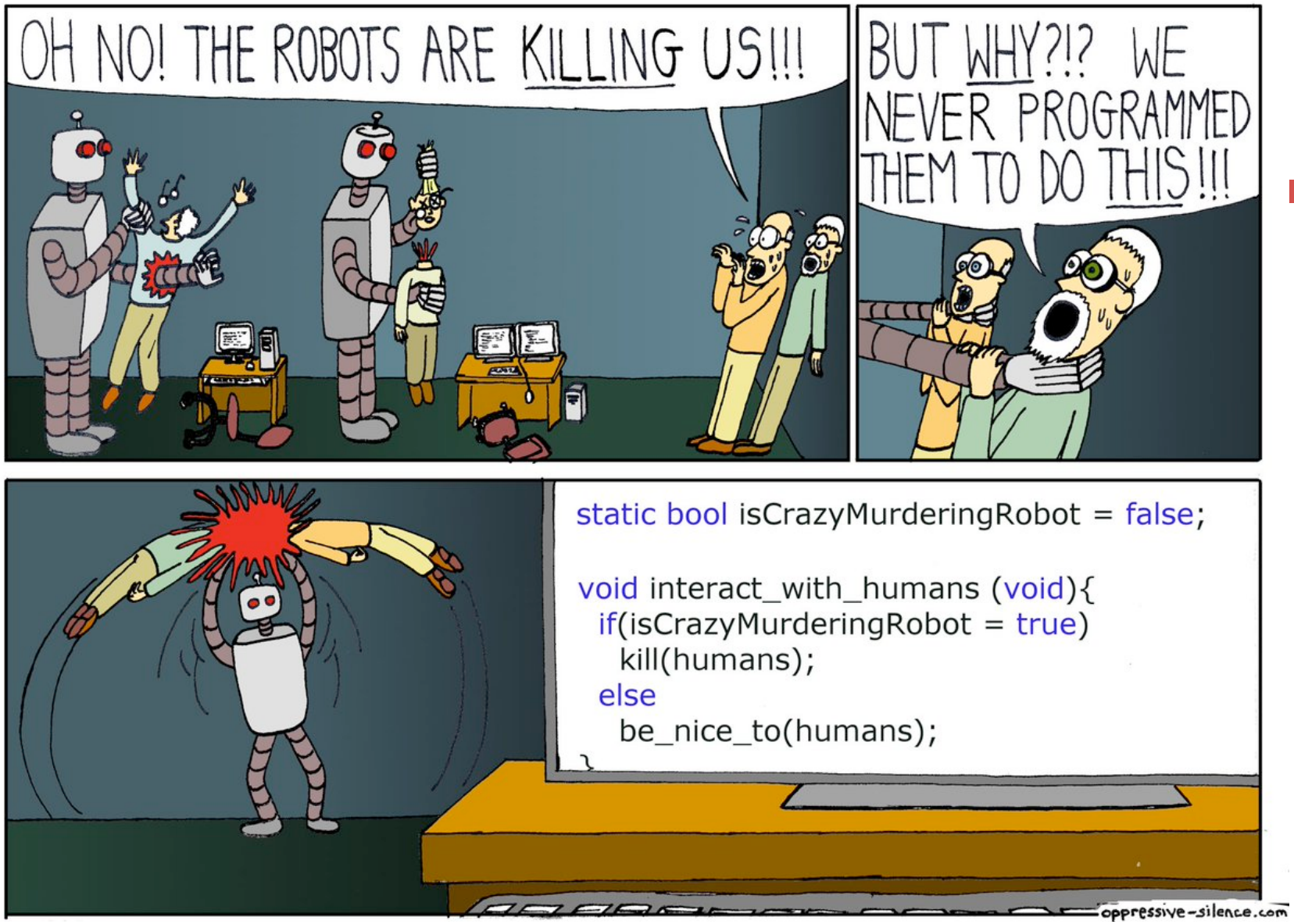
"Secure" Coding Practices
Nicholas Weaver

based on David Wagner's slides from Sp 2016

Administrivia


OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
static bool isCrazyMurderingRobot = false;
void interact_with_humans (void){
if(isCrazyMurderingRobot = true)
kill(humans);
else
be_nice_to(humans);
}
oppressive-silence.com

