Password Systems: Problems and Solutions

Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
via G. Caruso 16, 56126 Pisa, Italy

***Abstract*—In a security environment featuring subjects and objects, we consider an alternative to the classical password paradigm. In this alternative, a key includes a password, an object identifier, and an authorization. A master password is associated with each object. A key is valid if the password in that key descends from the master password by using a validity relation expressed in terms of a symmetric-key algorithm. We analyse a number of security problems. For each problem, a solution is presented and discussed. In certain cases, extensions to the original key paradigm are introduced. The problems considered include the revocation of access authorizations; bounded keys expressing limitations on the number of iterated utilizations of the same key to access the corresponding object; repositories, which are objects aimed at storing keys, possibly organized into hierarchical structures; and the merging of two keys into a single key featuring a composite authorization that includes the access rights in the two keys.**

Keywords*—*Access authorization; key; password; revocation; security

I. Introduction

We will refer to the classical security paradigm featuring active entities, called *subjects*, that generate access attempts to passive entities, called *objects* [15], [16], [20], [24], [29]. A subject can be a process, or the activity generated by the occurrence of an event, e.g. a hardware interrupt. Objects are typed. The definition of the type of a given object includes the specification of a set of values, and a set of operations that act on these values. For each operation, the type definition specifies the *access authorization*, i.e. the set of access rights, that is necessary to execute this operation successfully.

In an environment featuring subjects and objects, a basic problem is to allow subjects to certify permission to access objects, i.e. the subject should possess the corresponding access authorization [6]. A classical solution is based on the association of a number of *passwords* with each object, one password for each significant access authorization [4], [13]. In this solution, a subject that holds a password for a given object, and is aimed at executing a given operation on that object, presents the password to the object. If the password is valid, and the access authorization associated with the password includes the required access rights, then the execution of the operation is permitted.

Password proliferation is an inherent problem in password systems. Let us refer to an object type defining four access rights, for instance. In this type, up to fifteen passwords are necessary, if all access right combinations are meaningful. Significant memory requirements follow from the necessity to store these passwords within the internal representation of each object. Alternatively, we can associate a password with each access right. This solution reduces the memory requirements, but is prone to significant complications of the whole password management process. For instance, a subject that should be granted a full access authorization that includes all the access rights for a given object must possess all the passwords defined for that object. The arguments of an operation requiring several access rights must include as many passwords. A subject that is aimed at passing an access authorization to a recipient must transmit one password for each access right in the authorization.

In a different approach, we associate a *master password* with each object. This password is generated at random when the object is created. Master passwords should be large and sparse, according to the overall security requirements of the system. A subject certifies its own right to access an object whose identifier is id by presenting a *key* K referencing this object. The key has the form $K = (psw, id, au)$, where au specifies an access authorization, and psw is a password. The key is valid if the password is valid, i.e. if $psw = E_{mp}(id \,||\, au)$, where the $||$ symbol denotes a concatenation. In this *validity relation*, E denotes a symmetric-key algorithm, the *password cipher*, which is universally known. The password is valid if it is the result of the application of the password cipher to the concatenation of the identifier and the authorization. The encryption key is the master password of the object identified by id. If the password is valid, possession of the key grants access to the object, to carry out those operations that are authorized by the access rights in au, according to the specification of the object type. The au field features one bit for each access right. If a given bit is asserted, the authorization includes the corresponding access right. Thus, for instance, an au field of all 1's corresponds to a full access authorization including all the access rights. A subject certifies possession of a full access authorization for the given object by a single key, i.e. key $(psw, id, 11\dots1)$.

In this approach, a single password, the master password, needs to be stored into the internal representation of each object, and a single key is necessary in each operation to certify possession of the access authorization required by that operation. A subject that holds an access authorization expressed in terms of a given key can transmit this authorization to a recipient by copying the key to the recipient.

The rest of this paper presents the background, first (Section II). Afterwards, with reference to a key-based method of password specification and storage, a collection of significant problems is analysed, which are connected with password utilization and management. The problems considered include the revocation of access authorizations (Section III); bounded keys aimed at forcing upper limits to the number of successful key utilizations (Section IV); key repositories, which are containers for collections of keys that can be connected to form a hierarchy (Section V); and a mechanism to merge two or more keys into a single key including all the access rights in the respective authorizations (Section VI). For each problem, we present a solution in terms of the corresponding password treatment approach. Extensions to the key format are introduced.

