Beyond automated tools and Frameworks: the shellcode injection process

Author: Dr Craig S Wright GSE GSM LLM MStat

Abstract / Lead

This article is going to follow from previous articles as well as going into some of the fundamentals that you will need in order to understand the shellcode creation process, how to use Python as a launch platform for your shellcode and that the various system components are. This is designed as a precursor to the actual injection process where we will in a later article next month introduce the actual injection process and start to move from automatic exploit frameworks (such as Metasploit) into being able to create and execute one's own exploit successfully. In order to do this, we need to start understanding just how code works and to know where to find the fundamentals of the Python programming language. This article will start a monthly series designed to take the reader from a novice to being able to create and deploy their own shellcode and exploits.

Electronic copy available at: https://ssrn.com/abstract=3153488

Introduction

Automated frameworks (including Metasploit) have simplified the testing and exploitation process. This of course comes with a price. Many penetration testers have become tool jockeys with little understanding of just how software functions. This script kiddie approach to code testing does have its place. It has allowed us to drastically increase the number of people working on testing systems for vulnerabilities and in assessing the risks these pose. At the same time, if these individuals do not progress further, simply relying on the ability to leverage the efforts of others, we will hit bottlenecks in the creation of new tests and processes.

In previous articles, we have covered a number of topics to do with the creation of shellcode. In this one we shall start to introduce the means that will allow the tester to use that code without having to rely on an external framework. In subsequent articles this will be expanded into the creation of standalone exploit kits.

In order to do this, we also need to take a step back and explain the system and the tools we will use in more detail. To achieve this, we will start with describing the various components that are used and to providing an introduction to the Python programming language. This will also extend into a simple method to analyse shellcode using GCC such that we can come to understand what the shellcode others have created is designed to do. This is a useful skill when reversing malware as well as a good way to learn from the existing code base and even to leverage some of the various tools that are freely available already.

What is shellcode again?

First and foremost, shellcode is an essential part of any exploit. This does not really inform us as to what it is or how it functions. Simply put, shellcode is an array of hex values that are arranged in such a way as to act as a set of assembly language instructions in order to make the machine it is run on do some function. This function can be to spawn a shell or to execute arbitrary code of the testers design. It is a fundamental part of any buffer overflow attack, but that is not all it is used for.

Just about any exploit that you may discover in the wild or as a part of a framework will target vulnerabilities using shellcode. This can be as the payload of a buffer overflow attack, a heap spray or other form of injection.

