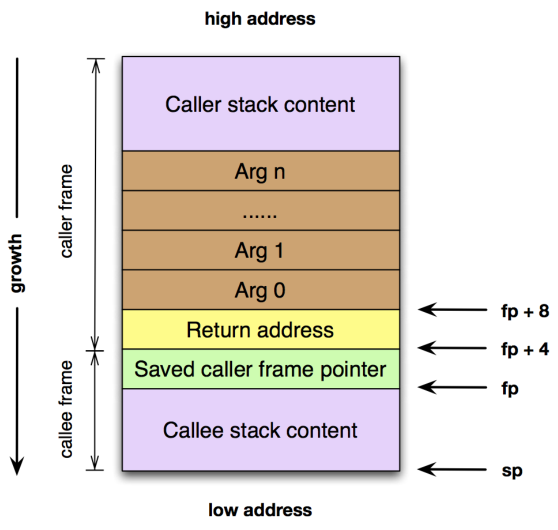
CS/240/Lab/7

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 on the stack

You will need `callstack.c`, `tweetIt.h`, `libtweetIt.a` and `badguy1.c`. `callstack.c` is shown below, and they must not be modified. Your task is to implement the `badguy` function in `badguy1.c` to print the call stack.

```
static char *correctPassword = "ceriaslyserious";
char *message = NULL;
int validateSanity(char *password) {
    for(int i=0;i<strlen(password);i++)
        if(!isalpha(password[i]))
            return 0;
    unsigned int magic = 0x12345678;
    return badguy(password);
}
int validate(char *password) {
    printf("--Validating something\n", password);
    if (strlen(password) > 128) return 0;
    char *passwordCopy = malloc(strlen(password) + 1);
    strcpy(passwordCopy, password);
    return validateSanity(passwordCopy);
}
int check(char *password, char *expectedPassword) {
    return (strcmp(password, expectedPassword) == 0);
}
int main() {
    char *password = "wrongpassword";
    unsigned int magic = 0xABCDE;
    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");
    }
}
```

The code forms a call chain, starting from `main` up to `badguy`. `validate` and `validateSanity` validate that the password is usable, and allow 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`, `validate` and `validateSanity`'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 `validateSanity`'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 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.

Compilation Directives and Note on 32-bit Code
Explicit compilation commands are not given in this lab. You have to figure out appropriate compilation commands based on your experience from previous labs and instructions that follow.

- 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` along with other flags like `-std=c99` to the gcc command line to force it to generate 32-bit code.
- 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.
- Since you experiment with stack in this lab, you have to explictly turn off gcc stack smashing protection mechanisms that disallow your experiments. You can do that by passing `-fno-stack-protector` flag to gcc.

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.

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`:

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

This program will print "fence" after calling `init` and the `print_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 `print_secret` invoked after "fence" is printed. The desired output is:

```
fence
Secret is dragonwarrior!
```

However, you will get full credit if you manage to invoke print_secret even without printing correct secret. That is, following output is also valid.

```
fence
Secret is <garbage>!
```

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 of the caller, return address, etc. You job is to make sure that a buffer overflow does occur and that `secret` is called while having the pointer to secret at appropriate address on the stack. Observe that main's secret is already on stack when init is being executed. (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. Note that parameters of a function are always at fixed offset from its frame pointer (fp). The C code does not make the offset explicit, but you can see it in the machine code generated by gcc. To generate human readable machine code from gcc, pass -S flag while compiling.
3. 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.

A Note on Makefiles
As you might have observed during your labs, keeping track of compilation flags for each file and manually building the dependencies when you make some change to your code is a very tough job and requires you to be very diligent. To automate this process there is a `make` utility in unix. When you run `make` on your command line, it looks for a file called `Makefile` in your current working directory. Makefile should contain instructions to build your code and create an executable. A link to tutorial on how to create makefiles is given on course webpage. Since the upcoming projects would require you to create Makefiles, it is a good exercise to create a simple Makefile for this lab that compiles your code and creates executables for all three questions in this lab. Your Makefile should contain separate targets to build separate executables and an `all` target that builds three executables, namely `badguy1`, `badguy2` and `overflow`. Essentially, your Makefile should respond correctly to following commands -

```
make badguy1
make badguy2
make overflow
make
```

Please note that your Makefile is not graded. This is an optional, although very useful exercise.

Turning in

Turnin your files using autograder. The lab is due Wednesday, October 17th before midnight. No late labs accepted.

Grading criteria

- source files are named `badguy1.c`, `badguy2.c`, and `overflow.c`
- 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 to invoke function print_secret