Security as an Add-On Quality in Persistent Object Systems[1]

Andreas Rudloff Florian Matthes
Joachim W. Schmidt
University of Hamburg, Dept. of Computer Science,
Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany
rudloff@informatik.uni-hamburg.de

Abstract

System security services like authentication, access control and auditing are becoming increasingly critical for information systems particularly in distributed heterogeneous environments. Since information system architectures are moving rapidly from centralized, *grand unifying architectures* towards open, service-oriented and communication-based environments ("Persistent Object Systems") constructed with well-organized component technologies it is essential that such structural changes are reflected adequately in the architecture of system security services.

In this paper we present an open, library-based approach to the security of Persistent Object Systems which generalizes and unifies the protection mechanisms that traditionally come bundled with database, communication or operating system services. More specifically, we illustrate how polymorphic typing can be exploited to abstract from particular commercially available security services, such as Kerberos, and how higher-order functions allow the user to add value to existing security services. Furthermore, we demonstrate how higher-order functions, first-class modules and reflection provide a technical framework for the realization of domain-specific security policies and for the systematic construction of security-enhanced activities.

1 Introduction

Customer demands on information systems in the commercial, public and private sectors require an ever-increasing number of services such as bulk data storage, data persistence, transactions, network data communication, window-based data visualization or data retrieval. Consequently, there is a strong market pressure to factor-out such services into generic libraries and to facilitate the safe use and free combination of such services. Furthermore, the required functionality is offered through a variety of service providers such as operating systems, database systems, language processors, directory servers, communication services or "middleware" toolkits.

[1]This research is supported by ESPRIT Basic Research, Project FIDE, #6309 and by a grant from the German Israeli Foundation for Research and Development (*bulk data classification*, I-183 060).

The Tycoon[2] *Persistent Object System* addresses this situation and combines a higher-order polymorphic language (**T**ycoon **L**anguage TL) with orthogonal persistence to implement directly the desired services and to integrate existing external services via type-safe interfaces to foreign libraries (written in languages like C or C++). Tycoon achieves thereby type-safe "plug and play" interaction between objects on screen, objects in memory, objects on disk, and objects on the wire in open heterogeneous environments. This freedom in the manipulation of persistent objects immediately raises security issues like the protection of sensitive or personal data held in shared repositories or the control and billing of resource consumption (time, space, communication bandwidth).

Traditionally, the security mechanisms required for authentication and authorization came hard-wired with a specific service provider like a database management system. Since it is impossible to cover an entire application domain that involves different service providers with a unique security model and it is also impossible to tailor a hard-wired security mechanism to specific application requirements. In this paper an add-on approach to security is proposed based on generic security libraries. It is clear that an add-on approach to security requires a certain degree of built-in security support in an integrated framework that contains all computational objects like values, functions and threads.

Section 2 introduces basic security notions. A presentation of the Tycoon framework is given in section 3 including an insecure example of a classical database application. The reinterpretation of security in the Tycoon framework and its integration is shown in section 4 together with a secure version of the example presented above. Since all this is done on a purely application level, the underlying built-in system support is described in section 5. Using the library-based security capabilities of Tycoon the correct development of secure applications can be simplified using generators as detailed in section 6. Final conclusions are discussed in section 7.

2 Basic Security Notions

Security[3] in persistent application systems is based on *authentication* and *authorization*. The semantics of these terms can be explained by the example of a distributed environment. In such an environment active entities can be identified (and named) uniquely on a purely logical level not restricted by machine or application boundaries. These active entities which are mainly users or processes (acting for itself or on behalf of a user) are the *principals* [BAN89] of the environment. Their counterparts are the passive entities which are part of an application having an identity only inside their application and which are controlled by a principal. A typical passive entity is a table in a relational database controlled by a database management system.

An active environment is characterized by the interaction of its principals. Since this may happen across machine boundaries via open potentially insecure networks there is no identity guaranteed by an operating system. This raises the need for a trusted authentication mechanism [SNS88, Lin93] between the

[2]Typed Communication Objects in Open Environments

[3]Security aspects like availability and integrity are not covered by this paper.

principals. The heart of such a mechanism is an *Authentication Server* administered by a (environment-) global authority. Using these protocols based on cryptographic algorithms [DES77, RSA78] each and every principal can construct a *credential* [FL93] containing the cryptographic information allowing another principal to verify the claimed identity. In addition, this information enables the creation of passive entities labeled with the identity of their creating principal which guarantees the provability of origin and lack of modifications. On a technical level this is done by adding a cryptographic checksum as *signature* to the entities. These *safe* objects can further more be made *private* through enciphering.

