PROJECT 8 - Security Implications of Open CoreCLR (CoreRun.exe)

Completed on 05-Feb-2016 (26 days) -- Updated on 23-Nov-2016

Introduction >

Open .NET & CoreCLR

As far as Project 7 already includes my impressions about open .NET and my first participation there, visiting that project is precisely the best way to get some preliminary ideas. More specifically, the following (Introduction) sections:

- Open .NET Overview
- First Contributions
- General Impression

I came up with the idea of creating the current Project 8 after realising about the peculiarities associated with the compilation/validation of the CoreCLR code. Note that compiling this whole code (or one of its main parts, like *mscorlib.dll*) is not strictly required when working on it. For example: I didn't consider whole-CoreCLR-compilation aspects while performing all the relevant tests and drawing the main conclusions in Project 7.

In summary, this project will be dealing with CoreCLR by focusing on compilation aspects and closely-related issues (e.g., *CoreRun.exe* and its impact on security), rather than on specific pieces of code (what Project 7 is precisely about).

OVER-8-MONTH-LATER UPDATE

As explained in the also-updated Project 7, Microsoft has modified the way in which .NET programs can communicate with the CoreCLR libraries (and with the ones in other repositories, like CoreFX). This fact has a big impact on the current project, as far as its whole purpose was precisely analysing what is now an obsolete proceeding. Nevertheless and except for this clarification, I will keep it as originally written because of describing one of the preliminary steps within the evolution of a complex framework.

The most relevant ideas can be summarised in the following points:

- Originally, *CoreRun.exe* could be used with any .NET executable right away (e.g., in a folder including the required DLLs, *CoreRun.exe* and *MyApp.exe*, *MyApp.exe* could be run on the new libraries by typing "CoreRun.exe MyApp.exe" in the command prompt). In fact, I firstly thought about starting this project because of intuitively considering that such a proceeding was intrinsically unsafe.
- The (main) new approach relies on the .NET Core Runtime to build the source code being tested (this new proceeding is explained here). There is also a reference to a

CoreRun.exe-based alternative (described here), which differs from the old one. Additionally, the CoreFX tests rely internally on *CoreRun.exe*; what can be easily confirmed by running any of these tests from Visual Studio and seeing that it is the associated process. In any case, the old performance dropdown isn't occurring anymore.

- At the moment, *CoreRun.exe* cannot run other .NET programs as described in the current project. Note that even previously-running-fine backups aren't working now on the same computer.

Compilation & CoreRun.exe

As already explained in the first section and in various parts of the previous project, the quality of the code in CoreCLR (but also in CoreFX & Roslyn) is certainly high: good and clear structure, efficient algorithms, helpful comments, etc. Any experienced .NET programmer should be able to start working with it almost immediately. On the other hand, compiling a more or less relevant portion of the code is a different story.

Note that the .NET Framework is a huge and very complex project which has been systematically improved during the last quite a few years by a relevant number of different people; only this issue should be enough to not expect it to be immediately adapted to a so relevant change like "suddenly" becoming open source. But there is still another issue, the ultimate responsible for having created this project, which is much more relevant here: the bipartition underlying the whole .NET Framework and what it implies. That is, only part of the information required by a .NET executable is contained in the file itself; the remaining bits are taken from the .NET libraries present in the given computer (e.g., automatically installed with Windows). The CoreCLR repository includes the source code of some of these libraries (the most basic ones) and that's why validating the compilation outputs isn't easy. How telling the aforementioned executable to rely on the .NET libraries generated by the last compilation?

Building and using a new version of the CoreCLR libraries requires from a not-too-simple multi-step process which is explained in detail in the corresponding documentation. One of the outputs of this process is the file *CoreRun.exe*, which answers the aforementioned question (how telling the executable...?) and represents the main reason for having started this project. Basically speaking, *CoreRun.exe* allows to directly modify the pure essence of any .NET executable (i.e., telling it to rely on a modified version of some basic libraries) and this project analyses whether this fact represents a real security threat. people.

