Buffer Overflows 3

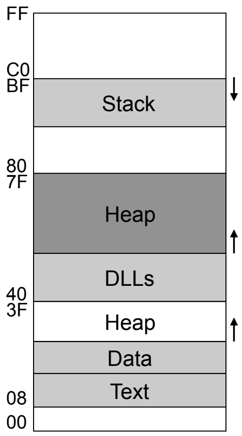
Heap Overflows

and

Defenses

Heap Buffer Overflow

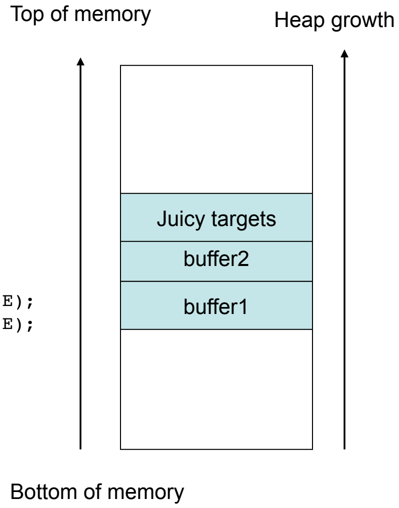
- Global variables
- Static variables
- Dynamically allocated memory
  - malloc()



Heap Operation

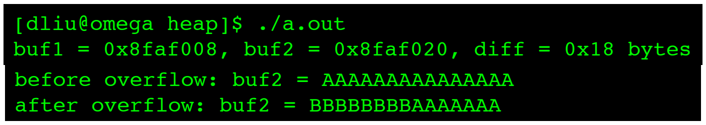
- Opposite of stack

```
#define BUFSIZE 16
int main()
   {
      u_long diff;
      char *buf1 = (char *)malloc(BUFSIZE);
      char *buf2 = (char *)malloc(BUFSIZE);
      // declare and allocate mem for
      // juicy targets here
```



Ex. Vulnerable Program I

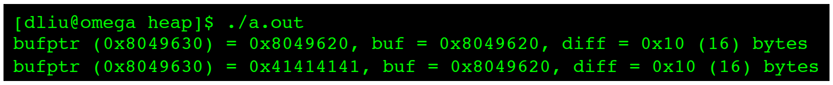
```
#define BUFSIZE 16
 #define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */
 int main()
 {
    u_long diff;
    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);

    diff = (u_long)buf2 - (u_long)buf1;
    printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);

    memset(buf2, 'A', BUFSIZE-1);
    buf2[BUFSIZE-1] = '\0';

    printf("before overflow: buf2 = %s\n", buf2);
    memset(buf1, 'B', (u_int)(diff + OVERSIZE));
    printf("after overflow: buf2 = %s\n", buf2);
    return 0;
 }
```



```
[dliu@omega heap]$ ./a.out
buf1 = 0x8faf008, buf2 = 0x8faf020, diff = 0x18 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow: buf2 = BBBBBBBBAAAAAAA
```

Ex. Vulnerable Program 2

```
#define BUFSIZE 16
#define ADDRLEN 4 /* # of bytes in an address */

  int main()
  {
     u_long diff;
     static char buf[BUFSIZE], *bufptr;

     bufptr = buf, diff = (u_long)&bufptr - (u_long)buf;

     printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
            &bufptr, bufptr, buf, diff, diff);

     memset(buf, 'A', (u_int)(diff + ADDRLEN));

     printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
            &bufptr, bufptr, buf, diff, diff);

     return 0;
  }
```

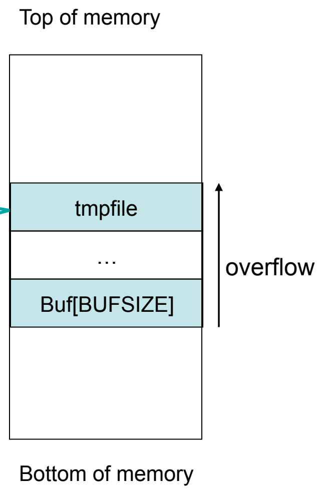


```
[dliu@omega heap]$ ./a.out
bufptr (0x8049630) = 0x8049620, buf = 0x8049620, diff = 0x10 (16) bytes
bufptr (0x8049630) = 0x41414141, buf = 0x8049620, diff = 0x10 (16) bytes
```