Authentication is a necessary requirement for answering the central security question of as to whether a *principal has the authorization to execute an action on an entity*. While the desired access can be done only by an active entity, a principal, the accessed entity can be either an active one (another principal) or a passive one. It is not necessary, however, that every principal has to be able to be both accessor and accessee. For this reason, authorization distinguishes between *subjects* which are principals in the role of an accessor, and *objects* [Mul91], which are active or passive entities in the role of an accessee. Assuming a successful authentication, it is the task of the principal representing or controlling the object the object to decide whether this access is to be granted or not. This task requires an internal information base structured by rules describing a specific *access control model (security model)*.

If an access control model allows the *creator* of an entity to decide on the subjects to which entity access will be granted, the model realizes *discretionary access control* [DRKJ85, Vin85]. A typical example of discretionary access control models are *access control lists*. They are attached to the object where access should be controlled and contain subjects and the type of access which will be allowed for these subjects.

This is in contrast to *mandatory access control models* [BL73, BN89, Mil89, TCS85] where a system-wide security model defines the allowed access patterns without further influence of the entity creators. Most of these models define a partially ordered set of security levels and assign a security level to every object and every subject. Read access will only be granted if the security level of the accessing subject dominates the security level of the accessed object; write access will only be granted if the security level of the object dominates the security level of the subject. One main difference in the mandatory access control models is that they require strict enforcement of the security model in the whole system environment whereas discretionary access control is restricted to the application environment. Therefore, discretionary access control can be used on top of a system guaranteeing mandatory access control.

3 The Rationale Behind Tycoon

The Tycoon Persistent Object System is an example of a software system that gives users flexible, problem-oriented, safe access to large sets of long-lived objects of various types [SM93, Mat93]. In Tycoon, the main abstractions necessary for the construction of such systems, namely functions, polymorphic types and persistence, have been identified and generalized. The system consists of three layers related hierarchically.

The *top layer* consists of Tycoon's higher-order language TL where *functions* and types are treated as first-class language objects thus allowing the user to write generic libraries and generators without leaving Tycoon's language framework. The semantic model of Tycoon is based on higher-order type theories [Car88]. The core semantic entities of Tycoon are values, types, bindings and signatures [BL84, Car89]. Values and types can be named in bindings for identification purposes and in order to introduce shared or recursive structures at the value and type levels. Signatures act as (partial) specifications of static and dynamic bindings. Bindings are embedded into the syntax of values, i.e. they can be named, passed as parameters, etc. Accordingly, signatures appear in the syntax of types to describe these aggregated bindings.

Higher-order functions imply *function parameterization* which enables programmers to pass functions dynamically as arguments to other functions and *function generation*, i.e., the possibility of returning functions as the result of functions. In supplying generic data structures like relations or stacks in the application framework, user-defined higher-order type operators are provided. They denote parameterized type expressions that map types or type operators to types or type operators. For example, the type operator *Pair* takes any type X that is a subtype of the trivial type **Ok** and returns a tuple type with two fields of type X:

Let *Pair*=***Oper****(X <:****Ok****)* ***Tuple*** *fst:X snd:X* ***end*** type operator binding

These generic types can be instantiated later with type parameters to construct application-specific types like

Pair(Int), Pair(String), Pair(Pair(Int)) type operator applications

Large TL programs are typically divided into modules, interfaces and hierarchically nested libraries with support for separate compilation and dynamic linking by the language processor. They are translated to *TML* (Tycoon Machine Language) [GM94] thereby entering the *second layer*. TML is a minimal intermediate language based on an untyped lambda calculus and extended with imperative constructs that serve as a low-level, portable TL program representation in distributed heterogeneous environments. TML was designed to support efficient host-specific target code generation as well as dynamic optimizations analogous to query and transaction rewriting in database systems. Execution is performed by the Tycoon Machine, TM, represented by a set of *threads* that act as the unit of execution for TML code. These threads are first-class objects and as such they are available as TL-values and can also be made persistent [MS94].

Finally, persistence of all *values* is realized in the *third layer* via *TSP* (Tycoon Store Protocol), a data-model-independent object store protocol based on the notion of a *persistent heap* that shields TML evaluators (and TL programmers) from operational aspects of the underlying persistent store like access optimization, storage reclamation, concurrency or recovery. By forcing all higher levels of the system to use the TSP (software) protocol, it provides a starting point to add system functionality at the object store level (e.g. distribution transparency). A key contribution of the TSP to the overall Tycoon system functionality is support for *orthogonal persistence* [AB87]: data of any type (including functions) can exist for any length of time or as short as required by the application. Programmers do not need to write explicit code to move data between persistent and volatile store.

