Cyclone

Safe C-level Programming

CMSC 412, Fall 2004

Michael Hicks

Credit where credit is due …

- Cyclone is a research language, the product of the labors of many people:
  - Greg Morrisett (Harvard)
  - Dan Grossman (Washington)
  - Trevor Jim (AT&T)
  - Mike Hicks

1988? 2004?

- "In order to start copies of itself running on other machines, the worm took advantage of a buffer overrun...
- ...it is estimated that it infected and crippled 5 to 10 percent of the machines on the Internet."
- Fact: half of CERT advisories involve buffer overruns.

1998: Missile Cruisers

- "The controversy began when the USS Yorktown ... suffered a widespread system failure ... a crew member mistakenly entered a zero into the data field of an application ... caused a buffer overflow ... which turned into a memory leak ... eventually brought down the ship's propulsion system.
- The result: the Yorktown was dead in the water for more than two hours."

Building Secure Software

- Today, our economy, government, and military depend upon the proper functioning of our computing and communications infrastructure.
- That infrastructure is coded in low-level, error-prone languages (*i.e.* C).
  - device drivers, kernels
  - file systems, web servers, email systems
  - switches, routers, firewalls

But C is a lousy language

- Must bypass the type system to do even simple things (e.g., allocate and initialize an object.)
- Libraries put the onus on the programmer to do the "right thing" (e.g., check return codes, pass in large enough buffer.)
- For efficiency, programmers stack-allocate arrays of size K (is K big enough? does the array escape downwards?)
- Programmers assume objects can be safely recycled when they cannot and fail to recycle memory when they should.
- It's not "fail-stop" --- errors don't manifest themselves until well after they happen (e.g., buffer overruns.)

But it's also very useful:

- Almost every critical system is coded in C:
  - language run-times, operating systems, device drivers, servers, switches, etc.
- because it provides a lot of good things:
  - ported to lots of architectures
  - low-level control over data structures, memory management, instructions, etc.
  - good performance
- We need safety for these infrastructures.

What can we do?

- Rewrite the code in Java or some other type-safe language?
  - Not low-level enough.
    - no control over data representations.
    - no control over memory management.
    - performance isn't there?
  - Just not realistic.
    - any more than telling all of those businesses to re-code their Cobol code to avoid Y2K.
    - need an incremental solution.

Instead …

- We need a next-generation low-level language X with the following features:
  - The practical coding power of C.
    - need to build device drivers, kernels, etc.
  - Transparent interoperability with legacy C.
    - just can't switch the whole world over at once.
  - The safety and scalability of Java.
    - many errors caught at compile time
    - fail-stop behavior at run time.
  - A relatively painless path from C to X.

Cyclone: an experimental Safe-C

- Start with ANSI-C.
- Throw out anything that can lead to a delayed core-dump:
  - e.g., arbitrary casts, unchecked pointer arithmetic
- Add a combination of advanced typing mechanisms and dynamic checks to cover what's missing.
  - keep analyses intra-procedural.
  - programmer will have to specify additional details at procedure boundaries.
- Minimize re-coding for safe idioms.
  - best case: leave the code alone
  - next best: add typing annotations
  - worst case: re-write the code

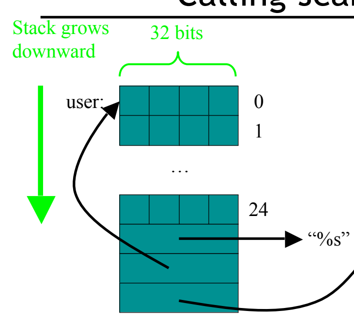
What is a C buffer overflow?

```
• #include <stdio>

• int login() {
•   char user [100];
•   printf("login: ");
•   scanf("%s",&user);
•   … // get password etc.
• }
```

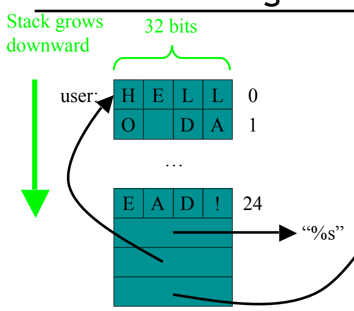
What happens if the user types In something that's more than 100 characters?

