Global Information Assurance Certification Paper

Designing Capability-Based Applications For The Web

by

Timothy Wood

submitted for satisfaction of

SANS Security Essentials GSEC Practical Assignment v1.3

1. Introduction

This paper is submitted in partial satisfaction of the SANS Institute's GIAC Security Essentials Certification requirements. It is also intended as a resource for information technology personnel to incorporate Capability-based design, and the security assurances it can bring, into their development efforts.

2. Abstract

The Capability security model confers several important assurances on the design of a computing system. In particular, the Capability model guarantees that no entity in the system has any access rights except those implied by the object references, or *Capabilities*, that the entity holds [1]. The Capability model is elegantly simple. In place of reliance on authentication protocols and access-control lists to produce access decisions, the Capability model provides that one's access rights are manifest and in-hand at all times. It is never necessary to consult an authority to obtain an access decision. One obtains access to an object straightforwardly by invoking one's Capability (if one possesses it) to the object. One may of course create one's own object, which yields the creator a Capability to that object. The creator may then hand to other entities Capabilities to the object. Capabilities used in appropriate combinations allow work to be done efficiently with large numbers of objects.

In the Web computing world, many technologies at various levels of abstraction, based on different security models, cooperate to implement applications and services. The challenge of designing a Capability-based application in this environment is considerably greater. But the potential rewards to Capability design: very high security assurance; fine-grained access control; high performance; flexible administration; and extensibility make the challenge worthwhile.

3. Capabilities in Theory

3.1. Confinement

A major consequence of the Capability model is that an entity cannot even *express* an attempt to access an unauthorized object. An entity either possesses a Capability, which contains within it particular authorizations, or the entity does not, hence it cannot even

express the access attempt. Thus Capability security supports *confinement* [2], [3], a key property of high-security systems. Confinement guarantees that an invoked program cannot access any objects except those objects to which the invoker grants the program access. In particular, the program cannot utilize any channel that would allow exporting, or leaking, data from the invoker's objects. Complete confinement is difficult to achieve in systems not architected for it. Many of the prevalent browser and email-client vulnerabilities stem from untrusted software (embedded in Web pages, contained in attachments, etc.) being allowed to execute in an unconfined context [10].

3.2. The Confused Deputy

Capabilities also solve the Confused Deputy problem [6]. The Confused Deputy problem is a least-privilege problem. It arises when an entity sees ambiguity about which of its access rights to use, those of the entity's invoker, or those pre-allocated to it to perform its function. This ambiguity can lead to damaging misuses of authority. Non-Capability-based systems typically authenticate a user, then make the user's access rights currently active in the session available to all programs (processing entities) the user initiates. In the example in [6], the user may specify a debug-output file to a compiler, which also runs with its own permission to write temporary files in a system area. If the user specifies the wrong debug-output filename (or a rogue version of the compiler was installed earlier surreptitiously), the compiler could damage critical files in the system area.

In contrast, Capability systems prevent the confusion because the compiler will hold a Capability that implements writing a temporary file (name irrelevant, but unique) to the system area, and be given another Capability by the user for writing debugging output. The compiler's temporary file Capability can even limit the maximum file size, to forestall a denial-of-service attempt. Since the user has no Capability for the system area, the Capabilities are independent, the compiler uses each for a specific function, so the compiler cannot use an authority for the wrong purpose.

Systems which rely on *ambient* authority, that is authority determined by group membership, user id or other environmental information, are vulnerable to Confused Deputy problems. Access control lists (ACLs) are generically vulnerable to Confused Deputy because no matter how detailed the grant (or deny) information in the list, the available authority depends both on properties of the subject and on properties of the entity acting on the subject's behalf. Confusion about which authority to apply can arise unpredictably from the dual dependency.

3.3. Capability Mechanics

What does a Capability "look like"? Intuitively, it's an *encapsulated object reference*. Like any object reference, the holder can use it to invoke methods on the referent object. Unlike an ordinary reference, though, the reference may only activate a subset of the methods defined on the object. Thus, varying levels of authority on an object can be expressed by creating different Capabilities to the object with varying subsets of methods active. An analogy is the familiar GUI dialog box with some of its controls grayed out.

