Problem Set 3 Solutions

Problem 3-1. Take/Grant Groups (Grant Emery, Brian Perrin, Emil Sit)

It may be desirable to have *groups* in our computer security system, such that all subjects in a given group share permissions on some set of files. We can use the take/grant model as a basis for our system when setting up our groups, but we will need to extend this model to fulfill all of the requirements for abilities related to a group.

Pure Take/Grant model

We can partially implement groups using the standard take/grant model. In order to create a group, a user creates a new subject. This subject functions as the group object. Group membership is given by giving the take permission to a new member. (This requires that the group creator have grant permission over the member.) If the creator wants to add permissions for a given file to the group, he grants those permissions to the group object. Thus, group members can exercise the rights of the group by taking them from the group object. This model also allows for groups within groups, since take permissions are transient.

Unfortunately, there is no real support for revocation. The take/grant model does not provide any mechanism for removing rights except by a user from himself. There is no clear mechanism for deleting subjects. So, once a user is placed in a group, the rights that he takes are his forever. Actual membership in the group (i.e. take rights over the group object) has no real impact on a subjects abilities.

Improving Take/Grant: a New Permission and its Function

In order to fully implement groups in our computer security system, we only need one new primitive, which we can demonstrate will not increase the complexity of solving permission problems.

Our new primitive is "ask". If subject A has ask permission on subject B, subject A can ask subject B to do anything B could normally do, and subject A will get the output as though A had performed the task. For example, if A had ask permission on B, and B had read permission on C, then A could ask B to read C, and A would get the output, allowing A to read C without having read permission on C. We also allow chaining of ask requests — when one subject asks another to do something, that subject can do it via asking yet another subject to perform the action.

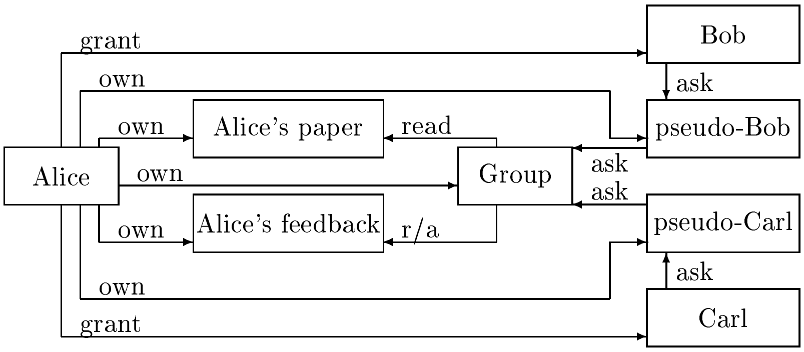
Note that when solving permission problems under this new system, we can model all ask permissions as take permissions, reducing this system to a normal take/grant system. The ask permission is simply be interpreted as a weaker version of take. The same bridges can be constructed, except that no new edges need be created. The net result is the same regardless of whether we are using take or ask.

Group Model with Ask

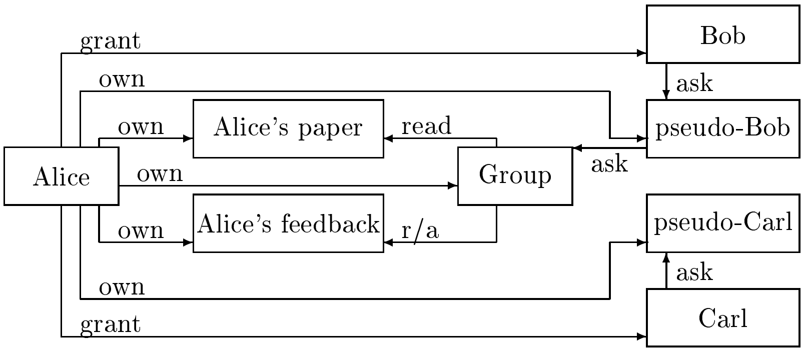
The first problem we want to fix is the inability to remove rights from the group. The first step is to have the creator have ask permission on the group, which should be inherent given that the user created the group. When the user wants to remove permissions from the group, the user can ask the group to remove the appropriate permissions. This prevents future access to users who have not already taken the appropriate permissions. The second part of this fix involves giving ask permission to all subjects in the group instead of giving them take permission. In this way, subjects cannot take permissions from a group, and when permissions are removed from a group, we have ensured that no subjects can continue to use old permissions.

