CS 33

Storage Allocation Problems

CS33 Intro to Computer Systems XXIII–1

Some of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Dereferencing Bad Pointers

- **The classic `scanf` bug**

```
int val;

...

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


Supplied by CMU.

Reading Uninitialized Memory

- **Assuming that heap data is initialized to zero**

```
/* return y = Ax */
int *matvec(int A[][N], int x[]) {
   int *y = (int *)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;
}
```


Supplied by CMU.

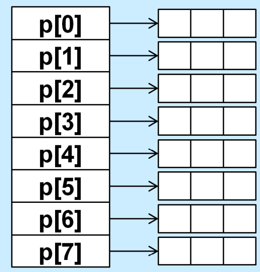
Overwriting Memory

- **Allocating the (possibly) wrong-sized object**

```
// set up p so it is an array of
// int *'s, allocated dynamically
int **p;

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

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



Supplied by CMU.

The problem here is that the storage allocated for p is of size N*sizeof(int), when it should be N*sizeof(int *) — on a 64-bit machine, p won’t have been assigned enough storage.

Overwriting Memory

- **Off-by-one error**

```
int **p;

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

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


Supplied by CMU.

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**

Supplied by CMU.

Going Too Far

- **Misunderstanding pointer arithmetic**

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

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

   return p;
}
```


Supplied by CMU.

Referencing Nonexistent Variables

- **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```


Supplied by CMU.

Freeing Blocks Multiple Times

- **Nasty!**

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

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


Supplied by CMU.

Referencing Freed Blocks

- **Evil!**

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


Supplied by CMU.

Failing to Free Blocks (Memory Leaks)

- **Slow, long-term killer!**

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


Supplied by CMU.

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;
}
```


Supplied by CMU.

Total Confusion

```
foo() {
   char *str;
   str = (char *)malloc(1024);
   ...
   str = "";
   ...
   strcat(str, "c");
   ...
   return;
}
```

It Works, But ...

- **Using a hammer where a feather would do ...**

```
funky() {
   int *x = (int *)malloc(1024*sizeof(int));
   Use(x, 1024);
   free(x);
   return;
}
```

```
better_funky() {
   int x[1024];
   Use(x, 1024);
   return;
}
```