EXPLOIT GENERATION AUTOMATION WITH WINDBG

White Paper

Csaba Fitzl
fitzl.csaba@gmail.com

Table of Contents

1.0 Introduction

This section gives a high lever overview of the tool, and the motivation behind creating it.

1.1 Motivation

The motivations behind creating this simple tool were to show how powerful is to script WinDBG and to simplify the typical BoF (buffer overflow) exploit development process. As the method is basically the same every time, a lot of manual work can be saved with automating the process. There are plenty of helper scripts (far the best and well known is mona.py), but none of them does the entire job for you.

1.2 Typical BoF exploit writing steps

The typical steps to create a BoF exploit are:

1. Find EIP overwrite location (offset in the buffer)
2. Examine memory layout, registers
3. Somehow jump to the buffer with the help of registers
4. Find bad characters
5. Generate shellcode
6. Put it all together

These are the classic steps, and altough there are corner cases, most of them works this way. This is a heavily manual process especially the part, when you start the debugger, attach the process, run your PoC code, discover memory layout, or debug errors you made, and then start again. When it comes to discovering bad characters, that is especially time intense.

1.3 Overview of the tool

The goal with creating this script/tool was to automate as much as I can, so no manual interaction is needed to create a working exploit from a crash PoC.

The script is written entirely in Python and uses the pykd library to intercat with WinDBG. The library is specific to this debugger, so others can't be used.

It is still in very early development stage, so there might be bugs, especially if interaction with the application is needed, and not all cases are covered.

1.3.1 What can the tool do as of today?

The tool currently can automate the process of creating classic BoF exploits from crash PoCs. The PoC has to be written in specific format (see later), so the script can interact with it, but if it's done, it can do the job. It will start the application, WinDBG, attach the process, etc...

As a first step it will run the crash PoC to determine the offset on the buffer, which overwrites EIP, and then discover the memory layout, how much space we have, which registers points to the buffer. Then it will start finding bad characters, which can't be used in the exploit. The next step is to find out how to jump to the actual buffer with the help of registers identified before, it will try to find various assembly instructions in memory, which can do this for us. If needed we can tell the tool to search only non-ASLR enabled modules, so essentially it can also bypass ASLR with this method.

Once all of these are done, it will call metasploit to generate a calc.exe shellcode, and put the entire buffer together, and launch a working exploit.

I differentiated two types of applications:

1. Network communication based
2. File based

In the 2nd case the file with the exploit has to be generated first, before launching the application, the tool can cover both types.

2.0 Setting up the environment

1. Install Debugger Tools for Windows from http://msdn.microsoft.com/en-US/windows/hardware/hh852363
2. Install an older version of Python (e.g.: 2.7.3), download from: https://www.python.org/downloads/windows/
3. Download latest 3.x version of pykd from https://pykd.codeplex.com/releases/view/614442
4. Extract the zip file contents
5. Copy the pykd.pyd file to "C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86\winext"

2.1 Verify Installation

1. Launch a custom x86 application
2. Attach the debugger
3. Type: ".load pykd.pyd", you shouldn’t get any errors showing up
    a. In case WinDBG terminates, try an older version of Python
4. Type "r" to get the values stored in registers
5. Start Python in WinDBG, type: "!py"
6. Type: "hex(reg('eip'))", you should get the same value for EIP, what you saw at step #4

3.0 Using the autoexp.py script

3.1 The exploit file

This class defines the exploit itself, and has functions to modify/interact with the exploit code during development. It has functions to modify the buffer, and get info from it. The class .py file has a wrapper function defintion, which will allow the exploit to run in asyn way:

```
def run_async(func):
    @wraps(func)
    def async_func(*args, **kwargs):
            func_hl = Thread(target = func, args = args, kwargs = kwargs)
            func_hl.start()
            return func_hl

    return async_func
```

3.2 Populate the exploit class with PoC/crash info

The exploit code has to be migrated to Python, and defined in the "exploit()" function. It will have one variable currently, which is the buffer itself, which overwrites the stack. This buffer will be modified by the autoexp.py script. An example:

```
    def exploit(self):
            """
            This function runs the actual exploit
            """
            sleep(1)
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.connect(('127.0.0.1',80))
            #Test 1
            message = "GET " + ''.join(self.buffer) + " HTTP/1.1\r\n\r\n"
            sock.send(message)
            sock.close()
```

In general we have to insert the code snippet, where we would put our buffer:

```
''.join(self.buffer)
```

The reason for the join is that the script will build the malicious code, from multiple parts, each of them will be stored in a string, and overall it will be in an array.

Additionally we need to set the initial buffer, along with a marker (egg), as an array:

```
self.buffer = [self.egg*2, "A"*2992]
```

The default egg is 4 byte long (EGGG), so from the crash PoC buffer length substract 8, and put that into the second part.

3.3 Additional info to populate

You need to define the program path:

```
self.command = 'C:\\Program Files (x86)\\Easy RM to MP3 Converter\\RM2MP3Converter.exe /cf'
```

If it’s an application, where the exploit will be in a file, you need to set the following variables as well:

```
self.filename = 'c:\\autoexploit\\list.m3u'
self.file_based = True
```

3.4 Saving the exploit

Once the exploit is generated we want to save it for future use. In case of a file based exploit we don’t need to do anything since the file is already generated, however in other cases we will want to save it. This is basically writing out the python script to a file with unescaping the characters in the buffer. Like this:

```
    def save(self):
            f = open('minishare_exploit.py','w')
            f.write("import socket\n\n" +
                            "sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)\n" +
                            "sock.connect(('127.0.0.1',80))\n" +
                            "message = 'GET " + ''.join('\\x%02x' % ord(c) for c in ''.join(self.buffer)) + " HTTP/1.1\\r\
\n\\r\\n'\n" +
                            "sock.send(message)\n" +
                            "sock.close()\n")
            f.close()
```

In file based versions we can keep it simple:

```
    def save(self):
            pass
```

3.5 Running the autoexp.py script

These are the script options:

```
c:\autoexploit>autoexp.py -h
usage: autoexp.py [-h] -e EXPLOIT [-a] [-t TYPE]

Automated exploit generation

optional arguments:
  -h, --help            show this help message and exit
  -e EXPLOIT, --exploit EXPLOIT
                        exploit class file
  -a, --aslr            Try to use modules w/o ASLR enabled
  -t TYPE, --type TYPE  Exploit type (bof,...)
```

The type is currently ignored, it’s added for future.