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
"\x42\x42\x42\x42";
```

Code 1: Shellcode sample[1]

We will start by analysing this small segment of shellcode (Code 1). To do this, we will start by compiling this shellcode source into an executable file.

[1] This sample of shellcode has been taken from Zillion (2002). This page goes into detail as to the operation of the shellcode and the reader is encouraged to step through this. The reader will find countless many examples online with a simple Google search and many good examples are also included within the Metasploit framework.

Electronic copy available at: https://ssrn.com/abstract=3153488

Disassembling Shellcode

We will start by first compiling and then disassembling the shellcode sample we have introduced above (Code 1). The debugger ***gdb, objdump*** or ***nasm***[2] work well for this task. To do this, we take the shellcode from above and paste it into a file using ***vi***. That is, "*vi shellcode.c*" as can be seen in figure 1.

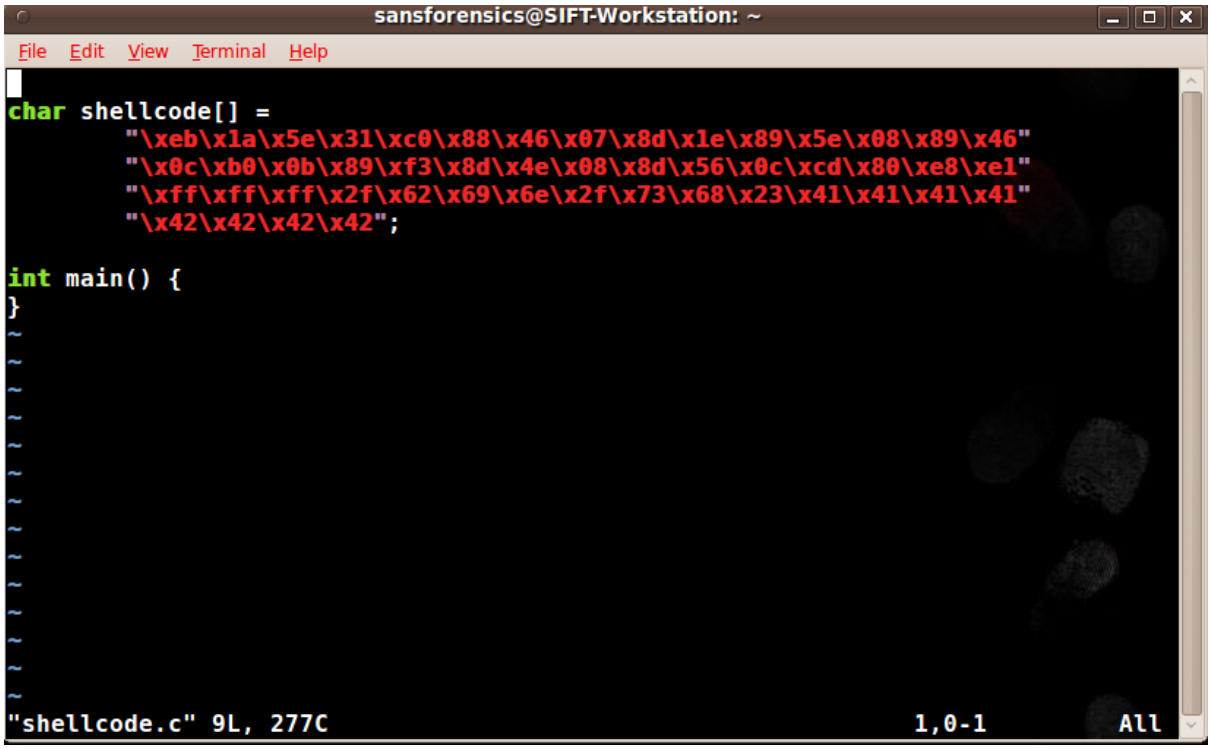

Figure 1: Making our shellcode executable, "vi shellcode.c".

We have added a ***main()*** statement and placed our shellcode sample into a character array to aid in compiling it. We need to compile the sample so that we can load it into a disassembler or debugger. These tools will interpret the hex instructions for us and return a series of assembly instructions for us to analyse. This C "*skeleton program*" can be compiled into a binary executable using gcc (see Code 2).

```
sudo gcc -g -o shellcode shellcode.c
```

Code 2: Using GCC to compile the shellcode sample

We can see in figure 2 that we have created an executable file called "shellcode". I have used the SIFT binary image for this analysis, but there are many good pre-configured analysis images based on Linux that are freely available. In addition, using one of the standard Linux distributions and installing tools such as NASM will enable you to create a shellcode analysis system very easily.

The sample of shellcode used in this article is by no means the simplest, but it is well known. Many examples of shellcode will actually be far smaller than this, but some can also be quite complex at the same time.

---

[2] OllyDby, Immunity and IDA Pro all work well when analysing shellcode on a Windows host.

```
sansforensics@SIFT-Workstation:~$ vi shellcode.c
sansforensics@SIFT-Workstation:~$ sudo gcc -g -o shellcode shellcode.c
sansforensics@SIFT-Workstation:~$ ls -l shellcode
-rwxr-xr-x 1 root root 9012 2012-02-06 22:13 shellcode
sansforensics@SIFT-Workstation:~$
```

Figure 2: Our compiled executable, "shellcode".

Loading the compiled code into GDB is simple. Just execute the command, "***gdb shellcode***". Once it is loaded into GDB, we just have to use the gdb disassemble command with the "shellcode[]" array we created as the object it will analyse as gdb will interpret the contents of the shellcode array as if it was assembly code. This is displayed in figure 3.

