# Understanding Common Security Exploits
# Problem Set 1

The purpose of this exercise is to write an exploit for the simple program shown below. It can also be found at `http://www.mit.edu/iap/exploits/overflow.c`. The steps of this problem set will walk you through this process.

```
#include <stdio.h>
void foo(void)
{
  char buffer[4];
  gets(buffer);
}
main(void) {
  foo();
}
```

You may want to try answering the following questions for both Sun Solaris and for Intel Linux. We have confirmed that the program has an exploitable buffer overflow on both platforms. Note that the Athena dialups (running Solaris) have disabled executable stacks, so you won't be able to test your shell code there. Try using a cluster machine instead.

1. Identify the buffer overflow. Overflow the buffer and confirm you can get the program to dump core.

2. Find where the buffer is placed within the stack frame either by looking at the assembler input or by running the program in the debugger . Find the offset of the return address you are going to overflow from the end of the buffer. Remember that for SPARC you can only overflow return addresses that are spilled out of register windows.

3. Confirm that if you overflow the return address you get either a segfault on the return instruction or a segfault reading the next instruction. Confirm that if you do not overflow the return address but do overflow all the data up to that point, the behavior changes.

4. Now, it is time to write shell code so you have something to stick in the buffer. Start by writing a small C program that uses the `execve` syscall to start a shell. You can probably use an empty environment—that is the first member of the array of character pointers can be null. You want to use `execve` because it is actually a syscall; many of the other variants of `exec` are actually just C library functions. Confirm your program works correctly.

5. Compile your program statically linked against the C library. If you dynamically link it, you may have extra complexity to understand when jumping into the C library. Step through the program in the debugger and understand how the registers are set up for the call into the kernel.

6. Start working with the assembly output of the compiler. Replace the call to `execve` with code to set up registers correctly and trap into the kernel. Confirm this works.

7. Now, rework the code to be location independent. You will probably want to move your strings into the text segment instead of the data segment where they are generated. You will probably need to dynamically generate the `argv` and `environ` arrays on the stack. Confirm the new code works.

8. Construct an appropriate landing pad, so that your code works even if someone starts a few bytes into your code.

9. Extract the binary of your code into a file. Disassemble this file and make sure you didn't get extra code from somewhere else and that the extraction worked.

10. Now, construct input for the program that overflows the buffer, sets the return address to point to your code and includes your code to launch a shell. Confirm that a shell is started. Note many ways of giving the input to the program will end up starting a shell that immediately exits. Why? How can you work around this?