II. Background

Password capabilities are a well-known implementation of the password concept, which was introduced in Section I [5], [9], [10], [14], [21]. Several computing systems implementing an object referencing approach based on password capabilities were designed and actually implemented in the past. Examples are Annex [19], Walnut [4], Mungi [9], Opal [5], and the Password Capability System [1]. In a password capability environment, a set of passwords is associated with each given object, one password for each significant access authorization. A password capability is a pair (psw, id) where psw is a password, and id is an object identifier. A subject that possesses a given password capability can access the named object to carry out those operations which are made possible by the access rights in the authorization associated with the password. If passwords are sparse, large, and generated at random, it is virtually impossible for an attacker to generate a valid password capability from scratch. It follows that password capabilities can be freely mixed in memory with ordinary information items. In this respect, password capabilities are an important improvement over the capability concept [12]. In a classical capability environment, the specification of the access authorization is part of the capability [7], [17], [22]. Consequently, capabilities should be segregated, into reserved memory regions [11], or by taking advantage of memory tagging techniques [2], [3], [18], [28].

Password capabilities suffer from the password proliferation problem. For a type defining several access rights, many passwords should be stored into the internal representation of an object of that type, one password for each meaningful access authorization. Negative effects follow in terms of complicated password management and high memory costs for password storage, especially for forms of fine-grained object access security featuring small-sized objects.

Consider a subject that holds a password capability including the password for a given access authorization. The subject may transfer the authorization to a recipient by passing the password capability to the recipient. In turn, the recipient can transmit the authorization further, by new actions of a password capability copy. Now suppose that the original subject is aimed at revoking the grant from the recipients. If the subject owns the object, it can modify the password. This form of revocation extends automatically to all subjects that hold a password capability expressed in terms of that password. However, we cannot reduce the authorization by eliminating a subset of the access rights associated with the password, and we cannot limit the revocation to a specific subset of the subjects.

Of course, after changing a password, the object owner can proceed to a new distribution of password capabilities with the new password to selected recipients. The whole process is much more complicated than implied by the desired effect. This is especially the case for those subjects that received the password capability through intermediate recipients, which may well be unwilling, or even unable, to cooperate in the new distribution. In fact, one of the main advantage of password capability systems is simplicity in access right transmission between subject. This simplicity should be also preserved for revocation.

No bound exists on the transmission ability of a subject that holds a given password capability. In fact, the subject is free to pass the password capability to an unlimited number of recipients. In turn, each recipient can transmit the password capability further. In the original definition of the password capability concept, no mechanism is provided to limit this form of password capability proliferation.

The password capability format does not include an authorization field. It follows that we cannot argue the authorization granted by a given password capability by inspection of the password capability itself. In fact, in the original password capability paradigm, the association between a password for a given object and the authorization granted by that password is part of the internal representation of the object. An *ad-hoc* operation would be necessary to convert passwords into the corresponding authorizations.

III. Revocation

At the security system level, the key-based approach introduced in Section I is supported by a collection of system primitives for object and key management (Fig. 1). A first example is $K \leftarrow new(T, arg_0, arg_1, \dots)$. In the execution of this primitive, the constructor of type T is used to create a new object of this type, according to the specifications of the type. Arguments $arg_0, arg_1, \dots$ are transmitted to the constructor. The primitive returns a key referencing the new object, with a full access authorization that includes all access rights. Primitive $delete(K)$ uses the destructor of the type of the object referenced by key K to delete this object. The execution terminates correctly only if K specifies an access authorization that includes access right OWN. Primitive $exec(K, op, arg_0, arg_1, \dots)$ executes operation op on the object referenced by key K. Arguments $arg_0, arg_1, \dots$ are transmitted to op. The execution terminates correctly only is K specifies an access authorization that includes all the access rights required by op. Finally, let $K_0 = (psw_0, id, au_0)$ be a key, and msk be a mask having the same size as an authorization. Primitive $K_1 \leftarrow reduce(K_0, msk)$ returns a key $K_1 = (psw_1, id, au_1)$ referencing the same object as key K_0, with the reduced authorization au_1 that results from relation $au_1 = au_0 \ \& \ msk$, i.e. the bitwise AND of authorization au_0 and mask msk. The execution of this primitive uses the validity relation and the master password of the object identified by id to evaluate the new password psw_1.