A classical persistent application is the storage of bulk data. For example, a database storing personal records consisting of a name and an address can be modeled by an abstract data type (ADT) `Person.T` and implemented in Tycoon as follows:

```
interface Person
  T <: Ok
  new(name :String  address :String) :T
  get(name :String) :T
  name(person :T) :String
  address(person :T) :String
end
```

It is assumed that the corresponding implementation of this interface does not contain any security mechanisms. Furthermore, the application requires that access to the personal data is restricted to authorized users and that the set of authorized users for the name attributes differs from the set for the address attributes of a specific person tuple (of type `Person.T`). The use of the Tycoon security libraries (see section 4.2) will be demonstrated by securing this simple application.

4 An Add-On Approach to Security

Traditionally, security is being handled as a *built-in* system feature. This can be observed in the fact that security is not a feature where it is up to the user to apply it or not but it is a *restriction* that must be enforced, especially with respect to users who want to circumvent this restriction intentionally. The high cost of the the built-in approach has as a consequence that only a few systems, mainly database and operating systems, possess integrated security components.

Another key disadvantage of built-in security components is their lack of flexibility and exchangeability in cases where they do not fit the application requirements. In addition, it is a much too narrow view that only values in a database or operating system objects must be put under access control. In a programming environment access to every creatable object regardless of its type has to be controllable by a matching access control model. This requirement leads to the development of an open add-on approach to security as exemplified in the following sections by the Tycoon security model.

4.1 Tycoon's View on Security

In section 2 security control is defined as the task of defining and deciding whether *a principal has the authorization to act on an entity.* This *security task* is tackled in the Tycoon framework by relating the security concepts introduced in section 2 (i.e., principal, action, entity) to Tycoon's computational entities as outlined in section 3 (i.e., thread, function, value).

Intuitively speaking in Tycoon the three central security questions are answered as follows:

Who is active principal? All application activities are represented and controlled by Tycoon *threads*.

What action is performed? Application actions are abstracted by Tycoon *functions*.

Which entity is involved? All abstractions of Tycoon are *values* and have first-class status.

This leads to a reformulation of our security task:

Has the TL thread at hand the right to apply the intended TL function to a given TL value?

In the example in section 3 a principal is a client's application program being executed by a thread (on behalf of a user) that accesses an ADT-value (of type `person.T`) with ADT-functions.

In section 4.2 we present the security concepts introduced in section 2 as Tycoon add-on libraries. The central issue of how such security libraries can be *securely* added to a Tycoon kernel and which basic security support must already be built-in remains open until section 5.

4.2 Add-On Security Libraries

In the Tycoon environment with its inherent add-on approach [MS93] the applications security needs are realized by implementing the basic security concepts presented in section 2 as polymorphic libraries which are described in this section. It should be noted that activities needing security in a distributed environment are characterized by at least two principals, one client and one server. For activity execution they have to communicate with each other in a secure way. The underlying communication abstractions (interprocess communication, RPCs, ...) including details of the authentication protocols on top of the security libraries are outside the scope of this paper and will not be discussed in the following.

4.2.1 Authentication

Authentication in a distributed programming environment requires agreement on a protocol and a supporting infrastructure which consists mainly of authentication servers administered by a trusted authority. On the one hand all of these services could be implemented completely in Tycoon itself but, on the other hand existing standardized authentication systems with a C-API (like, for example, the Kerberos system [SNS88]) are to be preferred. In the latter case, access to the authentication services is done via a type safe TL-interface using the C-call mechanism of the Tycoon system. In both cases the authentication services are presented to the application programmer by a common interface with exchangeable implementations.

Under the assumption of an existing authentication infrastructure, the libraries have to contain abstractions for principals and their credentials as described below. Since all authentication is based on cryptographic algorithms, an abstraction for these algorithms must also be made available. In combination with a credential they can be used to make arbitrary objects *private* or *safe*.

Component `Principal` The principal abstraction contains two type definitions for principals according the two different roles in which a principal can be used (`Principal.T` and `Identity.T`). This distinction on the type level helps to increase the correctness of the application programs.

```
interface Principal export
  T <: Ok
  Identity <: Ok
  get(name :String) :T
  proveIdentity(p :T  secret :String) :Identity
  ...
end;
```

- A principal is simply an identifier of an active entity. Such an identifier is described by the type `Principal.T`. Values of this type, called *simple principals* in the following, support the management of principals but do not include the possibility of acting on behalf of a principal.
- If a user wants to act as the named principal he/she has to get the *identity* of this principal (a value of type `Principal.Identity`). For proving that he/she really is the intended principal a *secret* has to be presented which normally is a password. Internally this is used for authentication against the authentication server or unlocking stored encryption keys depending on the underlying authentication system. Only the owner of an identity can acquire credentials.

