Secure Coding

Secure coding in C

- Since the language provides few guarantees, **developers** must use **discipline**
- Good **reference guide**: **CERT C Coding Standard**
  - https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Coding+Standard
    - Similar guides for other languages (e.g., Java)
  - See also: *David Wheeler*: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html
  - And *Matt Bishop*: http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Combine with ***advanced code review and testing*** *—*
will consider in depth later in the course

Design vs. Implementation

- In general, we strive to follow principles and rules
  - A **principle** is a design goal with many possible manifestations.
  - A **rule** is a specific practice that is consonant with sound design principles.
    - The difference between these can sometimes be fuzzy
- Here we look at **rules** for **good C coding**
  - In particular, to **avoid implementation errors** that could result in violations of memory safety
- In a **future** course module we **consider principles and rules more broadly**

Rule: Enforce input compliance

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = MIN(len,strlen(buf));
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
 }}
```

Recall earlier example

Read integer

(partial) message

Sanitizes input to be compliant

May exceed actual message length!

Rule: Enforce input compliance

```
char digit_to_char(int i) {
  char convert[] = “0123456789”;
  if(i < 0 || i > 9)
    return ‘?’;

}
```

possible overflow

- **Unfounded trust** in received input is a recurring **source of vulnerabilities**
  - We will see many more examples in the course

General **Principle**: *Robust* coding

- **Like defensive driving**
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about ***minimizing trust***
- Each module **pessimistically checks its assumed preconditions** (on outside callers)
  - Even if you "know" clients will not send a NULL pointer
  - … Better to throw an exception (or even exit) than run malicious code

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = “fine”;
strcpy(str,”hello”); // overflows str
strcat(buf,”day to you”); // overflows buf
```

- Safe versions check the destination length

```
char str[4];
char buf[10] = “fine”;
strlcpy(str,”hello”,sizeof(str)); //fails
strlcat(buf,”day to you”,sizeof(buf));//fails
```

Replacements

- … for string-oriented functions
  - `strcat` ⇒ `strlcat`
  - `strcpy` ⇒ `strlcpy`
  - `strncat` ⇒ `strlcat`
  - `strncpy` ⇒ `strlcpy`
  - `sprintf` ⇒ `snprintf`
  - `vsprintf` ⇒ `vsnprintf`
  - `gets` ⇒ `fgets`
- Microsoft versions different
  - `strcpy_s`, `strcat_s`, …

Rule: Don’t forget NUL terminator

- Strings require one additional character to store the NUL terminator. Forgetting that could lead to overflows

```
char str[3];
strcpy(str,”bye”); // write overflow
int x = strlen(str); // read overflow
```

- Using safe string library calls will catch this mistake

```
char str[3];
strlcpy(str,”bye”,3); // blocked
int x = strlen(str); // returns 2
```

Rule: Understand pointer arithmetic

- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

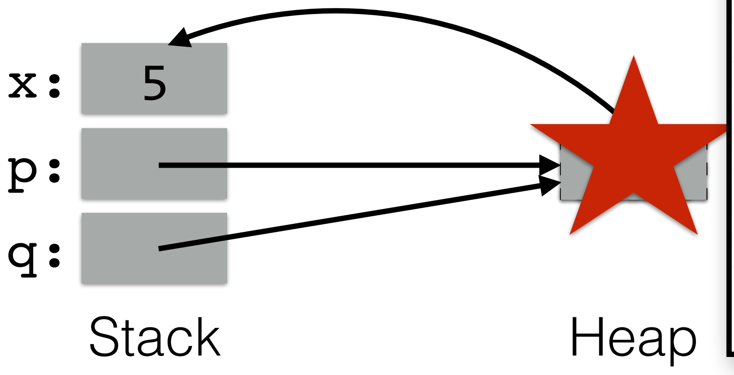
```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

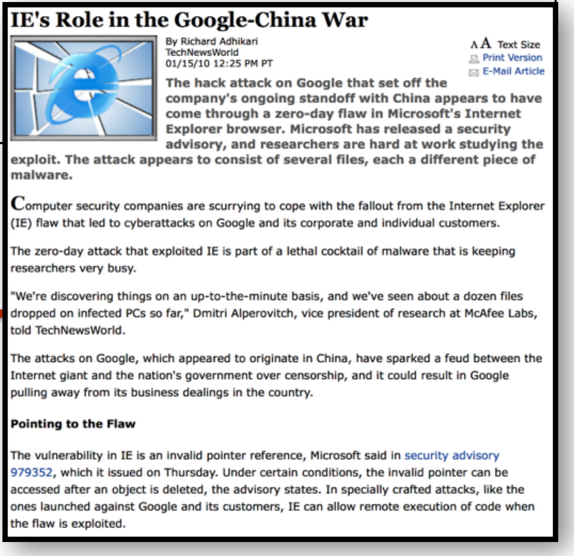
- So, **use the right units**