Usage of CoreRun.exe >

Minimum Requirements

This project comes from the assumption that *CoreRun.exe* can be a security threat because of allowing to easily modify .NET applications. It has to be noted that "easily" is a very important part of the previous sentence, equivalently to what happens in the definition of any other software security analysis. This point is specially worth noting here on account of the fact that the source code of *CoreRun.exe* has been made publicly available.

The aforementioned paragraph explains why this section and the next two ones, Windows Version and Visual Studio & .NET Version, are needed: the default constraints of *CoreRun.exe* (i.e., how easily the default version can be used under different scenarios) is a key part of the current analysis. Additionally, there are other issues to bear in mind in order to properly understand this project:

- Neither *CoreRun.exe* nor the Visual Studio files associated with it (e.g., *CoreRun.sln*) can be found in the CoreCLR repository, but just the C++ files containing its source code. The executable and the VS files are included among the outputs generated when building the whole CoreCLR code (i.e., running *build.cmd*); in Windows machines, these files are stored under *\bin\obj\Windows_NT.x64.Debug\src\coreclr\hosts\corerun\*.
- I did all my tests on Windows 8 & 10 (64-bit). That's why I am not completely sure whether the conclusions of this project are also applicable to other operating systems, although this seems the most likely scenario.
- The majority of this project has been focused on testing *CoreRun.exe* under different conditions, rather than on analysing its source code. Nevertheless, I did skim through the code and did perform slight modifications on it as a way to confirm certain assumptions. Doing a more detailed analysis of the code would have been against the aforementioned expectations of this project (i.e., *CoreRun.exe* being an immediate security threat).

Basic *CoreRun.exe* requirements:

- 64-bit Windows version. Its target platform is 64-bit, equivalently to what happens with all the CoreCLR code.
- Visual Studio installed. *CoreRun.exe* requires certain files which are present in the supported versions of VS.
- Using it with a .NET executable. For example: typing "CoreRun.exe MyExecutable.exe" in the command prompt, where *MyExecutable.exe* was built on a .NET language.
- Including all the required libraries in the same directory than *CoreRun.exe*; at least, *coreclr.dll* and *mscorlib.dll*. In this context, "required" means used by the given .NET executable. For example: if *MyExecutable.exe* relies on `Decimal.Parse`, the library including this method would have to be included (i.e., *mscorlib.dll*).

Windows Version

As explained in the first section, I got a quite good impression about the overall quality of the open .NET code (i.e., not just in CoreCLR, but also in CoreFX and Roslyn): very clear structure, efficient algorithms, descriptive comments, etc. All the parts analysed here (i.e., various methods in the file Number.cs) are certainly not an exception to this statement.

This project is focused on the optimisation of `ParseNumber` and associated resources. All its code is written in (almost-exclusively-)unmanaged C#, although C++ is also used in *Number.cs* even in parts which are closely related to `ParseNumber`. The following three different methods will be considered:

- `Boolean ParseNumber(ref char* str, NumberStyles options, StringBuilder sb, Boolean parseDecimal)`: it analyses the inputs to the given parsing method, like `Decimal.Parse` (i.e., string to be converted into number and arguments taking care of additional issues, like culture-related format); then outputs the numerical characters which will be later transformed into the given type (i.e., `decimal`) by `NumberBufferToDecimal`. All this code is unmanaged and highly optimised (e.g., relevant usage of bitwise operations). Most of the performance-improvement modifications of this project were precisely done to this method.
- `char* MatchChars(char* p, char* str)`: it is called from `ParseNumber`, while looping through the characters in the input string, and returns the position after the given target (`str`). There is a second overload (`char* MatchChars(char* p, string str)`) allowing the target to also be `string`.
- `Boolean IsWhite(char ch)`: very simple method checking whether the given character is blank. Anecdotally and unlikely the two aforementioned methods, this is managed code.