How does an entity obtain Capabilities? Four ways are explained in [11]:

1. At birth;
2. To each object the entity creates;
3. Passed to the entity from another entity (as depicted in the "Granovetter diagram");
4. In the initial conditions—ambient (or replicated to every primordial object). E.g., everyone with working lungs has a Capability to the oxygen in the air.

3.4. Requirements and Non-Requirements on Capabilities

3.4.1. Requirements: Non-Bypassable, Non-Forgeable, Complete, Transferable

Since invocation of a Capability is the sole security primitive, systems that implement Capabilities must protect the integrity of Capabilities.

Capabilities must be non-bypassable: It must be infeasible to obtain a valid Capability except by invocation of another Capability whose contract provides the requested Capability. Capabilities must be non-forgeable: It must be infeasible for an entity to fabricate a Capability which the entity does not already hold. Capabilities must be complete: It must be infeasible to message any object except through a valid Capability, and that Capability defines exactly the messages that may be sent. The Capability itself is the authorization to access the referenced object.

An entity is free to share or transfer its Capabilities with another entity with which it has a Capability to communicate. The model places no transfer restrictions on the holders of Capabilities. At first, this appears to pose a significant risk of misappropriation of authority, or misattribution of actions by a user. However, the risk posed is no greater than with ACL-based systems, and in practice will be less.

In ACL-based systems, a user's identity is checked against the access list associated with an object to find a list item which specifically grants (or denies) access to that user. This is the basic security check for all access attempts. If the user gives his authentication tokens to another user, or the tokens are stolen, the receiving user can immediately act with the full authority of the first user. The principle of Least Privilege is not maintained because a user can either transfer no authority (by not sharing authentication tokens) or too much authority (by sharing them) [17]. Sharing authority at a fine-grained level is difficult with ACLs because they usually require an additional entry by the object owner to grant authority or to allow a user to grant authority to someone else. In contrast, a Capability gives a specific authority on the object it references, but transitively allows transfer of that authority to others. A user can even give another user a subset of the authority in a Capability by creating a new object that uses the original Capability in limited ways, then handing a Capability to the new object to the other user. In 3.4.3 below, we discuss how to maintain accountability for the results of such transfers.

3.4.2. Non-Requirement: Revocability

Revocability, or the ability of an entity to invalidate outstanding Capabilities to an object the entity owns, is not a requirement of the Capability model. However, most practical security policies will require the ability to revoke at least some Capabilities. An object owner will not in general be able to recall outstanding Capabilities to its objects, because it would be difficult to limit the authority of the message to discard a given Capability, and the entity receiving the message could just ignore it. Instead, the owner must destroy the object whose Capabilities the owner wishes to revoke. If the owner wishes to revoke from some entities but not others, the owner must organize the shared objects and hand out Capabilities to them according to a pattern that allows revocation to the desired degree of generality. Specific examples are discussed in Section 5, below.

3.4.3. Non-Requirement: Accountability

Capabilities, as the combination of object reference with authorization, make either object owners or the Capability infrastructure responsible for maintaining accountability, usually by use of a logging service API.

If the object owner is responsible for logging, then the owner must know the stated identity of the subject to which it gives a Capability to an object, and arrange to log accesses to the object. But Capabilities are transferable between entities, so use of a Capability by other entities would be incorrectly associated with the original holder (though the delegation of the authority is relevant). This misleading association presents no greater risk than ACL-based systems, however. Recall from 3.4.1 that a subject can transfer its authentication tokens to another subject, just as easily as the subject can transfer its Capabilities to another subject. Either way, the second subject's actions can be misattributed to the first. The risk with Capabilities is actually less, because Capabilities only give access rights to their particular referenced objects. Whereas transferring a subject's authentication tokens allows someone else to act with all the authority of that subject.

If the infrastructure is responsible for logging, then any access to any object may potentially cause logging. Some form of subject authentication, security sessions and services to manage them would be necessary to associate a Capability use with a subject. These dissimilar architectural approaches illustrate the flexibility of the Capability model toward accountability mechanisms.

