```
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Preparing to throw
Catching VEH!
Exception code: C00000FD
Info: E02630
Context: E02B0C
Going back to Eip: 2FE4E98
I'm done
Press any key to continue . . .
```

Abusing type system

Abusing type system

Methods assume parameters have correct types.

Types are checked by the C# compiler and when loading the module.

call/callvirt instructions do not check types.

Methods do not check types.

Instance is just a bunch of bytes. It has no methods associated.

Abusing type system

```
namespace AbusingTypeSystem
{
    public class A
    {
        public virtual void Print()
        {
            Console.WriteLine("A.Print");
            Console.WriteLine(this.GetType());
        }
    }

    public class B
    {
        public virtual void Print()
        {
            Console.WriteLine("B.Print");
            Console.WriteLine(this.GetType());
        }

        // Make this method non-virtual to see that it doesn't crash
        -references
        public virtual void Print2()
        {
            Console.WriteLine("B.Print2");
            Console.WriteLine(this.GetType());
        }
    }
```

Serialization

```
namespace SerializeNonSerializable
{
    [Serializable]
    public class Root
    {
        public int RootField { get; set; }
        public NonSerializableChild Child { get; set; }

        public void Print()
        {
            Console.WriteLine($"Root.Print {RootField}");
            Magic(Child);
        }

        public void Magic(NonSerializableChild child)
        {
            Console.WriteLine($"Delegating to child {child.GetType()}");
            child.Print();
        }
    }

    public class NonSerializableChild
    {
        public int ChildField { get; set; }

        public void Print()
        {
            Console.WriteLine($"NonSerializableChild.Print {ChildField}");
        }
    }

    [Serializable]
    public class SerializableChild
    {
        2 references
        public int ChildField { get; set; }

        0 references
        public void Print()
        {
            Console.WriteLine($"SerializableChild.Print {ChildField}");
        }
    }
```

Multiple Inheritance

```
static void Main(string[] args)
{
    MultipleBase child = new FakeChild1
    {
        currentState = new CurrentState(new Base1(), new Base2(), new Base3(), new Base4())
    };

    Console.WriteLine("Base1");
    Base1 base1 = child.Morph<FakeChild1, Base1>();
    base1.field = 123;
    base1.PrintInt();
    Console.WriteLine();

    Console.WriteLine("Base2");
    Base2 base2 = child.Morph<FakeChild2, Base2>();
    base2.field = 456.0f;
    base2.PrintFloat();
    Console.WriteLine();

    Console.WriteLine("Base3");
    Base3 base3 = child.Morph<FakeChild3, Base3>();
    base3.field1 = 789;
    base3.field2 = 987;
    base3.PrintFields();
    Console.WriteLine();

    Console.WriteLine("Base4");
    Base4 base4 = child.Morph<FakeChild4, Base4>();
    base4.field = "Abrakadabra";
    base4.PrintString();
    Console.WriteLine();

    Console.WriteLine("Base3 again");
    base3 = child.Morph<FakeChild3, Base3>();
    base3.PrintFields();
    Console.WriteLine();

    Console.WriteLine("Base2 again");
    base2 = child.Morph<FakeChild2, Base2>();
    base2.PrintFloat();
    Console.WriteLine();

    Console.WriteLine("Base1 again");
    base1 = child.Morph<FakeChild1, Base1>();
    base1.PrintInt();
}
```

Summary

You need to remember about all platform components.

You need to understand Operating System and .NET interoperability.

You need to know your CPU architecture.

And you need to be careful.

Ultimately — it’s just a bunch of bytes.

Q&A

References

Jeffrey Richter - „CLR via C#”

Jeffrey Richter, Christophe Nasarre - „Windows via C/C++”

Mark Russinovich, David A. Solomon, Alex Ionescu - „Windows Internals”

Penny Orwick – „Developing drivers with the Microsoft Windows Driver Foundation”

Mario Hewardt, Daniel Pravat - „Advanced Windows Debugging”

Mario Hewardt - „Advanced .NET Debugging”

Steven Pratschner - „Customizing the Microsoft .NET Framework Common Language Runtime”

Serge Lidin - „Expert .NET 2.0 IL Assembler”

Joel Pobar, Ted Neward — „Shared Source CLI 2.0 Internals”

Adam Furmanek – „.NET Internals Cookbook”

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md — „Book of the Runtime”

https://blogs.msdn.microsoft.com/oldnewthing/ — Raymond Chen „The Old New Thing”

References

https://blog.adamfurmanek.pl/2020/01/11/net-inside-out-part-14-calling-virtual-method-without-dynamic-dispatch/ — Calling virtual method without dynamic dispatch

https://blog.adamfurmanek.pl/2018/10/06/async-wandering-part-5/ — Catching exceptions from async void

https://blog.adamfurmanek.pl/2018/10/13/net-inside-out-part-9-generating-func-from-a-bunch-of-bytes-in-c-revisited/ — Generating Func from a bunch of bytes in C# revisited

https://blog.adamfurmanek.pl/2017/05/27/how-to-override-sealed-function-in-c-revisited/ — How to override sealed function in C# Revisited

https://blog.adamfurmanek.pl/2020/02/29/net-inside-out-part-15/ — Starting process on different desktop

https://blog.adamfurmanek.pl/2017/06/03/capturing-thread-creation-to-catch-exceptions/ — Capturing thread creation to catch exceptions

https://blog.adamfurmanek.pl/2018/04/07/handling-stack-overflow-exception-in-c-with-veh/ — Handling Stack Overflow Exception in C# with VEH

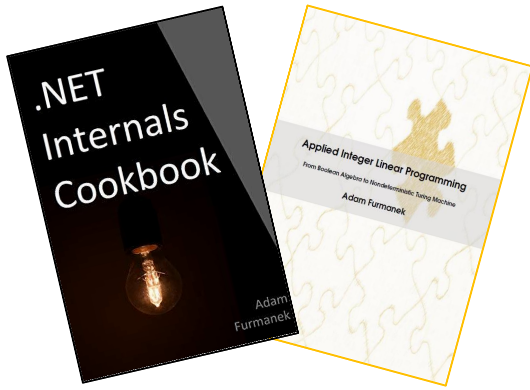
https://blog.adamfurmanek.pl/2020/04/11/net-inside-out-part-16/ — Abusing type system

https://blog.adamfurmanek.pl/2020/04/18/net-inside-out-part-17/ — Abusing types to serialize non-serializable type

https://blog.adamfurmanek.pl/2016/04/30/custom-memory-allocation-in-c-part-2/ — List copying objects

https://blog.adamfurmanek.pl/2019/08/10/custom-memory-allocation-in-c-part-12/ — Hiding objects from GC

https://blog.adamfurmanek.pl/2019/09/14/custom-memory-allocation-in-c-part-13/ — In-place serialization



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM