Arguments and Return Value

- **Return value**
  - in register, typically r0
- **Arguments**
  - in registers or on stack

Snippet 8 - An example

```assembly
b:   ld   $0xfffffff8, r0  # r0 = -8 (frames size)
     add  r0, r5           # create frame on stack

dealloc bar frame (1)
return

allocate bar frame (2)
body
dealloc bar frame (2)
return
```

Arguments on Stack (s9-args-stack.s)

```
before jump to frame Foo
```

Arguments in Registers (s9-args-reggs.s)

```
before jump to frame B
```

Arguments and Return Value

- **Return value**
  - in register, typically r0
- **Arguments**
  - in registers or on stack

Creating the stack

Every thread starts with a hidden procedure
- Its name is start (or sometimes something like crt)
- 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

```
import java.lang.Thread;
public class A {
    static int add (int a, int b) { ret addr: $oneret }
    int add (int a, int b) { return a+b; }
    static int foo (int x) { return 2*x; }
}

void bar (int x) {
    // r5 = 2000
    one ();
}
```

Arguments and Return Value

- **Return value**
  - in register, typically r0
- **Arguments**
  - in registers or on stack

Creating the stack

Every thread starts with a hidden procedure
- Its name is start (or sometimes something like crt)
- 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

```
import java.lang.Thread;
public class A {
    static int add (int a, int b) { ret addr: $oneret }
    int add (int a, int b) { return a+b; }
    static int foo (int x) { return 2*x; }
}

void bar (int x) {
    // r5 = 2000
    one ();
}
```

Arguments on Stack (s9-args-stack.s)

```
before jump to frame Foo
```

Arguments in Registers (s9-args-reggs.s)

```
before jump to frame B
```

Arguments and Return Value

- **Return value**
  - in register, typically r0
- **Arguments**
  - in registers or on stack

Creating the stack

Every thread starts with a hidden procedure
- Its name is start (or sometimes something like crt)
- 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

```
import java.lang.Thread;
public class A {
    static int add (int a, int b) { ret addr: $oneret }
    int add (int a, int b) { return a+b; }
    static int foo (int x) { return 2*x; }
}

void bar (int x) {
    // r5 = 2000
    one ();
}
```

Arguments on Stack (s9-args-stack.s)

```
before jump to frame Foo
```

Arguments in Registers (s9-args-reggs.s)

```
before jump to frame B
```
Security Vulnerability in Buffer Overflow

Find the bug in this program

 void printPrefix (char* str) {
  char buf[18];
  // copy str up to '.' input buf while (str[0]){
  char* bp = buf;
  // 
  for (int i = 0; str[i] != 0; i++) {
    // a[i] < ' ' 
    // char = buf[i + 1];
    buf[i + 1] = str[i];
  }
  buf[17] = 0;
  puts(buf);
}

The "overflow" buffer bug

• if the position of the first '.' in str is more than 10 bytes from the beginning of this loop it will write portions of str into memory beyond the end of buf

Finding Location of Return Address

use debugger with long test string to see return address when it crashes
• keep "converse" output from printPrefix()

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]
The address is 0xbfff140

Write Worm: Part 2

void write_worm () {
  char str[1024];
  // copy str into buf
  strcpy(str, buf);
  // copy str up to '.' input buf
  for (int i = 0; str[i] != 0; i++){
    str[i] = buf[i + 1];
  }
}

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 procedure prologue to combine 2 machine code
• enter machine code as data in your input string

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 calling procedure

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
• store your input in memory, ignoring how it will get there for real attack
• a portion of the stack can be used for local variables and saved registers (e.g., save r6)
• procedure prologue allocates space on stack for local variables and saved registers
• right after procedure call deallocates space on stack used for arguments
• accesses local variables and arguments
• static offset from stack pointer (e.g., r5)
• grows from bottom up
• stack is managed by code that the compiler generates
• push by subtracting
• accessing local variables and arguments
• gets return value (if any) from ebp

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

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 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

Applying Techniques

• put worm address in r6
• jump to worm

Why is this so ugly
• the attacker can change printPrefix’s return address
• buf[0] can overwrite return address frame (that should go back to caller)
• what power does this give the attacker

Approximate Locations

• post 3: send the worm around the world (please don’t)