4. Capabilities in Current Practice

Capability operating systems have sufficient control over the computing domain that the above axioms can be implemented using established system programming techniques. Capability-based machines (proposed [4], and implemented [5]) implement Capability addressing in the CPU architecture and memory-management unit. It is a simpler problem to implement a secure operating system on a single Capability machine, or a secure application on a Capability OS, because the machine or OS have the ability to

confine the application.

4.1. Operating Systems

The KeyKOS Operating System [7] was a commercial product for the IBM System 370 line of mainframe computers. Its feature set focuses on:

- Minimal kernel footprint and overhead,
- Very high performance of primitive operations (e.g., calling through a Capability),
- An integrated persistence facility, where a subset of, or the entire low-level state of the system could be saved to mass storage for later restart.

KeyKOS is not a full-service operating system; it does not contain a file system, network stack, or user interfaces. Rather, KeyKOS is a hardware multiplexor, capable of hosting virtual operating system instances, or OS components such as file systems, within *domains*. The domain in KeyKOS is the basic client of the KeyKOS kernel. All non-kernel code runs within one or more domains. The KeyKOS kernel's responsibilities are to perform Capability calls and returns and manage the persistent store. The primary strength of KeyKOS is to support mutual confinement of domains with high assurance.

The EROS Operating System ([8], [9]) brings the KeyKOS architecture to the PC in an open-source implementation. This system can be used as a virtual server platform, where multiple, isolated instances of various commercial operating systems can run within a collection of domains. The Capability architecture allows communication channels among the operating systems to be specified exactly and arbitrarily limited. The persistence feature allows restart of the system and resumption of processing with very little latency.

4.2. Programming Languages and Libraries

4.2.1. Netscape Capabilities API

The Netscape Capabilities API, [12], addresses a prevalent security problem: protecting a client computer from damage caused by executing software downloaded from the World Wide Web. The API allows a Java applet writer to enumerate the privileges the applet may wish to have on the client machine, as the applet executes within the Netscape browser. At runtime, the applet may prompt the user to authorize the applet to use a particular privilege. This model is in contrast to the earlier, rigid, "sandbox" rule for applets, which imposed a restrictive blanket policy on all access by the applet to the client system. It also avoids the all-or-nothing proposition of ActiveX, where downloaded code has privileges up to the full extent of the user's authorizations on the client system if the user expresses trust in the stated creator of the downloaded code. Besides not supporting the principle of least-privilege, the ActiveX authentication scheme is vulnerable to a particular human error with improperly-issued certificates [13].

Paradoxically, the Netscape Capabilities API does not quite offer Capability security. In particular, it does not generate encapsulated object references to the objects that the applet writer wishes to get authorization to use. Rather, the programmer uses the `netscape.security` classes to enable ambient privileges within a critical section of code (a method subgraph or just a portion of a method). The presence of privilege is reflected in the stack frames of the active method calls, so the privilege is restricted to the thread that obtains it. The Java 2 Privileged Block API [16], works in a similar manner, except the higher privilege takes effect in the `run()` method of a separate object created for the purpose of executing the privileged operation.

4.2.2. The E Programming Language

The E programming language ([14], [15]), is an interpreted, Capability-based programming language with some important influences from Smalltalk, such as runtime-only type-checking. It also features the ELib subcomponent, written in Java, which implements the distributed communication features of E. E itself is currently implemented on top of the Java Virtual Machine, that is, the E compiler and interpreter generate JVM opcodes. An E programmer can make use of the standard Java Runtime Environment classes via an import mechanism.

E promulgates a model of distributed computing that is:

- Capability-based, and
- Deadlock-free.

E (via the ELib library) preserves the requirements of Capabilities discussed earlier (non-bypassability, non-forgeability, completeness and transferability) across distributed components by use of encryption and unguessable identifiers, also called *Swiss numbers*. Such numbers may be generated by a pseudo-random number generator periodically reseeded from a source of highly random bits.

