15-213

"The course that gives CMU its Zip!"

Memory Management III: Perils and pitfalls Mar 9, 2000

Topics

- **Memory-related bugs**
- **Debugging versions of malloc**

C operators

Operators / **Associativity**

```
()  []  ->  .                                   left to right
!  ~  ++  --  +  -  *  &  (type)  sizeof        right to left
*  /  %                                         left to right
+  -                                            left to right
<<  >>                                          left to right
<  <=  >  >=                                    left to right
==  !=                                          left to right
&                                               left to right
^                                               left to right
|                                               left to right
&&                                              left to right
||                                              left to right
?:                                              right to left
=  +=  -=  *=  /=  %=  &=  ^=  !=  <<=  >>=     right to left
,                                               left to right
```

Note: Unary +, -, and * have higher precedence than binary forms

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Memory-related bugs

Dereferencing bad pointers
Reading uninitialized memory
Overwriting memory
Referencing nonexistent variables
Freeing blocks multiple times
Referencing freed blocks
Failing to free blocks

Dereferencing bad pointers

The classic scanf bug

```
scanf("%d", val);
```


Reading uninitialized memory

Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```


Overwriting memory

Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```


Overwriting memory

Off-by-one

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Overwriting memory

Off-by-one redux

```
int i=0, done=0;
int s[4];

while (!done) {
   if (i > 3)
     done = 1;
   else
     s[++i] = 10;
}
```


Overwriting memory

Forgetting that strings end with '/0'

```
char t[7];
char s[8] = "1234567";

strcpy(t, s);
```


Overwriting memory

Not checking the max string size

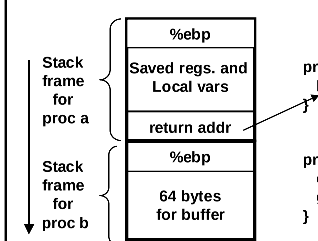
```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

Basis for classic *__buffer overflow attacks__*
- 1988 Internet worm
- modern attacks on Web servers

Buffer overflow attacks

Description of hole:

- Servers that use C library routines such as gets() that don't check input sizes when they write into buffers on the stack.
- The following description is based on the IA32 stack conventions. The details will depend on how the stack is organized, which varies between machines

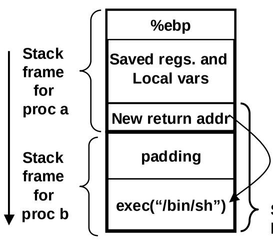



```
proc a() {
  b(); # call procedure b
}

proc b() {
  char buffer[64]; # alloc 64 bytes on stack
  gets(buffer);      # read STDIN line into stack buf
}
```

Buffer overflow attacks

Vulnerability stems from possibility of the gets() routine overwriting the return address for b.

- **overwrite stack frame with**
  - machine code instruction(s) that execs a shell
  - a bogus return address to the instruction




```
proc a() {
   b(); # call procedure b
}       # b should return here, instead it
        # returns to an address inside of buffer

proc b() {
   char buffer[64]; # alloc 64 bytes on stack
   gets(buffer);     # read STDIN line to stack buffer
}
```

Stack region overwritten by gets(buffer)

Buffer overflow attacks on servers

Example attack: classic *buffer overflow attack*

- **Early versions of the finger server (fingerd) used gets() to read the argument sent by the client:**
  - *finger droh@cs.cmu.edu*
- **To attack fingerd, send a binary string that puts a program to execute a shell on the stack followed by a new return address to that stack location, padded with enough bytes so that it overwrites the real return address.**
  - *finger "binary program padding new return address"*
- **After the finger server reads the argument from the client, the client has a direct TCP connection to a root shell running on the server!**
  - STDIN and STDOUT on the server are bound to an open TCP socket
- **Bottom line: client can now execute any command on the server.**

Famous buffer overflow attack: The 1988 Internet Worm

***Worm*: an independent program that replicates itself across the host machines on a network.**

November 1988: Thousands of Sun and DEC machines on the Internet are attacked by a "worm" written by Cornell grad student Robert Morris.

Because of a bug in the worm, it replicated itself multiple times on many of the Internet hosts, causing them to crash.

- **had the effect of a denial of service attack**

Resulted (after a similar attack weeks later) in the formation of CERT (Computer Emergency Response Team) and increased awareness of security.

Overwriting memory

Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Overwriting memory

Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```


Referencing nonexistent variables

Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;
   return &val;
}
```


Freeing blocks multiple times

Nasty!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```


Referencing freed blocks

Evil!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to free blocks (memory leaks)

slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```


Failing to free blocks (memory leaks)

Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head =
               malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
   ...
   free(head);
   return;
}
```


Dealing with memory bugs

Conventional debugger (gdb)

- **good for finding bad pointer dereferences**
- **hard to detect the other memory bugs**

Debugging malloc (CSRI UToronto malloc)

- **wrapper around conventional malloc**
- **detects memory bugs at malloc and free boundaries**
  - memory overwrites that corrupt heap structures
  - some instances of freeing blocks multiple times
  - memory leaks
- **Cannot detect all memory bugs**
  - overwrites into the middle of allocated blocks
  - freeing block twice that has been reallocated in the interim
  - referencing freed blocks

Dealing with memory bugs (cont.)

Binary translator (Atom, Purify)

- **powerful debugging and analysis technique**
- **rewrites text section of executable object file**
- **can detect all errors as debugging malloc**
- **can also check each individual reference at runtime**
  - bad pointers
  - overwriting
  - referencing outside of allocated block

Garbage collection (Boehm-Weiser Conservative GC)

- **let the system free blocks instead of the programmer.**

Debugging malloc

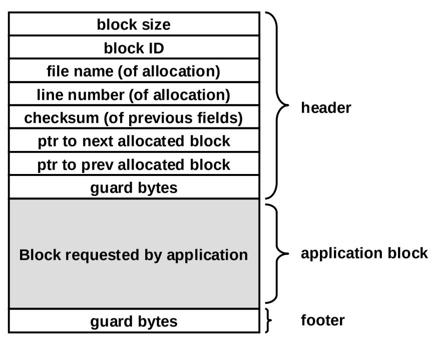
```
mymalloc.h:

#define malloc(size) mymalloc(size, __FILE__, __LINE__)
#define free(p) myfree(p, __FILE__, __LINE__)
```

```
Application program:

ifdef DEBUG
#include <mymalloc.h>
#endif

main() {
   ...
   p = malloc(128);
   ...
   free(p);
   ...
   q = malloc(32);
   ...
}
```


Debugging malloc (cont.)

```
Debugging malloc library:

void *mymalloc(int size, char *file, int line) {
   <prologue code>
   p = malloc(...);
   <epilogue code>
   return q;
}

void myfree(void *p, char *file, int line) {
   <prologue code>
   free(p);
   <epilogue code>
}
```


Debugging malloc (cont.)



Debugging malloc (cont.)

mymalloc(size):

- p = malloc(size + sizeof(header) + sizeof(footer));
- add p to list of allocated blocks
- initialize application block to 0xdeadbeef
- return pointer to application block

myfree(p):

- already free (line # = 0xfefefefefefefefe)?
- checksum OK?
- guard bytes OK?
- free(p - sizeof(hdr));
- line # = 0xfefefefefefefefe;