The most important functionality of the interface is to lookup a principal (function `get`) by giving the name of the desired principal and to obtain the identity of a principal by presenting the correct secret (function `proveIdentity`).

Components `Credential` and `Encryption` If a principal (in the role of a client) who has successfully received his identity has to prove this identity against a peer principal (in the role of a server) a credential is required. This contains the cryptographic information necessary for the peer to verify the claimed identity. This credential will be transported to the peer using the communication medium selected by the application.

```
interface Credential export
  T <:Ok
  new(p :principal.Identity) :T
  valid(p :principal.T  c :T) :Bool
  ...
end;
```

Credentials in the interface are described by values of type *Credential.T* which hides the specific structure of a credential used by the authentication system. The *new*-function takes a principal identity as parameter and returns the credential whereas the *valid*-function used by the server to check whether a received credential proves the claimed identity of a client returns only a simple principal. Additional functions are provided for credentials valid only for a specific server which are needed by some authentication protocols.

The encryption component contains the abstractions for encryption keys and functions for en- and decrypting. Since their use is restricted to the credential component and the safe/private components described below, they will not be discussed in detail here.

Components `Private` and `Safe` Following a successful authentication by the exchange of credentials these credentials are used to guarantee the safety or privacy of objects created by the principals. These objects can also be exchanged by a communication mechanism. The corresponding abstraction component in the Tycoon libraries uses the cryptographic information like encryption keys stored in the credentials and the corresponding cryptographic algorithms. The component `Safe` is described as an example:

```
interface Safe export
  Signed(A<:Ok) <:Ok
  Signature(A<:Ok) <:Ok
  signIt(A <:Ok c :credential.T object :A)  :Signed(A)
  signedBy(A <: Ok c :credential.T object :Signed(A)) :Bool
  contents(A <: Ok c :credential.T object :Signed(A)) :A
  ...
end;
```

The type operator `Signed` is parameterized with the type of objects to be signed and describes signed values. A signed value of type `Signed(Int)` is a pair of a value of type Int and a hidden signature. Signatures itself are described by values of type `Signature(V)` where V denotes the type of the signed value. The function `signIt` takes the credential of the signing principal as parameter and returns a signed object. The verification that a signed object was signed by a specific principal is carried out by the function `signedBy`, again using the credential of this principal (the match between a principal and a credential can be checked using the principal component); the function `contents` works similarly but returns the value and raises an exception if the value was not signed by the principal represented by the credential. Analogous functions using signatures only are also available.

Again, the use of type operators in combination with the static typing of TL enforces at a language level that arguments to functions are signed and that every access to a parameter value must be preceded by a call to a function that returns the value and checks its authenticity (and integrity). Polymorphic typing avoids a type loss during the `sign` operation.

4.2.2 Authorization

Authorization decides on the question of whether a subject is allowed to access an object or not. A subject can be a principal but authorization can be based also on other granularities. For example, in many systems access has to be granted to groups or roles.

In each of these cases, the *access granularity* is represented by a principal who is the basis of authentication. It is the task of the access control mechanism to decide whether a principal is a valid representative of its claimed access granularity.

Component `Subject` The subject abstraction of the Tycoon libraries built on top of the principal and credential abstractions supports multiple access granularities and describes subject values by a type operator `Subject.T` parameterized with the type of the accessing granularity.

```
interface Subject export
 By(B <:Ok) <:Ok
 new(B <:Ok
     equal(:B  :B) :Bool
     isPrincipal(:principal.T  :credential.T) :Bool
     representative(:B  :principal.T) :Bool) :By(B)
 T(B <:Ok) <:Ok
 Identity <:T
 get(B <:Ok  by :By(B)  baseEntity :B  p :principal.T) :T(B)
 prove(B <:Ok by :By(B) s :T(B) c :credential.T) :Identity(B)
 fromSystem() :Identity(principal.T)
 baseEntityOf(B <:Ok by :By(B) s :T(B)) :B
 principalOf(B <:Ok by :By(B) s :T(B)) :principal.T
 credentialOf(B <:Ok by :By(B) s :Identity(B)) :credential.T
 byPrincipal :By(principal.T)
  ...
end;
```

As is the case for principals (see section 4.2.1) it is necessary to distinguished between *simple subjects* and *subject identities*. Whereas a simple subject is only an identification for a subject, a subject identity proves its identity based on a claimed principal, a credential and a check that the principal belongs to the access granularity. The management of access rights can be done only on simple subjects, access granting decisions have to be based on subject identities. The twofold character of subjects is reflected by the additional type operator `Identity` which is a subtype of `T` and therefore also parameterized by the type of the accessing granularity.