Recall that a Capability is an encapsulated object reference. The referent of the Capability may be a local or remote object. The system must guarantee that messages to the object only propagate via a Capability. The encryption protects the authentication between two parties that need to communicate across the distributed link, and ensures that the communication between the parties is safe from eavesdropping and tampering. The unguessable identifiers make it infeasible for either party to successfully fabricate a Capability to some other object on the remote side in lieu of receiving it via a message.

E (or a Java program written to the ELib API) prevents distributed deadlock by replacing synchronous calls across components on the network with *event-loop concurrency* executing in a *Promise-based* architecture. Event-loop concurrency replaces the multiple threads that may be activated to handle concurrent service requests with a single thread, called a *vat*, that services an event queue, reminiscent of GUI programming models. The primary rule of event-loop programming is that the service thread must not block, except in the event mechanism when awaiting a new event. In traditional synchronous programming, the service thread could make a call to a remote component that would block the thread. Since there is only one service thread, this is unacceptable. Instead,

calls across components do not block, but immediately return placeholder objects called *Promises*. A Promise represents a commitment by its maker (the called component) to either provide the actual object requested (a *fulfilled* Promise) or provide indication that the call has failed (a *broken* Promise). The fulfillment of a Promise by a called component is asynchronous; the fulfillment (or breaking) of the Promise is simply queued to the caller's event queue via the communication layer. If the caller is ready to service the call immediately, it simply returns a fulfilled (or broken) Promise to the caller. The Promise acts as a wrapper around the requested object or object reference (if fulfilled) or error indication (if broken). A Promise can be requested with a timeout, so that the Promise is automatically broken if there is no fulfillment or break received in the timeout period.

Figure 1 below depicts an interaction diagram for a hypothetical, simple securities trading system. All distributed components of the system are designed as ELib-managed event loops. The Client component wishes to fetch a stock quote and pass it to a rule-based Portfolio Service for possible trading action. The "active" periods for each component each indicate one pass of the internal event loop. Event loop passes are independent; any number (limited by system resources) may take place in a time period.

The Client requests a quote from the Quote Service. The ELib call to make the request immediately returns a Promise for the result and forwards the request with the ID of the issued Promise to the Quote Service. The Client, in the same event loop pass, then requests a trading action from the Portfolio Service, passing `quotePromise` as an argument to the call. The Client receives `tradeResultPromise`, then returns from the event loop pass. Meantime, the Quote Service is obtaining the quote. This is likely an asynchronous action, but not necessarily an ELib exchange. When the Quote Service has a `Quote`, it calls ELib to fulfill the `QuotePromise` that ELib within the Client has associated with the request. At this point, the Client can fulfill `quotePromise` held by the Portfolio Service, and allow that service to make its trading decision based on the `Quote`. Finally, the Portfolio Service fulfills the Client's `TradeResultPromise` issued earlier and the Client may now use the `TradeResult`.

Note that the Client runs concurrently with the other two services when setting up the request. Because the Promise architecture allows a pending object (the unfulfilled Promise) to be passed to other calls (which may in turn yield other Promises), the system as a whole can exhibit a lot of concurrency with a constant number of threads.

