Exercise Set 2

helge206 Andrew Helgeson
wayt0012 Ethan Waytas
maure113 Brian Maurer

1)

Many biometric authentication schemes produce a "confidence" value that allows a tradeoff between "false positive" and "false negative" errors. Password schemes are not typically considered in this light. List some reasons why you think this might be.

For the following paragraphs, confidence value is assumed to incorporate the false negative and false positive rates. A confidence value that is "high" implies a secure system, a confidence value that is "low" implies an insecure system. If the confidence value increases or decreases, the security of the system has changed.

The main reason that password systems are usually not considered in the false-positive or false-negative light is based on where the source of the mistake originates from. In a typical password system, it is not the system that is making a mistake, it is the user. Assuming the system is implemented properly, it will only fail if the user fails to remember or type their password correctly. This contrasts with other biometric authentication system failures because they are often based on the system's inability to properly recognize a given user's input due to slight amounts of biometric input variability. For example, it is not a really considered a mistake if a user cuts their finger, and then the next time they use fingerprint authentication the system rejects them. It is the system that is unable to recognize that it is the same user, not the user making an input mistake.

Additionally, another reason a confidence value might not be an accurate measurement for a password system would be the variability with the confidence value itself, or lack of. For example, say a website does not restrict users on the types of passwords they can choose. Some users will have easy to guess/crack passwords, while others will be difficult or impossible to crack. In this situation, the false positive rate is probably high (easy to guess passwords) and the false negative rate is low (easy to remember passwords). Now, say this website suddenly requires its users to change their passwords to a minimum length of eight characters, with at least one uppercase character, special character and number. The false positive rate will drop, as passwords have now become very difficult to guess or crack. The false negative rate, however, will increase, as passwords are harder to remember and are easier to mistype. In this new situation, the confidence value may not change, or it may become worse as many users struggle to remember their passwords. The website has become more secure, but the password confidence value has decreased. By only examining the confidence value, it would be implied that password security has dropped, when in fact it has not.

We could change the way we check passwords to produce a confidence value; for example, the edit distance between a login attempt and the stored password. What are the (security)challenges of

this approach compared with the standard use of passwords?
To do edit distance matching of passwords they must be stored in plain text, which is a security vulnerability. In a regular system passwords are hashed so even in the case of a breach all that will be obtained is the hash and not the actual password text. Since password hashes are non-reversible to the exact password which created them (disregarding luck), you cannot implement this edit distance scheme using them.

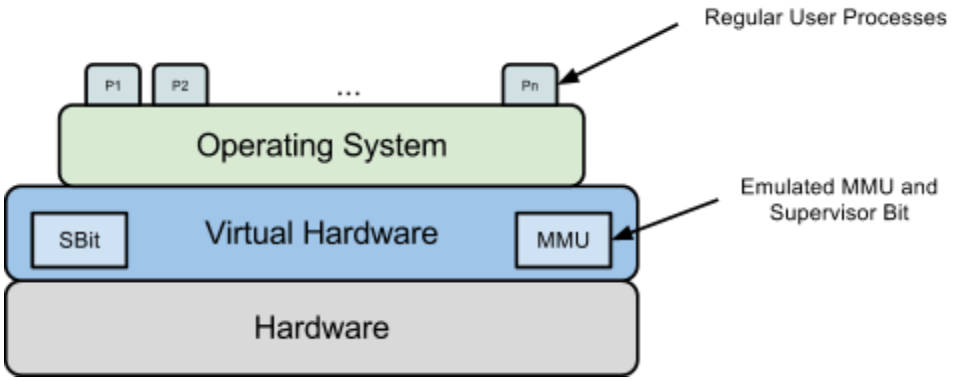
First we need to make a decision on what edit distance to use. If you chose a static edit length it could be too loose on short passwords, and to tight on long passwords. For example, an edit distance of 1 on a 2 letter password would result in 1/26 chance that another two letter password is a match. This is BAD. A better solution would be to allow increasing edit distances as the password gets longer so that all length passwords have the same probability of false positive and false negative. This would require some clever probability calculations to figure out the best edit-distance-allowed for a given password length.

How would you measure the EER of a password system using this approach? (Would your measurement be meaningful?)
The false positive rate would be the probability that somebody tries to login in as somebody else, and types a password that is accepted because it is within the allowed edit distance.
The false negative rate would be the probability that a user types their own password wrong and the edit distance is outside the acceptable range for the system, so they are rejected.
The ERR is when those two probabilities are equal.

This is a meaningful measurement because the authentication error rate is now dependent upon both the system and the user. In the regular password system the password either matches or it doesn't so there was no meaningful error measurement for the system, only the user (see first section). Like other biometrics, in determining whether to accept or reject, using an edit distance password system has more decision making rather than the binary decision of the traditional password system.

2) *Access control without hardware support*

As mentioned in lecture, in the absence of hardware access control mechanisms you have two options, no access control or virtualization. Here we present our ideas for virtualizing the hardware to add back the control we had with an MMU and the supervisor bit.



