CSE 484 Lab1 FAQ and hints

Understanding the frame pointer (ebp)

The stack frame pointer is a register used to keep track of the *start* of the current stack frame. In x86 32-bit this is the `ebp` register.

Tracking the location of local stack variables

Understanding the usage of ebp is easier once you understand *why* it might exist.
Consider the following two approaches to how a compiled program can try to refer to local (stack) variables:

1. Explicit tracking of offsets throughout the function.

It could keep track of where the stack pointer is relative to the start of the stack frame, and do the computation *at compile time* for where a variable is.
For example, if our function has the following two local variables:

```
int local1; // sub 4, $esp
char buf[32]; // sub 32, $esp
```

And it now wants to refer to `local1`, the compiler knows that will start at `$esp+32`, so will use that when trying to load or store to `local1`

If later in that same function we allocate more locals:

```
char buf2[8]; //sub 8, $esp
```

and then the function wants to refer to `local1` again, we need to refer to it differently (`$esp + 40`). This is entirely doable, and there are numerous clever optimizations for this.

2. Static reference point for the function duration.

We could instead, keep track of the START of the stack frame (with a frame pointer: `$ebp`), and then always use that as a static reference point.
In this case, `local1` is always going to be at `$ebp-4`, and buf will always be at `$ebp-36`, regardless of how many more allocations are made in this stack frame.

So what is the *saved* frame pointer?

Traditionally, programs used the frame pointer. These days compilers and such are good enough it is not necessary, we can compute the offsets at compile time.

Net effect: if the program is compiled with a stack frame pointer (e.g. we DIDN'T pass `-fomit-frame-pointer`) then it is going to use `$ebp` to keep track of the start of the current frame.

When we call a new function, we are then going to need to set `$ebp` to the top of *this new* stack frame. Which means we need to save the *old* `$ebp` somewhere. And that is the 'saved frame pointer' that we have in our stack diagrams

`snprintf` exploitation

FAQs and hints

It is worth rewatching the lecture recordings, and working through simple examples on paper+gdb. Understanding how variable argument functions in C work is critical to making this exploit work.
Solve the `printftoy` programs. Debug them and understand why they work. Solving them but not understanding why will likely not help.
Write small programs that just call `printf` so you understand exactly how it works under normal circumstances. Same for `snprintf`. (If you are having trouble replicating the build environment, just edit the `printftoy` programs.)

Background and reminders

Reminders for calling conventions

Normal functions

All functions in Lab 1 use classic C calling conventions: All arguments are pushed onto the program stack before calling, in reverse order.
E.g.
`foo(int a, int* bptr)`
When called from `bar()` will have a stack like

```
[...]
[end of bar's local variables]
[bptr (pointer)]
[a (int)]
---------------^ bar's frame above, foo's below ------------------
[saved return address that points into bar's code]
[saved ebp/frame pointer that points into the stack frame for bar]
[...]
```

Variable argument functions

When you call a *variable argument function* this means it is unclear how many arguments are on the stack! E.g. `printf(char* format_string, ……)`
We just don't know how many arguments will be there until we parse the `format_string`!

A good call to `printf()` from `baz()` might be: `printf("Hi %s, today's number is %i\n", username, 5);`

```
[...]
[end of baz's local variables]
[5 (integer]
[username (pointer)]
[formatstring (pointer)]
---------------^ bar's frame above, printf's below ------------------
[saved return address that points into baz's code]
[saved ebp/frame pointer that points into the stack frame for baz]
[...]
```

If `format_string` is supplied by an *adversary* then `printf` is going to get confused. We assume that `baz()` called `printf(attackerstring);` here.

```
[...]
[end of baz's local variables]
[formatstring (pointer)]
---------------^ bar's frame above, printf's below ------------------
[saved return address that points into baz's code]
[saved ebp/frame pointer that points into the stack frame for baz]
[...]
```

If your format string contains any `%` operators, then `printf` is going to go rummaging around in `baz`'s stack frame expecting data there to be arguments!

