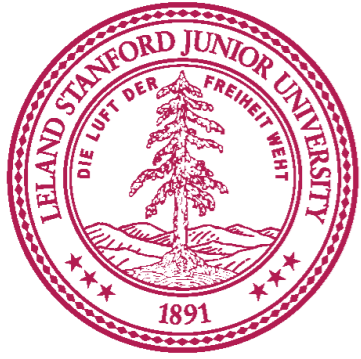


# CS 155 Project I

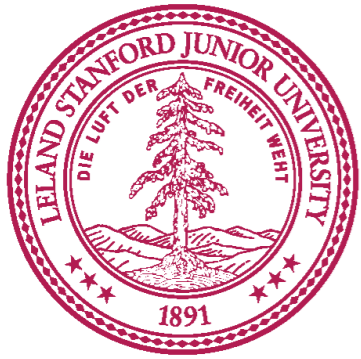
Gary Luu  
Spring 2008





# Setting up the Environment

- Download VMware Player
  - <http://www.vmware.com/products/player>
- Available on PUP Cluster
- If prompted, click “I copied it”
- Should be configured with NAT, check w/ ifconfig
- Demo



# target1.c

```
int bar(char *arg, char *out)
{
    strcpy(out, arg);
    return 0;
}

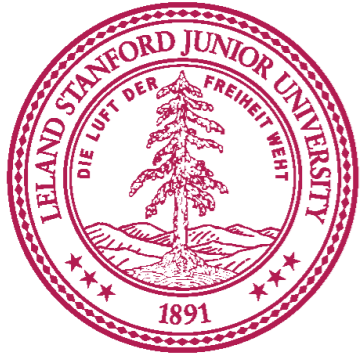
int foo(char *argv[])
{
    char buf[128];
    bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}
```



# Stack During Call to foo

- Target the local buffer “buf” inside of foo
- What’s on the stack after the end of “buf?”
- Stack layout dependent on OS and compiler
- arguments to foo, then return address, then saved frame pointer, then “buf”
- Explore stack with gdb, read “Smashing the Stack”



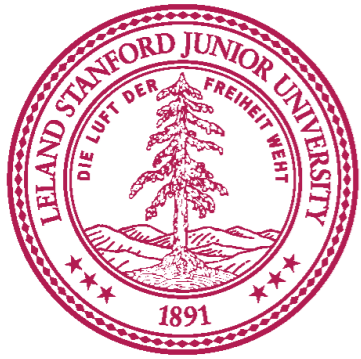
# sploit1

- Want to overwrite return address of foo()
- Need to insert shellcode in “buf”
- Distance from “buf” and return address on stack
  - Remember, this is dependent on compiler/OS
  - Make sure your exploits work in VM!



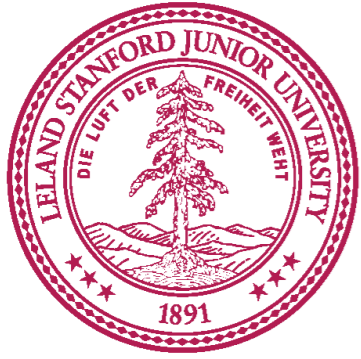
# Address of “buf”

- How to obtain?
  - Examine stack frame using “info frame”
  - Use “x buf” when in foo’s frame
- Stays the same everytime program is invoked
  - Address changes when invoked from exec()
  - Get address using `gdb -e sploit1 -s /tmp/target1`



# Crafting the Exploit String

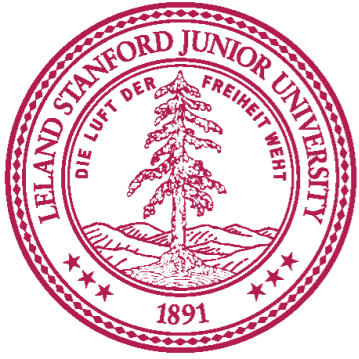
- Place shellcode at the start of the string
- Return address (\$ra, or saved \$eip) exists at offset 132 on our VM
  - 128 bytes of buf, 4 bytes frame pointer
  - Write address of “buf” to \$ra, 0xbffffd78
- Remember to null terminate your string (strcpy)



# Hints

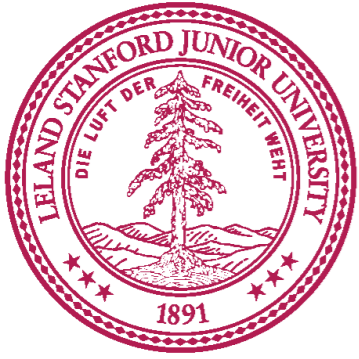
- There are other ways to attack besides overwriting the return address
- Understand what assembly instructions are doing
  - README contains links to Intel x86 assembly manuals
  - Understand what registers \$esp, \$ebp point to
  - What happens when LEAVE and RET called?





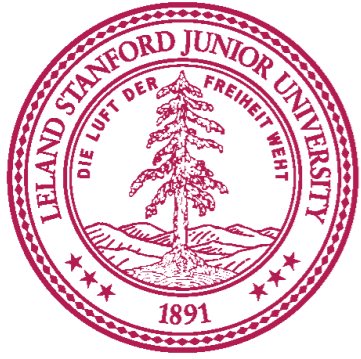
# IA-32 Review

- x86 is little endian
- `$esp`: Stack Pointer: points to the top of stack (which way does the stack grow on x86?)
- `$ebp`: Frame Pointer: points to fixed location within an activation record
  - Used to reference local vars and parameters since the distance from the frame pointer to these objects stays constant, while stack pointer changes
- `$eip`: instruction pointer (aka `$ra`) (`$ebp+4`)



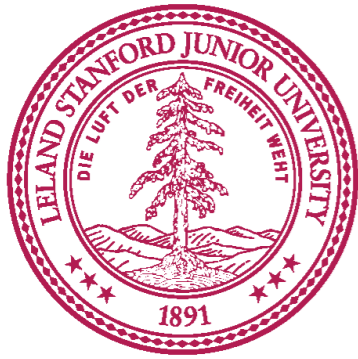
# IA-32 Review (cont'd)

- When CALL procedure foo()
  - Push \$eip onto stack, (return address)
  - Push \$ebp, saving previous frame
  - Copy sp into fp, \$ebp = \$esp
  - Decrement \$sp for allocations (like buffers)
- When LEAVE procedure p()
  - Process is reversed
  - Load \$ebp into \$esp
  - Restore \$ebp from stack



# Interaction Between \$esp, \$ebp, \$eip

- During CALL, value of \$eip register pushed onto stack
- Before RET, programmer should make sure stack pointer (\$esp) is pointing to saved \$eip on the stack
  - Move contents of \$ebp into \$esp
  - Increment \$esp by 4
  - \$esp should now point to address of saved \$eip
  - RET will load saved \$eip into \$eip register, processor will execute instruction in \$eip register



# Advice

- Start early, you'll need to read
  - “Smashing the Stack” - Aleph One
  - “Basic Integer Overflows”
  - “Exploiting Format String Vulnerabilities”
  - “How to hijack the Global Offset Table...”
  - “Once upon a free”
  - Reference IA-32 guide (on syllabus with papers)
- Part 2 MUCH harder than Part 1.
- Make a diagram of the stack using gdb