```
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/sansforensics/shellcode...done.
(gdb) disassemble shellcode
Dump of assembler code for function shellcode:
0x0804a040 <shellcode+0>:       jmp    0x804a05c <shellcode+28>
0x0804a042 <shellcode+2>:       pop    %esi
0x0804a043 <shellcode+3>:       xor    %eax,%eax
0x0804a045 <shellcode+5>:       mov    %al,0x7(%esi)
0x0804a048 <shellcode+8>:       lea    (%esi),%ebx
0x0804a04a <shellcode+10>:      mov    %ebx,0x8(%esi)
0x0804a04d <shellcode+13>:      mov    %eax,0xc(%esi)
0x0804a050 <shellcode+16>:      mov    $0xb,%al
0x0804a052 <shellcode+18>:      mov    %esi,%ebx
0x0804a054 <shellcode+20>:      lea    0x8(%esi),%ecx
0x0804a057 <shellcode+23>:      lea    0xc(%esi),%edx
0x0804a05a <shellcode+26>:      int    $0x80
0x0804a05c <shellcode+28>:      call   0x804a042 <shellcode+2>
0x0804a061 <shellcode+33>:      das
0x0804a062 <shellcode+34>:      bound  %ebp,0x6e(%ecx)
0x0804a065 <shellcode+37>:      das
0x0804a066 <shellcode+38>:      jae    0x804a0d0
0x0804a068 <shellcode+40>:      and    0x41(%ecx),%eax
```

GBD to disassemble the code

Figure 3: Our disassembled "shellcode".

Altering this shellcode sample can be as simple as changing the shell being called. In "Code 1", we can change the call to spawn "/bin/sh" to something such as a call to "/bin/ksh" by changing the hex entry, "`\x2f\x62\x69\x6e\x2f\x73\x68`" to "`x2f\x62\x69\x6b\x6e\x2f\x73\x68`".

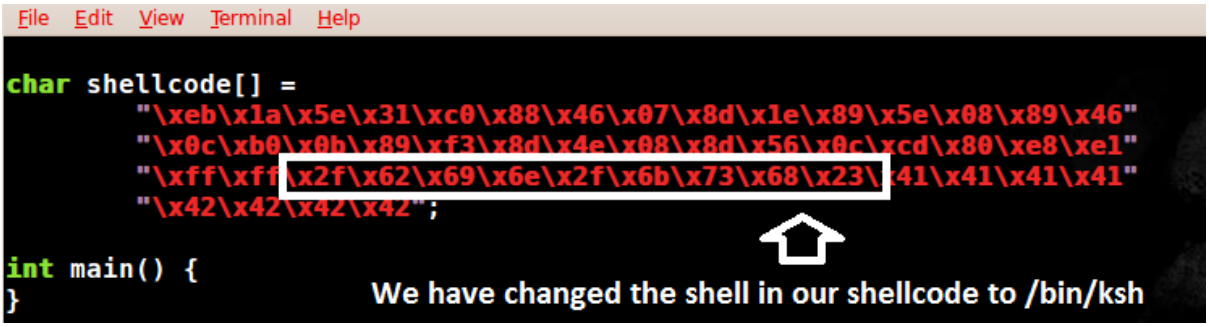

Figure 4: Altering the "shellcode".

With this simple change, we have altered the shellcode in a manner that allows us to control its objective.

What if I use Windows?

For those of us who are more aligned to using Windows, we can load and disassemble our shellcode in the same way using IDA Pro or OllyDbg. Starting with a tool such as "***Shellcode2Exe.py***" (Zeltser, 2011) or "***ConvertShellcode***" we can convert our shellcode sample into an executable in Windows and load it into IDA Pro.

Loading the shellcode into IDA Pro (Fig 4) we see a visually simplified view of the same disassembly we completed in GDB (Fig 3). This display can be of use to people who are less familiar with coding as it displays the jumps (and conditional jumps) in a more structured manner.

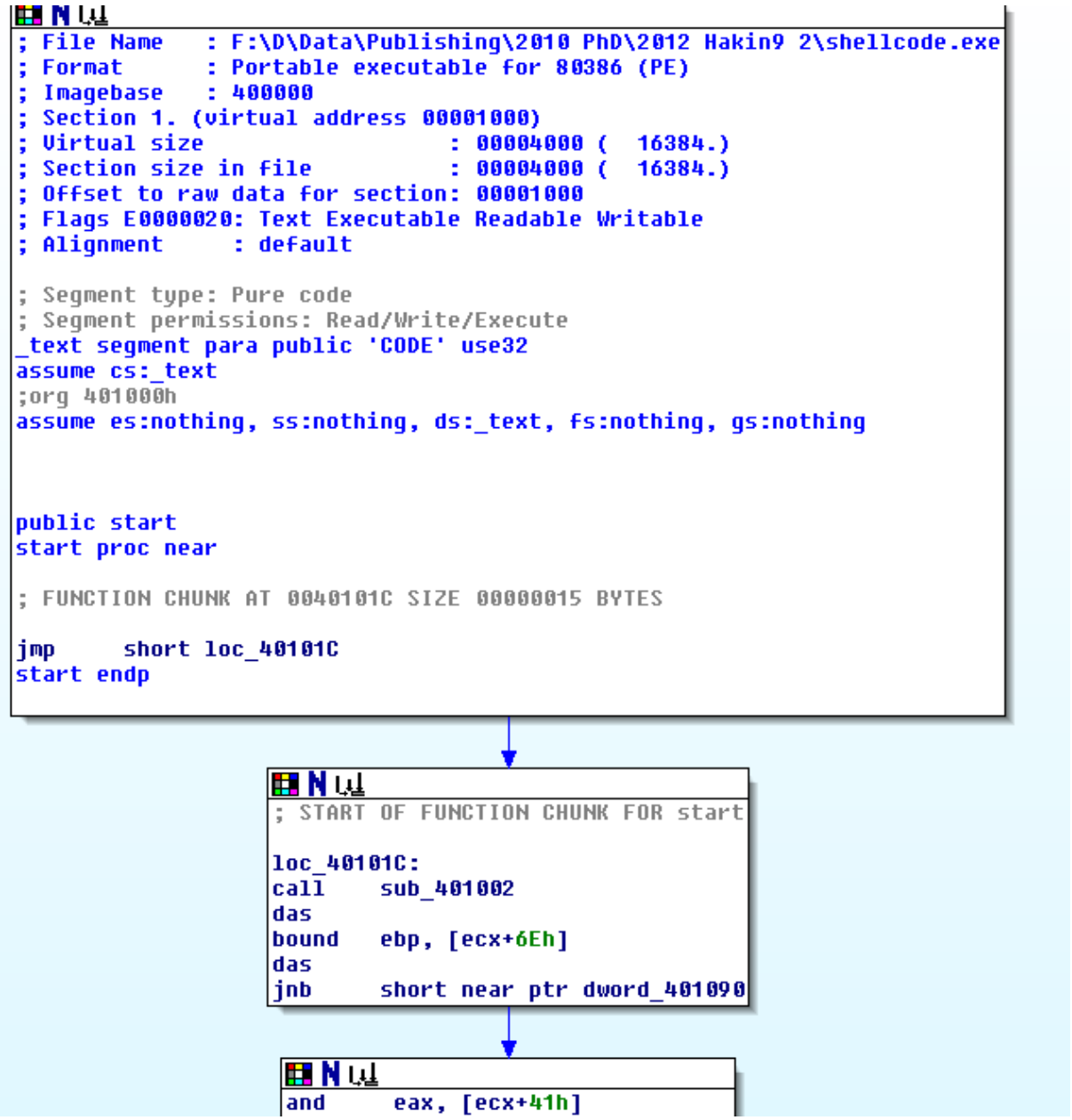

Figure 5: IDA disassembled "shellcode".

Now we have to start learning to interpret what these assembly instructions actually mean.

Assembly Instructions

In the last article[3] we started to detail what a system register is. We need to extend this and to start to look at other parts of the system. To understand assembly processing, we need to understand the concept of LIFO or Last in First out. Think a stack of dishes when you think of the stack in memory. Just as you do not try to take a disk from the base of a large stack (well not without consequences), stack calls pull the last item entered and return that moving one by one to the previously added entries until the first to have been added to the stack is removed as the last item.

[3] "Starting to write your own shellcode" in Hakin9, Jan 23 2012.

In particular, when working with the stack, the two assembly commands that you need to have a strong understanding of are *push* and *pop*.