Ideally, this algorithm should be optimised by completely rethinking how the problem is being faced; specifically, the not-too-practical/efficient way in which the input information is stored provokes unnecessary problems. For example: over-complicated/less-efficient algorithms or over-usage of resources (e.g., most of potential inputs aren't analysed). Nevertheless, such an expectation is beyond the scope of the current project and its final goal (i.e., pull request to CoreCLR, whose likelihood to be accepted would be much lower in case of including modifications in existing features). The `decimal.TryParseExact` method which I am planning to create in the near future (likely to be part of Project 9) will certainly take care of these issues, also account for various functionalities not supported by the current parsing approaches.

Visual Studio & .NET Version

CoreRun.exe is not a .NET application, although its whole purpose is precisely dealing with .NET applications and that's why the installed .NET Framework is also relevant to it. The question is: can *CoreRun.exe* deal with any .NET Framework version? The answer seems to be yes, mainly on account of the traditionally-quite-reliable .NET backwards compatibility.

CoreRun.exe relies on certain Visual Studio files and that's why having this IDE installed is one of its requisites. I did some tests on a VS-free machine to see how easily the required conditions might be replicated without performing a whole installation. The results were quite discouraging (i.e., good from a safety point of view): firstly, clearly-defined errors which were quickly fixed after some research; then, the errors stopped appearing but the .NET executable didn't work as expected either. Thus, having a Visual Studio version installed doesn't seem replaceable; at least, not according to the "easy enough or no threat" basic assumption of this project.

Regarding the Visual Studio version, it is expressly stated that CoreCLR requires VS 2013 or 2015 to be installed. This somehow curious requirement seems to indicate that the CoreCLR code relies on files/features only present in VS 2013/2015. Such an assumption was confirmed when I tried to run *CoreRun.exe* on a computer where only VS 2012 was installed; I even recompiled the code with VS 2012 (by updating the "Platform Toolset" value for all the projects in the solution). The aforementioned situation was repeated: neither errors nor warnings, but the .NET executable didn't work as expected either. After installing VS 2015 on that computer, *CoreRun.exe* and the .NET executable started working fine.

In summary: *CoreRun.exe* can deal with .NET executables of any version (less safe), but only on computers where Visual Studio 2013 or newer is installed (safer).

Security Analysis >

Sample Scenario

CoreRun.exe can modify the behaviour of any .NET executable by forcing it to rely on an altered version of some of the libraries which define its pure essence. Thus, any potential threat scenario has to include, additionally to *CoreRun.exe*, an existing application built on a .NET language and (modified versions of) various CoreCRL libraries.

Let's imagine that employees at bank XYZ rely on the .NET program IJK to browse through the information of their clients. On their computers, they run a local copy of IJK which is connected to the central database. While dealing with client C, employee E runs IJK via *CoreRun.exe* together with a set of specially-modified CoreCLR libraries (i.e., by typing "CoreRun.exe ijk.exe" rather than the usual "ijk.exe"). This provokes IJK to run notably slower and to show slightly different screens and even errors; but nothing of this is noticed or seen as a problem by the client.

What kind of modifications can E do on IJK? Theoretically, he might change anything. Examples:

- Redefine basic types. Any action might be triggered by the constructor of any type, by consequently having full control on any variable of this type or on a specific subgroup of them (e.g., the ones whose names include the words "account" or "connection"). For example: all the values assigned to `string` variables might be written to an external file.
- Alter OS-related information. The values of any `System.IO` variable might be changed, what would seriously affect IJK's I/O system. For example: it might be forced to read from/write to specific files and locations.
- Affect the behaviour of basic elements of the programming language. Certain type of events (or even all of them) might be modified in any way. For example: a new window asking for a password might be triggered when clicking on a button with certain name.

The proposed example depicts the ideal conditions under which *CoreRun.exe* might be used to modify the behaviour of a .NET executable against its original purpose. A presumably trustworthy person (bank employee) using a program apparently as expected; and without the victim fully understanding or seeing the performed actions (errors or weird behaviours might occur). The required requisites and assumptions will be analysed in the next section.