This is a Remarkably Typical C Problem


```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

- Someone attempted to add this checking code into the Linux kernel back in 2003
  - It goes caught only because they didn't have proper write permission so it was flagged as anomalous
- If you use the proper compiler flags, it ***should*** gripe when you do this

Why does software have vulnerabilities?

- Programmers are humans.
  And humans make mistakes.
  - Use tools
- Programmers often aren't security-aware.
  - Learn about common types of security flaws.
- Programming languages aren't designed well for security.
  - Use better languages (Java, Python, …).

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We need to test for the **absence** of something
    - Security is a negative property!
      - "nothing bad happens, even in really unusual circumstances"
  - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
  - Random inputs (fuzz testing)
  - Mutation
  - Spec-driven
- How do we tell when we've found a problem?
  - Crash or other deviant behavior
- How do we tell that we've tested enough?
  - Hard: but code-coverage tools can help

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We need to test for the **absence** of something
    - Security is a negative property!
      - "nothing bad happens, even in really unusual circumstances"
  - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
  - Random inputs (fuzz testing)
  - Mutation
  - Spec-driven
- How do we tell when we've found a problem?
  - Crash or other deviant behavior
- How do we tell that we've tested enough?
  - Hard: but code-coverage tools can help

Test For Failures...
Not Just Successes

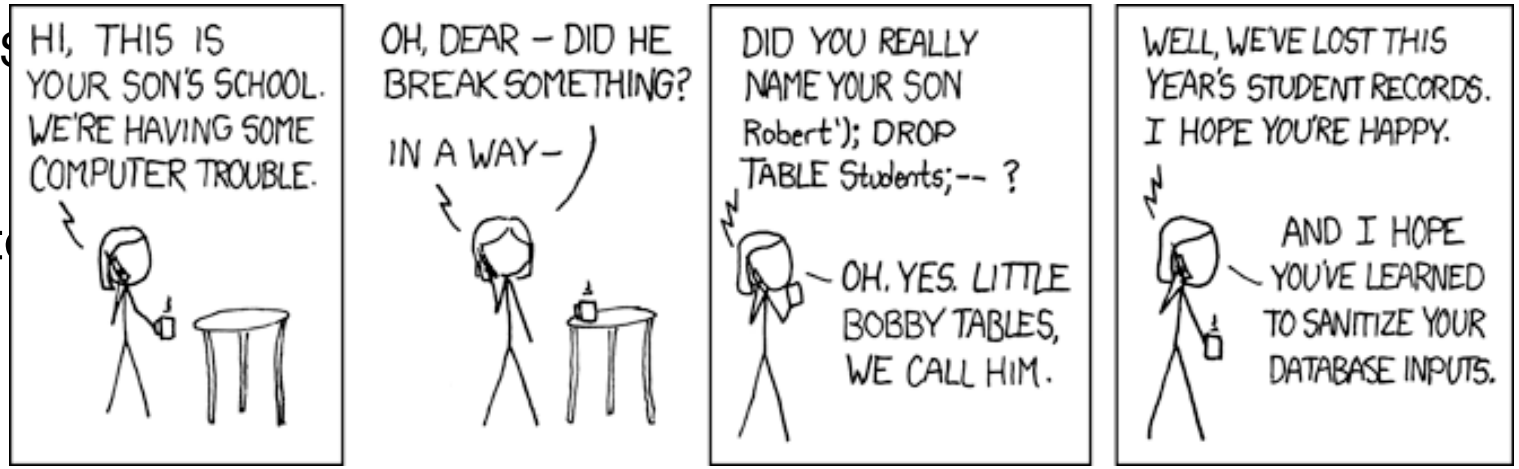
- Think about how your program might fail, not just succeed
  - Because the bad guys are going to look there
- "Edge cases" are where your problems likely lie
  - Either barely erroneous or barely correct
  - E.g. if your function accepts strings up to length ***n***
    - Be sure to test lengths 0, 1, n-1, n, n+1, 2n-1, 2n, and 2n+1
- A good guide by @eevee:
  - https://eev.ee/blog/2016/08/22/testing-for-people-who-hate-testing/

This Applies to Both Sides...

- When making your program robust, think like an attacker would
  - "Hmm, what if I spew random junk?"
  - "Hmm, what if I go for obvious corner cases?"
- When attacking software, think like a dumb programmer?
  - "Hmm, what mistakes have I made in the past? Lets try those!"

Try to Eliminate entire classes of problems

- Stack Overflows:
  - Turn on compiler protections
- Memory corruption attacks more generally
  - Use a safe language
  - Or barring that, turn on ALL defenses:
    - W^X/DEP + 64b AS
- SQL Injection
  - Only use paramet



Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …
- What's hard about patching?
  - Can require ***restarting*** production systems
  - Can ***break*** crucial functionality
    - Vendor ***regression tests*** should prevent this but don't always!
  - Management burden:
    - It never stops (the "***patch treadmill***")
  - User burden:
    - "Flaw in Flash, you need to manually update it..."
- But absolutely essential: 0-days are pricey, N-days are free

News

info security
STRATEGY /// INSIGHT /// TECHNIQUE

IT administrators give thanks for light Patch Tuesday

07 November 2011

Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

Only one of the bulletins is rated critical by Microsoft, which addresses a flaw that could result in remote code execution attacks for the newer operating systems – Windows Vista, Windows 7, and Windows 2008 Server R2.

The critical bulletin has an exploitability rating of 3, suggesting

Working Towards Secure Systems

- Along with securing individual components, need to keep them up to date …
- What's hard about patching?
  - Can require restarting production systems
  - Can break crucial functionality
  - Management burden:
    - It never stops (the "patch treadmill") …
    - … and can be difficult to track just what's needed where
- Other (complementary) approaches?
  - Vulnerability scanning: probe your systems/networks for known flaws
  - Penetration testing ("pen-testing"): pay someone to break into your systems …
    - … provided they take excellent notes about how they did it!

RISK ASSESSMENT / SECURITY & HACKTIVISM

ars technica

Extremely critical Ruby on Rails bug threatens more than 200,000 sites

Servers that run the framework are by default vulnerable to remote code attacks.

by **Dan Goodin** - Jan 8 2013, 4:35pm PST

HARDENING 38

Hundreds of thousands of websites are potentially at risk following the discovery of an extremely critical vulnerability in the Ruby on Rails framework that gives remote attackers the ability to execute malicious code on the underlying servers.

The bug is present in Rails versions spanning the past six years and in default configurations gives hackers a simple and reliable way to pilfer database contents, run system commands, and cause websites to crash, according to Ben Murphy, one of the developers who has confirmed the vulnerability. As of last week, the framework was used by more than 240,000 websites, including Github, Hulu, and Basecamp, underscoring the seriousness of the threat.

"It is quite bad," Murphy told Ars. "An attack can send a request to any Ruby on Rails sever and then execute arbitrary commands. Even though it's complex, it's reliable, so it will work 100 percent of the time."

Murphy said the bug leaves open the possibility of attacks that cause one site running rails to seek out and infect others, creating a worm that infects large swaths of the Internet. Developers with the Metasploit framework for hackers and penetration testers are in the process of creating a module that can scan the Internet for vulnerable sites and exploit the bug, said HD Moore, the CSO of Rapid7 and chief architect of Metasploit.

Maintainers of the Rails framework are urging users to update their systems as soon as possible to

Reasoning About Safety

- How can we have confidence that our code executes in a safe (and correct, ideally) fashion?
- Approach: build up confidence on a function-by-function / module-by-module basis
- Modularity provides boundaries for our reasoning:
  - **Preconditions**: what must hold for function to operate correctly
  - **Postconditions**: what holds after function completes
- These basically describe a contract for using the module
- Notions also apply to individual statements (what must hold for correctness; what holds after execution)
  - Stmt #1's postcondition should logically imply Stmt #2's precondition
  - Invariants: conditions that always hold at a given point in a function

