Attacking .Net at Runtime

By

Jonathan McCoy

Abstract

This paper will introduce methodology for attacking .NET programs at runtime. This attack will grant control over the targets variables, core logic, and the GUI. This attack is implemented with .NET code, and is heavily based on reflection.

This paper has a C# implementation of this attack: DotNetSpike

Introduction

This attack can navigate and control a live program by using the rules of the Runtime system to control other .NET applications. Some rules can be bent others can be broken. Once access to another program's Runtime is gained almost absolute control is at hand. Most every aspect from Objects to Events can be accessed, and most of the time modified. This allows for simple attacks like changing an Object's values or calling functionality, and more complex attacks like introducing or changing the basic logic of the target can be done with ease.

With this level of control the target can be forced to divulge protected information, carry out subverted functionality and send corrupted signals. This attack will also allow for accessing the code base and Object structure live. This platform can allow an attack to be developed and implemented in a matter of minutes or hours.

Once inside of the target program the necessary references need to be found, and then full power of .NET can be used. After gaining all the references it can be a matter of changing one variable or introducing one line of code to subvert a program's logic.

Access Live .NET Program

The first step is to establish a connection inside of the target's Runtime; this is done in a number of ways. This can range from compromising the .NET Framework[1] to exploiting a glitch in a specific application. Each method of accessing another application's Runtime has a different impact on stability, foot print and security alerts. Also the method of entry used will change what references are at hand and what if any constraints will be imposed.

The method of access used in this paper is injection by the Windows OS. This method is platform dependent as well as highly detectable by Anti-Virus programs, and starts with no references. However, this method has a fast development cycle.

The security constraints come into play at this point. Is the target a weak program running on a controlled system? Can it be attacked, dissected and restarted a million times? Or perhaps it is a heavily secured program on a server somewhere, and it must not be crashed or restarted. Depending on the environment and the goal a delivery method would be selected that best fits.

No matter what method of entry is used the end effect is to create a way of gaining access to the targets program's Appdomain and Objects. Once this is accomplished it will allow the payload to dig into the logic and structure of the target application.

This paper has a demo implementation of injection into a .NET application: Injector this is a C# program, with C++ unit, that will inject .NET application running on Microsoft Windows compatible with both 32 bit and 64 bit.

[1] .NET Framework Rootkits By Erez Metul - at http://www.applicationsecurity.co.il/english/NETFrameworkRootkits/tabid/161/Default.aspx

Controlling the Runtime

After gaining access to the target's Runtime, it is necessary to get a reference to some Object(s). This is the most difficult part of the attack as you have none of the keys, but luckily you also have no doors. So how do you get to an Object if you don't have any references? Luckily most programs have a GUI and the OS we can retrieve a reference to this Object. In addition, looking for other types of Objects, like global variables and events can be an easy source of references.

After establishing a reference to an Object inside of the target's Runtime, it can be leveraged to gain more references to other Objects. Depending on the layout of the target application and the manner of entry, it will take more or less work to get references and implement the necessary changes. As each Object is traversed, the code for each function can be easily accessed. This along with clean readable code naming can make for an intuitive attack.

Say the goal is to attack the GUI on a target program. Start out by asking the Windows OS about the target's GUI window handles (using an on screen location or PID) and use that information to build a Reference to the GUI Object. This GUI Object will be inside of the target's Runtime, and as such can now be accessed and controlled. Once this reference is created it can be used to find references to other Objects. Iterating over all the Form's child controls and their Event's can gain a far number of references. Perhaps we find a key Object of the application, such as a timer or SQL connection, that can be leveraged to affect the system.

Anything from changing a variable to invoking functionality can be implemented at this point. Objects can with some skill and effort be replaced live. This allows the attackers to change the core logic granting a god mode of sorts.

What is an Object at Runtime

Objects can be asked what Type they are, and what they implement. This in turn gives a Class list, on complex Objects this can be a few levels of inheritance and interfaces.

Once the Class of an Object is found it can be referenced, and the Class can be queried as to what functions it implements and what variables it contains. With this information, it is possible to access constructors, Events, variables, and properties. This allows an Object to be manipulated and controlled.

GUI

The Graphical User Interface(GUI) also known as a Form is a key part of most programs, and is an easy entry point for this attack. Using the Windows OS to get a reference to the GUI Objects will get a good foothold into the heart of the program. After establishing a reference to the main GUI, a reference to each child control is easily accessed, and in turn will lead to most of the core Objects in the program.

A good place to start looking for references is in variables or Event lists. Take a Button on the main Form, assuming good N-Tier design, the Button will have a Click Event connected to some business logic deep in the program.

Your target could be the GUI it self, perhaps the need is to access to a disabled Check Box or override the functionality of a Text Box sanitization event. This is done with ease as the GUI Controls are Objects subject to the same modification and influences as any other Object. However, the GUI imposes an extra constraint of thread safety. Anytime modifications are made to a GUI Control it must be done from a parent thread. To satisfy this move the execution point to inside of a parent thread or call Invoke on a parent Object.

Events

Events are a key aspect of the logic flow to most programs today as well as a probable link to functionality and references. Events are an Object and as such can be controlled.

