Memory Safety (cont'd)
Software Security

CS 161: Computer Security

Prof. Raluca Ada Popa

January 17, 2016

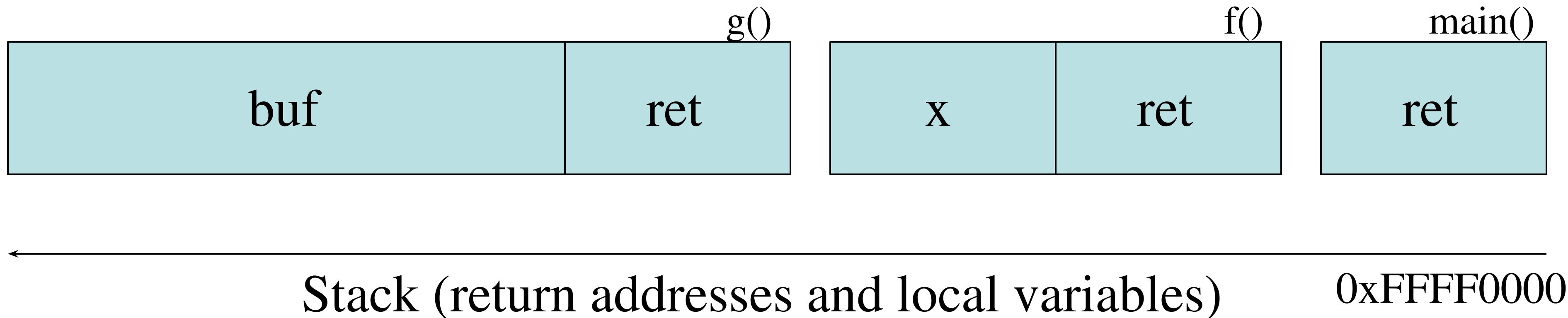
Some slides credit to David Wagner and Nick Weaver

Announcements

- Discussion sections and office hours start next week
- Join Piazza
- Homework 1 out Monday, due next Monday
- CS 61C Review session: Friday (tomorrow) 6:30-8:30pm in Soda 306

Memory safety (cont’d)

Recall: code Injection

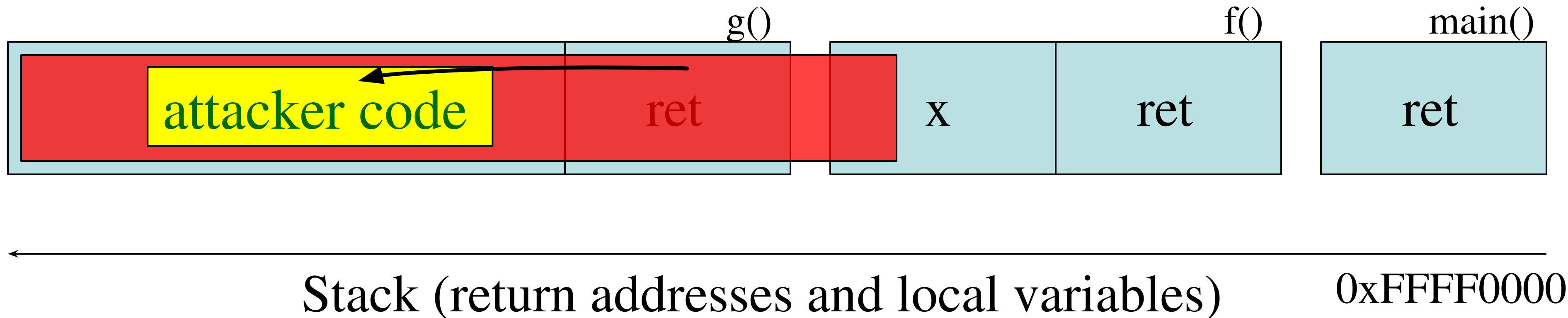



```
main() {
  f();
}
```

```
f() {
  int x;
  g();
}
```

```
g() {
  char buf[80];
  gets(buf);
}
```

Recall: code Injection




```
main() {
  f();
}
```

```
f() {
  int x;
  g();
}
```

```
g() {
  char buf[80];
  gets(buf);
}
```

Basic Stack Exploit

- Overwriting the return address allows an attacker to redirect the flow of program control.
- Instead of crashing, this can allow *arbitrary* code to be executed.
- Example: attacker chooses malicious code he wants executed ("shellcode"), compiles to bytes, includes this in the input to the program or part of the buffer overflow so it will get stored in memory somewhere, then overwrites the return address to point to it.

