GitHub - TheWover/donut: Generates x86, x64, or AMD64+x86 position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters

By TheWover

Current version: v1.1

Table of contents

1. Introduction

Donut is a position-independent code that enables in-memory execution of VBScript, JScript, EXE, DLL files and dotNET assemblies. A module created by Donut can either be staged from a HTTP server or embedded directly in the loader itself. The module is optionally encrypted using the Chaskey block cipher and a 128-bit randomly generated key. After the file is loaded and executed in memory, the original reference is erased to deter memory scanners. The generator and loader support the following features:

- Compression of input files with aPLib and LZNT1, Xpress, Xpress Huffman via RtlCompressBuffer.
- Using entropy for API hashes and generation of strings.
- 128-bit symmetric encryption of files.
- Overwriting native PE headers.
- Storing native PEs in MEM_IMAGE memory.
- Patching Antimalware Scan Interface (AMSI) and Windows Lockdown Policy (WLDP).
- Patching Event Tracing for Windows (ETW).
- Patching command line for EXE files.
- Patching exit-related API to avoid termination of host process.
- Multiple output formats: C, Ruby, Python, PowerShell, Base64, C#, Hexadecimal, and UUID string.

There are dynamic and static libraries for both Linux and Windows that can be integrated into your own projects. There's also a python module which you can read more about in Building and using the Python extension.

2. How It Works

Donut contains individual loaders for each supported file type. For dotNET EXE/DLL assemblies, Donut uses the Unmanaged CLR Hosting API to load the Common Language Runtime. Once the CLR is loaded into the host process, a new Application Domain is created to allow for running Assemblies in disposable AppDomains. When the AppDomain is ready, the dotNET Assembly is loaded via the AppDomain.Load_3 method. Finally, the Entry Point for EXEs or public method for DLLs specified by the user is invoked with any additional parameters. Refer to MSDN for documentation on the Unmanaged CLR Hosting API. For a standalone example of a CLR Host, refer to code here.

VBScript and JScript files are executed using the IActiveScript interface. There's also minimal support for some of the methods provided by the Windows Script Host (wscript/cscript). For a standalone example, refer to code here. For a more detailed description, read: In-Memory Execution of JavaScript, VBScript, JScript and XSL

Unmanaged or native EXE/DLL files are executed using a custom PE loader with support for Delayed Imports, TLS and patching the command line. Only files with relocation information are supported. Read In-Memory Execution of DLL for more information.

The loader can disable AMSI and WLDP to help evade detection of malicious files executed in-memory. For more information, read How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code. It also supports decompression of files in memory using aPLib or the RtlDecompressBuffer API. Read Data Compression for more information.

As of v1.0, ETW is also bypassed. Like with AMSI/WLDP, this a modular system that allows you to swap out the default bypass with your own. The default bypass is derived from research by XPN. Read Hiding your .NET - ETW for more information.

By default, the loader will overwrite the PE headers of unmanaged PEs (from the base address to `IMAGE_OPTIONAL_HEADER.SizeOfHeaders`). If no decoy module is used (module overloading), then the PE headers will be zeroed. If a decoy module is used, the PE headers of the decoy module will be used to overwrite those of the payload module. This is to deter detection by comparing the PE headers of modules in memory with the file backing them on disk. The user may request that all PE headers be preserved in their original state. This is helpful for scenarios when the payload module needs to access its PE headers, such as when looking up embedded PE resources.

For a detailed walkthrough using the generator and how Donut affects tradecraft, read Donut - Injecting .NET Assemblies as Shellcode. For more information about the loader, read Loading .NET Assemblies From Memory.

Those who wish to know more about the internals should refer to Developer notes.

3. Building

There are two types of build. If you want to debug Donut, please refer to documentation here. If not, continue reading for the release build.

Clone

From a Windows command prompt or Linux terminal, clone the repository.

```
git clone http://github.com/thewover/donut.git
```

The next step depends on your operating system and what compiler you decide to use. Currently, the generator and loader template for Donut can be compiled successfully with both Microsoft Visual Studio 2019 and MingGW-64. To use the libraries in your own C/C++ project, please refer to the examples provided here.

Windows

To generate the loader template, dynamic library donut.dll, the static library donut.lib and the generator donut.exe. Start an x64 Microsoft Visual Studio Developer Command Prompt, change to the directory where you cloned the Donut repository and enter the following:

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

To do the same, except using MinGW-64 on Windows or Linux, change to the directory where you cloned the Donut repository and enter the following:

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

Linux

To generate the dynamic library donut.so, the static library donut.a and the generator donut. Change to the directory where you cloned the Donut repository and simply type make.

Python Module

Donut can be installed and used as a Python module. To install from source requires pip for Python3. First, ensure older versions of donut-shellcode are not installed by issuing the following command on Linux terminal or Microsoft Visual Studio command prompt.

```
pip3 uninstall donut-shellcode
```

After you confirm older versions are no longer installed, issue the following command.

```
pip3 install .
```

You may also install Donut as a Python module by grabbing it from the PyPi repository.

```
pip3 install donut-shellcode
```

For more information, please refer to Building and using the Python extension.

Docker

Building the docker container.

```
docker build -t donut .
```

Running donut.

```
docker run -it --rm -v "${PWD}:/workdir" donut -h
```

Support Tools

Donut includes several other executables that may be built separately. This include "hash.exe", "encrypt.exe","inject.exe", and "inject_local.exe". The first two are used in shellcode generation. The latter two are provided to assist with testing donut shellcode. "inject.exe" will inject a raw binary file (loader.bin) into a process by its PID or process name. "inject_local.exe" will inject a raw binary file into its own process.

