Meta Objects for Access Control: Extending Capability-Based Security

Thomas Riechmann, Franz J. Hauck

University of Erlangen-Nürnberg
Department of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
{riechmann,hauck}@informatik.uni-erlangen.de
http://www4.informatik.uni-erlangen.de/~{riechman,hauck}/

published in: Proceedings of ACM New Security Paradigms Workshop 97, Great Langdale, UK

Abstract

Object-based programming is becoming more and more popular and is currently conquering the world of distributed programming models. In object-based systems, access control is often based on capabilities, as capability-based security is a well-known paradigm. It has been extended by means to restrict, revoke, and expire capabilities.

On the other hand, capabilities have serious drawbacks. First, in object-based systems, programming is based on the frequent exchange of object references (i.e., capabilities). Thus, it is hard to check which parts of an application are able to gain control of a certain capability. This becomes even harder if we consider distributed object-based systems like Java RMI and CORBA. Second, a capability usually cannot prevent method invocations from leaking unprotected references as return values. Transitive access control is not possible in a transparent way, which is independent of the code describing the invocation.

We present a new security paradigm based on meta objects. Meta objects can be attached to object references and control access to the corresponding objects. Meta objects offer the same functionality as capability-based security. In addition, they can be used for implicit and transitive access control of object references passed as a parameter or as a result. Such a reference can be automatically protected by the meta object by attaching itself or another meta object to the reference before passing it on.

Meta objects can implement arbitrary and user-defined security policies. They help to separate security policies from application code, and thus support reuse.

1 Introduction

The object-based programming paradigm is becoming more and more popular. Currently, it is conquering the world of distributed programming models. CORBA [OMG95] and DCOM [BrK96, Mic96] are two strong candidates for very popular models of the future. In object-based systems, access control is often based on capabilities, as capability-based security is a well-known paradigm [DeH66, Lev84]. In reference-based object systems a reference to an object is per se a capability: if you have a capability then you have full access to the object. Thus, capabilities are a very natural security paradigm for object-based programming.

Some capability-based systems allow restriction of access by attaching access rights to the capability (e.g., Amoeba [Tan86] and Mach [Ras86]). Access is denied if the client's capability does not include the necessary rights. Other systems allow capabilities to be revoked—that is, withdrawn by the issuer (e.g., CORBA [OMG95a]). An exception is raised, if a client tries to access the corresponding object after the revocation. Other systems allow capabilities to expire and thus to be valid only for a certain period of time (e.g., Kerberos V5 [KoN93], which is used in DCE [OSF92]). After expiration, clients have to get a new capability or cannot access the corresponding object any longer. Some newer security models allow user-defined restriction of a capability by attaching a script to it which implements the appropriate access security policy (Active Capabilities, [CQL+96]).

On the other hand, capabilities have serious drawbacks. First, object-based programming is based on the frequent exchange of object references (i.e., capabilities). For large applications, it is almost impossible to verify and guarantee that a part of the application cannot gain access to a certain capability. This becomes even harder if we consider distributed object-based systems like Java RMI [Sun96] or CORBA. While reviewing such an object-based system, in this case the *Spring* operating system [Sun95a][1], we found that this problem led to an almost procedural style of programming of the application parts. Object-reference passing happens rarely and there are only few references between application parts. This led to

1. We refer especially to the interaction between user level and kernel and between different nodes.

very large interfaces and objects. Security aspects seem to contradict basic principles of object-based programming and force programmers to dilute their design principles.

Second, a capability usually cannot prevent method invocations from leaking unprotected references to untrusted objects. In Hydra [WCC+74], it is possible to restrict propagation of references to other objects, but this is too restrictive because object-based programming is heavily based on the propagation of references. Thus, we do not want to stop propagation of capabilities but add access control. In capability-based systems, this can be done by explicitly restricting the access rights. The involved classes (probably even from class libraries) have to be reviewed and modified to perform security checks and return or pass restricted references. As normal object-based programming implies a frequent exchange of object references, many classes and methods have to be adapted to enable full access control. Access control becomes nonorthogonal and restricts the reuse of code. What we need is implicit and transitive access control.