Overwriting File Pointers

```
#define BUFSIZE 16

int main(int argc, char **argv)
{
  FILE *tmpfd;
  static char buf[BUFSIZE], *tmpfile;

  tmpfile = "/tmp/vulprog.tmp";
  printf("before: tmpfile = %s\n", tmpfile);

  printf("Enter one line of data to put in %s:
", tmpfile);
  gets(buf);

  printf("\nafter: tmpfile = %s\n", tmpfile);

  tmpfd = fopen(tmpfile, "w");
  if (tmpfd == NULL) exit(ERROR);

  fputs(buf, tmpfd);
  fclose(tmpfd);
}
```

"/etc/shadow"

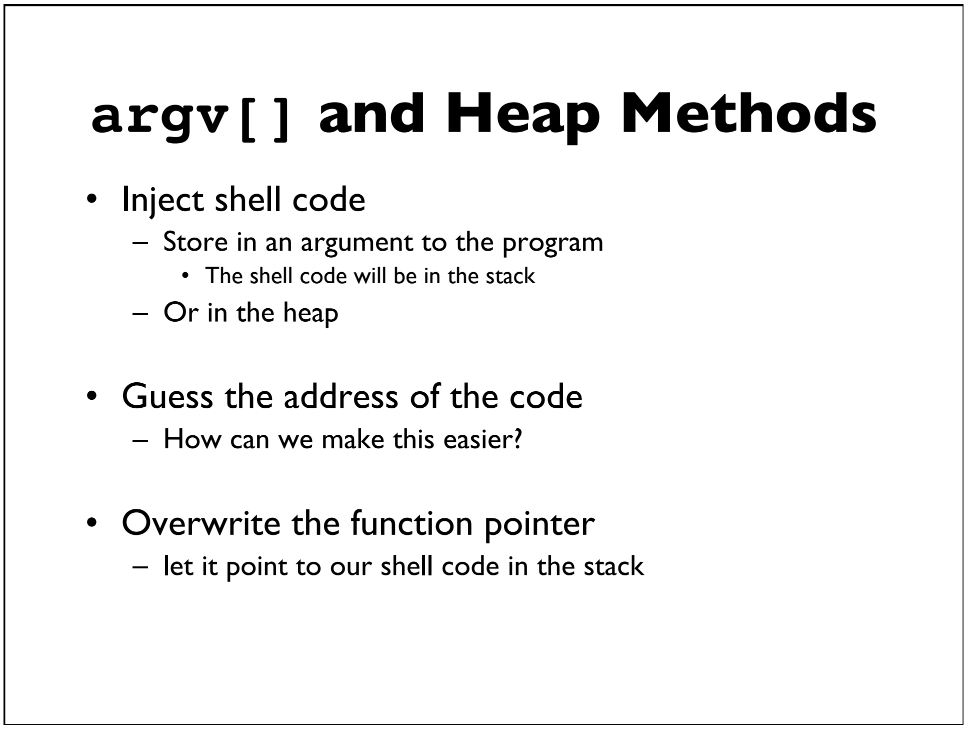


Overwrite Function Pointers

- Dynamically modify a function
  - E.g., `int (*funcptr)(char *str)`
- Taking advantage
  - `System()` method: using library functions
  - `argv[]` method: store shell code in the input
  - Heap method: store shell code in the heap