One of the main advantages of a password-based environment is simplicity in access right distribution. Consider a subject that holds the password corresponding to a given authorization. The subject can grant this authorization to one or more recipients by simply distributing a copy of the password to these recipients. In turn, each recipient can grant the authorization to additional subjects, by further password copy

---

$K \leftarrow new(T, arg_0, arg_1, \dots)$
Uses the constructor of type T to create a new object of this type. Arguments $arg_0, arg_1, \dots$ are transmitted to the constructor. Returns a key referencing the new object, with a full access authorization that includes all access rights.

$delete(K)$
Uses the destructor of the type of the object referenced by key K to delete this object. K should specify access right OWN.

$exec(K, op, arg_0, arg_1, \dots)$
Executes operation *op* on the object referenced by key K. Arguments $arg_0, arg_1, \dots$ are transmitted to *op*. K should specify all the access rights required by *op*.

$K_1 \leftarrow reduce(K_0, msk)$
Returns a key $K_1 = (psw_1, id, au_1)$ referencing the same object as key $K_0 = (psw_0, id, au_0)$, where $au_1 = au_0 \ \& \ msk$, the & symbol denotes a bitwise AND, and mask msk has the same size as an authorization.

Fig. 1. Primitives for Object and Key Management.

actions. We will now consider the case that the original subject modifies its own intention, and is aimed at revoking the grants from the recipients. Revocation is especially useful to comply with the *principle of least privilege*: at any given time, each subject should possess only those access privileges that are necessary at that time for its legitimate purposes [23], [25]–[27].

Of course, by changing the password associated with a given authorization we obtain a form of revocation that includes only this authorization. The revocation involves all the subjects that received the authorization in the form of that password. Restricting revocation to a subset of these recipient subjects is a problem that is hard to solve. We can modify the password, and then proceed to a distribution of the new password to the desired recipients. However, consider the case of a recipient that was reached by means of two or more distribution steps through intermediate subjects. Collaboration will be necessary in the new distribution, but these intermediate subjects may well be unwilling, or even unable, to cooperate.

In our key-based environment, consider a subject that holds a key for a given object, and distributes a copy of this key to one or more recipients. A form of total revocation that involves all access authorizations can be obtained at little effort by a system primitive having the form $K_1 \leftarrow mpReplace(K)$. The execution of this primitive modifies the master password of the object referenced by key K, whose authorization should specify all access rights. The primitive returns a key K_1 defined in terms of the new master password and including all access rights. After the execution of this primitive, all keys generated by using the old master password are revoked; it will no longer possible to use these keys for successful object accesses. However, it is impossible to take advantage of an approach of this type to implement forms of revocation that involve only a subset of the recipients, or only a subset of the access rights. We will now introduce more flexible approaches to the solution of the revocation problem.

A. Instances

A first approach is based on a different form of the access authorization field. In *au*, we associate more than a single bit with each access right. This means that we can have several

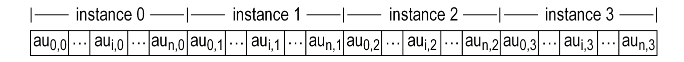
instance 0
instance 1
instance 2
instance 3
$au_{0,0}$ ··· $au_{i,0}$ ··· $au_{n,0}$ $au_{0,1}$ ··· $au_{i,1}$ ··· $au_{n,1}$ $au_{0,2}$ ··· $au_{i,2}$ ··· $au_{n,2}$ $au_{0,3}$ ··· $au_{i,3}$ ··· $au_{n,3}$

Fig. 2. An Access Authorization Field Featuring Four Instances of n Access Rights.

instances of the same access right. Fig. 2 considers an example of an access authorization field featuring four instances of n access rights. For the i-th access right in the authorization, these instances are named $au_{i,0}$ to $au_{i,3}$. If bit $au_{i,j}$ is asserted, then the access authorization includes instance j of access right i. The internal representation of each object is modified to contain an authorization *mask*, which applies to all keys referencing that object. The structure of the mask is similar to that of an authorization field. If $mask_{i,j}$ is cleared, then $au_{i,j}$ is revoked.