```
int deref(int *p) {
    return *p;
}
```

Precondition?

```
/* requires: p != NULL
                (and p a valid pointer)
*/
int deref(int *p) {
    return *p;
}
```

Precondition: what needs to hold for function to operate correctly

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

Postcondition?

```
/* ensures: retval != NULL (and a valid pointer) */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

Postcondition: what the function promises will hold upon its return

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

Precondition?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access?

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* ?? */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires?

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

Let's simplify, given that a never changes. (It gets ***much worse*** when we have multiple threads)

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

The 0 <= i part is clear, so let's focus for now on the rest.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

How to prove our candidate invariant?
n <= size(a) is straightforward because n never changes.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

What about i < n ?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

What about i < n ? That follows from the loop condition.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

At this point we know the proposed invariant will always hold...

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
         0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

… and we're done!

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
         0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

A more complicated loop might need us to use *induction*:

Base case: first entrance into loop.

Induction: show that *postcondition* of last statement of loop plus loop test condition implies invariant.

```
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
     size(a) >= n &&
           ???
*/
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
         total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
      size(a) >= n &&
      for all j in 0..n-1, a[j] != NULL
*/
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
         total += *(a[i]);
    return total;
}
```

```
char *tbl[N]; /* N > 0, has type int */
```



```
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct postcondition for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
s)--0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <=? h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                        /* 0 <= h */
  while (*s)                         /* 0 <= h */
    h = 257*h + (*s++) + 3;          /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                          /* 0 <= h */
  while (*s)                           /* 0 <= h */
    h = 257*h + (*s++) + 3;            /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                          /* 0 <= h */
  while (*s)                           /* 0 <= h */
    h = 257*h + (*s++) + 3;            /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```



```
/* ensures: 0 <= retval && retval < N */
unsigned int hash(char *s) {
  unsigned int h = 17;          /* 0 <= h */
  while (*s)                    /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  unsigned int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
void foo(int *a){
  int i, j, sum;
  sum = 0;
  j = 0;
  for(i = 1; i < 10; ++i){
    sum += a[j];
    j = a[j];
  }
}
```

Common Coding Errors

- Memory safety vulnerabilities
  - In a "safe" language they cause immediate faults
    - May result in a "denial of service", aka crash, but not control flow hijack
  - In an "unsafe" language they may cause unpredictable (and likely exploitable) errors
- Input validation vulnerabilities
  - "You mean you ***trusted*** me when I said my name was "robert'); drop table students;--??!?"
- Time-of-Check to Time-of-Use (TOCTTOU) vulnerability (later)

Input Validation Vulnerabilities

- Program requires certain assumptions on inputs to run properly
- Programmer forgets to check inputs are valid => program gets exploited
- Example:
  - Bank money transfer: Check that amount to be transferred is non-negative and no larger than payer's current balance
  - SQLi: Accept string as input into an SQL command
  - Format String vulnerability: Accept string as the format string for printf

If Time Left: Real World Security Research: Click Trajectories