- "PUSH" saves the contents of a register onto the stack.
- "POP" grabs the saved contents from the stack and puts it into a specific register.

In this, we can think of the stack as the endless tape in a turning machine[4]. As we move along the stack we are adding and removing values we have saved previously.

A good and quick introduction to assembly coding is available in "*Assembler: The Basics In Reversing*"[5]. In this you will find a small but critical list of instructions that you will need to know when you start to create shellcode. It includes:

- ADD (addition)
- AND (logical AND)
- CMP (compare)
- TEST (test two values)
- DEC (decrement)
- DIV (division)
- INC (increment)
- INT (interrupt handler)
- LEA (load effective address)
- MOV (move)
- MUL (multiplication)
- NOP (no operation)
- OR logical OR)
- POP (See above in the stack section)
- PUSH (See above in the stack section)
- XOR (exclusive OR)

From here we start to move to conditional statements and jumps. That is, we can branch to different functions based on the input or other values we have in the registers and flags. Changing the execution path updates the value held in the EIP register. This is used to add jumps, calls and loops to your code. The main jump statements and their uses are displayed in *Table 1*.

Jumps are enacted unconditionally (JMP, CALL, RET) when the code always branches to another location in memory. These are fixed controls that always behave in a set manner if the code reaches a point where it will execute them.

A conditional jump (JCC, JE, JZ etc.) are used to control execution branches (See Table 1). These forms of jumps are used to branch into different execution paths based on the values retuned to the jump statement. That is, these control the branching of a code segment based on whether certain conditions have been fulfilled. Often, these conditionals are based on the returned value in a Boolean statement (see Table 2).

Most conditionals actually change the values held in a destination operand or register. There are some that do not (CMP, TEST). CMP is an implied SUB and TEST an implied ADD function.

---

[4] Here we are of course ignoring the necessary memory and processor constraints.
[5] This paper is currently available online at http://flip-edesign.com/basics_of_assembler.pdf

Jump	Action	Flags
JA	Jump if (unsigned) above	CF = 0 and ZF = 0
JB	Jump if (unsigned) below	CF = 1
JC	Jump if carry flag set	CF = 1
JE	Jump if equal	ZF = 1
JG	Jump if (signed) greater than	ZF = 0 and SF=OF
JGE	Jump if (signed) greater than or equal to	SF = OF
JL	Jump if (signed) less than	SF != OF
JLE	Jump if (signed) less than or equal to	ZF = 1 and SF != OF
JMP	Jump	Always jumps
JNE	Jump if not equal	ZF = 0
JNZ	Jump if not zero	ZF = 0
JZ	Jump if zero	ZF = 1

Table 1: Jcc Assembly Jumps

The effects of a conditional instruction are stored in the *flags* register. For instance, in "*Code 3*", the statement "*Cmp EBX EAX*" will not result in the ZF bit (zero flag in the flags register) being set as EBX and EAX are not equal. In this operation, the values stored in both EAX and EBX will not change and will remain the same after the operation as before.