Defenses

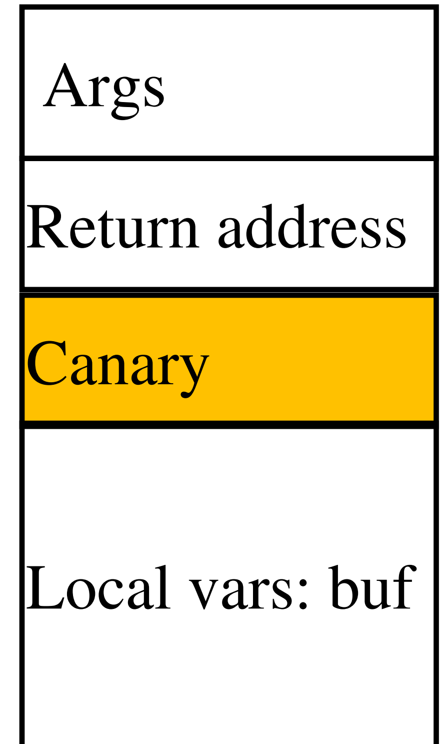
- Discuss with your partner some ideas

Defense #1

- The real solution to these problems is to avoid C or C++ if you can. Use memory safe languages such as: Java, Python, Rust, Go, …, which check bounds and don’t permit such overflows
- Still, a lot of code is written in C
  - Performance
  - Legacy code
  - Low level control

Defense #2

- Insert a **canary** = a random value just before the return address in each stack frame
  - Before returning, check that the canary still has the unmodified stored value



Q: Why below return address and not after?

A: to prevent return address overwrite without modifying canary

Q: Why random and not a fixed value 0x324a0b?

A: so attacker does not know it

Q: Even with canary, how could an attacker read the return address with a buffer overflow?

A: buffer overrun in inputs, args, that copies arg value at negative indices into some buffer returned to attacker

Defense #3: Non-Executable Stack Space…

- Make stack non-executable
- The overwritten return address from the attacker could point to code on stack which was similarly injected via a buffer overflow attack. With the stack nonexecutable, this code cannot execute

Q: does it protect against all buffer overflows?
A: No. For example, it does not protect against those that overwrite variables such as passwords, and others.

Defense #4:
Data Execution Protection/ W^X (write or execute)

- Ensure each piece of memory is either writeable or executable but not both
  - Q: What does this stop?
  - A: So an attacker can no longer inject code and execute it
- But some attacks are still possible: return oriented programming when the return address points to an existing snippet of code such as standard libraries like libc
  - Set up a series of return statements to execute “gadgets” in the code
  - This is not easy to understand, we won’t go into detail but there are tools to do this for you automatically: ROPgadget
  - Open source: https://github.com/JonathanSalwan/ROPgadget/tree/master

Idea #4: Lets make that hard to do

- Address Space Layout Randomization…
  - Randomized where library code and other text segments are placed in memory
  - Q: Why?
  - A: so the attacker does not know the address to “return” to

- Particularly powerful with W^X
  - Since bypassing W^X requires only executing existing code, which requires knowing the address of existing codes, but ASLR randomizes where the existing code is.
- Good idea but…if you can get the address of a single function in a library, you’ve defeated ASLR and can just generate your string of ROP gadgets at runtime

Idea #5: Write "Secure" code…

- Always bounds check, think of type overflow
- Difficult in C..

If nothing works…

Just run machine learning…

[joking]

Software security

Why does software have vulnerabilities?