Calling scanf()



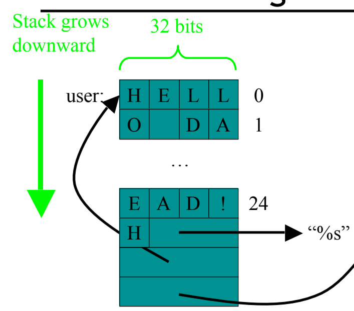

```
• int login() {
•   char user [100];
•   printf("login: ");
•
  scanf("%s",&user);
•   // return here
• }
```

Calling scanf()



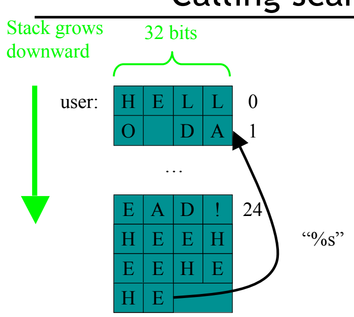

```
• int login() {
•   char user [100];
•   printf("login: ");
•
  scanf("%s",&user);
•   // return here
• }
```

Calling scanf()




```
• int login() {
•   char user [100];
•   printf("login: ");
•
  scanf("%s",&user);
•   // return here
• }
```

Calling scanf()




```
• int login() {
•   char user [100];
•   printf("login: ");
•
  scanf("%s",&user);
•   // return here
• }
```

Can make the return address point anywhere, including into the buffer.

How to Prevent This?

- Don't allow dereferencing a buffer unless compiler can prove it's safe
  - Too conservative
- Have two separate stacks, one for data, one for return addresses
  - Violates standard calling convention
  - Could still work around this
- Prevent dereferencing with *dynamic* checks

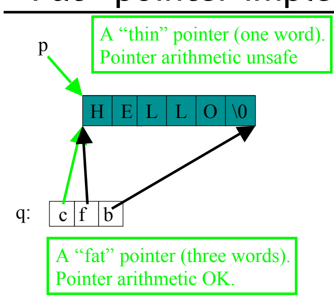
Bounds Checking

- I would like scanf to check each time it writes to its buffer to make sure that it's not about to "go off the end."
- To do this, I must provide not only the buffer memory, but the bounds on it.
- Then I can check that every dereference is within bounds.
- This is what Java does, too.

"Fat" pointers

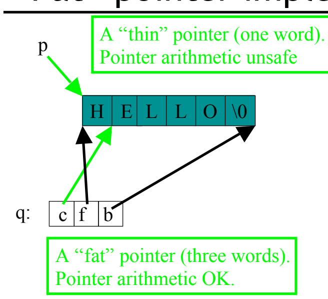
- What kind of bounds do I need?
  - Just the length of the array
    - This is what Java does
    - But, what happens with pointer arithmetic?
  - A pointer to the current location, and a pointer to the end of the array
    - Allows forward arithmetic. (x++)
    - But what about backward arithmetic? (x--)
  - Answer: pointers to the beginning and end of the buffer, and a pointer to the current location.