Multiple access granularities are supported by the type operator `By`. A value describing the framework for a specific granularity is created with the `new`-function. This function takes three functions as arguments to describe the structure of a specific access granularity. All other functions must be parameterized with such a value; the coherence of the access granularity type of all parameters is guaranteed by the type parameter `B` expressing an inter-parameter constraint. For example, the function *prove* takes a granularity value, a simple subject and a credential restricted to the same access granularity type. It returns a subject identity, only if all necessary checks have been passed. The function `fromSystem` returns a subject identity based on the system authentication as described in more detail in section 5.

Access Control Models Based on the subject abstraction described above a wide range of access control models can be constructed. Access control lists are such an example for a discretionary access control model and will be described.

```
interface ACL export
 T(ObjectT, SubjectT <:Ok) <: Ok
 new(ObjectT, SubjectT <:Ok
     equal(:subject.T(SubjectT)
           :subject.T(SubjectT)):Bool)
              :T(ObjectT  SubjectT)

 addSubject(ObjectT, SubjectT <:Ok
            s :subject.T(SubjectT)
            acl :T(ObjectT  SubjectT)) :Ok

 deleteSubject(ObjectT, SubjectT <:Ok
               s :subject.T(SubjectT)
               acl :T(ObjectT  SubjectT)) :Ok

 grant(ObjectT, SubjectT <:Ok
       s :subject.Identity(SubjectT)
       acl :T(ObjectT  SubjectT)) :Bool
  ...
end;
```

An access control list is attached to the object to which access should be controlled. It simply contains the subjects to which access will be granted. The interface defines a type operator `T` describing access control list values. It is parameterized with the type of the controlled object as its first parameter `ObjectT`; as explained in section 4.2.2, a subject represents an access granularity that is defined by the second parameter `SubjectT`.

The interface contains functions to create access control lists for a given object and subject type and to add and delete subjects (of the correct access granularity type) from an access control list. These two functions still work on simple subjects whereas the grant function which checks whether a subject is contained in the access control list uses subject identities (requiring the use of an authentication check function of the subject component).

A major difference between the access control lists in the Tycoon libraries and traditional access control lists is the apparent lack of an access type specification (like *read* or *write*). In an environment where functions are one of the main abstractions and all activities are done by functions, the *application access type* is expressed by a function value and there is only one access type available on functions, namely to *execute* them.

The access control list above can be attached directly to the functions or objects to be controlled by the application developer. If this does not fit the application requirements, a predefined `access control list manager` can be used. This component manages pairs of objects and access control lists.

```
interface ACLManager export
 T(ObjectT, SubjectT <:Ok) <: Ok
 new(ObjectT, SubjectT <:Ok
     equal(:ObjectT  :ObjectT) :Bool
     equal(:subject.T(SubjectT)
            :subject.T(SubjectT)) :Bool)
              :T(ObjectT  SubjectT)

 addObject(ObjectT, SubjectT <:Ok
            manager :T(ObjectT  SubjectT)
   object :ObjectT) :Ok

 grant(ObjectT, SubjectT <:Ok
       manager :T(ObjectT  SubjectT)
       s :subject.Identity(SubjectT)
       object :ObjectT) :Bool
 ...
end;
```

Again the controlled objects can be of arbitrary type including function types. The necessary object-equality function can be constructed easily with the existing function-equality-test function of the Tycoon library. It should be noted that such a function can only exist in a homogeneous environment where functions are first-class values.

Other access control models can be realized in the same manner. This includes also mandatory access control models under the assumption of an adequate system environment. Activities controlled by some access control model can in turn be used by other activities which may be controlled by another model. This makes it possible to build up complex access control structures depending on the application needs.

4.3 Example: A Secure Person Database

In securing the person database of section 3 the first step is the identification of the objects to be protected following the application needs. These are the ADT-values like `peter` in

```
let peter = person.get("Peter")
```

The access types for these protected objects are defined on the application level by the ADT-functions `person.name` and `person.address`. The targets for the access control can be modeled in TL by the tuple `ObjectT`:

```
Let ObjectT = Tuple
  accessType :Fun(person.T) :String
  protected :person.T
end
```

The secured version of the `Person` database is now constructed on top of the existing `Person` module, the `Subject` component and an access control model. Since the application requires a differing authorization for `secureName`