- Programmers are humans. And humans make mistakes.

Why does software have vulnerabilities?

- Programmers are humans.
  And humans make mistakes.

- Programmers often aren’t security-aware.

- Programming languages aren’t designed well for security.

Why does software have vulnerabilities?

- Programmers are humans.
  And humans make mistakes.
  - Use tools.
- Programmers often aren't security-aware.
  - Take CS 161 ;-P
  - 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?

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*)

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: change certain statements in the source code and see if the tests find the errors
  - Spec-driven: test code of a function matches spec of that function
- How do we tell when we've found a problem?
  - Crash or other deviant behavior; now enable expensive checks

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
  - Management burden:
    - It never stops (the "*patch treadmill*") …

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 …

arstechnica

RISK ASSESSMENT / SECURITY & HACKTIVISM

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
- These 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?
(what needs to hold at the time of entering the function for the function to operate correctly)

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

Precondition?
(what needs to hold at the time of entering the function for the 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

```c
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

```c
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?

```c
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.

```
/* 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;
}
```

```c
/* 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:

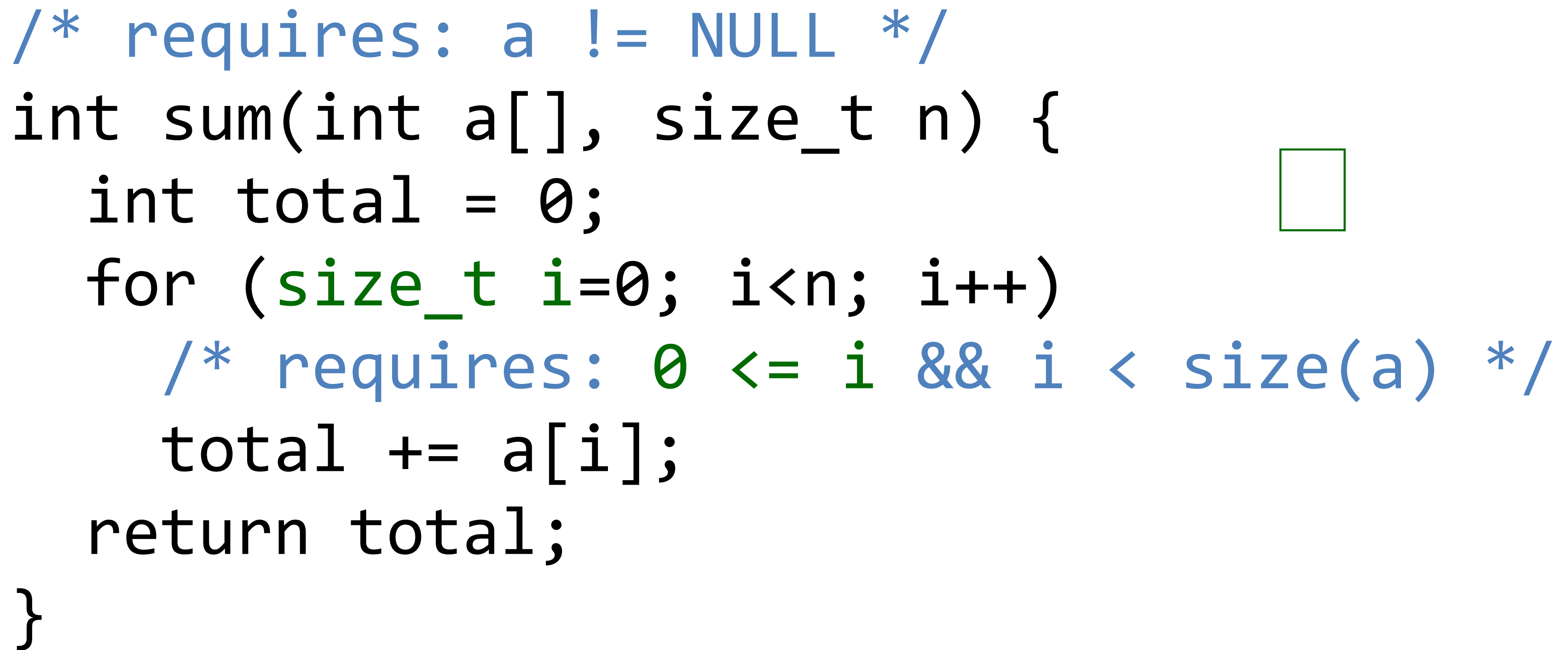
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* 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;
}
```

?

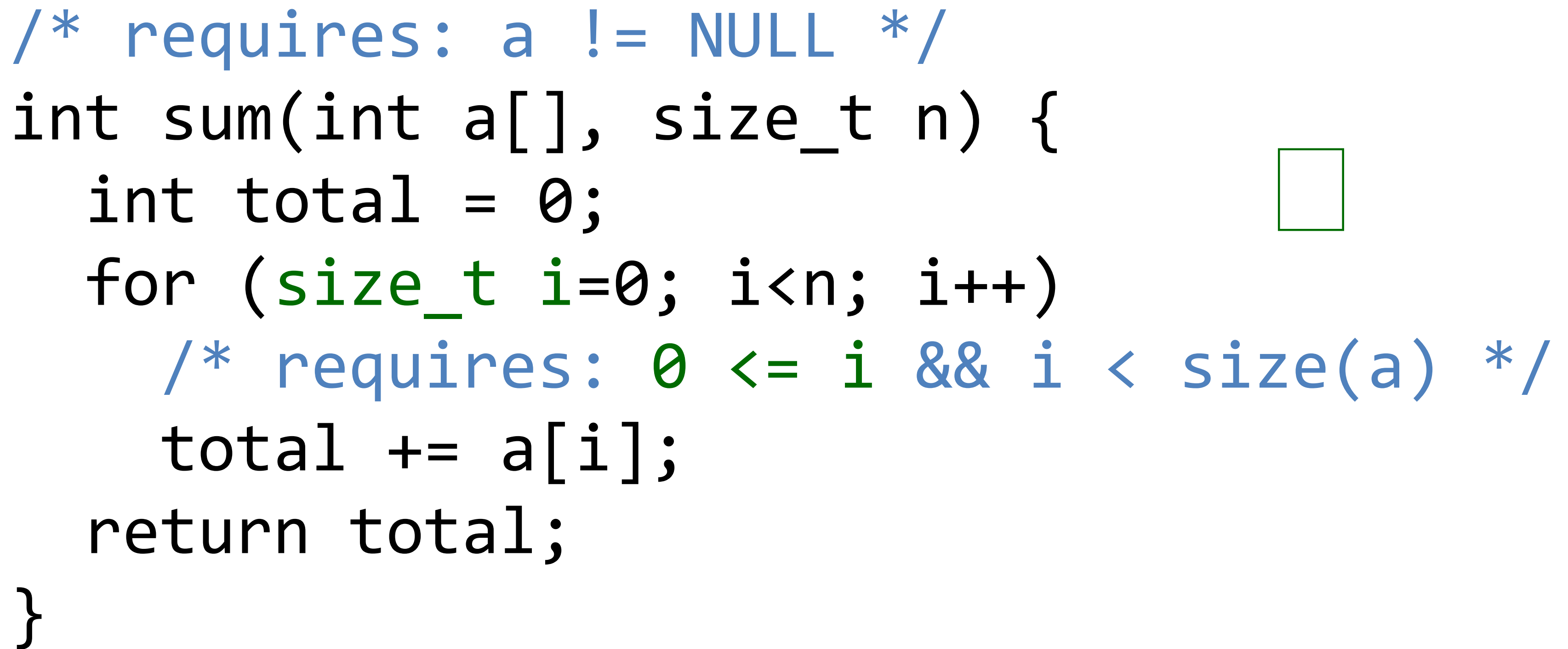
General correctness proof strategy for memory safety:

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



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?



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

```c
/* 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.

Questions?