When a subject generates an access attempt to a given object, the key presented by the subject is considered to certify the access. The access is permitted if, for each access right required by the access, at least one instance is asserted in the authorization field of the key, and this instance is not revoked by the corresponding mask bits. Conversely, the object access is negated if all instances of one or more of the required access rights are cleared, or are revoked by the mask. Let *eff* denote the *effective* authorization resulting from the bitwise AND of the authorization field and the mask; thus we have *eff* $= au \ \& \ mask$. The i-th access right is granted by the key if at least one instance of this access right is asserted in *eff*, that is, for m instances, $\mathit{eff}_{i,0} \vee \mathit{eff}_{i,1} \vee \dots \vee \mathit{eff}_{i,m} = 1$. In the mask, the bits corresponding to instance 0 are always asserted for all access rights. It follows that, in a key, an access right in instance 0 is never revoked.

The primitives to create and access objects, introduced previously and illustrated in Fig. 1, should be modified to deal with effective authorizations. When primitive *new* is issued to create a new object, the mask of that object is set to all 1's, to validate all access right instances. The key returned by *new* features an access authorization that includes all access rights in all instances. The execution of primitive $delete(K)$ terminates successfully only if the effective authorization granted by key K includes access right OWN. In the execution of primitive $exec(K, op, arg_0, arg_1, \dots)$, the effective authorization should include the access rights required by the operation specified by argument *op*.

The mask of a given object can be modified by issuing primitive $mask(K, msk)$. Argument K is a key referencing the object, argument msk is the new mask, which should specify all 1's for instance 0. The execution accesses the internal representation of the object to modify the mask. The operation terminates successfully only if key K specifies all access rights in instance 0.

B. Categories

In a different approach to access right revocation, we extend the key format to include the specification of a *category*. A key assumes the form $K = (psw, id, t, au)$, where the t field specifies the category. Each category has a *degree*. The degree of a given category expresses a limitation on the access rights granted by every key in that category.

$changeDegree(K, t, d)$
Assigns degree d to category t of the object referenced by key K, which should specify category 0.

$K_1 \leftarrow changeCategory(K, t)$
Returns a new key in category t for the object referenced by key K, with the same access authorization. K should specify category 0.

Fig. 3. Primitives for Category Management.

Let us refer to an object type featuring n access rights. As seen in Section I, authorization field au is encoded in n bits, one bit for each access right. If a given bit is asserted, the corresponding access right is part of the authorization. The internal representation of each object is modified to include a *category table* featuring an entry for each category. The entry for a given category contains the degree of that category. A degree is encoded in n bits. For a given key, the *effective* authorization results from the bitwise AND of the au field and the degree of the category specified by the t field. It follows that a degree of all 1's for a given category implies that all the access rights in the authorization of a key in that category are effective. Conversely, a degree of all 0's means that all these access rights are revoked.

In an extended key environment featuring categories, the primitives for object management, introduced previously and illustrated in Fig. 1, should be modified to deal with degrees. Primitive *new* assigns degree 0 to all categories, and returns a key featuring category 0. This category is special in that its degree always is all 1's, and cannot be changed. The execution of primitives *delete* and *exec* considers the effective access authorization granted by key K, as follows from the compound effect of the authorization field and the degree of the category specified by the key.

Category management is supported by two primitives (Fig. 3). Primitive $changeDegree(K, t, d)$ makes it possible to modify category degrees. Its execution assigns degree d to category t of the object referenced by key K. The execution is successful only if K specifies category 0. Primitive $K_1 \leftarrow changeCategory(K, t)$ returns a new key in category t for the object referenced by key K, with the same access authorization. The execution is successful only if K specifies category 0.

C. Comparison

Let us refer to the classical properties of an access right revocation system [8]. In the approach based on access right instances, revocation is *partial*, that is, we can revoke any desired subset of the access rights. To this aim, we clear the mask bits corresponding to these access rights in all instances. Revocation is *selective*, that is, we can revoke an access right from a subset of the recipients of that access right. To this aim, we clear the mask bits of the instances specified by the keys held by these recipients. Revocation is *independent*, that is, keys received from different distributors can be revoked independently of each other, if these keys specify different instances of the same access rights. Revocation is *transitive*, that is, it propagates to all copies of the same key, independently of the path followed by the copy to reach its recipient; and in fact, a key copy cannot be distinguished from the original. Revocation is *temporal*, as it can be reversed through the same mechanism, i.e. the mask, by setting the mask bits that were cleared for revocation.