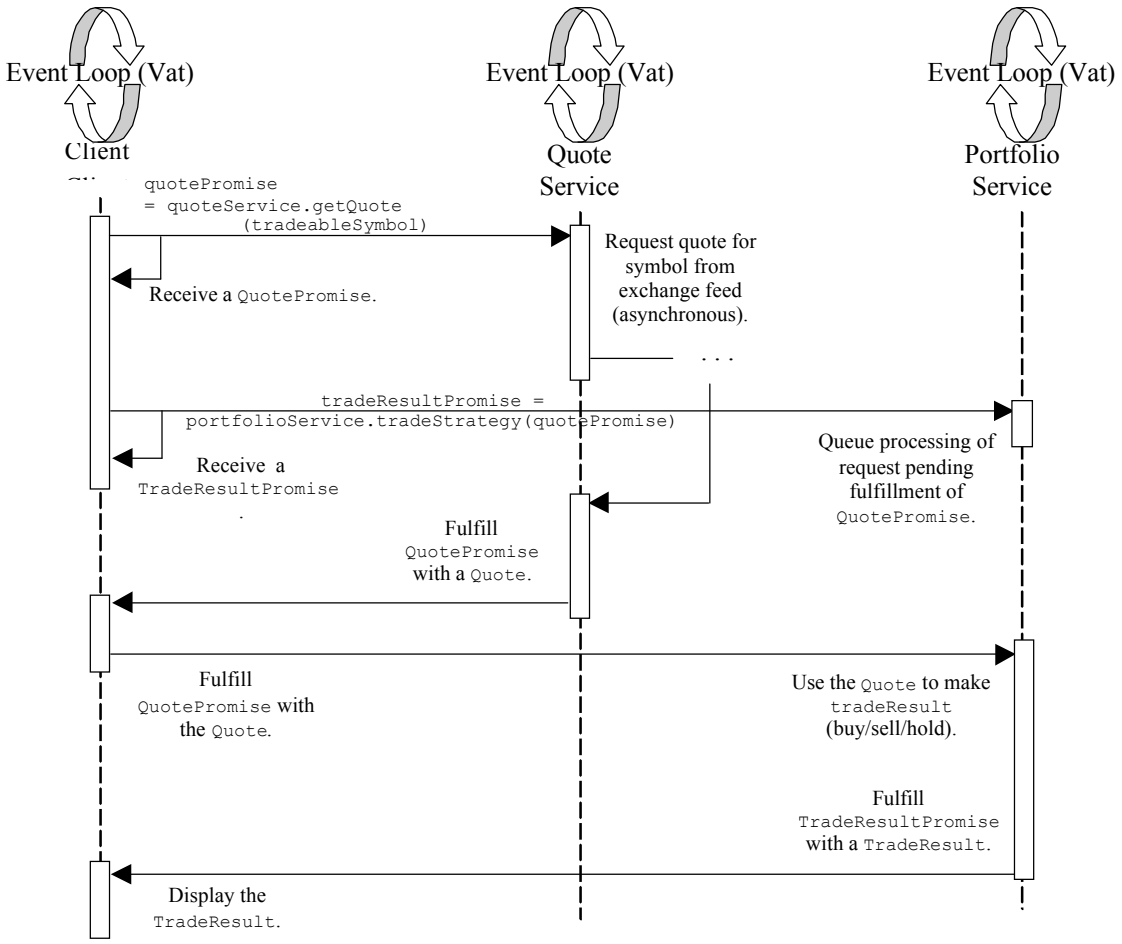

Figure 1. Interaction diagram for simple trading scenario

The major effect of event-loop concurrency and Promises on system design is to drastically simplify the system synchronization protocol, a very valuable effect as synchronization problems are notorious for being among the most difficult to diagnose. These design features reduce the static and dynamic frequency of thread blockage. The lower frequencies lead to higher thread duty cycles, fewer context switches, and lower resource consumption than designs that use a worker-thread pool approach.

5. Techniques for Application Design Using Capabilities

Extending the Capability architecture to distributed applications and services on today's major platforms is a harder class of problem, because the basic properties of Capabilities must be preserved across machine, operating system, network, middleware and application technologies not designed with Capabilities in mind. The remainder of this paper focuses on this problem, some existing technologies that address aspects of it, and offers some further approaches to solving it.

5.1. Facets

A logical service on the network may define a complex, feature-rich interface. Security

policy may require that subsets of the interface are made available to subjects according to the subjects' access rights. *Facets* provide a convenient pattern for supporting this policy in the design of the service [18]. A subject is given a Capability to a facet, or interface subset, of the service upon request if the subject is authorized to use the facet. The Capability references the facet object, which implements some subset of all the messages defined for the service. Clearly, one could create a facet for every possible combination of messages taken from the full interface. However, the pattern is most economical when the full service interface is divided into a manageable number of major facets which are each associated with certain access rights. For example, a simple policy would divide the interface to a service into general query, privileged query, update, and administrative groups of methods. The service implementation has wide latitude in choosing design patterns for its constituent objects. It is only necessary to ensure that a subject with a Capability to one facet cannot message another facet, except through another Capability that references that facet.