Likelihood of Threat Occurrence

The conditions of the scenario proposed in the previous section are threat-favourable enough to allow general conclusions about the likelihood of a threat to occur at all. Different conditions might certainly provoke relevant changes on the relevance of some of the aspects below, although the final conclusions should remain unaltered.

The following requisites have to be met in order to allow the threat in the proposed scenario to become effective:

- As explained in some of the previous sections (i.e., Minimum Requirements, Windows Version and Visual Studio & .NET Version), *CoreRun.exe* cannot be run on any computer independently upon its configuration. Examples: supported version of Visual Studio installed and 64-bit OS.
- Additionally to meeting the aforementioned requirements, *CoreRun.exe* needs to access various files in OS-privileged locations (e.g., *Windows* directory when running on Windows). Hence, employee E would need to have administrative privileges on the given computer; this is considered an unsafe practice and, consequently, rarely happens in this kind of situations.
- Employee E (or his associates) needs to have a solid background in .NET, programming and IJK; ideally, he should also have a good understanding of IJK's source code.
- Employee E (or his associates) will have to perform a relevant number of tests to make sure that the IJK behaviour will be as normal in appearance as possible. Note that guessing the implications (i.e., without in-depth tests) of relying on modified CoreCLR libraries is virtually impossible when dealing with a more or less complex piece of software.
- All the aforementioned points might even be ignored on account of the tremendous difficulty associated with deciding the modifications to be performed. Theoretically, IJK might be completely changed; practically, even the slightest variation in the CoreCLR libraries might provoke a huge cascading effect which will be very difficult (or even impossible) to be fixed. For example: redefining `string` would not only affect all the variables of this type in the IJK source code, but also any used .NET functionality relying on this type at any point (i.e., virtually all the in-built functionalities would be affected).

In summary, using *CoreRun.exe* to modify the behaviour of a random .NET executable without having access to the source code is impractically complex. Even in case of having access to the source code, the associated difficulty would be very high. Thus, the conclusion of this section is that the likelihood of threat occurrence (i.e., using *CoreRun.exe* to make a .NET executable behave against its intended purpose) is virtually zero.

Possible Solutions

As explained in the previous section, a *CoreRun.exe*-based threat is very unlikely to happen due to the associated difficulties. No real threat means no problem and, consequently, no solution is strictly required.

Nevertheless, the following ideas can further improve *CoreRun.exe*'s security:

- Constraining its utilisation even more. For example: being able to only deal with executables built on a specific .NET version or to only be run on certain Windows version.
- Exclusively dealing with properly-identified .NET executables. They might be identified by forcing some of its defining features to meet specific conditions (e.g., including certain copyright reference or version).
- Rather than releasing a standalone application and its source code, relying on a cloud-based approach. It might consist in a simple webpage accepting the given inputs (i.e., .NET executable and modified libraries) and delivering the final result (i.e., what is output by this executable when relying on the modified libraries).

Conclusions

The first and most important conclusion of this project is that *CoreRun.exe* isn't a security threat. In the sense that there is little chance of someone using it to make a .NET executable perform unintended actions. I am not sure whether Microsoft paid special attention to these aspects or whether this is just a fortuitous consequence from a so complex reality; either way the final result seems good enough from a security point of view.

On the other hand, the chosen approach doesn't seem ideal and concluding its suitability to be a threat or not is certainly not trivial. Note that *CoreRun.exe* can modify the most essential parts of virtually any .NET application by just taking it as an argument (i.e., writing in the command prompt "CoreRun.exe NameOfFile.exe"); this seems a bit too excessive on account of the *CoreRun.exe*'s purpose (i.e., validating modifications in some .NET libraries). In the most likely scenario, programmers will create small applications to test their modifications on the CoreCLR libraries, rather than using random .NET executables (i.e., what *CoreRun.exe* unnecessarily allows).

CoreRun.exe delivers what is expected and doesn't represent a clear threat to the .NET security. On the other hand, it is a quite peculiar approach giving out much more than strictly required, what is rarely a good idea when dealing with software.