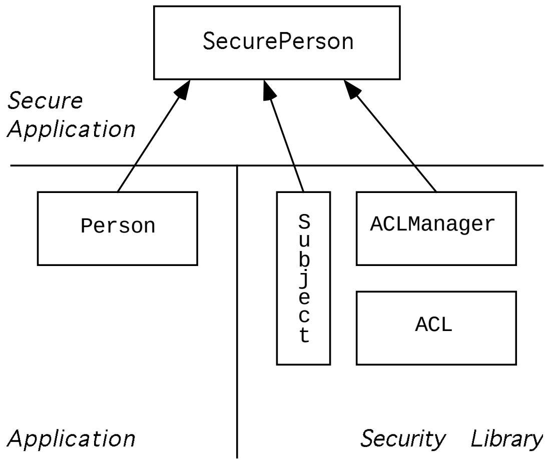

Figure 1: The component structure of `SecurePersons`

and `secureAddress` the `ACLManager` (see section 4.2.2) is chosen which uses itself ACLs. This results in a new component `SecurePerson` (see figure 1).

Access should be granted to principals determining them as access granularity. The authentication uses for simplicity the information gained during the system-authentication (see section 5). In the component `SecurePerson` an acl-manager (a value `personACLs` of type `ACLManager.T`) is created during the linking phase using the functions `objectEqual` and `principalAsSubjectEqual`:

```
let objectEqual(o1, o2 :ObjectT) :Bool = ...
let principalAsSubjectEqual = ...
let personACLs = aclManager.new(objectEqual
                                principalAsSubjectEqual)
```

For all security relevant functions `f` in `Person` the component `SecurePersons` contains a secure variant `secureF`, which combines security management with the base functionality of `Person`. The `secureNew`-function creates a new person using `person.new` and adds a value of type `ObjectT` denoting the protected person and the access type to the ACLs.

```
let secureNew(name :String  address :String) :person.T =
  begin
    let p = person.new(name  address)
    aclManager.addObject(personACLs
                         tuple person.name p end)
    aclManager.addObject(personACLs
                         tuple person.address p end)
    ...
end
```

The secure access functions like **secureAddress** are responsible for authentication (reusing the system-authentication in this example) and the following authorization. Only if all checks have been passed successfully is the base access function called; otherwise an error is raised.

```
let secureAddress(p :person.T) :String = begin
  let subjectId = subject.fromSystem()
  if aclManager.grant(personACLs  subjectId
                     tuple person.name p end) then
    person.address(person)
  else
    raise authorizationError
  end
end
```

It should be remarked that this example neglects details of a possible communication context between client and server (requiring additional authentication) and the administration of the access rights.

5 System Security Support

The flexibility of authentication and in particular authorization mechanisms in Tycoon is in strong contrast to existing systems equipped with a hardwired security system. All security models must be protected against misuse and this cannot be done by a purely add-on approach. Since hardwiring *all* security mechanism is far too strong a mechanism, a measure of security support from the system will suffice to gain the flexibility of the add-on approach. Security can be enforced on three levels in the Tycoon environment.

Application Level The Tycoon security libraries following the add-on approach are discussed in detail in section 4.2.

Machine Level Execution of Tycoon code is finally done by the threads of the TM. During this execution access control to store objects based on access types determined by the used machine functions can be done.

Objectstore Level In a Tycoon environment all objects are allocated (and persistently stored) in a persistent object store. This gives the possibility of access control during access to the objects in the store.

The levels on which security support must be used are determined by the level on which the sphere of control over the execution unit switches from the client to the server depending on the configuration of the Tycoon system in a distributed environment. In principle all activities done under the control of the protected server or an execution unit trusted by the server can be considered as *security enforcing*, all activities controlled by clients must be considered as unsafe.

5.1 Configuration Scenarios

Different configurations are characterized primarily by configuration parameters determining

- single- or multi-user mode (clients and protected server are running concurrently on the same object store);
- who has control over the thread or the object store;
- whether arbitrary TML-code generated by the client can be executed by the server (remote code execution).

The consequences of different selections of parameters from the protected servers view should be clarified by a few examples. In the simplest case of a single-user Tycoon system without remote code execution the system can still be used as a server by using some communication mechanisms like RPCs or sockets. All execution takes place under the control of the server and a security mechanism on the application level is sufficient. With the additional possibility of remote code execution which is carried out by the server's thread additional security mechanisms on the machine level must be used.