The virtual hardware (VH) is most similar to a modern hypervisor except for simplification it only supports running a single operating system. To add security above the raw hardware this VH will implement a software emulated supervisor bit and MMU. First, we will list the primary security benefits lost from not having this hardware support and then argue that the VH gives us this security back.

MMU (memory management unit)

- Without the mmu we lost the separation between physical and virtual memory which is important for process isolation. Without this, processes could access any memory location such as the OS or other user processes.

Supervisor (Kernel) Bit

- Without the supervisor bit the hardware cannot restrict operations based on what process is running. Without this the MMU would not be meaningful since any user process could modify the MMU translation tables or simply use direct memory access.

Not having this hardware support is not acceptable if you want to run a multi-process operating system where you will likely have untrustworthy processes. Since we are told these machines will have sufficient CPU power and RAM virtualization becomes a feasible solution. For the virtualization to work, system calls (interrupts) and memory requests must be redirected toward our virtual hardware. The virtual hardware could accomplish this through binary rewriting or some other method but we will continue assuming this will work. Now we will argue that the emulated versions of the hardware access control mechanisms will provide us the same security.

System calls which are initiated through a TRAP instruction will be captured in the virtual hardware where the emulated supervisor bit can be set before entering operating system code. To the operating system this will work just as before. This new mechanism still doesn't solve the return to user attacks that you have with normal hardware but we are aiming to provide an equivalent security mechanism. The supervisor will also protect any privileged operations, such as modifying MMU entries or the page table. As assumed earlier, this requires that the VH can

modify the running processes to trap these requests.

While it will make memory access potentially much slower, the MMU could be easily emulated in software. The updating of the MMU tables is a privileged instruction which could be enforced by the VH using the emulated supervisor bit. Similarly, any process making memory requests will have to go through the MMU and access requests that are outside the process' memory segment would require the supervisor bit to be set. In actuality you would probably reject any such request unless the supervisor bit was set and the physical addressing mode was requested (which requires the SB). With this in place no process can change its own memory mapping in the MMU and process virtual memory spaces are still enforced as with a hardware MMU.

Using VH we have created a software emulated MMU and SB that will protect against any unauthorized memory accesses and protect privileged hardware operations. With these two things we can ensure secure process isolation to the same extent that normal hardware would provide.

3)

What are some of the potential security problems with this approach? (e.g. suppose Bob can read and write some of Alice's files but not others;can he use alice-write and alice-read to gain access to files he shouldn't? Are there potential attacks that could allow third parties to read/write Alice's files?)

When using *alice-write,* permissions are only checked on the *out* file and on *alice-read* she only checks permissions on the *in* file. This would allow a user to write out content of a file they don't have permissions for to a file that they do have permissions for. Users could also take the contents of a file they do have permissions for and use *alice-write* to write them to a file they shouldn't be able to write to.

If a user within Alice's permission boundaries sends information to a party outside (3rd party), that 3rd party would be able to see something they are not allowed to see. Essentially, Alice does not have control outside her program so insiders can leak information out. Additionally, if she has a reason to make a file public she would end up opening all her files to the outside users due to the weakness of her interface.

Suggest ways to change the interface and/or implementation of alice-write and alice-read to avoid your attacks.

First, we should check the permissions on both *in* and *out* for both *alice-read* and *alice-write*. This will prevent the half authorized *alice-reads* and *alice-writes*. So even if data is leaked it can only be data which Alice has entrusted a given user with, not all her files. Similarly, writes to Alice's files can only come from people she has permitted write access for the file, like she

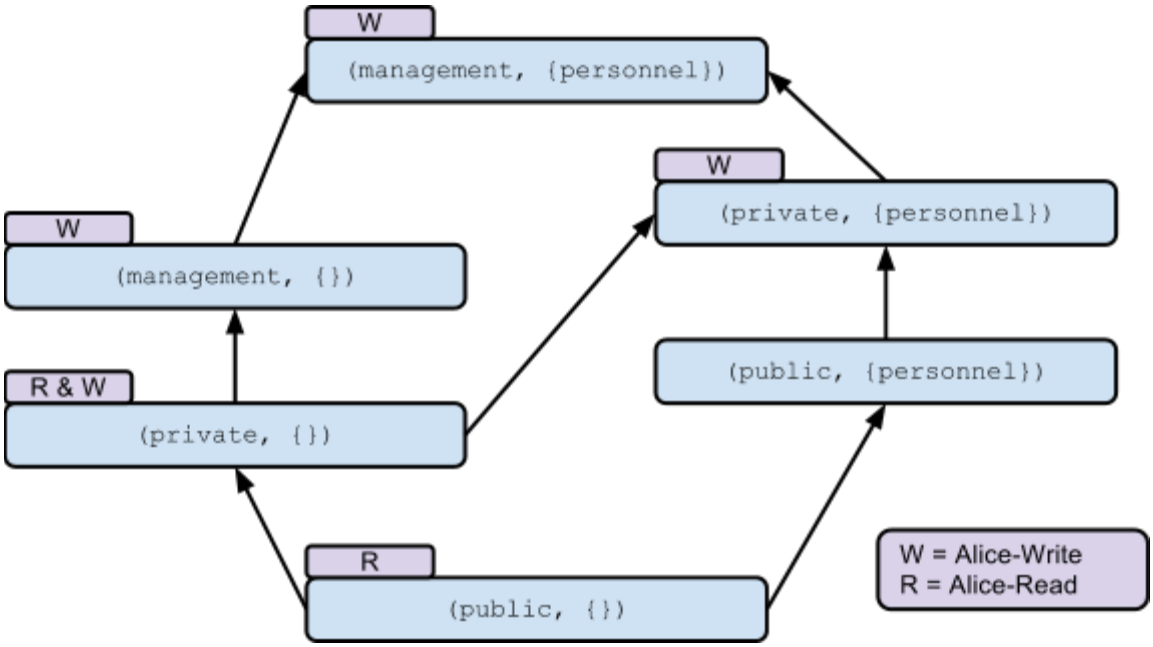
originally intended.