Some object-based systems (e.g., Java [Fla96],[Sun95b], DSOM [BBN96] and Legion [WWK96]) try to solve these problems with domain-based policies or ACLs, which are user-definable in DSOM and Legion. While domain-based policies do not entail the described problems, they suffer from proxy problems as we will see in Section 4. ACLs also suffer from the problem of leaking unprotected references to untrusted application parts.

In this paper, we present a novel security paradigm based on security meta objects (SMOs). One or multiple meta objects can be attached to an object reference. They control access to the target object via this reference. A method of the meta object is automatically invoked by the run-time system when a method is called using this reference. The meta object's check method can decide whether access is to be granted or not. Meta objects offer the same functionality as capabilities. In addition, they can be used to provide implicit and transitive access control for object references passed as a parameter or as a result. These references can be automatically protected by the meta object by attaching itself or another meta object to each of them before passing them on.

Meta objects have the advantage that they can implement arbitrary and user-defined security policies. They help to separate security policies from application code, and thus support reuse. Above all, the application objects can be designed without considering security.

This paper is structured as follows: In Section 2, we present our model for access control by SMOs and provide the mechanisms for implementing the same functionality as with capabilities. Then, in Section 3, we describe how SMOs extend the usual behavior of capability-based security by introducing transitive access control. Section 4 will briefly compare SMOs to domain-based security systems. Finally, in Section 5, we present our conclusion and refer to future work.

2 Access Control by Meta Objects

In this section, we describe our security model for access control using security meta objects. We assume that we have an object-based programming model that does not allow any access to objects except by invoking methods using object references, as is the case, for example, with Java and its Bytecode Verifier [Sun95]. For simplicity, we do not allow access to instance variables through references, but our model could easily be extended to handle this. We also assume that object references are safe, that is, they are only generated and controlled by a trusted runtime system and cannot be faked.

As in most object-based models, we see object references as capabilities. If a client has an object reference at hand, it can access the corresponding object. If a client cannot get an object reference to an object, it is not able to access it. We extend this simple model by adding the possibility to attach one or more special objects to an object reference. These special objects are invoked for each security-relevant operation on the object reference. The special objects are not visible to the application; that is, protected and unprotected object references look the same to the application. In general, such special objects can be considered as meta objects [Mae87]; we call them *security meta objects* or SMOs for short.

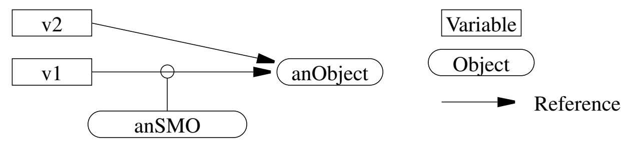

Fig. 2.1 A reference with an attached security meta object

SMOs are attached on a per-reference basis. There may be many references to an object, each with a different set of attached SMOs. Fig. 2.1 shows an object reference stored in variable `v1`, which is protected by the meta object `anSMO`. The meta object restricts the accessibility of the object `anObject` via this reference. There may be other references to the same object in the system, like the one stored in variable `v2`, which are not protected or protected by a different meta object.

If a client invokes a method via a protected reference, a special check method of the meta object is implicitly invoked. This method gains access to some meta information, such as name and parameters of the method to be invoked. The check method can decide whether it wants to grant access or not. To grant access, it returns control to the run-time system, which continues with the method invocation. If access is to be denied, an exception is raised or the invocation is terminated with an error result.

We allow anyone to attach an SMO to an object reference. Several SMOs can be bound to the same reference. In this case these meta objects are asked sequentially before access is granted. A single SMO can be used to protect multiple references. Of course, it is not possible to detach SMOs from a reference unless the security meta object removes itself.

2.1 Modeling Capabilities

In principle, capabilities introduce three additional concepts to object references: restriction of access, revocation, and expiration. Restriction of access is easily implemented by using the meta information passed to the SMO with each call. The SMO can allow access to one method and restrict access to the other, simply by distinguishing the invocations by method names. A restriction depending on the values of parameters is also possible. With appropriate support from the object model we can, for example, implement a generic SMO that denies access to methods that change the object's state (read-only policy).

The revocation of a capability can be implemented by attaching an SMO that generally allows access, but disallows access as soon as a user-defined revocation method was invoked at the SMO. A user only has to keep a reference to the SMO to revoke the capability.