Try working with `printftoy` at this point. Make sure you understand *exactly* why it prints what it does, when it does. Draw the stack frames.

`printf`'s operators

Read the `printf` manual pages, the format string attacks paper, and try using them in a normal program.

You are mostly going to need *integer printing* specifiers, and the weird `%n` operator.

Integer ops

We generally use `%x` in class. It prints an integer, in hexadecimal.

`%p` is also useful. It is a shorthand for `0x%08x` (e.g. print out the value in a standard-looking pointer format, sized correctly for the machine.)

Pointer ops

You will also see us use `%s`. It expects a string as an argument, e.g. a `char*`. It will follow that pointer and print it as a string.

Formatting specifiers

You can specify a required number of characters to print, e.g. `%8x` will always print 8 characters no matter the number. `printf("%8x",1)` will print `       1` … which is annoying to read, so we usually specify what padding to use, and want it to be `0`. So `printf("%08x",1)` will print `0000001`.
There are other very useful specifiers. Read through the format strings paper!

%n and its internals

`%n` is the only `printf` operator that *writes* to an argument, rather than *reading* an argument. It expects a `int*` as the argument, and will write the number of characters printed thus far to it.

`%n` expects a pointer as an argument (e.g. `printf("abc%n", &someint);`)

It then writes the number of characters printed thus far to that pointer (so after that above line, `someint == 3`.)

If you have a target value you want to write you calculate how many characters have been printed thus far up to the `%n`, and then figure out how many more you need to print.

E.g. if I want to store the value `0x34` to `someint` above, I would need to print `52-3=49` more characters. One way to do that would be: `printf("abc%49x%n",someval, &someint);`
Now it is printing `3+49` characters, so the `%n` will write `52` (`0x34`) to `someint`. Note however, the way we did that was to *add another % operator to the string.* That also required us reasoning about another *argument* to `printf`.

Exploitation and `snprintf`

Approaching exploitation

Read the format strings paper, rewatch the recorded lectures.

Done? Great! You should have some idea of the flow of this exploit.
Remember that we need to have an objective: probably to overwrite a saved return address on the stack somewhere.

`%n` is your one tool for writing values. You need to ensure that when it triggers, it gets a *pointer* to wherever you wanted to write to as its argument. Go make sure you can solve and understand `printftoy2` at this point. If you can't ask a question about that.

For your sploit6, you need to construct a stack that will provide the arguments you want it to for the `%` operators you create.

snprintf

`snprintf(char* out, size_t maxlen, char* fmt, …)` is `printf(char* fmt …)`, but with:
An output buffer (`out`) instead of printing to the terminal.
A max number of characters to write (`maxlen`) to that buffer.

If you aren't sure what its doing, try writing a small test program using `snprintf` correctly (e.g. not an attacker string.)

We have given you a useful situation as the attacker: the caller of `snprintf` put the output buffer *on the stack* right before it called `snprintf`. Think carefully about what that means for what arguments `snprintf` will read if it needs arguments due to the format string.

Debugging

Don't try debugging `snprintf` itself. You won't make it out the other side!

Instead, try debugging by observing the *output* of `snprintf`.
Once `snprintf` is done executing, you can print `buf` as a string (e..g `print buf`) or examine it byte by byte (e.g. `x /300x` or something).

Did `snprintf` crash instead of doing what you expected? Your string probably tried to use something as a pointer that wasn't a pointer! Swap your `%n` for a `%p`, and check what gets printed to the `buf`. Note this may not do what you'd like if your `%n` comes after printing a ton of characters… (`buf` does have a size limit!)

Another trick is to pick somewhere innocuous on the stack, and try and write values there before you try overwriting the saved return address.

Always check that your write did what you expect after `snprintf` finishes execution. Don't try and do more than 1 write until you have 1 write working successfully.