In the category-based approach, we can obtain a partial revocation by clearing the bits of the degree field corresponding to the access rights to be revoked from the category. Selective revocation is intrinsic in the category model, and will be simply obtained by clearing the degree of selected categories. Independent revocation can be obtained at little effort for keys belonging to different categories, by modifying the degree of only those categories that are involved in the revocation; the other degrees will be left unaltered. Transitive revocation is implicit in the key model. Finally, temporality can be obtained by simply setting the bits of the degree that were cleared for revocation to reverse the effects of the revocation.

Instances imply no modification of the key format. In fact, instances need to be introduced only in those object types for which an option for revocation is necessary. In these types, the access authorization field will be extended to include several bits for each access rights. The size of the extension will be decided on a type basis. In fact, we can have different numbers of instances for different types. Conversely, in the category based approach, the key format should be extended to include the category field. This modification applies to all types. The number of categories is fixed for all types, and is determined by the size of the category field.

The memory costs of instances are connected with mask storage and the size of the access authorization field, which is increased to include several bits for each access rights. For four access rights and four instances of each access right, a two-byte access right field and a two-byte mask will be necessary. These memory costs are to be paid only for those object types for which revocation is necessary. On the other hand, a four-bit category field is sufficient to implement up to 16 categories. The memory costs for storage of the category table in the internal representation of each object are quite limited. Let us refer to an object type defining four access right, for instance. In a situation of this type, for 16 categories, a 64-bit word will be sufficient.

IV. Bounded Keys

The password paradigm, as is implemented by the key construct, implies no limitation on iterated utilizations of the same given key to access the corresponding object. We will now present an extension of the key concept aimed at forcing an upper bound to the number of successful applications of the same key.

We modify the key format by introducing a new field, the *bound field* b. In the new format, a *bounded key* B is a quadruple $B = (psw, id, au, b)$ (Table I). The validity relation is modified to take the bound field into account. The key is valid if $psw = E_{mp}(id \mid\mid au \mid\mid b)$, where *mp* is the master password of the object identified by id, au is the authorization granted by the key, and b identifies the bound. A *bound table* is associated with each object. The bound table features an entry for each bound. The entry for a given bound contains the *extent* of that bound. The extent is the total number of times that bounded keys in that bound can be successfully used to access the object (if the extent is 0, these keys can no longer

TABLE I. Bounded Key Format

$B = (psw, id, au, b)$: bounded key
$psw = E_{mp}(id \mid\mid au \mid\mid b)$: validity relation
psw: password
id: object identifier
au: access authorization
b: bound
E: password cipher
mp: master password

be used). If the bound is 0, then the key is a *primary key* that has no bound, and can be used for an unlimited number of accesses to the object.

The bound table of a given object will be stored as part of the internal representation of the object. The memory requirements for storage of the bound table are determined by the number of bounds and the maximum extent permitted for each bound. For instance, a bound field of three bits allows for up to 7 bounds (bound 0 being reserved to specify a primary key). If an extent is encoded in 16 bits, for each bound we can have up to 65,535 executions of the *exec* primitive using a bounded key in that bound. In a configuration of this type, the whole bound table can be contained in two 64-bit words.

A. Primitives

The primitives for object management, introduced in Section III and illustrated in Fig. 1, should be modified to deal with bounds and extents. Primitive $B \leftarrow new(T, arg_0, arg_1 \dots)$ uses arguments $arg_0, arg_1, \dots$ in the constructor of type T to create a new object of this type, and returns a primary key for that object, which includes all acces rights. The extents of the bounds of the object are all equal to 0. This means that the object can only be accessed by using the primary key, until one or more bounds are *recharged* to specify new extents (see below). Primitive $delete(B)$ uses the destructor of the type of the object referenced by bounded key B to delete the object. B should be a primary key, and should include an access authorization that specifies access right OWN. Primitive $exec(B, op, arg_0, arg_1, \dots)$ uses arguments $arg_0, arg_1, \dots$ to execute operation *op* on the object identified by *id*. The execution is successful if B specifies an access authorization that includes all the access rights required by *op*. B should be a primary key, or the extent of the bound of B should be greater than 0. If this is the case, the extent is decremented by 1.