The basic idea of the Event is to execute a list of function calls. This can be used to gain a reference to an Object or change logic flow. For instance, a timer Object that is the heart beat for the programs could have a number of Objects connected directly to the Timer's Elapsed Event. Each entry in the Event list is a connection to a function that could lead to a reference. If just one event call is intercepted it could alter the flow of the application.

A fair amount of key logic is connected to Events, such as, on a Button, Text Box, or Timer. This logic will (most of the time) be on key Objects. This can be used as a shortcut to get directly to specific logic and Object.

Accessing Source Code

The raw structure is maintained in IL, for example x passed into FOO(int varIN) is

```
0002 : ldarg.0
0003 : ldfld int App.Form1 ::x
0008 : call instance System.Void App.Form1::foo()
```

Assuming that the code is well written and not obfuscated we can move between Objects in an application with confidence. If we can see well-named variables and functionality it is no harder then working on someone else's code. If we do not have well-named Objects it can slow the process down, but with a good understanding of the basic functionality it would most likely only take a few(10-60) minutes to reconstruct a small peace of strongly obfuscated logic(code).

Dynamic function can be created and destroyed at Runtime, giving a nice path for incorporation of new logic. This logic can also be unloaded in an attempt to hide the foot print of this attack.

As for the language the target program is written in, it makes little difference to this attack. Because it should be expected that the code will be obfuscated and thus cannot be reversed (fully) to any wrapper language. So IL should be chosen as the primary language for long term work in this area.

Legitimate uses

This level of control over other applications has legitimate uses as well, such as to extend a program or implement a upgrade.

With this it is possible to extend or reuse another application in new and different ways. It is also possible to combine other applications to form a new system, taking them far beyond the original purposes.

Conclusion: Going in Blind

If the target is an unknown program and only the Runtime is at hand to work with you will have to work in its world, but you can bring your tools. If the OS or Framework has been compromised, and can be uses to help, it will tip the balance of power in our favor, if the target has other programs or security setup to protect itself, the attack will be that much harder. Once a suitable way into the target is found the tactics of this attack should work the same.

Some basic methods of learning about the target should be done. Such as running the attack a few times to see the flow of the program. This will give insight into the target's design and infrastructure. Jumping around the GUI controls and classes can get directly to objects. Also if possible dissecting the target will give information and help in developing the attack.

The real strength of this attack is in how fast and easy it is to adapt and control a running program. The tools used in this attack are in the core of .NET, and for the most part, are in every version and as such should apply to any app.

Tools and rules of the Runtime:

Objects are derived from and instantiated by classes, and must be referenced by a chain of execution. This links every Object together in a predictable way.

A **Class** has a list of variables and functions that can be accessed (both public & private). This can be used to learn about an Object and it's connects, or to control and manipulate it.

The **code** is in IL (most of the time) and can be examined, but cannot be edited (this rule can be broken). The code is solid but the logic can be manipulated. This will allow the behavior of the program to be controlled as the attacker wishes.

A **reference** is needed in order to access an Object (this rule can be bent). Once a few objects are referenced. It will be easy to get around inside of a program to find the necessary other references, as most everything is connected. Additionally some references can be created from information.

Reflection is a complex topic, so in short, allows for information to be gathered about an Object. The long version would be, it is the part of .NET that allows for introspection of Objects, where do they come from, what are they made of, how can they be accessed, and what they will do. This is a key tool to understand for this attack but in order to cover this would take a paper in itself. *If you would like to find out more about reflection and how it works at the code leave check out my paper: Reflection's Hidden Power*

Background & Basics of .NET

The .NET framework is an open standard implemented on a number of different platforms[2]. .NET has the largest developer community ever, with a cross platform portability never before seen; surpassing the last generation of languages such as C++ and JAVA.

.NET run's on Intermediate Language(IL) code, this can be thought of as a meta language. Most people code in a wrapper language such as C#, VB.Net, MC++ and more are crated each year. The wrapper language allows programmers to wonk on a platform resembling the language they are accustom to.

IL is the code that runs inside of .NET, it is code generated from compilation and processing of a wrapper language. IL is a base set of commands that strongly resembles assembly code. Every command in a wrapper language is converted into it's component IL command(s). This is in turn converted at some later point on a users' computer to machine code for a specific hardware set.

.NET is a framework consisting of a Common Language Infrastructure(CLI) that houses the Common Language Runtime(CLR). The CLR is a virtual machine at the heart of .NET; it is commonly known as the .NET Runtime or just the Runtime. The Runtime is what most people think of as the core of .NET as it manages the Just-in-Time(JIT) compiler, threads, IO, garbage collector and more. The Runtime is predominantly what the attack in this paper is exploiting.

[2] .NET is supported on - Linux, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, Solaris, OS X, ARM, MIPS, IPhone, Nokia, Blackberry, Windows Mobile, Web and more.

References and Influences

James Devlin
www.codingthewheel.com

Sorin Serban
www.sorin.serbans.net/blog/

Erez Metula
paper: .NET reverse engineering
& .NET Framework Rootkits

Thanks to

L~~~~ A~~~~~~~
Thank you for the mentorship and training in forensics

D~~~~~~ D~~~~~~~
Thank you for the help on research and vulnerability analysis

A~~~~~~~ ------ K
Thank you for the advanced IT support

A~~- -~~~ (Redacted)
Thank you for the IT support; specifically networking and hardware