The below diagram shows a basic multi-faceted network service in use by several clients. Each client already has authenticated to some service that issues an authentication token, serving as proof (to some degree) of identity. Each client has a Capability to the Director/Authorizer, which hands out Capabilities based on authorizations granted to a holder of an authentication token. Some Capabilities are public, meaning that no authentication token is needed to obtain them.

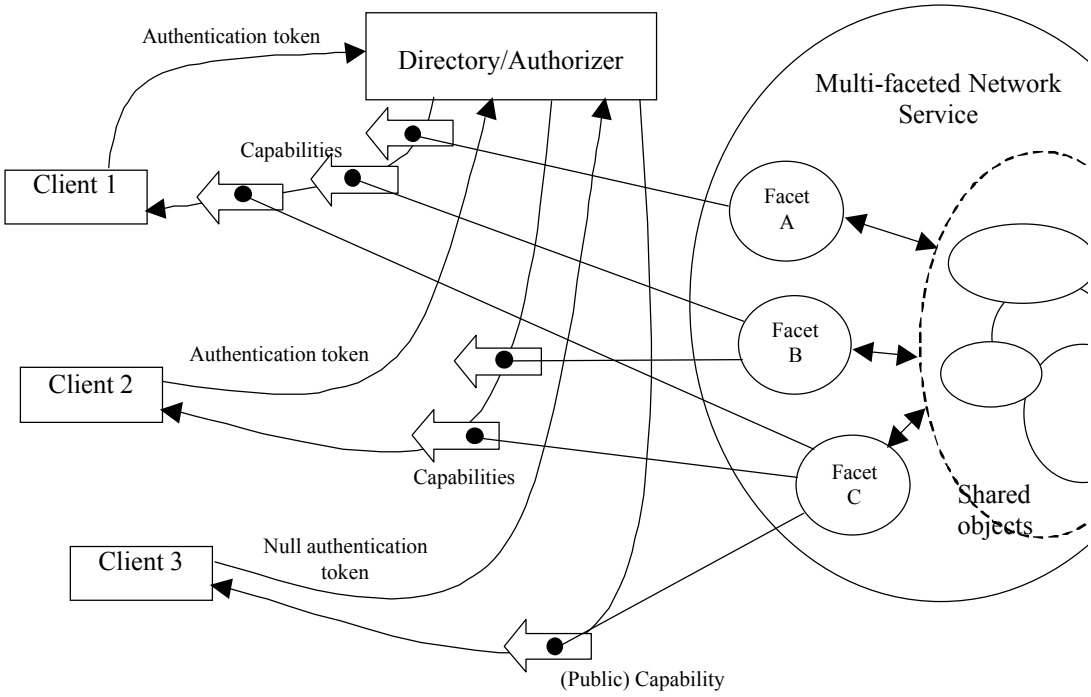

Figure 2 - Use of a multi-faceted network service

5.2. Façades

If facets decompose a large interface into smaller interfaces in order to authorize use of each interface separately, then a *façade* has the converse purpose: To aggregate facets of a service's interface into a larger interface, possibly with more abstract methods in place of the specific methods of the facets. This abstraction can make authorization and invocation more convenient [19]. However, Capabilities make the pattern stronger than described in the reference. A Capability to a façade does not confer Capabilities to the interface facets that make up the façade, though those facets can still be authorized separately.

5.3. Protecting Capabilities in Existing Application Frameworks

There is a great variation in security policies in today's application frameworks. Support ranges from native Capability security in E and ELib to responsibility being left entirely to the programmer (CORBA, RMI). The challenge of implementing Capability-secure systems lies in preserving the requirements of the Capability model across the collection of technologies used to build the system. This section will demonstrate that it is easier to use the framework security features only to meet the requirements of Capabilities rather than to compose the framework policies together with Capabilities and try to make sense of the resulting hybrid model.