The other problem involves being able to remove subjects from a group. We cannot expect all subjects to give the creator of the group ask permission on themselves, for the purpose of allowing the creator to ask the subject to remove the ask permissions on the group, because the creator could use this ask permission to do anything the subject could do. Instead, we will create pseudo-subjects, which are essentially imitations of the subjects we want to add to the group, but which we can allow the creator to have ask permission on. When the creator wants to add a subject to the group, the user creates a new subject, and grants that new subject ask permission on the group. The creator then grants ask permission on this intermediate object to the subject he wants to add to the group. In this way, the added subject can ask the intermediate subject to ask the group to do something. To remove the subject the creator can just ask the intermediate subject to remove its ask permission. Alternately, one could simply delete the intermediate subject.

For example, user Alice wants to create a group that can read her paper, and then provide feedback on it. She first creates the group, and gives the group read permission on her paper. She also creates another new subject, and gives the group read and append permission to it, so that subjects in the group can read what comments have already been made, and add to those comments (but not delete any existing comments). Alice wants to add subjects Bob and Carl to the group. The system would now look like this.



If Alice later decided to remove Carl from the group, she would ask the pseudo-Carl to remove its ask permission on the group, and the new system would look like this.



Remaining Problems

While all of the functionality required for a group is implemented, there are still a few drawbacks to this system. First, there may end up being many useless intermediate objects in our system, the results of subjects being added to a group and then removed from them. This could possibly be remedied by adding delete permissions, but this adds too many complications for it to be recommended. Alternatively, Alice could delete pseudo-Carl when she's done with it.

Another annoyance is that any subject in the group can remove permissions on any file. If the original object came from the creator of the group, then the creator can simply replace the removed permissions (perhaps after removing the offending subject). However, the subject may have been created by the group, in which case, it is lost forever. Essentially, all members of the group have delete access to any group-created file. While this is not necessarily a bug, it is something to be careful of in implementation.

Any subject can grant their ask permission on their intermediate subject to another subject, essentially adding other subjects to the group. This is not necessarily a problem since the subject could have done the same thing outside of the context of the group (for example, making a copy of files the subject can read, and granting read access to other users). If it is considered too great a problem, the creator of the group can remove the offending user when the violation is discovered.

Problem 3-2. Access-matrix Versus Stack Inspection (Adam Holt, Dilong Zeng, Svetlana Schnitser, Igor Tarashchanskiy)

The Java stack-inspection model involves scanning the method call stack from top (newest) to bottom (oldest) looking for explicit permission or denial for the requested permission. Recall that Java is set up in such a way that dangerous calls must go through the Security Manager. Default behavior upon falling off the end of the stack (without finding either condition) unfortunately varies. Amazingly: Microsoft and Sun allow access "to support legacy applications," while Netscape denies access "to support legacy applets."[1]

This general behavior can be represented as a series of access matrix consultations, one for each stack frame. It is worth noting that graphs are always expressible as (typically sparse) matrices. So the privilege flags of each stack frame can be represented not only as a series of fixed access matrices, but these can be merged to represent a snapshot of any thread's entire runtime stack. During a dangerous method call, we can verify in the larger stack-equivalent access matrix that the subject procedure indeed has said permission over its requested subject. This larger access matrix corresponds to a thorough assertion set of current privileges.

Java's stack inspection is often criticized as ad hoc, but Felten and Wallach show[2] that this access control corresponds to proving statements in a belief logic known as ABPL. They further demonstrate that java stack inspection is expressible with finite pushdown automata, a simpler model than undecidable mutable access matrices where security is unprovable. Mutable access matrices belong to a more general class of problems due to halting problem on Turing machine equivalence. Pushdown automata offer no final security guarantee however. No ship is leakproof, but we can certainly now run a better-engineered more leak-resistant ship.

While a more expressive security model allows more advanced applications, it often tends to weaken overall trust. A real world example of the additional power (and complexity) offered by access matrices is suggested by putting ALL of your most personal photos online using http passwords. You not only discover the limitations of in-or-out black-and-white security; in fact you quickly discover that even the more "shades-of-gray" benefits similar to java's stack inspection are not enough. Your more-and-less web-shy friends each want different (often contradictory) fine-grained group access policies suggested by a richer mutable access matrix. And your friends quickly want to create their own transitive security policies so their friends and parents can view the online pictures. Still, whatever Java's inheritance-like model might lack in expressivity, its construction offers much stronger assurances of access control.