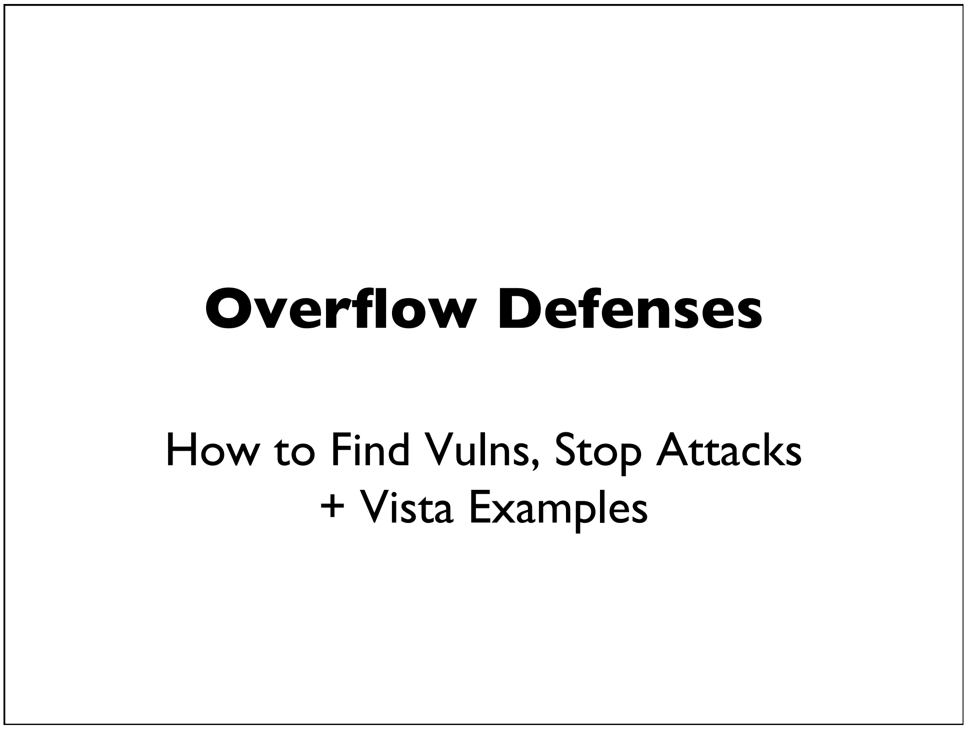
`System()` Method

- `System()` function
  - library function
  - Address is usually fixed
- Steps
  - Guess the address of `system()` function.
  - Overwrite the function pointer in the heap
  - Let it point to the `system()` function.



`argv[]` and Heap Methods

- Inject shell code
  - Store in an argument to the program
    - The shell code will be in the stack
  - Or in the heap
- Guess the address of the code
  - How can we make this easier?
- Overwrite the function pointer
  - let it point to our shell code in the stack



Overflow Defenses

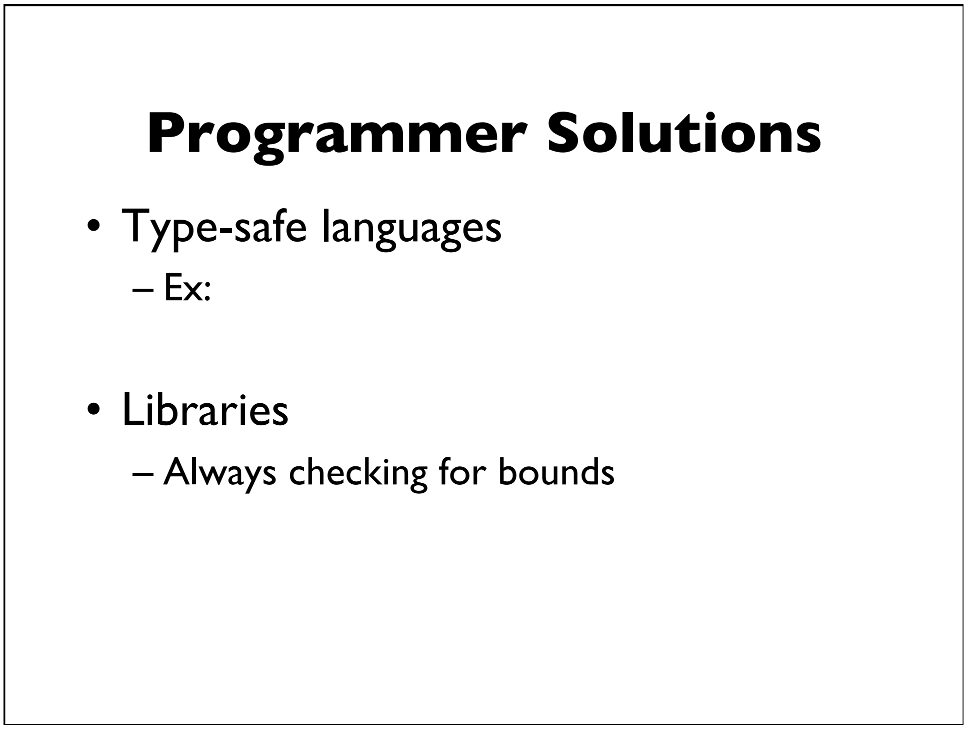
How to Find Vulns, Stop Attacks
+ Vista Examples

