Hijacking Arbitrary .NET Application Control Flow

Topher Timzen
Dept. of Computer Science
Southern Oregon University
2015

Ryan Allen
Dept. of Computer Science
Southern Oregon University
2015

Abstract

This paper describes the use of Reflection in .NET and how it can be utilized to change the control flow of an arbitrary application at runtime. A tool, Gray Storm, will be introduced that can be injected into an AppDomain and used to control the executing assembly instructions after just-in-time compilation.

I. Introduction

The .NET Framework is used in a variety of enterprise applications such as fingerprint readers and SQL databases with the C# programming language. Through the use of Reflection the framework grants the ability to look at metadata for assemblies, classes, and methods within an application. This ability allows a malicious application to read the metadata of a specific entity in a running AppDomain. That information can be utilized to modify the behavior of a target process's AppDomain at run-time.

.NET AppDomains are equivalent to a process and are containers for running code. An application may have several AppDomains and all of them can be accessed through Reflection. Because of the nature of just-in-time compilation (JIT/Jitter), memory is marked readable, writable and executable (rwx) as the JIT compiler needs to write generated machine code from Intermediate Language (IL) to memory for execution.

Gray Storm takes advantage of the way the .NET Framework uses Reflection and JIT. It allows an attacker to read, write and change existing methods. This can be done with compiled on-the-fly C# code or assembly payloads.

II. .NET Basics

When an application is compiled, IL code is generated through implicit compilation [5]. The .NET Framework will then generate machine code at runtime. The common language runtime (CLR) is used by the framework to generate assembly code from IL code. IL code is an independent set of instructions that are converted to native machine code at JIT [6]. When a method is about to be executed, the framework uses JIT to generate assembly code that the CPU can execute by accessing a JIT stub. IL is fairly human readable and shows which methods a given method calls. Utilizing IL, individuals are able to trace the control flow of an application easily.

A. JIT

The CLR reads the metadata from the IL and allocates memory for the machine code. This memory is readable, writable and executable as the JIT needs to be able to write code to memory and have it execute. The following is output from WinDbg showing rwx memory on a method that has been just-in-time compiled.

```
0:008> !address 1EB4F8
Usage: <unknown>
Base Address: 00000000`001e0000
End Address: 00000000`001ed000
Region Size: 00000000`0000d000
State: 00001000 MEM_COMMIT
Protect 00000040 PAGE_EXECUTE_READWRITE
Type: 00020000 MEM_PRIVATE
Allocation Base: 00000000`001e0000
Allocation Protect: 00000001 PAGE_NOACCESS
```

The JIT compiler leaves memory rwx after it writes executable code to memory. That memory can then be overwritten with arbitrary assembly instructions.

B. Reflection

A .NET Assembly is a definition of types, manifests and other metadata constructs [4]. Assemblies contain specific classes, whether they be libraries or programmer generated, and the methods within them. Reflection provides a library of classes in the .NET Framework that give the ability to look at the metadata of Assemblies. With Reflection the programmer can look at the return type and arguments of a method and the address of a method's generated assembly. Furthermore, Reflection shows all AppDomains within a process and all of the constructors within a class.

Gray Storm utilizes the ability of Reflection to see the function pointer of the rwx memory the JIT produces. Reflection contains a MethodInfo class that has the attributes of a method. MethodInfo contains a MethodHandle that gets an internal metadata representation of a method. With that, a call to GetFunctionPointer() gives the address of the rwx memory where the JIT compiler will place the assembly code.

```
IntPtr funPtr = (IntPtr)MethodInfo.MethodHandle.
    GetFunctionPointer().ToInt64();
```

Reflection also allows the user to see all the binding flags used on a method such as static, public and private. This information can be used to generate code at runtime and overwrite the assembly in a target method with malicious code.

III. Gray Storm

Using the capabilities of Reflection in conjunction with rwx memory, Gray Storm takes advantage of the .NET Framework to operate maliciously in an AppDomain. Several attack chains have been implemented to work within a process. The attacks are able to adjust arbitrary call sequences to alter control flow, overwrite Method Tables and compile new C# classes on-the-fly. Furthermore the tool allows an attacker to change existing Method Table pointers to point to new methods, as well as granting the ability to replace original assembly with custom assembly payloads.