[1] "Access Control and Stack Inspection" `http://securingjava.com/chapter-three/chapter-three-6.html`
[2] "Understanding Java Stack Inspection" `http://www.cs.princeton.edu/sip/pub/oakland98.html`

Problem 3-3. Tripwire in the Matrix (Nick Estrada, Nick Feamster, Michael Freedman)

Part (a)

Alice intends to design a security model based on the Graham/Denning framework, with the modification that each S_i can be verified by its corresponding hash value H_i, which is created by the security monitor. The security monitor S_{sm} of course has its own hash value H_{sm}. Verification of a given subject is thus executed by comparing that subject with its corresponding hash value ("snapshot").

We do not want a subject S_i to be able to control/write its own hash H_i, in case the subject is compromised. If we ensure that *only* the security monitor S_{sm} can write to these hash objects, the chance for a compromised subject is reduced. A given subject should not be able to undergo changes to its own identity and subsequently have its hash value rewritten by something that is not the security monitor.

In addition to write access, the security monitor have control over these hashes by maintaining a list of $[S_i, H_i]$ pairs to keep track of all known subjects and their appropriate hash values. This list is the "state" of S_{sm}, and must be updated every time a new subject is added.

Note that a disadvantage of this approach is that the security monitor itself (including this table) could grow quite large as the number of subjects increases. We choose this implementation, however, because it reduces the possibility of compromised subjects.

Graham/Denning Rule Modifications

In the framework of the Graham/Denning model, the create and destroy rules would have to be changed, such that the hash H_i is created with the subject S_i in one *atomic* operation. This ensures that the hash value actually represents the subject that was just created, and that a subject cannot exist without a hash (or vice versa). Likewise, the hash should be destroyed with the subject atomically.

Additionally, as described below, since we choose to store the hash values in the security monitor itself, it is necessary to ensure the integrity of this list. In the context of our N-redundant scheme described below, a security monitor should not have write access to its own hash, but it can be hashed by other security monitors which are ensuring its integrity. We describe this scheme in the following section.

Part (b)

In order for the system to maintain integrity, the integrity of the security monitor itself is essential, because we are storing hash values for all subjects in the security monitor. The following sections address possible safeguards for attacks mounted against the security monitor.

Hashing the Security Monitor

There are two possible methods in which to handle hashing the security monitor:

- Hash S_{sm} instruction memory: Only keep a hash of the security monitor's logic, which only needs to be done upon creation (similar to some S_i.) Thus, we would know if an attacker compromises the logic of S'_{sm}. However, this approach leaves the subjects' hash values open to attack: an attacker could then replace certain hash values in the security monitor without detection.

- **Hash entire** S_{sm}: We hash both the instructional memory and state of the security monitor, therefore stopping an attacker from using the aforementioned exploit. However, this requires hashing the S_{sm} every time its state changes: whenever some S_i is added or deleted. This may be computationally intensive, but, in conjunction with an N-redundant system (described in part b), reduce the likelihood of a successful attack.

We recommend hashing the entire security monitor; although this method requires rehashing S_{sm} every time a new subject is created, it provides higher integrity for the system as a whole.

Protecting the Security Monitor: Redundancy

In order to prevent the possibility that the security monitor S_{sm} is compromised and is allowed to rewrite its own hash, there must be a check on the security monitor itself. Thus, the security monitor should not have the authority to rewrite its own hash. One potential scheme is a two security monitor scheme, whereby S_{sm_1} has write access to H_{sm_1} and S_{sm_2} has write access to H_{sm_1}. No object has write access to its own hash function. Therefore, we have created a cross-validation scheme.

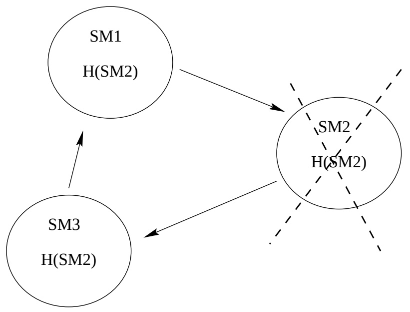


Figure 1: 3-Redundant Security Monitor

More generally, we can employ a N-redundant security monitor (similar in motivation to a RAID recovering from disk failure). If a security monitor is compromised, this will likely be detected by the security manager which holds its hash function (an adjacent security monitor in the diagram). This compromised security manager can subsequently be discarded, and replaced with an atomic copy of a valid security monitor, followed by a recomputation of the hash value which the compromised security monitor contained.

