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:

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

2. **Metasploit - Pattern offset** script to determine the required number of bytes for the EIP to be overwritten.
   
   The syntax is: `pattern_offset.rb <specify the memory address of EIP at the state of panic>`

[http://searchsecurity.techtarget.in/tutorial/Exploit-writing-tutorial-Part-1](http://searchsecurity.techtarget.in/tutorial/Exploit-writing-tutorial-Part-1)
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.

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

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

http://searchsecurity.techtarget.in/tutorial/Exploit-writing-tutorial-Part-1
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](http://www.epsilonlambda.wordpress.com)

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

More Tutorials

[http://searchsecurity.techtarget.in/tutorial/Exploit-writing-tutorial-Part-1](http://searchsecurity.techtarget.in/tutorial/Exploit-writing-tutorial-Part-1)
• **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**