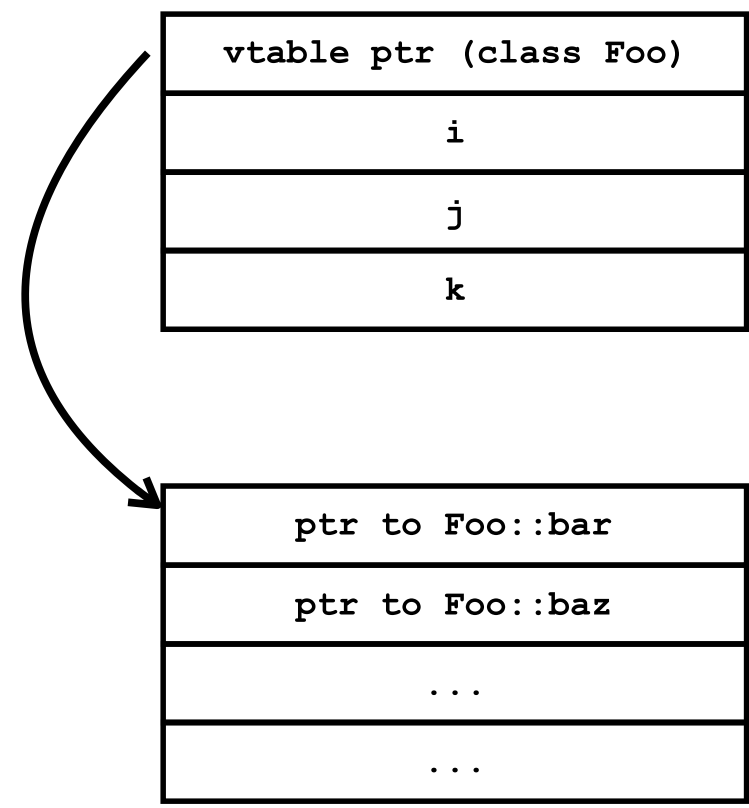
Software Security Continued

It isn't just the stack...

- Control flow attacks require that the attacker overwrite a piece of memory that contains a pointer for future code execution
  - The return address on the stack is just the easiest target
- You can cause plenty of mayhem overwriting memory in the heap...
  - And it is made easier when targeting C++
- Allows alternate ways to hijack control flow of the program

Compiler Operation: Compiling Object Oriented Code


```
class Foo {
   int i, j, k;
   public virtual void bar(){ ... }
   public virtual void baz(){ ... }
....
```



So Targets For Overwriting...

- If you can overwrite a vtable pointer…
  - It is effectively the same as overwriting the return address pointer on the stack:
    When the function gets invoked the control flow is hijacked to point to the attacker's code
    - The only difference is that instead of overwriting with a pointer you overwrite it with a pointer to a table of pointers...
- Heap Overflow:
  - A buffer in the heap is not checked:
    Attacker writes beyond and overwrites the vtable pointer of the next object in memory
- Use-after-free:
  - An object is deallocated too early:
    Attacker writes new data in a newly reallocated block that overwrites the vtable pointer
  - Object is then invoked

Magic Numbers & Exploitation…

- Exploits can often be ***very*** brittle
  - You see this on your Project 1: Your ./egg will not work on someone else's VM because the memory layout is different
- Making an exploit robust is an art unto itself: e.g. EXTRABACON…
- EXTRABACON is an NSA exploit for Cisco ASA "Adaptive Security Appliances"
  - It had an exploitable stack-overflow vulnerability in the SNMP read operation
  - But actual exploitation required two steps:
    Query for the particular version (with an SMTP read)
    Select the proper set of magic numbers for that version

A hack that helps: NOOP sled...

- Don't just overwrite the pointer and then provide the code you want to execute...
- Instead, write a large number of NOOP operations
  - Instructions that do nothing
- Now if you are a ***little*** off, it doesn't matter
  - Since if you are close enough, control flow will land in the sled and start running...

ETERNALBLUE(screen)

- ETERNALBLUE is another NSA exploit
  - Stolen by the same group ("ShadowBrokers") which stole EXTRABACON
- Eventually it was very robust...
  - This was "god mode":
    remote exploit Windows through SMBv1 (Windows File sharing)
- But initially it was jokingly called ETERNALBLUESCREEN
  - Because it would crash Windows computers more reliably than exploitation.

```
Plugin Category: Special
========================
```

Current and former officials defended the agency's handling of EternalBlue, saying that the NSA must use such volatile tools to fulfill its mission of gathering foreign intelligence. In the case of EternalBlue, the intelligence haul was "unreal," said one

The NSA also made upgrades to EternalBlue to address its penchant for crashing targeted computers — a problem that earned it the nickname "EternalBlueScreen" in reference to the eerie blue screen often displayed by computers in distress.

```
[!] plugin variables are valid
[?] Prompt For Variable Settings? [Yes] :
```

Reasoning About Memory Safety

- Memory Safety: No accesses to undefined memory
  - "Undefined" is with respect to the semantics of the programming language
- Read Access:
  - An attacker can read memory that he isn't supposed to
- Write Access:
  - An attacker can write memory that she isn't supposed to
- Execute Access:
  - An attacker can transfer control flow to memory that they isn't supposed 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 (this particularly matters for loops)

```
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.

Needs to be expressed in a way that a *person* writing code to call the function knows how to evaluate.

```
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.

Likewise, expressed in a way that a person using the call in their code knows how to make use of.

```
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;
}
```

size(X) = number of *elements* allocated for region pointed to by X
size(NULL) = 0

This is an abstract notion, *not* something built into C (like **`sizeof`**).

Gene

(1) Id

(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.

```
/* 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.

```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?: i < n && n <= size(a) */
    /* requires: i < 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;
}
```

This may still be memory ***safe***
but it can still have undefined behavior!

```
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.

```
=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.

```
=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.

```
=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.

```
=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.

```
=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 */
  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);
}
```

Or an alternative: *FFS Don't Use C or C++!!!!*

- Do you honestly think a human is going to go through this process for all their code?
  - Because that is what it takes to prevent undefined memory behavior in C or C++
- Instead, use a safe language:
  - Turns "undefined" memory references into an immediate exception or program termination
  - Now you simply don't have to worry about buffer overflows and similar vulnerabilities
- Plenty to chose from:
  - Python, Java, ***Go (project 2)***, Rust (if you need C's mostly-deterministicish performance), Swift... Pretty much everything other than C/C++/Objective C