Expiration is implemented in a similar way. The SMO checks, at each request, the current time and date against the expiration time. If the capability has expired the SMO denies access.

All these concepts can be implemented by a special SMO attached to a single reference, or by three different SMOs that are all attached to the same reference.

2.2 Example

Let us consider a simple example. We have a list and we want to implement an expiring reference to that list. To achieve this we just have to connect an SMO to the reference, which implements this policy. In Listing 2.1, we present a code example written in a Java-

```
List l = ..... ;                // a list reference
SecurityMeta s =                // create SMO
   new MetaExpire(new Date(12,12,1998));
s.attachTo ( l );               // connect SMO to
                                // object reference
...                     // now the reference 'l' is protected.
l.Get (...);            // the call to the list is checked

// The implementation of the MetaExpire class:
class MetaExpire extends SecuritySecurityMeta {
   final Date d;              // instance variable
                              // (expiration date)

   MetaExpire(Date d) {       // Constructor
      this.d = d;             // store date in "d"
   }

   void incomingCall
         ( Object o,          // this method checks
           Method m,          // calls via protected ref.
           ParameterList p) {
           // Check if the reference is expired:
      if (d.isBefore(getCurrentDate()))
      {               // throw exception, if ref. expired.
         throw (new SecException(...));
}  }  }
```

Listing 2.1 An SMO for expiring references.

like syntax. The first part shows the creation and the attachment of the SMO; the second part presents the class of the SMO. The method `incomingCall` is invoked on each method call done via a protected reference. It raises an exception if the reference has expired. Otherwise, access is granted.

The protected reference `l` can then be passed to untrusted parts of the application. In fact, assignments and parameter passing duplicate a reference including all attachments. Fig. 2.2 shows the graphical representation of the example. The reference stored in `l` has been propagated to an untrusted application part, which has stored it in variable `v`. All these references will expire after the expiration date. The trusted part may keep an additional reference, which allows unlimited access.

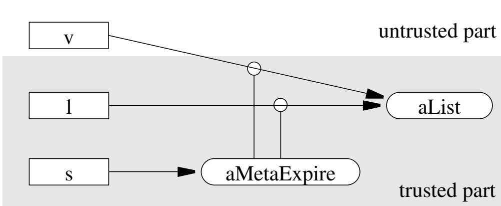

Fig. 2.2 The expiring reference example.

Note that the list does not need any support for security and also note that the implementation of the security meta class does not contain any special support for the list; it could be used for any class. Of course, we cannot achieve this quite as elegantly if we want to implement method-dependent security.

3 Extending Capability-Based Security

In this section, we demonstrate that SMOs are much more expressive than pure capabilities. We concentrate on access control for passing references to untrusted parts of an application, as is often the case in large and distributed applications. An example is a Java Applet that runs in a browser and gets references to local objects while it is still connected to its origin—for example by Java RMI.

In pure capability-based systems, we face the problem that references cannot be transitively protected and a problem that we call the *Trojan Horse Problem* of references. Both can be solved using suitable SMOs.

3.1 Transitivity of Access Control

In the case of our initial list example, Listing 2.1, we can protect the list with a security policy implemented in the attached SMO, but we cannot prevent the untrusted part from getting additional references to objects in the trusted part if they are passed as a result of a method call at the list object. For example, the list may have a method `Get` that allows retrieval of objects from the list. The references to those objects are, by default, not protected (unless the list implements its own security policy, which is not what we want). We need some transitive protection for those references.

```
class MetaExpire extends SecurityMeta {
   final Date d;
   MetaExpire(Date d) { this.d = d; }

   void incomingCall(...) { ... }

   void outgoingRef(Object o) {
            // object reference 'o' is to be returned
      this.attachTo(o);
                        // protect 'o' with myself
   }
}
```

Listing 3.1 Protecting all outgoing references

To achieve this transitivity of access control we have to protect all references that are returned as a result of a method invocation. With SMOs this can easily be implemented. When a reference is returned from a method which has been invoked via a protected reference, a special method of the SMO is called. In Listing 3.1, we show a reimplementation of the `MetaExpire` SMO. If a method has been called via a protected reference, the meta object's method `outgoingRef` is implicitly invoked for each reference that is to be returned by the method. In the example, the meta object attaches itself to all of these references, thus propagating its security policy to them.

