# CSE 303: Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 27— More Memory-Management Idioms; A Tiny Taste of Software Security

# Where are we

- A week ago: started looking at common ways to approach the very difficult problem of manual memory management (none are always appropriate)
  - To review: unique pointers
  - To discuss: reference-counting and regions
- Then: Why secure code is extra hard to write
  - And some rules of thumb / mottos

# Why is memory-management hard?

This is not really the hard part:

```
free(p);
...
p->x = 37; // dangling-pointer dereference
```

These are:

```
p = q; // if p was last reference and q!=p, leak!
```

```
lst1 = append(lst1,lst2);
free_list(lst2); // user function, assume it
                 // frees all elements of list
length(lst1); // dangling-pointer dereference
              // if append does not copy!
```

There are an infinite number of *safe idioms*, but only a few are known to be simple enough to get right in large systems...

# Idiom 1: Unique Pointers

*Ensure there is exactly one pointer to an object.* Then you can call `free` on the pointer whenever, and set the pointer's location to `NULL` to be "extra careful".

Furthermore, you *must* free pointers before losing references to them.

Hard parts:

1. May make no sense for the data-structure/algorithm.
2. May lead to extra space because sharing is not allowed.
3. Easy to lose references (e.g., `p=q;`).
4. Easy to duplicate references (e.g., `p=q;`) (must follow with `q=NULL;`).
5. A pain to return unfreed function arguments.

# Relaxing Uniqueness

This does not preserve uniqueness:

```
void g(int *p1, int*p2) { ... }
void f(int *p1, int*p2) {
  if(...)
    g(p1,p1);
  else
    g(p1,p2);
  ...
  free(p1);
  free(p2);
}
```

Wrong if g frees an argument or stores an alias somewhere else.

Also notice true-branch creates aliases just in the callee.

# Relaxing Uniqueness

Instead, have some "unconsumed" pointers:

- Callee won't free them
- They will be unique again when function exits

More often what you want, but changes the contract:

- Callee must *not* free
- Callee must not store the pointer anywhere else (in a global, in a field of an object pointed to by another pointer, etc.)

# Idiom 2: Reference-Counting

*Store with an object how many pointers there are to it. When it reaches 0, call* `free`.

- Literally a field in each pointed to object.
- `p=q;` becomes `incr_count(q); decr_count(p); p=q;`
- In practice, you can "be careful" and omit ref-count manipulation for temporary variables.

```
struct Example { int count; ... };
void incr_count(struct Example * p) { ++p->count; }
void decr_count(struct Example * p) {
  --p->count;
  if(p->count == 0)
    free(p);
}
```

# Reference-Counting Problems

1. Avoids freeing too early, but one lost reference means a leak.
2. Reference-count maintenance expensive and error-prone (C++ tricks can automate to some degree).
3. CYCLES! (Must break them manually.)

# Idiom 3: Arenas (a.k.a. regions)

*Rather than track each object's "liveness", track each object's "region" and deallocate a region "all at once".*

Revised memory-management interface:

```
typedef struct RgnHandle * region_t;
region_t create_rgn();
void destroy_rgn(region_t);
void * rgn_malloc(region_t,int);
```

So now you "only" have to keep track of a pointer's region and the region's status. (In theory, no simpler? In practice, much simpler!)

And even the library is easier because it knows there is no `free`, only `destroy_rgn`.

# Arena Uses

Examples:

- Scratch space for lots of lists with sharing. When you're done, copy out the one answer and destroy the region.
- Callee chooses size, number of objects, aliasing patterns.
  Caller choose lifetime (and passes in a *handle* as an argument).
- You can track handles and inter-region pointers via other means (e.g., reference-counting) while "ignoring" intra-region pointers.

# Conclusions

Memory management is difficult; each "general approach" has plusses and minuses.

As with any "design pattern", knowing vocabulary helps communicate, assess trade-offs, and reuse hard-won wisdom.

Key notions: reachability, aliasing, cycles, "escaping (e.g., storing argument in global)". Each approach restricts one of them to some degree.

# Security

Computer security is a huge area; we can't cover it in 30 minutes.

Robust software (no buffer-overflows, uncaught exceptions, etc.) is necessary but *in no way* sufficient to achieve "security".

Non-coding examples:

- Guessable passwords
- File permissions
- Tricking humans (email clicks, ...)

Subtle coding examples:

- Timing and other *covert* channels (classic example: early-exit password checking)
- File-system tricks (relative paths and races on accesses)

Security is hard!

# Security is Worst-Case

What you cannot say when writing a function:

- I can't imagine anyone ever passing arguments like that.
- If the arguments don't make sense, "behavior is unpredicatable".
- I'll get it working now, and close the security holes later when I have time.

Some mottos/axioms:

- Principle of least privilege: Give each entity no more rights than it needs to accomplish its task.
- Obscurity is not security.
  - Guessing URLs, phone-tree numbers, back-doors, etc.

Also: simple, well-defined security policies separate from implementations

# Security-breach impact

Also be clear about what is at stake:

- Resource-consumption (denial-of-service)
- Data revealing (privacy, theft)
- Data corruption (destruction of property)
- Full machine compromise (run arbitrary programs on attacked computer)

Take preventative measures. Example: the CSE department uses a different webserver for CGI scripts and it does *not* have access to home directories.

# Check your inputs

Any nontrivial program has *untrusted inputs* (command-line args, files, mouse-clicks, etc.)

- What properties do you expect them to have (integers, buffer-lengths, alphabetical characters, ...)
- **Check these properties!!!**

Bad examples:

- `printf(argv[0])`
- `char x[256]; strcpy(x,argv[1]);`
- SQL-injection attacks
  - See bash analogy.

A sad tale: The FUZZ studies

# Input-checking is more general

"Defensive programming" is good software-engineering practice regardless of security:

- Check your function inputs (at least of public) methods
  - At compile-time if possible
  - At run-time if possible
  - Not just in debug-mode (if not too expensive)
  - Not everything is possible
- Assertions are so common, Java added them to the language.
- Preaching: How can assertions be more important during testing?!

# Copy your inputs/outputs

In the presence of mutation (as in C and Java), checking inputs is not always enough:

- What if the untrusted source can change the inputs after you check them but before you use them.
- What if you give out pointers to internal data and untrusted recipient assigns through the pointers.

Java example (security flaw in JDK1.1) a class's permissions:

```
public class Class {
  private Identity[] signers;
  public Identity[] getSigners() { return signers; }
  ...
}
```

Another motto: "copy-in/copy-out"

# A note on ethics

An analogy: Engineers learning how to make strong glass might learn about the weakenesses of glass, how one can throw rocks through glass, etc.

It is still illegal to break a window that is not yours and dangerous to throw rocks.

Giving examples of hacks can make you a more secure programmer. Unleashing hacks can lead to long "government-sponsored vacations".