In a multi-user system where the threads are under the control of a trusted Tycoon kernel the security mechanism on the application level must be enhanced by security mechanisms on the machine level. Otherwise unauthorized clients can try to directly access protected objects by circumventing the security enforcing functions of the application level. In the secure version of the person database this for example means that users must be forced not to use directly the person.address-function but only the secureAddress-function. If the threads are running under the control of the clients they again cannot be forced to use the security mechanisms of the thread. In this case security mechanisms on the store level are required. This in turn only works in a secure way if the store access is done under the control of the trusted Tycoon kernel. If they are done by the clients themselves no security can be guaranteed. It is possible to encrypt the store in this situation but the clients will have enough time to crack the ciphertext off-line without being detected.

5.2 Machine Level Security

Security on the machine level should only guarantee the enforcement of the application level security. There is no need for high flexibility with regard to different access control models. In the following the case of a multi-user Tycoon system with a tycoon-kernel controlled TM is considered. Access control information for all relevant thread operations is managed by the kernel. It consists of a principal denoting the owner of the object affected by the operation and one access right determining whether only this principal or all other principals may perform this operation. By default all operations affecting objects of linked application may only be executed by their owner.

In order to use the Tycoon Environment a client must request the initiation of a TM with an initial thread by the Tycoon kernel. During this start-up an authentication (*system authentication*) needs to be performed. This authentication on the machine level is not a substitute for authentication on the application level. In combination with the access control described below it only defines a set of reachable objects for the applications. Inside this set the applications can perform their own authorization based on their own authentication. This gives the application the freedom to switch between different principals while the thread is still running under the same principal. Nevertheless the system authentication is still available as one of the possible authentications at the application level.

The activated TM and the first thread are marked with a *real principal* and an *effective principal* (similar to the user-ids of the Unix system), both are initialized with the starting principal. New threads inherit their principals from their parent thread. Since a thread can only execute functions owned by its effective principal there is to this point no possibility of creating servers. However, if a principal links an application as a server he has the opportunity to mark some functions as *secure*. As a consequence the access rights at the machine level for the machine operations executing these application functions are modified to allow access for all principals.

This marking can be done by extending the TL syntax with a corresponding keyword or through the use of higher-order functions which modify the kernel access control information. It is the responsibility of the application developer to mark only functions as secure which enforce the application level security mechanisms.

Only the functions marked as secure can be executed by all principals. The access rights of functions called by them remain unchanged, otherwise their security checks can be circumvented. This requires a change of the effective principal of the executing thread to the principal of the owner of the secure function directly after entering this function. This is followed by the security checks and eventually the function may be executed provided the security checks are positive.

5.3 Object Store Level Security

Security mechanisms on the object store level require a store designed by the client/server principle for realizing Tycoon kernel controlled store accesses. The strict enforcement of this is in contrast to the Tycoon environment's goal in supporting flexibly different stores and can only be done by a substantial extension of the TSP. There exists at present no need to investigate this aspect since Tycoon kernel controlled store accesses can be simulated by a thread, similarly under Tycoon kernel control, which checks access control information attached to the store objects at every access.

6 Securing Applications

The use of a Tycoon environment equipped with security mechanisms on the application and machine levels allows the application developer to create secure activities by using only the Tycoon security libraries. However, this forces him to deal with the correct implementation of the application semantics and the correct integration of the security model which leads to some disadvantages as follows:

1. Integration of a security model requires for the most part the securing of selected functions dedicated to the use by the intended clients. The repetition of this task contains the danger of inconsistencies and raises the possibility of programming errors.

2. Depending on the programming style of the developer it may be hard to exchange the security relevant code which is caused by a radical change in the security requirements. Also changes in the application semantics and the resulting updates of the implementation may unintentionally lead to modifications of the security relevant code. A special case is the introduction of a security model to an existing insecure application.

Of course all security mechanisms whether they are built-in or add-on can not guarantee security if they are used incorrectly. This can be avoided by automatic generation of the security relevant code which can be done in two alternative ways.

6.1 Generation by Higher-Order Functions

The capability of higher-order functions to take function values as parameters and to return function values opens up their use as *generators*. They directly fit the requirements securing a function by using a specific security model. In this context *securing* is a function which takes the function to be secured and a value representing the security model as input and returns the secured function which can be used by clients. As a side effect the securing function can call the functions necessary to set up the security mechanisms at the machine level (see section 5.2).

The dependence from the security model requires that the securing functions are part of the security model components and fixed in an interface. In the implementation of the securing functions function values of the type of the input functions have to be constructed. Inside the implementation the ariety of these function values and the types of their parameters have to be known. For this the signature of the securing function has to specify the input function type up to their ariety and only the parameter types itself can be parameterized. As consequence only securing functions for a fixed set of input function types with respect to their ariety can be realized. Of course this is no real restriction if this set is large enough.