With this implementation, the untrusted part cannot get unprotected references as a result of a protected method invocation. Fig. 3.1 provides a graphical representation of the system after the `Get` method of the list has been invoked and the result assigned to `v2`.

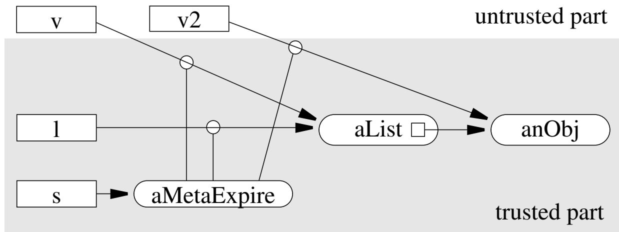

Fig. 3.1 Outgoing references are automatically protected.

Using this mechanism, we can apply generic security policies to all outgoing references. Thus, we can solve the problem of uncontrolled distribution of references. For example, we can implement a revocation meta object, similar to our expire meta object, that can be used to disable calls via all references it is bound to just by invoking a disable method on it. With such an SMO, we can not only revoke the initially protected reference but also all other references that have been collected using that initial reference.

3.2 The Trojan Horse Problem of References

There is another problem, which we cannot solve with the mechanisms introduced so far: the Trojan Horse Problem. Incoming references passed as parameters of a method call via a protected reference may act like a Trojan Horse, because the protected object may use them to invoke methods. Listing 3.2 shows a code snippet of our example list. A method `contains` can be invoked to check whether an object is in the list or not. The equality of objects is tested by a user-defined method `equals` that has to be implemented by each list element.

```
// method 'contains' of 'list':
boolean List::contains(Entry e) {
   for (Entry a=first; ....)
   {                              // iterate through the list
      if (e.equals(a))
         return true;             // entry found
   }
   return false;                  // entry not in the list
}
```

Listing 3.2 The `contains` method of the list example.

If an untrusted application part passes one of its objects (the Trojan Horse) to the `contains` method of the trusted and protected list, then this object's `equals` method will be called. It will get an object reference (the current value of `a`), which is probably an unprotected reference to one of the list elements. The Trojan Horse object can now keep this unprotected reference and propagate it in the untrusted part. In this sense, our security policy is not yet transitive and we have to introduce a new basic mechanism.

Fig. 3.2 shows a Trojan Horse that got an unprotected reference because the `aTrojH` object was passed as a parameter `e` to the `contains` method of the list.

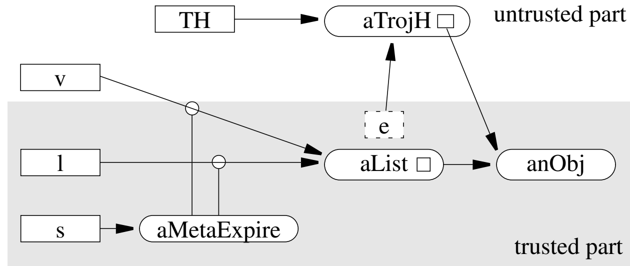

Fig. 3.2 A Trojan Horse can get an unprotected reference.

The solution is to also protect references that are passed as parameters of a protected method call. Therefore, the method `incomingRef` of the SMO is implicitly invoked for each reference passed as a parameter. In this special method, another SMO can be attached to the potential Trojan Horse. This second SMO works exactly the other way around from the first one. It protects all references passed as parameters of method invocations at the potential Trojan Horse with the first SMO (these are incoming references from the SMO's point of view) and all return values (outgoing references) by attaching itself. The first part ensures that the Trojan Horse can only get references with the original protection; the latter prevents the resulting references of a method invocation at the Trojan Horse from being Trojan Horses themselves.

As the second SMO does the same as the first one but in reverse, our basic mechanisms allow us to use the same meta object for this purpose. Instead of the usual attach method `attachTo` there is another one called `reverseAttachTo` that reverses the processing of special methods. A full implementation for an expiring meta object is shown in Listing 3.3.

```
class MetaExpire extends SecurityMeta {
   final Date d;
   MetaExpire(Date d) { this.d = d; }
   void incomingCall( Object o,
                      Method m,
                      ParameterList p) {
      if (d . isBefore(getCurrentDate()))
         throw (new SecurityException(...));
   }
   void outgoingCall( Object o, Method m,
                      ParameterList p) {}
   void incomingRef(Object o) {
      this.reverseAttachTo(o);
   }
   void outgoingRef(Object o) {
      this.attachTo(o);
}  }
```

Listing 3.3 Fully transitive implementation of `MetaExpire`.