To build these support executables separately you may use the MSVC makefile. For example, to build "inject_local.exe" to test your donut shellcode, you may run.

```
nmake inject_local -f Makefile.msvc
```

Releases

Tags have been provided for each release version of Donut that contain the compiled executables.

- v0.9.3, TBD
- v0.9.2, Bear Claw
- v0.9.1, Apple Fritter
- v0.9.0, Initial Release

Currently, there are two other generators available.

- C# generator by n1xbyte
- Go generator by awgh

4. Usage

The following table lists switches supported by the command line version of the generator.

Switch	Argument	Description
-a	*arch*	Target architecture for loader : 1=x86, 2=amd64, 3=x86+amd64(default).
-b	*level*	Behavior for bypassing AMSI/WLDP : 1=None, 2=Abort on fail, 3=Continue on fail.(default)
-k	*headers*	Preserve PE headers. 1=Overwrite (default), 2=Keep all
-j	*decoy*	Optional path of decoy module for Module Overloading.
-c	*class*	Optional class name. (required for .NET DLL) Can also include namespace: e.g *namespace.class*
-d	*name*	AppDomain name to create for .NET. If entropy is enabled, one will be generated randomly.
-e	*level*	Entropy level. 1=None, 2=Generate random names, 3=Generate random names + use symmetric encryption (default)
-f	*format*	The output format of loader saved to file. 1=Binary (default), 2=Base64, 3=C, 4=Ruby, 5=Python, 6=PowerShell, 7=C#, 8=Hexadecimal
-m	*name*	Optional method or function for DLL. (a method is required for .NET DLL)

-n	*name*	Module name for HTTP staging. If entropy is enabled, one is generated randomly.
-o	*path*	Specifies where Donut should save the loader. Default is "loader.bin" in the current directory.
-p	*parameters*	Optional parameters/command line inside quotations for DLL method/function or EXE.
-r	*version*	CLR runtime version. MetaHeader used by default or v4.0.30319 if none available.
-s	*server*	URL for the HTTP server that will host a Donut module. Credentials may be provided in the following format: `https://username:password@192.168.0.1/`
-t		Run the entrypoint of an unmanaged/native EXE as a thread and wait for thread to end.
-w		Command line is passed to unmanaged DLL function in UNICODE format. (default is ANSI)
-x	*option*	Determines how the loader should exit. 1=exit thread (default), 2=exit process, 3=Do not exit or cleanup and block indefinitely
-y	*addr*	Creates a new thread for the loader and continues execution at an address that is an offset relative to the host process's executable. The value provided is the offset. This option supports loaders that wish to resume execution of the host process after donut completes execution.
-z	*engine*	Pack/Compress the input file. 1=None, 2=aPLib, 3=LZNT1, 4=Xpress, 5=Xpress Huffman. Currently, the last three are only supported on Windows.

Payload Requirements

There are some specific requirements that your payload must meet in order for Donut to successfully load it.

.NET Assemblies

- The entry point method must only take strings as arguments, or take no arguments.
- The entry point method must be marked as public and static.
- The class containing the entry point method must be marked as public.
- The Assembly must NOT be a Mixed Assembly (contain both managed and native code).
- As such, the Assembly must NOT contain any Unmanaged Exports.

Native EXE/DLL

- Binaries built with Cygwin are unsupported.

Cygwin executables use initialization routines that expect the host process to be running from disk. If executing from memory, the host process will likely crash.

Unmanaged DLLs

- A user-specified entry point method must only take a string as an argument, or take no arguments. We have provided an example.

5. Subprojects

There are four companion projects provided with donut:

Tool	Description
DemoCreateProcess	A sample .NET Assembly to use in testing. Takes two command-line parameters that each specify a program to execute.
DonutTest	A simple C# shellcode injector to use in testing donut. The shellcode must be base64 encoded and copied in as a string.
ModuleMonitor	A proof-of-concept tool that detects CLR injection as it is done by tools such as Donut and Cobalt Strike's execute-assembly.
ProcessManager	A Process Discovery tool that offensive operators may use to determine what to inject into and defensive operators may use to determine what is running, what properties those processes have, and whether or not they have the CLR loaded.

6. Developing with Donut

You may want to add support for more types of payloads, change our feature set, or integrate Donut into your existing tooling. We have provided developer documentation. Additional features are left as exercises to the reader. Our suggestions:

- Add environmental keying.
- Make Donut polymorphic by obfuscating the loader every time shellcode is generated.
- Integrate Donut as a module into your favorite RAT/C2 Framework.

7. Questions and Discussion

If you have any questions or comments about Donut. Join the #Donut channel in the BloodHound Gang Slack

8. Disclaimer

We are not responsible for any misuse of this software or technique. Donut is provided as a demonstration of CLR Injection and in-memory loading through shellcode in order to provide red teamers a way to emulate adversaries and defenders a frame of reference for building analytics and mitigations. This inevitably runs the risk of malware authors and threat actors misusing it. However, we believe that the net benefit outweighs the risk. Hopefully that is correct. In the event EDR or AV products are capable of detecting Donut via signatures or behavioral patterns, we will not update Donut to counter signatures or detection methods. To avoid being offended, please do not ask.

Source: https://github.com/TheWover/donut