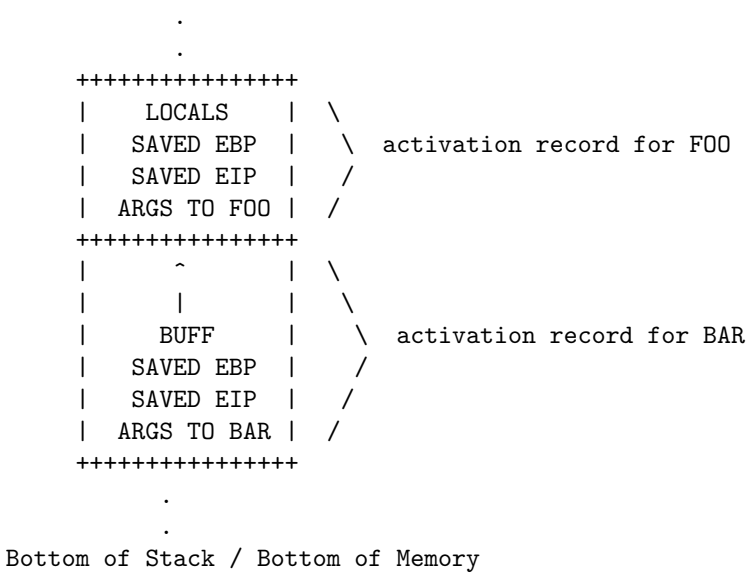
CS 155
Homework 1

Rafael Moreno Ferrer
SUID#: 05490330

Problem 1. *(Control Hijacking)*

a) In x86 the stack grows downwards. Explain how a stack-based overflow attack would work if the stack grew upwards instead.

For upward growing stacks overflowing an insecure buffer in the current activation record[1] will not generally result in a stack smashing attack since, at most, we will overwrite local variables but not any of the control data embedded in the stack (like saved eips, ebps or arguments). **However, stack smashing attacks can be done in this architecture by overflowing a buffer in a previous activation record.**



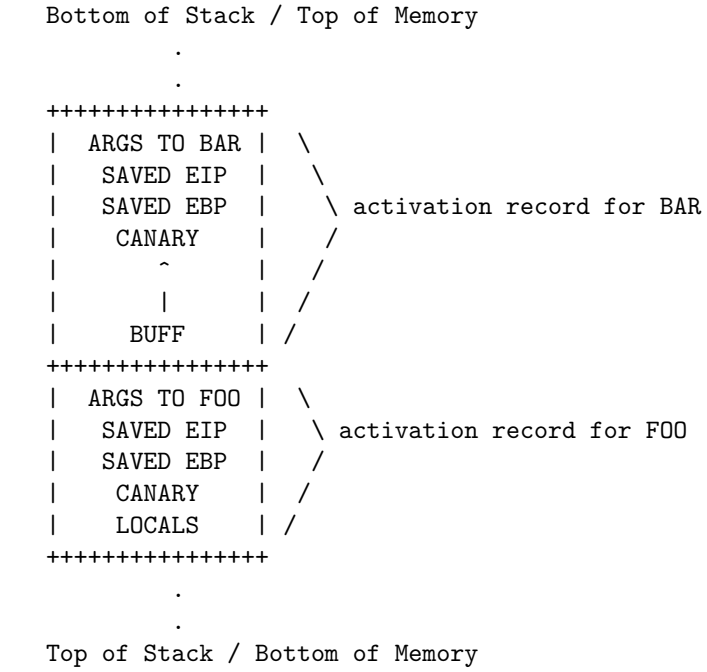
Suppose function BAR calls function FOO and function BAR has a local buffer BUFF. Inside function FOO there is code that copies data into BUFF unsafely. BUFF then can be overflown to overwrite the SAVED EIP for FOO and hijack control when FOO returns.

b) How would you implement StackGuard in an architecture where the stack grows upwards? What would be different from StackGuard on the x86?