If an SMO is attached with `reverseAttachTo` to a reference, `outgoingCall` is invoked instead of `incomingCall`; `incomingRef` and `outgoingRef` are swapped and used for resulting references and parameters, respectively.

The SMO presented in Listing 3.3 also works if the untrusted part puts a Trojan Horse into the list. The list would have a reference to an untrusted object, the Trojan Horse, and trusted clients of the list could not distinguish the new list element from trusted elements. With our SMO the list can only get a reference with the SMO reversely attached to it. Fig. 3.3 shows such an example.

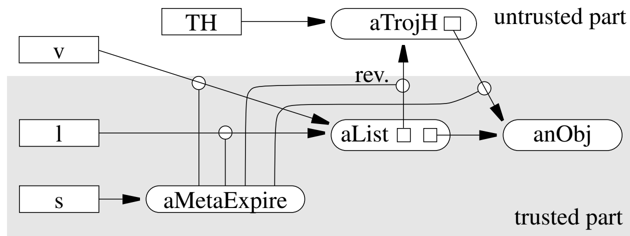

Fig. 3.3 A Trojan Horse can get only protected ref.

As shown in this example, it is possible to hand out a protected reference and ensure that all other references exchanged via the initial one will be protected. All references into the trusted part will get the initial protection, all references from the trusted part to the outside world will get a reversed protection. If all initial references between application parts are initially protected, every exchanged reference will be protected as well.

We can not only implement general transitive policies, like the transitive expire meta object, but also special transitive policies, for example a list-read-only meta object that may be attached to a list and protects all retrieved list entries (via `Get`, `Search`, etc.) with read-only meta objects. Note that such an implementation is completely transitive and also resistant to the Trojan Horse attack.

In our example, the `outgoingCall` method is not necessary but it could be used for implementing a stricter revocation semantics. On revocation, incoming and outgoing calls can be denied. Outgoing calls in this sense would be invocations done by the trusted part at nontrusted objects which have been obtained by interaction with the untrusted part.

4 Comparison to Domain-Based Security

In this section, we will compare our model to domain- and ACL-based security models as they are used in addition to capabilities in Java [Fla96] and CORBA [OMG95a]. A cross-domain call may be checked by the target domain. The target can use knowledge about the source domain and perhaps the called method to permit or refuse the call. The transitivity problem does not apply to this model, as every call from another domain is checked. In such models, we can implement policies like “domain B is not allowed to write my files.”

There are two major disadvantages of this model. First, the implementable policies are too coarse-grained. They cannot distinguish multiple clients within a domain. Thus, programmers have to fall back to individual ACLs for multiple clients, but then they again face the transitivity problem.

Second, these models suffer from what we call the proxy problem. Let us consider an example: Domain B has an object reference to a file object in domain A. Domain A implements the general policy that domain B may call only the `read` method. Domain B might know an object in domain C that is permitted to call the `write` method. It can try to use domain C as a proxy, for example by passing the file reference to an object in domain C saying: “write your contents to the file object I gave you.” As C is permitted to write, B would succeed. Thus, domain B would be able to circumvent the policy of domain A. Our model does not suffer from the proxy problem. As we have pure capabilities, no domain is able to circumvent the restriction—not even domain A itself.

5 Conclusion and Future Work

We presented a new paradigm for access control in object-based systems using security meta objects (SMOs). We showed that SMOs can implement arbitrary and user-defined security policies. SMOs can implement sophisticated capability-based security such as access restriction, revocation, and expiration. In addition to these traditional features, SMOs are able to implement transitive security policies. References that are exchanged by method invocations via protected references can be automatically and implicitly protected. Programmers can stick to a pure object-based style of programming and do not need to review all their classes and methods for security holes.