The primitives for bound management are summarized in Fig. 4. Primitive $e \leftarrow extent(B, b)$ returns the extent e of the bound b of the object referenced by bounded key B, which should be a primary key. Primitive $recharge(B, b, e)$ increments the extent of the bound b of the object referenced by bounded key B by quantity e. B should be a primary key. Primitive $B_1 \leftarrow newBound(B_0, b)$ returns a bounded key B_1 referencing the same object as bounded key B_0, with the same access authorization and bound b. B_0 should be a primary key.

V. Repositories

The *Repository* data type allows us to define objects aimed at key storage [15]. A name is associated with each key in a repository. The name is unique within the repository, that is, it will never be the case that two keys in a given repository are associated with the same name (on the other hand, the same key name can be freely used in different repositories).

$e \leftarrow extent(B, b)$
Returns the extent e of the bound b of the object referenced by bounded key B, which should be a primary key.

$recharge(B, b, e)$
Increments the extent of the bound b of the object referenced by bounded key B by quantity e. B should be a primary key.

$B_1 \leftarrow newBound(B_0, b)$
Returns a bounded key B_1 referencing the same object as bounded key B_0, with the same access authorization and bound b. B_0 should be a primary key.

Fig. 4. Primitives for Bound Management.

TABLE II. Access Rights in the *Repository* Type.

Access right	Operation
GET	*read*
PUT	*write*
INSPECT	*list*
OWN	*delete*

Table II enumerates the access rights that are included in the definition of the *Repository* type. For each access right, the table shows the operation whose execution is made possible by that access right. For a given repository, access right GET makes it possible to read those keys in the repository whose names are known. Access right PUT makes it possible to insert keys into the repository. Access right INSPECT allows us to read the names of the keys in the repository. Access right OWN allows us to delete the repository.

Fig. 5 presents the operations defined by the *Repository* type, and gives short indications of the effects of the execution of each of them. Primitive $K \leftarrow new(Repository)$ uses the constructor of the type to create a new, empty repository, and return a key referencing that repository, with a full access authorization that includes all the access rights. Primitive $delete(K)$ uses the destructor to delete the repository referenced by key K. This key should specify access right OWN.

The other operations of the *Repository* type are implemented taking advantage of primitive $exec$. A first example is $K_1 \leftarrow exec(K_0, read, nm)$. The execution of this operation accesses the repository referenced by key K_0 to return the key named nm in that repository. The execution is successful if K_0 specifies access right GET. Operation $exec(K_0, write, nm, K_1)$ adds key K_1 to the repository referenced by key K_0, and associates name nm to K_1. K_0 should specify access right PUT. Operation $lst \leftarrow exec(K, list)$ returns a list of the names of the keys contained in the repository referenced by key K. This key should specify access right INSPECT.

A. Hierarchies

The *Repository* object type makes it possible to organize keys into hierarchies. In an organization of this type, each repository can include keys for other repositories at a lower

$K \leftarrow new(Repository)$
Uses the constructor of the *Repository* type to create a new, empty repository. Returns a key referencing that repository, with an access authorization that includes all access rights.

$delete(K)$
Uses the destructor of the *Repository* type to delete the repository referenced by key K, which should specify access rigth OWN

$K_1 \leftarrow exec(K_0, read, nm)$
Returns the key named nm taken from the repository referenced by key K_0, which should specifies access right GET.

$exec(K_0, write, nm, K_1)$
Adds key K_1 to the repository referenced by key K_0, and associates name nm to K_1. K_0 should specify access right PUT.

$lst \leftarrow exec(K, list)$
Returns a list of the names of the keys stored in the repository referenced by key K, which should specify access right INSPECT.

Fig. 5. Operations of the *Repository* Type.

hierarchical level. Each repository can also include keys for objects of any other type, which represent the leaves of the hierarchy. In a given repository, the name associated with each key identifies the object referenced by that key.