Unfortunately, we cannot prevent insiders from sharing information they have access too with an outsider. For example, an insider might transcribe the file to a hand written page or copy the file to an email. Both of these examples are very hard to prevent because they deal with the user maliciously trying to get data out of a system by non-preventable means (a piece of paper does not have read/write groups). Security, for these types of attacks, begins to combine both computer and user, where the user must also be a “trusted” system. By trusting a user, a system will always be opened up to the potential of information leakage through unique ways.

4)

(a)

Suppose Alice has current clearance (private, ∅). Draw the lattice of classifications in this system (there are 6 classifications) and mark with an “r” each classification that Alice should be able to read under the BLP policy and a “w” each classification that Alice should be able to write to under the BLP policy.



(b)

What are some of the problems with this approach? (i.e., are there conditions under which classified data is leaked to uncleared users? when and how? Can cleared users be prevented from reading classified documents? how?)

Permissions of *075* allows anyone to *cd* and *ls*, which means anyone could see file names, even if they are not part of the group on the directory. Revealing filenames could be considered a security breach if the names of files gives away some confidential information (ie

‘known_alien_planets.txt’). The rationale behind 075 permissions instead of 070 is to allow users at a higher classification to still be able to *cd* or *ls* to a directory below themselves since they aren’t part of the write group for lower classification files. This would be solvable if we could assign different permissions for different groups on the same file. Currently we can assign specific permissions to the “*write”* group that owns the directory, but if we want anybody else to be able to read the contents (ie a dominating classification) we have to give read permissions on the directory for everyone else.

When a user creates a directory/file there is room for a race condition between the time that the directory is created and the time that the daemon changes the permissions on the directory/file. Potentially, a user with a lower classification could read a file “above” them before the daemon changes the permissions, leading to information leakage.

In order to prevent higher level classifications from reading files, we must make some assumptions about how the daemon decides to change directory/file permissions. From the question, there appears to be a naming scheme (f-dir/f-data) that tells the monitoring daemon to change a file's permissions to be 0040 and a directories to 0075. A user, wanting to hide a file could create a file that breaks the naming scheme by not appending “-data” to the end. The daemon would not change the permissions and a user with a dominating classification could not read this file (however *ls* would still reveal the existence of the file) . Additionally, if the naming scheme applies to directories as well, a user could create a directory without “-dir” and the daemon would miss it. If the permissions on this directory are set to 0700, then only the user who created the directory could *ls* or *cd* on the directory. Depending on where the directory was created, it may be difficult to find for users with higher classification and could escape notice.

If a user writes a file at or above their classification, the daemon does not change the owner of the file, only the group. Per the man pages, the owner of a file is allowed to change the file permission as long as they are listed as the owner. A user could, after the daemon changes the group and permissions, change the permissions to 0700, which would prevent anyone other than the user from reading, writing or executing the file.

Suggest a better solution without using ACLs

To fix this problem we would first have the classified system contained within a file where only root has write permissions. Any time that a directory/file is going to be written, the specified write commands would have go through the permissions daemon. The daemon would handle write-permissions checking on the given directory/file and enforce it at time of creation. This scheme would avoid race condition issues that we discussed above. With this mechanism for creating directories and files you could do away with the naming scheme (f-dir/f-data) to provide another level of anonymity to the contents. The daemon would also set the owner of the file to be root, to prevent the original owner from changing permissions.

The next thing we would need to do is separate the “write” group and “read” group permissions onto separate and mirrored directories. This way we would put 070 permissions on the write

group folder, and 050 permissions on the read group folder. This would ensure that no directory is world readable and therefore eliminating the problem where anyone can *cd* into a directory and *ls* its contents. Just to reiterate, if a file is written, the daemon would manage the write, and update the mirrored file in the corresponding “read” directory so that they match.