```
…
1       Mov     EAX     04
2       Mov     EBX     10
3       Mov     ECX     04
4       Cmp     EBX     EAX
5       JZ      EDX
6       Cmp     EBX     EBX
7       JZ      EDX
8       Sub     EAX     ECX
9       JZ      EDX
…
```

Code 3: Testing conditionals

The statement "*Cmp EBX EBX*" will result in the ZF bit (zero flag in the flags register) being set as EBX – EBX would equal zero (that is they are the same value). Conversely, statement "*Cmp EAX ECX*" will result in ZF being set, but it will also result in the EAX register being modified.

In this example (Code 3), with the values set in lines 1 to 3, the statement at line 4 will not set ZF and as such the conditional at line 5 will not execute. The statement at line 6 will set the ZF flag and the jump at line 7 would take the execution to the value held in the EDX register. Hence, the test at line 8

and the conditional statement at line 9 will only execute if EDX contained the value referencing the memory location for line 8[6].

Logical Operation	Source	Destination	Result
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
NOT	0	N/A	1
	1	N/A	0

Table 2: Logical Operations

The TEST instruction is similar to the CMP instruction in that it does not change the values stored in the registers but acts as an AND instruction (Table 2).

In the next article we will continue this process by converting "If-Else" and "If-Else-If-Else" loops to and from assembly code. In this, we will take C/C++ statements and write them as assembly code and vice versa. This will continue into examining loops and stack functions.

Loops are important parts of any program code and in injected code are regularly deployed by attackers in order to:

- Encrypt and decrypt data and network traffic,
- Log keystrokes while checking for selected key codes,
- Port scan or ping sweep systems incrementally, and
- Connect to a C&C server by scrolling through a list of available systems.

Using Python to deliver shellcode

In next article we will start to use python to deliver your shellcode. Before you do this, you will need to have some experience with this scripting language. To do this, you need to start with a quick introduction to learning more about Python if you do not already have this skill.