6.2 Generation by Code-Generators

Generation using higher-order functions is completely done at the application level and allows easy securing of single functions. But this does not really reflect the process of developing secure applications which consists of creating components represented by interfaces where all functions should be secured. As a result an interface containing only secured functions should be produced. Although an interface in TL is simply a tuple type and its implementation a tuple value this cannot be realized by generator functions because the generator function implementation has to be aware of the single tuple fields for constructing the resulting tuple type. This knowledge consists of the name of the field and the type of its value. In the case of being a function type the restrictions mentioned in section 6.1 for the securing functions apply. Again this must be reflected in the signature of the generator functions thus disallowing the parameterization of them with the tuple type.

This problem can be solved by generating the source code of the implementation for the secured components (the interface remains unchanged). As input a code-generator function takes the interface to be secured and the interface of the security model to be used. From this it generates the source code for the secure interface and its implementation. The implementation primarily consists of calls to the secure function of section 6.1.

7 Concluding Remarks

A major goal of the Tycoon persistent object system is the realization of an add-on approach [MS93] to secure system construction. Security is expected to be a key requirement of tomorrow's *information highways* [YS93] and as such will become the next orthogonal dimension to be integrated into Tycoon and similar systems. This paper focuses on two aspects of security integration in Tycoon resulting from the add-on approach.

At first add-on security requires a kernel security support to be built into system environments. For this authentication and minimal access control information is attached to the threads [MS94] executing the Tycoon applications in varying distributed configurations. The task of this system security support is to enforce security mechanisms defined independently on the application level by Tycoon's security libraries. For the development of these generic libraries, the second aspect of add-on security, the full power of Tycoon's higher-order polymorphic language is available for example avoiding a type loss during enciphering operations. Customizable and exchangeable security models can now be added safely on application demands supported by Tycoon's generating facilities.

While basic security mechanisms for authentication and authorization are already available as higher-order polymorphic libraries in the current multithreaded version of the Tycoon system, future work will concentrate on the development of high-level security abstractions. These include sophisticated and flexible access control models, discretionary or mandatory, allowing the definition and enforcement of uniform environment-wide security policies.

References

[AB87] M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical report, DEC System Research Center, 1989.

[BL73] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Vol. 1, The MITRE Corporation, Bedford, Massachusetts, 1973.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[BN89] D.F.C. Brewer and J.W. Nash. The chinese wall security policy. In *Proceedings 1989 IEEE Symposium on Security and Privacy*, Oakland, California, 1989. IEEE Computer Society Press.

[Car88] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, California*, 1988.

[Car89] L. Cardelli. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, May 1989.

[DES77] Data encryption standard. Federal Information Processing Standards, no. 46, National Bureau of Standards, U.S. Department of Commerce, 1977.

[DRKJ85] D.D. Downs, J.R. Rub, C.K. Kung, and C.S. Jordan. Issues in discretionary access control. In *Proceedings 1985 IEEE Symposium on Security and Privacy*, pages 208–218, April 1985.

[FL93] W. Fumy and P. Landrock. Principles of key management. *IEEE Journal on Selected Areas in Communications*, 11(5):785–793, May 1993.

[GM94] A. Gawecki and F. Matthes. The Tycoon machine language TML: An optimizable persistent program representation. FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[Lin93] J. Linn. Practical authentication for distributed computing. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 31–40. IEEE Computer Society Press, 1993.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German.).

[Mil89] J.K. Millen. Models of multilevel computer security. *Advances in Computers*, 29:1–45, 1989.

[MS93] F. Matthes and J.W. Schmidt. System construction in the Tycoon environment: Architectures, interfaces and gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. To appear in the Proceedings of the Twentieth Conference on Very Large Databases, VLDB, 1994, Santiago, Chile, 1994.

[Mul91] S.J. Mullender. Protection. In S.J. Mullender, editor, *Distributed Systems*, chapter 7, pages 117–132. ACM Press, 1991.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[SM93] J.W. Schmidt and F. Matthes. Lean languages and models: Towards an interoperable kernel for persistent object systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering*, pages 2–16, April 1993.

[SNS88] J.G. Steiner, B.C. Neumann, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, February 1988.

[TCS85] Trusted computer system evaluation criteria. Department of Defense, DOD 5200.28-STD, 1985.

[Vin85] S.T. Vinter. Extended discretionary access controls. In *Proceedings 1988 IEEE Symposium on Security and Privacy*, pages 39–49, April 1985.

[YS93] M. Yap and D. Sng. Building public concurrent engineering frameworks on a national information infrastructure. In *Proceedings of 2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises*, West Virginia, U.S.A., April 1993.