Compromise However, it is conceivable that if S_{sm_i} can be compromised, $S_{sm_{1 \ldots i-1, i+1 \ldots n}}$ may also be compromised. In order for the exploit not to be initially detected by a still-valid S_{sm_j}, all security monitors needs to be compromised within a time *less than* the periodicity in which security monitors check each other's validity.

Recovery For example, if S_{sm_2} is compromised, it can be recovered by copying its data from S_{sm_1} (since we know is valid because the hash in S_{sm_3} will verify it). Subsequently, the hash value can be recovered by rehashing S_{sm_3}. Assuming that all dependent security monitors are not compromised simultaneously, the integrity can be maintained. One can imagine increasing the robustness by increasing interconnectivity of security monitors.

Circumventing the Security Monitor

There are several ways in which the integrity of the Graham/Denning security model may be compromised via Trojan Horse-type exploits.

- **Public hash function:** if an attacker replaces the public hash function $H()$ with their own function $H'()$, they can trick the security monitor by having $H'(S_i') = H(S_i)$. This is hard for good hash functions.

- **Security monitor reference:** if an attacker can replace the pointer to the S_{sm} in memory, they can effectively replace the security monitor with their own, compromised S'_{sm}.

Implications for Rules

Furthermore, in this situation, the framework of the rules must be such that one security monitor cannot delete these rights from one another, nor can a malicious security monitor transfer or grant its rights (using rule R1 or R2) to the other security monitor. This means that rule R1 either needs to be modified to prevent the copying of this write action, or that the copy flag should be prohibited in this case. It is more feasible in this case to **disable the copy flag on write access** to all hash functions. This also solves the potential problem of a malicious security monitor giving unauthorized rights to another subject.

Additionally, because of rule R2, it also follows that no security manager "controls" any other subject or security manager, in order to prevent granting these rights.

Part (c)

Given that a given object might be an executable, which includes state in registers, memory, etc., Alice's idea needs more refinement. As soon as a given object changes state, its "snapshot" is no longer valid (because values in memory, registers, other states) may have changed. A feasible alternative (in the case of an executable) could be to hash instruction memory for that executable, seeing that instruction memory is not designed to change over the course of the life of a program.

Thus, if a given subject consists of both an executable and state, an obvious alternative is simply to hash the portion of the subject which defines that subject's identity (i.e., its instruction memory). Another potential alternative, if the subject can take on different states, is to include the state transition table (i.e., for the finite state machine defined by the program) in the hash.

Including a state transition table in the hash for a given subject may provide greater integrity for the subjects, because a modification of the program to execute instructions in data memory could potentially be detected via an altered state transition table. However, determining these entire tables may have such extreme computational complexity that this method becomes useless in practice.

Hashing Subsets and Bookkeeping

Which subset of a particular subject should be hashed depends on the nature of the subject itself. As a result, now each subject has three entities associated with it: the subject S_i, the hash value H_i, and an indicator Q_i which indicates which subset of S_i the hash value H_i corresponds too. It is logical for this value Q_i to be stored with H_i in the security manager. As discussed in the following section, hashing the value of Q_i with the subset automatically might be a feasible countermeasure (i.e., $H_i = H(S_i, Q_i)$).

Vulnerabilities

Hashing a subset of each subject introduces new vulnerabilities. Assume that only the static portions of a subject are hashed (i.e. variable components such as data memory are not hashed). One possibility is that it might be possible for an attack to be mounted against the program counter, such that the program counter is given a `JMP` command to a location in data memory, which is a portion of the executable not checked by the hash function.

Another possible vulnerability is that if the state, memory, or some other component of S_i which is not a portion of the hashed subset is compromised, the hash value H_i will not reflect this compromise. For example, by altering a non-hashed component such as the register which holds the stack pointer or the linkage pointer, the behavior of the program can be altered without altering the actual instructions.

Finally, an attacker could potentially compromise Q_i such that the part of S_i which he wishes to compromise is no longer in the subset which is hashed (this is possible because, Q_i is not hashed), and subsequently compromise a component of S_i which is no longer hashed. A countermeasure against this vulnerability is to **automatically include Q_i in the hash**.

The ability to hash and maintain integrity on Q_i follows the same method as that described in part A. Namely, an N-redundant set of security monitors will ensure integrity of the hashes.