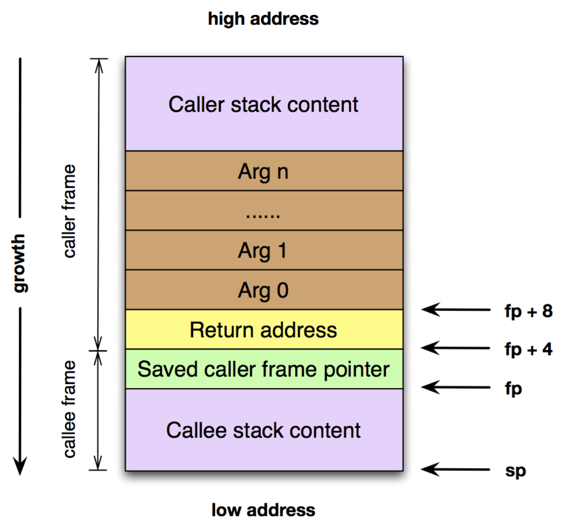
CS/240/Lab/6

Understanding the call stack is an important step towards mastering C. However, you usually do not have opportunities to exercise it, as stack management code is inserted by the compiler. The goal of this lab is to turn the call stack from an abstract, fuzzy concept in to a concrete entity.

Call stack

Understanding the layout of call stack is critical in this lab. The figure below shows a stack for 32-bit architecture that grows from high to low addresses. Upon a function call, the caller pushes the arguments for its callee onto the stack in reverse order and saves the return address. The callee is responsible for saving the caller's frame pointer (fp) and setting up its own frame and stack pointer (sp).



Q1: Spy the stack

You will need `callstack.c`, `tweetIt.h`, `libtweetIt.a` and `badguy1.c`. `callstack.c` is shown below, and must not be modified. Your task is to implement the `badguy` function in `badguy1.c` to print the call stack. Since compiler optimizations affect the size of the call stack, do not use `-O2` or any other optimization flags in this lab.

```
static char *correctPassword = "ceriaslyserious";
char *message = NULL;
int validate(char *password) {
    printf("Validating %p\n", password);
    if (strlen(password) > 128) return 0;
    char *passwordCopy = malloc(strlen(password) + 1);
    strcpy(passwordCopy, password);
    int valid = badguy(passwordCopy);
    return valid;
}
int check(char *password, char *expectedPassword) {
    return (strcmp(password, expectedPassword) == 0);
}
int main() {
    char *password = "wrongpassword";
    char *expectedPassword = correctPassword;
    if (!validate(password)) {
        printf("Invalid password!\n");
        return 1;
    }
    if (check(password, expectedPassword)) {
        if (message == NULL) {
            printf("No message!\n");
            return 1;
        } else {
            tweetIt(message, strlen(message));
            printf("Message sent.\n");
        }
    } else {
        printf("Incorrect password!\n");
    }
    return 0;
}
```

The code forms a call chain, starting from `main` up to `badguy`. `validate` validates that the password is usable, and it allows an external validation routine (`badguy`) to help. `check` checks that the password is actually correct, then sends a message via Twitter if one is available. At the moment `badguy` is invoked, `main` and `validate`'s call frames are already on the stack, so `badguy` is able to access all of their local data. It can print a list of <address>: <value> pairs using:

```
printf("%p: %x\n", p, *p); // p has type int*, pointing to an address on the stack
```

You must print the stack in `badguy`, from high to low addresses. The printout should start at the address of the `main`'s local variable `password` and end at the address where `validate`'s frame pointer is stored. In order to stop at the correct address, you should identify in each function 1) the argument, 2) the return address, and 3) the frame pointer.

Use trial and error to discover how large each call frame is. Remember that frame pointers are addresses, and those addresses point to locations on the stack, so the stored address should be close to the address where it's stored. (Hint: `p` should not be fairly close to `*p` when you've found a frame pointer.) So long as you don't modify `callstack.c`, the stack layout should be the same for every invocation, regardless of the content of the `badguy` function.

32-bit Code
This lab assumes that the compiled code is 32-bit (where pointers are 4 bytes wide). Since the lab machines are running a 64-bit environment, gcc generates 64-bit programs by default. You must add the flag `-m32` to the gcc command line to force it to generate 32-bit code. This should be part of your `CFLAGS`. Moreover, to prevent the compiler from attempting to mangle the stack for optimization, do not add any optimization flags. Adding `-O0` will explicitly disable all optimizations.

Makefile
You must provide a Makefile which contains the default target `all` that builds the program `badguy1` from `badguy1.c`, `callstack.c` and `libtweetIt.a`. The `CFLAGS` variable in your Makefile must contain `-m32` and `-O0`.

Q2: Send a tweet

After successfully spying on the stack, you're going to play a little trick and convince `main` to actually send a tweet, even though the password as entered is wrong. Change the values in its stack, as well as `message`, to send a tweet of your choice. Please note that you will need to allocate space for `message`, using a string literal won't work (`libtweetIt` requires that its strings be writable). This task must be achieved by modifying the stack, not by directly calling `tweetIt` or modifying other global state.
Implement the `badguy` function in `badguy2.c` to do this trick.

Makefile
Your Makefile's `all` target must also build `badguy2`, from `badguy2.c` and `callstack.c`.

Q3: Buffer overflow

You will now mount a simple buffer overflow attack. A program that has a buffer overflow bug carelessly allows users to write beyond the end of an array, into space used by the program for other purposes. An attacker can exploit such a bug to trick the program to do something unanticipated. Write code to attack the program given in `vulnerable.c`:

```
char* name = "abc";
void secret() { printf("Secret!\n"); exit(0); }
void wrong() { char buf[4]; strcpy(buf, name); }
void fence() { printf("fence\n"); }
int main(){ init(); fence(); wrong(); return 0; }
```

This program will print "fence" after calling `init` and the `secret` function will never be called. What you have to do is: with no modification to `vulnerable.c`, implement a malicious `init` function in `overflow.c` to get `secret` invoked after "fence" is printed. The output should be:

```
fence
Secret!
```

At the first glance, this seems impossible. However, there is buffer overflow bug in `wrong`. It does not check the length of the string held in `name` before invoking `strcpy`. As the variable `buf` is allocated on the stack, overflowing it can overwrite critical information such as the frame pointer, return address, etc. You job is to make sure that a buffer overflow does occur and that `secret` is called. (Hint: use function pointers!)

Some tips:

1. You will probably need to estimate with various string lengths for your attack. Attackers usually use a technique to increase the chance of success: they write the same value over again, rather than only writing it to the exact address to be exploited.
2. Remember that casting pointers is always legal, but the size of the values depends on what type you cast them to. Keep this in mind while trying to get pointers into a string.

Makefile
Your Makefile's `all` target must also build `overflow`, from `vulnerable.c` and `overflow.c`.

Turning in

As this submission includes multiple files, your submission will be in the form of a tarball. Make sure you include all necessary files, even the ones we provide.

```
tar zcf turnin.tgz Makefile source1.c source2.c ...
```

Lab is due Monday, February 27th before midnight. No late labs accepted.

Grading criteria

- source files `badguy1.c`, `badguy2.c`, and `overflow.c`
- Makefiles contain target `all` that builds `badguy1`, `badguy2`, and `overflow` respectively
- code compiles and runs without error
- the portion of call stack is printed as required
- badguy2 convinces callstack to send a tweet
- successfully carry out buffer overflow attack