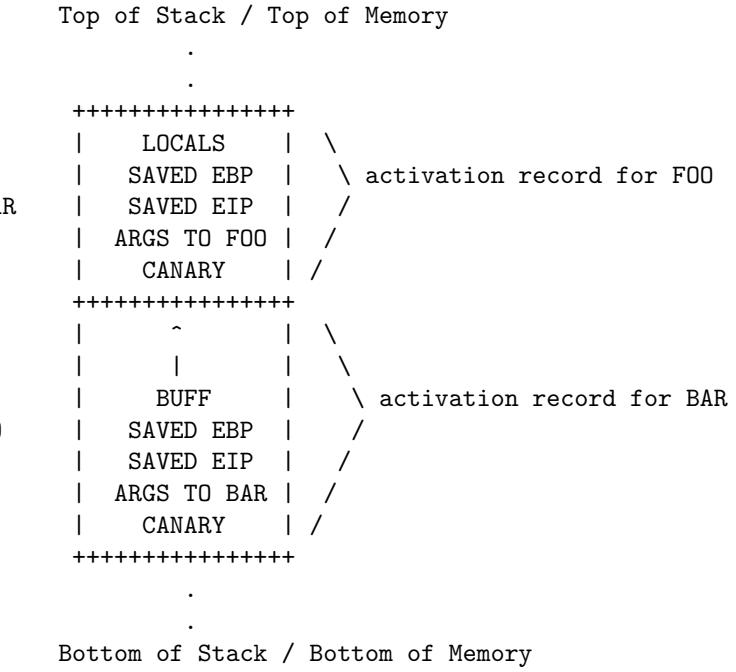
From the picture above we see that in order to overflow BUFF and overwrite the SAVED EIP for FOO we need to stomp first over ARGS TO FOO and then its SAVED EIP. To implement StackGuard in the upward growing architecture we can make the compiler **place a canary in the stack right before the arguments for a function call to the stack. Prior to function return we will verify the integrity of the current activation record by checking the canary** and kill the process if we detect corruption. This differs from x86 in fact that the canary in x86 has to defend against a buffer overflow in an activation record overwriting the saved eip of that activation record as opposed to a buffer overflow in previous activation record corrupting the saved eip of a subsequent activation record. Thus the placement of the canary has to be different. The following diagram shows the different placements.

[1] the activation record of the function being current ly executed

a) StackGuard layout for downward growing stack



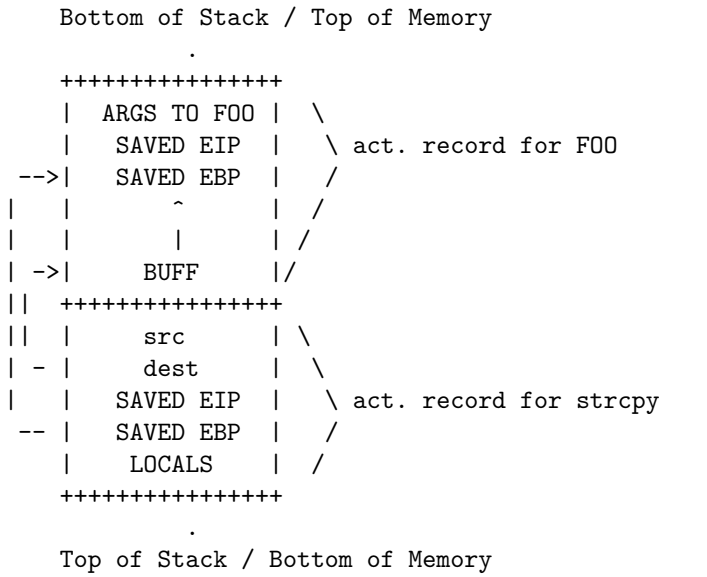
b) StackGuard layout for upward growing stack