```
while (!done() && buf_ptr < (buf + SIZE)) {
  *buf_ptr++ = getnext(); // stays in bounds
}
```

Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```





IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

Text Size
Print Version
E-Mail Article

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

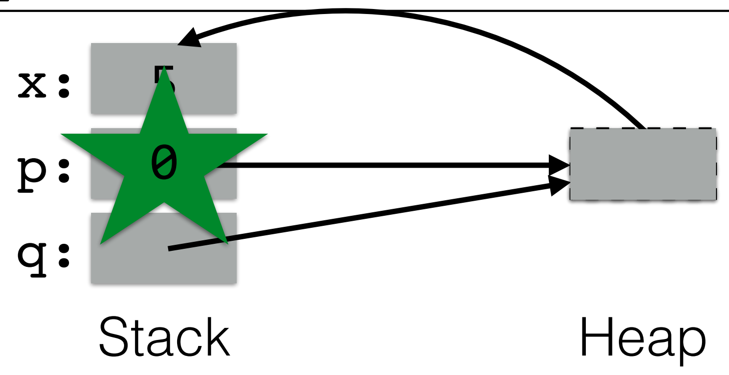
The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



Manage memory properly

```
int foo(int arg1, int arg2) {
  struct foo *pf1, *pf2;
  int retc = -1;

  pf1 = malloc(sizeof(struct foo));
  if (!isok(arg1)) goto DONE;
  …
  pf2 = malloc(sizeof(struct foo));
  if (!isok(arg2)) goto FAIL_ARG2;
  …
  retc = 0;

 FAIL_ARG2:
  free(pf2); //fallthru
 DONE:
  free(pf1);
  return retc;
}
```

- **Rule: Use *goto chains* to avoid duplicated or missed code**
  - Encodes try/finally in languages like Java
- Confirm your logic!
  - *Gotofail* bug

(Better) **Rule**: Use safe string library

- Libraries designed to ensure strings used safely
  - **Safety first**, despite some performance loss
- Example: Very Secure FTP (**vsftp**) **string library**

http://vsftpd.beasts.org/

```
struct mystr; // impl hidden

void str_alloc_text(struct mystr* p_str,
                    const char* p_src);
void str_append_str(struct mystr* p_str,
                    const struct mystr* p_other);
int str_equal(const struct mystr* p_str1,
              const struct mystr* p_str2);
int str_contains_space(const struct mystr* p_str);
…
```

- Another example: **C++ `std::string`** safe string library

Rule: Favor safe libraries

- **Libraries** encapsulate **well-thought-out design**.
  Take advantage!
- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard
- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient

Rule: Use a safe allocator

- ASLR challenges exploits by making the base address of libraries unpredictable
- **Challenge heap-based overflows** by making the **addresses** returned by `malloc` **unpredictable**
  - Can have some negative performance impact
- Example implementations:
  - **Windows Fault-Tolerant Heap**
    - http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764(v=vs.85).aspx
  - **DieHard** (on which fault-tolerant heap is based)
    - http://plasma.cs.umass.edu/emery/diehard.html