A subject that knows the name of a key in a given repository, and possesses a key for that repository with access right GET, can access the repository to read the key. If the subject does not know the key name, it can use the *list* operation, but an action of this type requires access right INSPECT for the repository.

B. Access Right Amplification

The *read* operation implements a form of access right amplification, whereby a subject that possesses access right GET for a given repository can read the keys in that repository independently of the access rights specified by these keys. For instance, consider the case of a repository R_0 that includes a key K referencing another repository R_1, and K specifies all access rights. A subject that possesses a key for R_0 featuring a single access right, GET, can read K from R_0. In this way, the subject acquires a full access authorization for R_1, which is an amplification of the authorization that the subject holds for R_0.

VI. Merging Keys

Let us refer to a subject S that holds two keys that reference the same object, say $K_0 = (psw_0, id, au_0)$ and $K_1 = (psw_1, id, au_1)$, where id is the object identifier. Let us suppose that the type of the object includes an operation *op* whose execution requires both the access rights in au_0 and the access rights in au_1. Primitive $exec(K, op, arg_0, arg_1, \dots)$ features a single key, whose authorization field should include all the required access rights. It follows that subject S is not in the position to execute *op*, unless the two keys K_0 and K_1 are merged to form a single key including the union of the access rights in the authorizations. To this aim, primitive $K \leftarrow merge(K_0, K_1)$ can be provided. The execution of this operation returns a key $K = (psw, id, au)$ for the object identified by id. K features an authorization au that includes the union of the access rights in au_0 and au_1.

It should be noted that *merge* implement a form of access right amplification. In the example above, by using *merge*, subject S amplifies its own execution ability to include operation *op*, which would be negated in the absence of *merge*. In fact, the decision to include *merge* in the set of primitives of the security system is a design choice. If this form of access right amplification should be permitted, *merge* will be made available.

In a more flexible approach, an *ad-hoc* access right, the JOIN access right, will be required in all the keys involved in an access right merging activity. In this case, in the example above, a successful execution of primitive *merge* will be possible only if both keys K_0 and K_1 include JOIN in the respective authorizations.

Primitive *merge* also supports a form of cooperation between subjects. Consider two subjects S_0 and S_1 that hold keys K_0 and K_1, respectively. These subjects cannot execute an operation requiring the union of the access rights in au_0 and au_1, unless they agree to merge the two keys.

VII. Conclusion

With reference to a security system featuring subjects and objects, we have considered a paradigm of object access, which is an alternative to classical password-based environments. Our paradigm takes advantage of keys. In particular:

- The key definition includes a password, an object identifier, and an authorization.
- A master password is associated with each object. A key is valid if the password descends from the master password by using a validity relation expressed in terms of a symmetric-key algorithm.

We analysed a number of security problems, which include:

- The revocation of access authorizations.
- Bounded keys expressing limitations on the number of iterated utilizations of the same key to access the corresponding object.
- Repositories, which are objects aimed at storing keys, possibly organized into hierarchical structures whereby each repository may include keys for other repositories at a lower hierarchical level.
- The merging of two keys into a single key featuring a composite authorization that includes all the access rights in the two authorizations.

For each problem, we have proposed a solution expressed in terms of a key treatment approach. Extensions to the original key format have been introduced and discussed.

Acknowledgment

This work was supported in part by the Italian Ministry of University and Research (MUR) in the framework of the *CrossLab* project (Departments of Excellence).

References

[1] M. Anderson, R. D. Pose, and C. S. Wallace, "A Password-Capability System," *The Computer Journal*, vol. 29, pp. 1–8, February 1986.

[2] K. M. Bresniker, P. Faraboschi, A. Mendelson, D. Milojicic, T. Roscoe, and R. N. Watson, "Rack-scale capabilities: fine-grained protection for large-scale memories," *Computer*, vol. 52, no. 2, pp. 52–62, 2019.

[3] J. Brown, J. Grossman, A. Huang, and T. F. Knight Jr, "A capability representation with embedded address and nearly-exact object bounds," tech. rep., Project Aries, ARIES-TM-005, Artificial Intelligence Laboratory, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.

[4] M. D. Castro, R. D. Pose, and C. Kopp, "Password-capabilities and the Walnut kernel," *The Computer Journal*, vol. 51, no. 5, pp. 595–607, 2008.