Vulnerable Code

- Unsafe library calls
  - Gets, strcpy, strcat, sprintf, scanf
- Safer ones
  - fgets, strncpy, strncat, snprintf

High-level Defense Approaches

- Programmer
- Compiler
- System



Programmer Solutions

- Type-safe languages
  - – Ex:
- Libraries
  - – Always checking for bounds

Programmer Solutions

- Improve programming
- Limited
  - But OpenBSD has a good record

Compiler Solutions

- Compiler help
  - May not work for speed-critical programs with lots of pointers

Preventing Stack Smashing

- Backwards stack?

StackGuard

- [Cowan et al., 1998]
- Buffer overflows
  - “Canary in the coal mine”
- Canary

Stack Shield

- GCC add-on

Randomization

- Randomize addresses

Other Addresses

- The return address
  - Is protecting it enough?
- Function pointers

```
void (*foo)(int);
foo = &my_int_func;
```

  - Generalized code
    - Generic sorting (numbers, strings, objects, reverse sorting, etc.)
  - Callback functions, e.g. for a GUI
    - Create a button as a generic call
    - Not one function for each type of button

Non-Executable Stack

- Hardware protection
    - AMD - NX (No eXecute) bit
    - Intel - XD (eXecute Disabled)
- Some cost
    - A slight performance hit
    - Some functions need executable stack
        - Linux signal handler

Non-Executable Stack

- Not a guarantee
    - Stack overflow and point to code in the heap
    - Return-to-libc
        - Alter the return address,
        - Direct return to a C library function
            - Not shell code
        - C Library function usually has fixed address
        - System("/bin/sh")

Windows Vista

- BO Protections
  - – Only applies to “unmanaged,” non-.Net code (C and C++)
  - – Support for no-execute bits (NX)
    - Data Execution Prevention (DEP)
    - Self-modifying code will fail
      - – Can specially mark the code

Windows Vista

- Randomization
  - – Address Space Layout Randomization
    - Different for each boot
    - Shell code is hard to find
  - – Heap randomization
  - – Stack randomization
- Heap corruption detection
  - – A variety of illegal operations

Visual C++

- More BO protections
  - StackGuard-based
    - /GS compiler flag
    - Enabled by default
    - Estimated 3% performance penalty
  - Move buffers higher in memory than other data
    - Why?

Visual C++

- More BO protections
  - Safe exception handling (SafeSEH)
  - Exception handlers
    - Address on the function’s stack frame
    - Can be overwritten
  - Store valid handler addresses
    - XP SP2 on, won’t use other addresses
  - Any performance impact?

MS Security Priorities

Defense	Priority
Address space layout randomization opt-in	**Critical**
DEP opt-in	**Critical**
/GS stack-based buffer overrun detection	**High**
/SafeSEH exception handler protection	**High**
Stack randomization testing	**Moderate**
Heap randomization testing	**Moderate**
Heap corruption detection	**Moderate**

The End