Our model separates security policies from application code. Security is configured at the only place where we know “where” and “why”: when we initially exchange object references. Our approach supports reuse of code, as in many cases SMOs can be programmed totally independently of the objects they protect and vice versa.

We are currently implementing a proof–of–concept prototype in the Java context. We are using the *MetaJava* system [KlG96], which already allows us to bind meta objects to object references. It is also possible to receive so-called events from the run-time system—for example, when a method is called via the reference. The *MetaJava* system has to be extended by a (meta) class library that extracts parameters and results, and implements the basic mechanisms described in this paper.

In the future, we will also have to deal with the remaining problems that arise from automatically protecting exchanged references. The set of SMOs that is attached to a reference may grow if the reference is passed back and forth between two application parts. We are thinking about automatic extraction of redundant meta objects. We are also thinking of allowing the owner of a meta object to detach it from a reference that he got back from untrusted parts, so that the owner may obtain unlimited access to his own object.

Although we have shown that there are many problems that can be solved with SMOs, there are problems where access control lists (ACLs) are needed. We intend to expand our model by adding principal information for the implementation of ACLs with meta objects.

6 References

BBN96 Benantar, M.; Blakley, B.; Nadalin, A.: Approach to object security in Distributed SOM, *IBM Systems Journal*, Vol. 35 No. 2, 1996, New York

BrK96 Brown, N.; Kindel, C.: *Distributed Component Object Model Protocol* — DCOM/1.0, Internet-Draft, November, 1996

CQL+96 Campell, R.; Qian, T.; Liao, W.; Liu, Z.: Active Capability: An unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation. White Paper. University of Illinois: 1996

DeH66 Dennis, J.B.; Van Horn, E.C.: “Programming Semantics for Multiprogrammed Computations”, *Comm. of the ACM*, March 1966

Fla96 Flanagan, D.: *Java in a Nutshell*, O’Reilly & Associates, 1st edition, Feb 1996

KlG96 Kleinöder, J.; Golm, M.: “MetaJava: An Efficient Run-Time Meta Architecture for Java”, *IWOOOS ‘96 workshop*, Seattle, 1996

KoN93 Kohl, J., Neuman, C.: *The Kerberos Network Authentication Service (V5)*, IETF Network Working Group, Request for Comments 1510, September 1993

Lev84 Levy, H.: *Capability-Based Computer Systems*, Bedford, Mass.: Digital Press, 1984

Mae87 Maes, P.: *Computational Reflection*, Ph.D. Thesis, Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987

MGH+94 Mitchell, J. ; Gibbons, J.; Hamilton, G. et.al.: An Overview of the Spring System. *Proc. of the Compcon Spring 1994 (San Francisco)*, Los Alamitos: IEEE, 1994

Mic96 Microsoft: *Windows NT Server*, DCOM Technical Overview, White Paper, 1996

OMG95 OMG: *The Common Object Request Broker*: Architecture and Specification, Revision 2.0, July 1995

OMG95a OMG: *CORBA Security*, OMG Document Number 95-12-1, 1995

OSF92 OSF: *Security in a Distributed Computing Environment*, Open Software Foundation, White Paper, 1992

Ras86 Rashid, R.: “Threads of a New System”. *UNIX Review*, 1986

Sun95 Sun Microsystems Comp. Corp.: *HotJava: The Security Story*, White Paper, 1995

Sun95a Sun Microsystems Comp. Corp.: S*pring Research Distribution 1.1 Source*, Source CD, 1995

Sun95b Sun Microsystems Comp. Corp.: The Java Language Environment, White Paper, 1995

Sun96 Sun Microsystems Comp. Corp.: *Java Remote Method Invocation Specification*, Revision 12, JDK 1.1 Beta Draft, 1996

Tan86 Tanenbaum, A. S.; Mullender, S. J.; van Renesse, R.: “Using sparse capabilities in a distributed operating system.” *Proc. of the 6th Int. Conf. on Distr. Comp. Sys.*, pp. 558-563, Amsterdam, 1986

WCC+74 Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F.: “HYDRA: The Kernel of a Multiprocessor Operating System”. *Communications of the ACM*, 1974

WWK96 Wang, C.; Wulf, W.; Kienzle, D.: A New Model of Security for Distributed Systems, In: *Proceedings of the 1996 ACM New Security Paradigms Workshop*, 1996