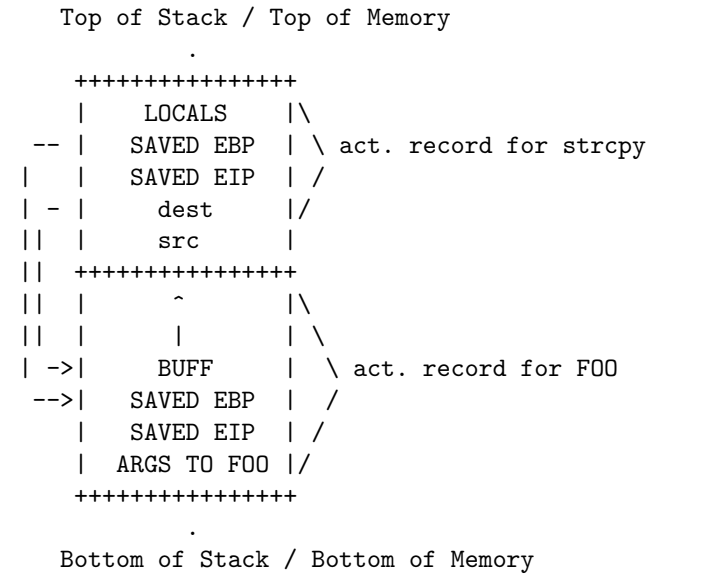
c) How would you implement LibSafe in an architecture where the stack grows upwards? What would be different from LibSafe on the x86?

The following diagram illustrates the stack layout for a call to strcpy from a function FOO in downward and upward growing stack architectures:

c) Libsafe layout for downward growing stack



d) Libsafe layout for upward growing stack



As discussed in lecture, in x86 Libsafe intercepts the call to `strcpy(dest, src)` and validates that there is sufficient space in the current stack frame via the test: `|SAVED EBP for strcpy - dest| > strlen(src)`. The picture clearly shows that **in the upward growing stack architecture in order to ensure `strcpy` does not overwrite `src` we would need the test: `|&src - dest| > strlen(src)`.** This approach of checking that the size of the data to be copied is no longer than the size of the space between the first argument to the Libsafe function and the start of the buffer being written to generalizes to other functions like `memcpy, strncpy, memset`, etc.

Problem 2. *(TOCTOU)*

Consider the following code snippet from `foo.c`:

```
1a.  if (!stat("./file.dat", buf)) return;   // Abort if file exists.
2a.  sleep(10);                              // Sleep for 10 seconds.
3a.  fp = fopen("./file.dat", "w" );
4a.  fprintf(fp, "Hello world" );
5a.  close(fp);
```

a) Suppose this code is running as a setuid root program. Give an example of how this code can lead to unexpected behavior that could cause a security problem. (try using symbolic links.)

Suppose there is a malicious program `bar.c` that simply does the following:

```
1b.  symlink ("/etc/passwd", "./file.dat");
```

i.e. it creates a symlink from `./file.dat` to `/etc/passwd` (assuming `./file.dat` does not exist). Suppose furthermore that the programs `foo` and `bar` are both running at the same time, with the same current working directory environment variable and that `foo` is running as a seteuid root program. Consider the following scenario: A file `./file.dat` does not exists when both programs start executing. `foo` is scheduled first and gets to line 1a. passes the test then goes on to line 2a. and gets switched out while sleeping. `bar` then gets scheduled in and executes line 1b successfully creating a symlink from `./file.dat` to `/etc/passwd` and exits. Then `foo` gets switched in again and continues to execute lines 3a through 5a. Since it has root EUID then `foo` will be allowed to open `/etc/passwd` and write "Hello world" to the beginning of the file.

This represents a security problem since the malicious program has achieved the password file to be overwritten, possibly corrupting the information for some user(s) and thus at least creating a denial of service attack against those user(s) since their credentials have been corrupted. Note: one can add a call to `remove("./file.dat");` before the `symlink` call in `bar` to make the attack work sometimes in cases where the file `./file.dat` existed prior to starting.

b) Suppose the sleep(10) is removed from the code above. Could the problem you identified in part (a) still occur? Please explain.

The problem in (a) can still occur, however it becomes less probable. The problem in (a) happened because `bar` got switched in while `foo` was sleeping. This was likely to occur since sleeping for 10 seconds will probably cause the scheduler to switch that process out and some other process in. By removing the `sleep` call we only decrease the probability of that exact interleaving, however it is still possible for `bar` to get switched in right before the call to `fopen` in `foo`, which would result in the same problem.

c) How would you fix the code to prevent the problem from part (a)?