```
python.exe <script_name.py> 127.0.0.1 80
```

Code 4: Shellcode sample[7]

[6] Interestingly, if the register EDX was to contain the memory location for statements 5, 7 or 9 this would result in an endless execution loop and a possible DoS against the system as the program looped endlessly to the same call.

[7] This sample of shellcode has been taken from Zillion (2002). This page goes into detail as to the operation of the shellcode and the reader is encouraged to step through this. The reader will find countless many examples online with a simple Google search and many good examples are also included within the Metasploit framework.

Python is a script interpreter and interactive interpreter at the same time allowing developers to either run and test code samples or to create executable scripts. Once we have found a site, we can use Python to simply deploy our exploit and shellcode. We see in "Code 4" just how simple it is to deploy an exploit script written in Python against a web server running on the local host. What we need to do is learn just how to write this script so we can deploy the shellcode we are learning to both write and to alter.

Learning Python

There are many good resources available for those who want to learn Python. Some of the ones I have found to really work are included below:

- Google Python Class http://www.youtube.com/watch?v=tKTZoB2Vjuk
- PythonLearn http://www.py4inf.com/
- The Python Tutorial http://docs.python.org/tutorial/
- Beginner's Guide to Python http://wiki.python.org/moin/BeginnersGuide
- Interactive Python tutorial http://www.learnpython.org/

Once you have the basic skills, we can take these and use them in developing deployment platforms. With these skills, you will find yourself able to do far more than use a framework.

Conclusion

Gaining a comprehensive understanding of shellcode takes time, but it is well worth the effort. In following articles, we will extend this process further as we start to detail how we can follow shellcode and interpret the jumps and calls made within it. In doing this, we will not only learn how to successfully modify the shellcode we have copied from others, extending its use, but to also learn to create our own. More, we will be able to reverse engineer hostile shellcode and to understand what purposes it has been created for.

Shellcode can be said to have a shelf life. As samples become popular and are used more widely in the underground community, they are slowly added into IDS and Anti-Malware signatures. Widely deployed shellcode, including that used in the Metasploit project, has a particularly low shelf-life. This is not to say that it will not be useful against many sites, but that it will be less likely to have value in testing highly secure sites.

In many cases, the alteration of small sections of the shellcode can result in the signatures that have been created to detect, alert and block it becoming ineffective. For instance, making small changes to a piece of existing shellcode to run "***/bin/csh***" in place of the standard call to "***/bin/sh***" can increase the useful life of the shellcode. As we start to learn how shellcode is created and formed, we can also start to alter it and extend it running different payloads or changing its form to avoid detection.

Python is a scripting language with a strong support for shellcode. A tester with a good understanding of Python and shellcode will be able to create their own exploit packages. This increases the value of the tester immensely. Instead of being one of many people who have the capability to run Metasploit or some other automated framework, you become one of a select few people who can actually create their own tools and exploits.

Anybody who desires to become an advanced pen-tester will find it essential to gain proficiency in at least one scripting language. This of course will require dedicated effort as there is no replacement for real world use in learning to code in any language (even a scripted language such as Python).

Python is the language of choice for many penetration testers and there is a wide community within the information security profession developing and extending it. For this reason, Python should be a high priority on any testers to learn list. There are innumerable examples of Python scripts, tools and methodologies to enhance your own tools readily available with the ease of a Google search. More, there are free tutorials and interactive lessons.

References

Foster, J., Osipov, V., Bhalla, N., and Heinen, N. (2005) "*Buffer Overflow Attacks: Detect, Exploit, Prevent*" Syngress, USA

Zeltser, L (2011) "http://zeltser.com/reverse-malware/convert-shellcode.html" Online at: http://zeltser.com/reverse-malware/convert-shellcode.html

zillion (2002) "Writing Shellcode" (safemode.org) Online at: http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html

Author's bio

About the Author:

Craig Wright (Charles Sturt University)is the VP of GICSR in Australia. He holds the GSE, GSE-Malware and GSE-Compliance certifications from GIAC. He is a perpetual student with numerous post graduate degrees including an LLM specializing in international commercial law and ecommerce law, A Masters Degree in mathematical statistics from Newcastle as well as working on his 4th IT focused Masters degree (Masters in System Development) from Charles Stuart University where he lectures subjects in a Masters degree in digital forensics. He is writing his second doctorate, a PhD on the quantification of information system risk at CSU.