The examples depicted below focus on only a client and a server tier, for simplicity. For multi-tier applications, intermediate tiers function as servers of some classes of objects and clients of others. The client and server patterns discussed can be reapplied between those tiers.

5.3.1. CORBA

The CORBA remote object framework implements inter-object distributed invocation and object reference lookup services [20]. A C++ or Java programmer specifies an interface for network-visible objects in Interface Definition Language (IDL), and generates "stub" and "skeleton" classes from the IDL with a generator program. The stub classes implement the client object reference to the remote object. The skeleton classes take care of dispatching calls received on the server side, from remote references, to the actual object implementation. The object request broker (ORB) handles binding of a client stub instance to a remote object instance. Once the binding is established, stub and skeleton code communicate directly via the Internet Inter-ORB Protocol (IIOP).

CORBA does not specify a security policy. The Borland implementation offers an SSL add-on that applies the SSL feature set (one- or two-way certificate-based authentication and encryption of traffic) to communication between clients and ORBs and stubs and skeletons. A security service that implements password and certification authentication, a Gatekeeper middle tier, and an API for server objects to create ACLs for authorization are also offered.

The CORBA security features include the familiar mechanisms of SSL, password-based authentication and ACL-based authorization. We now examine possible ways to use the CORBA security features to design a Capability-secure application, and some pitfalls that

could arise in realizing Capability security in a non-Capability system.

CORBA uses a name service that clients query to obtain references to remote objects. The name service must only know about objects that are public, i.e. ambient Capabilities. Otherwise objects responsible for handing out other Capabilities could be bypassed by asking the name server. Even with the Security Service add-on, once a client authenticates, it can obtain a reference to any remote object by name because the name service does not require authorization for references before returning them.

Rule: Only ambient Capabilities should be registered with the CORBA name service.

CORBA's IIOP sends object references over the wire in "stringified" (e.g., serialized) form. Unless the object reference strings are unguessable, e.g. by incorporating Swiss numbers, a client could construct a Capability to an unauthorized object if the client knows how to name the object in this manner. This violates the unforgeable property of Capabilities. Even if valid Capabilities to specific objects could not be feasibly constructed, clients could go on "fishing expeditions" for valid object references unless the space of object reference strings is extremely sparse and the strings are uniformly distributed in the space.

Rule: Objects that dispense Capabilities must name the referent object in a way that makes it infeasible for a client to fabricate a valid CORBA reference to the object.

The next diagram shows one possible arrangement for implementing revocable Capabilities in CORBA. The "shadow objects" are the ones to which clients actually receive references (Capabilities). (From the point of view of the Service Object, the shadow objects themselves are Capabilities.) References to shadow objects may be shared between clients, and separate shadow objects may be created to allow individual revocation. Each shadow object can be set up with particular methods enabled or disabled depending on the client's authorizations. Moreover, it is straightforward to generate the shadow object class definitions from the same IDL file used to define the actual CORBA object on the network, with a tool similar to the existing `idl2cpp` or `idl2java`. This model can be composed with the Facets approach (5.1), by creating a shadow object class for each facet. A CORBA Authorizer component would need to be written to hand out Capabilities to shadow objects to authorized clients [3].

