CSCI 5271 Exercise Set 2

Alex Biedny

October 21, 2020

Problem 1. Equal Error Rates and Passwords

The main security challenge associated with using confidence values to check passwords is the difference between how machines read passwords versus biometrics. In a biometric system, there needs to be some sort of sensor to read the fingerprint or iris or other biometric. This is imprecise by nature, and subsequent reads of the same biometric will rarely return identical values. Because of this, a confidence value scheme is used, which allows for biometric readings that are similar to a high enough degree to be verified as the same.

With passwords however, no sensor is required, and instead a precise method of input (the keyboard) is used. It is simple to input an identical password every time, and as such usually confidence values are not used to check passwords. If they were, it would decrease the security of the system, as then an adversary would not need to know the exact password, rather they could use a password that is simply similar enough. In addition, depending on how the confidence value is adjusted, there could be many false positive inputs.

To measure the EER of a password system that used a confidence value approach based on edit distance, you could measure false negatives by having the user enter their password a number of times, and record the rate that they are rejected at. Of course, this would vary significantly based on how the user types. To measure false positive rate, you could have an adversary attempt the same number of logins, and measure the rate that they are accepted.

The only result of this that would really have any meaning would be the false positives, as the adversary would not know the password, so if the false positive rate was too high, the acceptance criterion would need to be tuned further. If this experiment was taken to its logical end, then the false positive and false negative rates would be minimized, which would end up giving you a system that only accepts the original password.

Problem 2. Access control without hardware support

The most straightforward solution to this would be to use a Type 1 hypervisor, and have the toy computer OS run on top of that. Even with no physical hardware security mechanisms, with enough compute power (and the problem states that the processors are fast and have a lot of RAM available to them), you can simply emulate that hardware. As the hypervisor will expose virtual hardware, the guest OS will think that it is running on bare metal, and when the toy computer OS is programmed, it can still take advantage of security hardware - so long as the hypervisor exposes them to the guest OS. Thus the guest OS can be designed securely, even without access to security hardware.

Problem 3. Sharing files in Unix

The biggest vulnerability in these programs are the permission files themselves. Alice wrote the permision files, so she is the owner of them, and would naturally have read and write permissions to them. Since the programs she wrote will be executing as her, they will have the ability to change these permission files. Exploits such as ones that exploit race conditions between time of check for the file permissions and time of read/write could work to let users access her files.

A better implementation would get rid of the possiblity of exploits like that by getting rid of the `alice-read/alice-write in out` format, and replacing it with programs that could output the contents of Alice's files for read, and expose a simple editor for write. In addition, it would be better to not require users to provide files at all, and instead have the user run the program and be presented with a list of files that they are able to read/write. This would prevent any sneaky symbolic link or race condition exploits, along with a host of other vulnerabilities that can come along when direct user input is allowed.

Problem 4. Multilevel-secure file sharing in Linux

First, only the owner of a file is allowed to chmod it, so Bob's process that changes permissions would not work. If a user wanted to write a file, they would create one (and be the owner of it), and then when Bob's process attempted to chmod it, it would not work because Bob's process would run with his permissions and wouldn't be able to chmod the file.

In addition, file owners would not be able to read any files they own to begin with. Since the permissions are set to 0 for the user block, whoever owns the file couldn't read or write to it, regardless of the groups they may be in. If a cleared user wanted to read a file that they authored, with an access level that they should be able to read, they would not be able to.

A better implementation would be to ditch the directory stuff and just use groups. Having one central user own all the files so that user permissions are not considered would eliminate the issue with user permissions overriding group permissions. Put all the users in groups that prohibit them from writing below them, and have the files owned by groups respective to their access level. For example, management is in nowrite-private and nowrite-public. Do the same with read permissions, and then have the everyone else permissions set to both read and write.