The problem comes from the fact that checking in the file exists and opening it is not done atomically. According to the Linux man pages for open:

```
If O_CREAT and O_EXCL are set, open() shall fail if the file exists. The check for the existence of the
file and the creation of the file if it does not exist shall be atomic with respect to other threads
executing open() naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL
and O_CREAT are set, and path names a symbolic link, open() shall fail and set errno to [EEXIST], regardless
of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.
```

Hence we can use the `O_EXCL, O_CREAT` files to make the existence checking and file opening atomic in the above code as follows:

```
1a.  int fp = open("./file.dat", O_CREAT | O_EXCL, 0644);
2a.  if (fp == -1) return;
4a.  fprintf(fp, "Hello world" );
5a.  close(fp);
```

Another way to prevent this problem is by using mandatory file locks in `foo.c`.

Problem 3. *(Soundness and completeness for static code analysis)*

a) A false alarm occurs when a tool reports an error, but the program the tool is analyzing does not contain an error. Can a tool that reports a false alarm be sound? Complete?

Completeness: By definition, a false alarm means reporting/flagging as an error something that is not an error in reality. A complete tool only reports things that are errors in reality. **Hence, a tool that report false alarms can definitely not be complete**.
Soundness: Soundness requires the program to report all real errors, plus possibly some false alarms. **Hence, a tool that reports false alarms can be sound if and only if it also reports all real errors.**

b) Suppose a company sets up an Android app marketplace for its employees. The company is going to use an analysis tool to check apps for security vulnerabilities before it promotes them to its employees. Which tool property is critical to the company for this purpose: soundness or completeness?

Argument for soundness: Consider a company that has a very high sensitivity to vulnerabilities (e.g. a governmental organization that stores highly classified national security information or intelligence, or a company that stores very sensitive consumer data such as health records, etc) and hence wants to be *absolutely certain* that the apps it promotes to its employees do not contain errors. Its sensitivity to vulnerabilities is so high that the company does not mind rejecting apps that did not actually contain errors and does not have resources or want to spend them investigating whether the apps it forbids are actually insecure. A sound tool will fit its purposes best since it will report all the real errors plus some false alarms. The company can just forbid all apps for which the tool reported errors. This would to ensure all apps it promotes to its employees are secure. **Alternatively,** if the company is willing to spend a lot of resources to be certain that the apps it promotes to its employees do not contain errors while accepting as many apps as possible the company will then have to devote resources to sift through the reports and discern between false alarms and real errors.

I believe there are arguments for both why you would want soundness or completeness in the tool and they both have tradeoffs. The decision whether to prefer one property over the other has to do with the tradeoff between *sensitivity to vulnerabilities* in the apps and *resources spent investigating* the vulnerabilities. Of course, another consideration is the quality of the tool (i.e. if we have the trivial sound tool that reports everything, or the trivial complete tool that reports nothing then these arguments don't matter) but we will not worry about that and assume the tool has "decent" quality. An argument for completeness is given as a footnote.[2]

c) Theoretically, suppose a tool is both sound and complete. When the tool is used to analyze the following code that may contain a vulnerability, what property of the loop determines whether the tool will report an error?

```
int main()
{
  int x = 0;
  while ( x < 10 ) { /* loop while x is less than 10 */
       ...  /* do some stuff that might change the value of x  */
  }
  ... /* do something insecure */;
}
```

(We assume the loop does not contain any insecure code)

Termination of the while loop (without exiting from main) on some possible path of execution: A sound and complete tool will report an error *if and only* if there is a vulnerability in the code. There is a vulnerability in the code if and only if the code after the while loop ever gets executed under some possible path of execution. This happens if and only if the loop terminates without exiting from main under some possible path of execution.

[2] **Argument for completeness:** Consider a company which does not want to spend many resources finding vulnerabilities and does not mind too much not covering all the possible vulnerabilities. A complete tool provides no false positives and only true positives (though not all of them). Hence a complete tool provides **actionable information** that the company can use right away to locate vulnerabilities in the apps its promoting to its employees. Even though the tool might fail to report some vulnerabilities it will give information that the company does not need to further validate and hence spend resources on. This is in contrast to a sound tool that may provide a lot of false alarms, leaving the company with no immediately actionable information and no clear sense of where to start looking. Thus a complete tool suits well a company with low sensitivity for vulnerabilities who is not looking to spend too much finding vulnerabilities.