DigitalBodyGuard described an injection platform and Gray Storm utilizes similar methods to inject itself into a target [3]. The tool is packaged into a byte array and shipped with a C++ DLL, Gray Frost, that acts as a bootstrapper, which targets the main AppDomain in a .NET application. This bootstrapper process also determines the .NET runtime that a specific application is using and pivots from 4.0 to 2.0 if necessary. As of now remote DLL injection is used but there is work underway to implement a MetaSploit module and utilize reflective DLL injection. By using the bootstrapper approach, Gray Storm is injectable into any arbitrary .NET application.

A. Method Calls

While looking at the assembly code that the Jitter places into rwx memory, the authors noticed two different calling conventions used in x86. The Jitter appears to output both relative and far calls

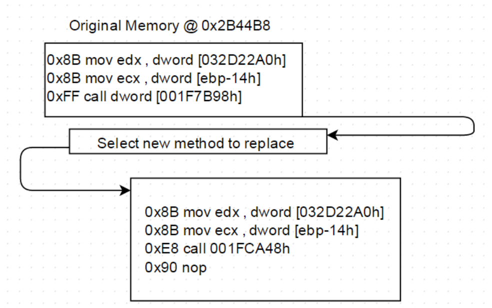
```
0xE8/Call [imm]
0xFF 0x15/Call dword [imm]
```

By looking at the MSIL code before the JIT, it can be accurately determined which calls call which methods. Calls can then be changed to an address under attacker control. For example, the following MSIL code generates the 0xE8 relative call.

```
MSIL: 0008 call System.Void memoryHijacker.abc.
      testCall(System.Int32)
ASM: 0xE8 call 730ECAD9h
```

The far call instruction is 6 bytes, 0xFF 0x15 0x## 0x## 0x## 0x##, and a relative call is 5 bytes, lengthOfCall below, 0xE8 0x## 0x## 0x## 0x##. An attacker can thus change a far call into a relative call by using the algorithm described in the IA32 manual [2] and add a NOP instruction to the 6th byte on the far call to change the instruction to a relative one.

```
relativeCallAddress = dstAddress −          (
      currentLocation + indexInMemoryToCall +
      lengthOfCall)
```


Original Memory @ 0x2B44B8
0x8B mov edx , dword [032D22A0h]
0x8B mov ecx , dword [ebp-14h]
0xFF call dword [001F7B98h]
Select new method to replace
0x8B mov edx , dword [032D22A0h]
0x8B mov ecx , dword [ebp-14h]
0xE8 call 001FCA48h
0x90 nop

Changing a relative call is fairly easy as you just overwrite the 4 bytes of the immediate with the new destination address. Gray Storm has the ability to use an arbitrary address in a call instruction thanks to Reflection.

B. Method Table

Ziad Elmalki described a way to replace Method Tables at runtime because Method Tables have the addresses of the JIT stubs for a class's methods [1]. He goes on to show that if a method has not gone through JIT it references the Method Table to generate addresses for method calls. Elmalki showed that the Method Table can be located in memory using Reflection and then changed so when referenced during JIT, the address of the call instruction can be altered.

Gray Storm utilizes this capability and allows any method to be the new target in a Method Table. On-the-fly C# methods or pre-existing ones can be used to change the control flow before a method is JIT compiled. The prototype to perform this change is listed below. All that is needed is two MethodInfo types so their Method Tables can be viewed with Reflection.

```
public static void ReplaceMethod(MethodInfo
      replacerMethod, MethodInfo targetMethod)
```

The above method will go through and find the MethodHandle of both methods and change the address of the targetMethods table to point to our replacerMethod.

Unfortunately, this technique requires that a method has not yet been JIT compiled. It is not yet known how to force the garbage collector to clean up a method so the table is referenced again. This technique does however provide a way to maintain persistence after garbage collection as Gray Storm can change a call address as shown in section 3.1 for the duration of an object.

The .NET Framework provides CSharpCodeProvider [7] to compile code at runtime. A user can input C# namespaces, classes and methods and run them as if they were compiled with the actual program. Using CSharpCodeProvider, Gray Storm can perform the aforementioned techniques to utilize newly created run-time methods.