"Fat" pointer implementation



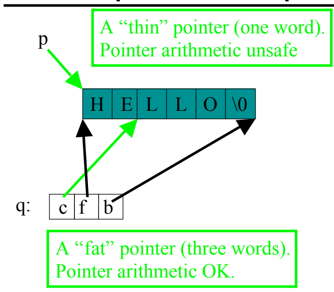

```
q++;
q++;
q--;
q++;
```

"Fat" pointer implementation



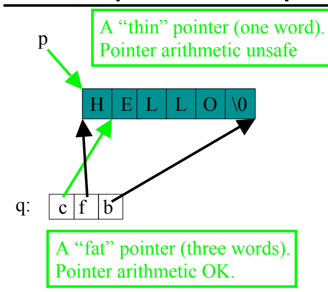

```
q++;
q++;
q--;
q++;
```

“Fat” pointer implementation



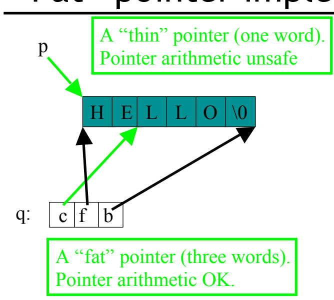

```
q++;
q++;
q--;
q++;
```

“Fat” pointer implementation




```
q++;
q++;
q--;
q++;
```

“Fat” pointer implementation




```
q++;
q++;
q--;
q++;
```

Thin, bounded pointers

```
#include <stdio>

int foo() {
  char buf[100] = {'h','e','l',…};
  int i;
  for (i = 0; i<100; i++) {
    putc(buf[i]);
  }
}
```

Do I really need bounds checks here?

No. Compiler can easily prove that all dereferences will be in bounds, so no need for extra information.

What about NULL?

```
#include <stdio>

int foo(char ?filename, char ?buf) {
  FILE *fp;
  fp = fopen(filename,"r");
  fwrite(fp,buf);
}
```

What happens if fopen failed, returning NULL?

Can result in a crash. C library assumes the user will check for NULL. In Cyclone we enforce this.

Not-null Pointers

- Two pointer types
  - int *
    - A possibly-null pointer to an int
  - int * @notnull
    - A definitely-not-null pointer to an int
    - Abbreviated int @
- Library functions can specify the latter, thus forcing the user to do a null check.

Not-null Pointer Usage

```
int *p = NULL;
int @q = NULL; // not allowed
int @r = p; // not allowed; type(p) != type(r)
int @r = (int @)p; // ok, does a null check

extern int fwrite(FILE @fp, char ?buf);
  // requires that fp be not-null
```

Pointer Summary

- Three kinds of pointers make intention clear:
  - fat pointers: int ?
    - represented as a triple: {base, upper, curr}
    - supports all operations that C does on int*
    - but any dereference is checked against bounds
    - ? makes representation change clear
  - thin, definite pointers: int @, int @{const-exp}
  - thin, possibly null pointers: int *, int*{const-exp}
    - bounds tracked statically -- same rep. as C
    - limited pointer arithmetic
    - * requires a null check.

Cyclone Hello World

```
#include <stdio.h>

int main(int argc, char ??argv) {
  if (argc > 1) {
    printf("Hello %s.\n",*(argv+1));
    return 0;
  }
  fprintf(stderr,"Usage: %s <name>\n",argv[0]);
  return -1;
}
```

Libraries are wrapped to prevent bad inputs

? denotes a "fat" pointer with bounds information

arguments to printf are wrapped with type information

pointer dereferences are checked either statically (optimized) or dynamically (typical)

Another Example:

```
typedef struct Point { int x,y; } pt;

void addTo(pt *p, pt *q) {
  p->x += q->x;
  p->y += q->y;
}

void foo() {
  pt a = {1,2};
  pt b = {3,4};
  pt *aptr = &a;
  pt *bptr = &b;
  addTo(aptr,bptr);
}
```

Many times, C code such as this compiles directly with no changes needed by programmer.

However, there may be additional run-time checks.

A Better Port

```
typedef struct Point { int x,y; } pt;

void addTo(pt @p, pt @q) {
  p->x += q->x;
  p->y += q->y;
}

void foo() {
  pt a = {1,2};
  pt b = {3,4};
  pt @aptr = &a;
  pt @bptr = &b;
  addTo(aptr,bptr);
}
```

By refining the types of variables, programmers can often get rid of the overheads.

Making Libraries Robust

```
struct FILE;
extern FILE *fopen(char ? name, char ? mode);
extern int putc(char, FILE@);

void foo() {
  FILE *f = fopen("/tmp/bar.txt","+wb");
  char s[] = "hello";
  int i;
  for (i = 0; i < 5; i++) { putc(s[i],f); }
}
```

most implementations core dump when given NULL.

type error here because f has type FILE* but putc demands FILE@.