[5] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey, "Lightweight shared objects in a 64-bit operating system," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Vancouver, British Columbia, Canada), pp. 397–413, ACM, October 1992.

[6] S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "Access control: principles and solutions," *Software – Practice and Experience*, vol. 33, no. 5, pp. 397–421, 2003.

[7] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal, "Efficient and provable local capability revocation using uninitialized capabilities," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–30, 2021.

[8] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 575–586, November 1979.

[9] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The Mungi single-address-space operating system," *Software – Practice and Experience*, vol. 28, pp. 901–928, July 1998.

[10] J. King-Lacroix and A. Martin, "BottleCap: a credential manager for capability systems," in *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, (Raleigh, NC, USA), pp. 45–54, ACM, October 2012.

[11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, (Big Sky, MT, USA), pp. 207–220, ACM, October 2009.

[12] H. M. Levy, *Capability-Based Computer Systems*. Bedford, Mass., USA: Digital Press, 1984.

[13] L. Lopriore, "Password capabilities revisited," *The Computer Journal*, vol. 58, pp. 782–791, April 2015.

[14] L. Lopriore, "Access right management by extended password capabilities," *International Journal of Information Security*, vol. 17, pp. 603–612, October 2018.

[15] L. Lopriore and A. Santone, "Extended pointers for memory protection in single address space systems," *Computers & Electrical Engineering*, vol. 82, pp. 1–14, 2020.

[16] M. S. Miller and J. S. Shapiro, "Paradigm regained: abstraction mechanisms for access control," in *Proceedings of the 8th Asian Computing Science Conference*, (Mumbai, India), pp. 224–242, Springer, December 2003.

[17] Y. Nakamura, Y. Zhang, M. Sasabe, and S. Kasahara, "Exploiting smart contracts for capability-based access control in the Internet of Things," *Sensors*, vol. 20, no. 6, p. 1793, 2020.

[18] P. G. Neumann and R. J. Feiertag, "PSOS revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference*, (Las Vegas, NV, USA), pp. 208–216, IEEE, December 2003.

[19] T. Newby, D. A. Grove, A. P. Murray, C. A. Owen, J. McCarthy, and C. J. North, "Annex: a middleware for constructing high-assurance software systems," in *Proceedings of the 13th Australasian Information Security Conference*, (Sydney, Australia), pp. 25–34, ACS, January 2015.

[20] F. Paci, A. Squicciarini, and N. Zannone, "Survey on access control for community-centered collaborative systems," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–38, 2018.

[21] R. Pose, "Password-capabilities: their evolution from the Password-Capability System into Walnut and beyond," in *Proceedings of the Sixth Australasian Computer Systems Architecture Conference*, (Gold Coast, Australia), pp. 105–113, IEEE, January 2001.

[22] A. Randal, "The ideal versus the real: revisiting the history of virtual machines and containers," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–31, 2020.

[23] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[24] P. Samarati and S. De Capitani Di Vimercati, "Access control: policies, models, and mechanisms," in *Foundations of Security Analysis and Design* (R. Focardi and R. Gorrieri, eds.), pp. 137–196, Berlin, Heidelberg: Springer, 2001.

[25] M. W. Sanders and C. Yue, "Mining least privilege attribute based access control policies," in *Proceedings of the 35th Annual Computer Security Applications Conference*, (San Juan, Puerto Rico, USA), pp. 404–416, December 2019.

[26] F. B. Schneider, "Least privilege and more," *IEEE Security & Privacy*, vol. 1, pp. 55–59, September 2003.

[27] H. Shrobe, D. L. Shrier, and A. Pentland, "Fundamental trustworthiness principles in CHERI," in *New Solutions for Cybersecurity*, pp. 199–236, MIT Press, 2018.

[28] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. Murdoch, R. Norton, M. Roe, S. Son, and V. Munraj, "CHERI: a hybrid capability-system architecture for scalable software compartmentalization," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, (San Jose, California, USA), May 2015.

[29] X. Zhang, Y. Li, and D. Nalla, "An attribute-based access matrix model," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, (Santa Fe, New Mexico, USA), pp. 359–363, ACM, March 2005.