Once a method is compiled, it can be invoked or used as a replacer to overwrite Method Table pointers. An attacker can compile any methods they want and if the exact return type and arguments of an original programmed method are matched, can execute a replacement method without crashing the injected application. This feature of the framework grants the ability to read MSIL and assembly code, rewrite a method in memory and then use it. For example, the authors have written methods in memory that send an encrypted e-mail of a users password as they login by changing the control flow of an application.

C. Assembly Level

Again because memory written by the JIT compiler is rwx, Gray Storm is able to write over it with arbitrary assembly instructions. Custom assembly payloads or Metasploit payloads can be used and placed over existing methods. Performing this action is as simple as finding the address of a method's executable memory from Reflection and writing over it with new assembly code. If a payload is longer than the space the current method takes in memory, a user can restructure their assembly code to support a form of hooking.

This hooking mechanism allows an attacker to supply an arbitrarily long payload while also allowing an attacker to restore the method as if no code had been changed. The hooking is achieved by creating a 7 byte sequence of the following assembly.

```
0xB8 0x## 0x## 0x## 0x## //mov eax memory
0xFF 0xD0 //call eax
```

The value of a payload is placed into the memory address that is moved into EAX and then called. In order to restore the original method, the payload needs to be constructed to make room for a payload cleaner that consumes 12 bytes before the payload returns. By creating a cleaner stub with the original return method value the original memory and EIP can be restored to the method prelude. This ensures that once a payload runs, it can restore the method as if no changes were made to control flow.

Gray Storm includes a shell code editor that allows a user to import arbitrary assembly code. Furthermore, it provides the ability to restore a method's original code should the attacker want to undo their actions.

D. Object Manipulation

Within Gray Storm there is an attack chain to find and use instantiated objects at runtime. A classes constructor can be discovered using Reflection and a new one can be instantiated locally. Once an object is constructed the attack chain finds the location of the managed heap, signatures the instantiated object, scans the managed heap and then converts the managed heap object pointers into raw objects [8]. Once objects are referenced locally all of their fields, properties and instance methods can be seen and utilized.

IV. Conclusion

The .NET Framework allows for an attacker to inject into an AppDomain and reflectively learn about the workings of an application. Because Microsoft chose to leave memory readable, writable and executable after JIT compilation, an attacker is able to control the executing assembly, change control flow and overwrite Method Tables. Through Gray Storm it has been demonstrated that the .NET Framework is insecure by design and an arbitrary application can be changed at an attackers whim.

V. Future Work

There are still improvements that can be made to the aforementioned attack chains. As described in 3.4, Gray Storm has the ability to hook a method for a one time use. Adding the ability to restore our payload without attacker interaction would be useful and will be implemented soon. Furthermore, the proof of concept for this tool was written in x86 assembly and work is currently underway to have full compatibility with x64.

VI. Acknowledgments

The authors would like to thank Jon of DigitalBodyGuard for assisting in researching .NET attack vectors and teaching them the power of using Reflection for attacking. Through his help they were able to inject Gray Storm successfully into arbitrary .NET applications. Furthermore, Dr. Lynn Ackler, assistant professor at SOU, for constantly being an encouragement and providing the cybersecurity research lab to all computer science students at the university.

REFERENCES

[1] Elmalki, Ziad. CLR Injection. Jun 2009. http://www.codeproject.com/Articles/37549/CLR-Injection-Runtime-Method-Replacer

[2] *Intel 64 and IA-32 Architectures Software Developers Manual*. Number 325383-053US. January 2015.

[3] Jon. Hacking .Net Application at Runtime. In OWASP APPSEC DC, Nov. 2010.

[4] Jon. Reflections Hidden Power. May 2002.

[5] Microsoft Corporation. .NET Framework 3.5. Compiling to MSIL.

[6] Microsoft Corporation. .NET Framework 3.5. Compiling MSIL to Native Code.

[7] Ponnupandy, Mercy. Dynamic Code Generation and Code Compilation. Dec 2002. http://www.codeproject.com/Articles/3289/Dynamic-Code-Generation-and-Code-Compilation

[8] Timzen, Topher. Acquiring .NET Objects from the Managed Heap. May 2015.