One way to fix:

```
struct FILE;
extern FILE *fopen(char ? name, char ? mode);
extern int putc(char, FILE@);

void foo() {
  FILE *f = fopen("/tmp/bar.txt","+wb");
  char s[] = "hello";
  int i;
  for (i = 0; i < 5; i++) { putc(s[i],(FILE @)f); }
}
```

dynamically checks that f is an actual file.

A better fix:

```
struct FILE;
extern FILE *fopen(char ? name, char ? mode);
extern int putc(char, FILE@);

void foo() {
  FILE *fn = fopen("/tmp/bar.txt","+wb");
  char s[] = "hello";
  int i;
  if (*fn != NULL) {
    FILE @f = (FILE @)fn;
    for (i = 0; i < 5; i++) { putc(s[i],f); }
  } else {
    throw new FileError("can't open /tmp/bar.txt!");
  }
}
```

Object Lifetimes: Spot the Bug

```
pt *add(pt *p, pt *q) {
  pt r;
  r->x = p->x + q->x;
  r->y = p->y + q->y;
  return &r;
}

void foo() {
  pt a = {1,2};
  pt b = {3,4};
  pt *c = addTo(&a, &b);
  c->x = 10;
}
```

r's lifetime ends here!

so dereferencing c here can cause problems...

Tracking Object Lifetimes

- Cyclone uses a *region-based* type system:
  - Each lexical block is treated as a distinct region.
  - Each pointer type has an associated region: **int*`r**
  - The heap is treated as a special region (`H) with a global lifetime (more on this later).
  - A pointer can only be dereferenced while the region is still live.

Simple Region Example

```
pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt @`H aptr = &a;
  pt @`foo bptr = &b;
  addTo(&a, &b);
}
```

a lives in the heap region, so &a has type pt @`H.

b lives in the activation record of foo so &b has type pt @`foo.

region inference can figure out the regions, so the programmer doesn't have to write them

Definite Initialization

```
void foo() {
  pt a;
  pt * aptr = &a;
  if (rand())
      { a.x = 1;
        a.y = 2;
      }
  aptr->x++;
}
```

Flow analysis determines that this may not be initialized.

Dangling Pointers

```
void foo() {
  int *x = malloc(sizeof(int));
  int *y;
  *x = 1;
  // do some stuff
  y = x;
  free(x);
  *y = 5; // freed storage!
}
```

Eliminating Dangling Pointers

- Garbage collection (simplest)
  - free() is removed
  - Memory is freed when it could not possibly be used by the program (reachability)
- Scoped memory management
- Safe malloc/free

- Cyclone supports all of these

Other things to be nervous about

- Unsafe casts

  int *p = (int *)1;
- Unsafe uses of union

```
union u { int x; int *p };
union u v;
v.x = 1;
*v.p = 5;
```

- varargs (as implemented in C)
- Cyclone prevents these bad usages

Performance

- Typically 1.5x C; up to 4-5x
- Bottlenecks
  - Array-bounds checks
  - Unoptimized libraries (e.g. string, file I/O libraries)

Cyclone: where we stand

- Cyclone compiler
  - ~100KL of Cyclone code
  - Bulk is the type-checker and dataflow analyses
  - Straightforward translation to C
  - Available for many architectures (Linux, BSD, Irix, Cygwin, Sparc, etc.)
- Ports
  - Libc and other libs (sockets, XML, lists, and more)
  - bison, flex, web server, cfrac, grobner, NT device driver … (~40KL total)
  - Typically differ from original C by 5-15%

Tools and Applications

- Lex, Bison, Memory profiler
- Semi-automated porting tools
  - Guess whether to convert a C * to Cyclone *, @, or ?
- In-kernel transport protocols (SOSP 03)
- Streaming data overlay networks (OPENARCH 03)
- In-kernel extensions (OPENARCH 02)
- Hardware description languages

Summary

- Research in safe, low-level languages is crucial.
- Programmer-controlled data representations and memory management are critical issues.
- We have good typing technologies at this point, but adapting them to practical settings is a lot of work.
- Cyclone isn't a full solution but it's moving in the right direction.

Obligatory URL

http://www.cs.umd.edu/projects/cyclone

- Includes code, papers, documentation, and more!