Messages
Messages
Messages

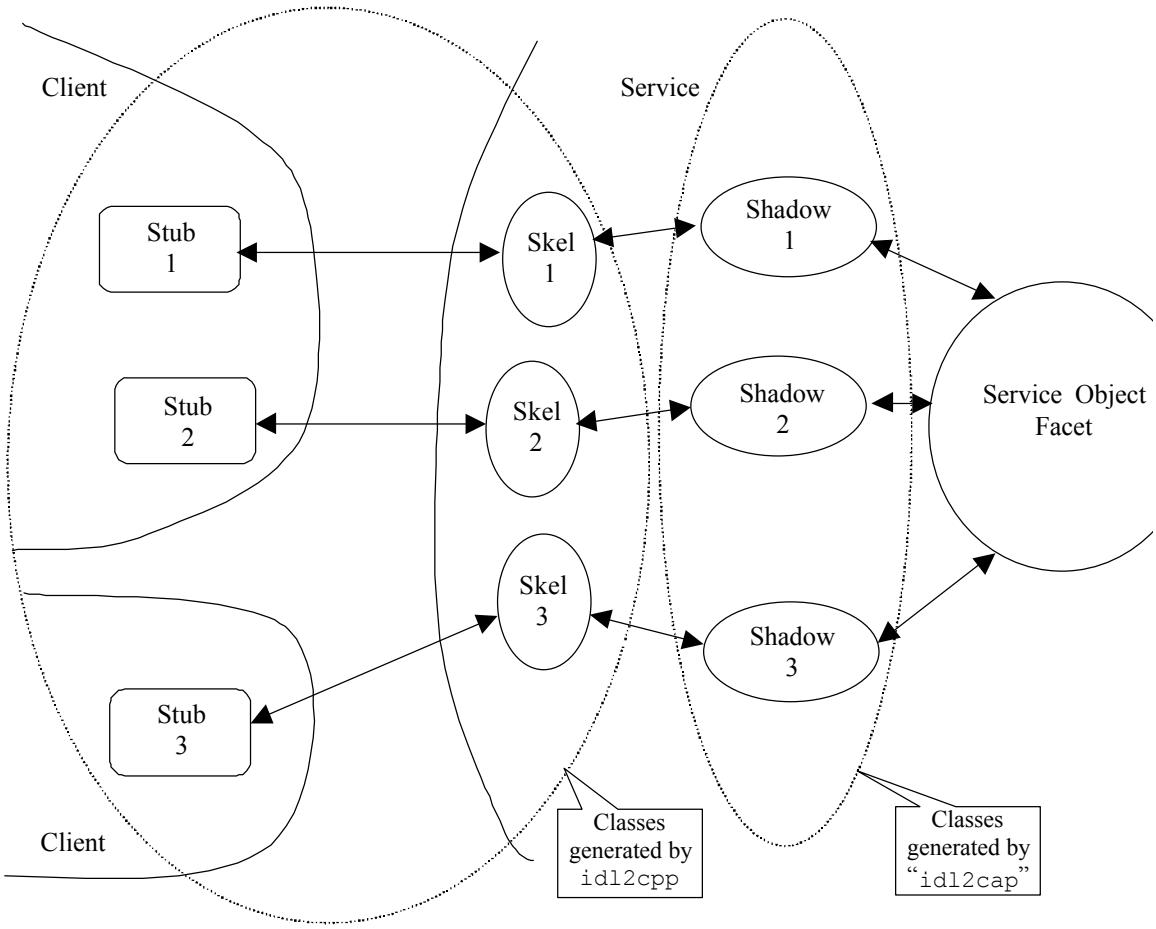

Figure 3 - Capabilities implemented with shadow objects

5.3.2. Java Remote Method Invocation (RMI)

The Java RMI facility [21] is similar in architecture to CORBA. It uses client side proxy classes ("stubs") generated by a compile-time tool, `rmic`, to message remote instances. Unlike CORBA, RMI does not use separate, server-side adapter classes (the "skeletons"). Services register with a name service, the RMI Registry, which clients use to find remote objects. RMI allows for another dimension of flexibility in that the service can use the Java class loader and serialization mechanisms to load the object instance from a method call argument, or from a file or the Internet.

The same rules for managing Capabilities in CORBA apply to RMI. The facet and shadow object patterns for a Capability-based design apply as well.

5.3.3. Comparing Frameworks

It is worth noting here that CORBA and RMI still operate on the synchronous-call model, unlike the E language and ELib library. That is, a remote object invocation blocks the thread that makes it. So even though CORBA and RMI solve important problems in designing distributed systems, they do not natively protect object references nor do they

offer a model for creating deadlock-free systems, as E and ELib do. On the other hand, E and ELib do not contain a pre-built facility for dispensing Capabilities to (authorized) clients, where as CORBA and RMI have basic, if insecure ones.

Whichever framework is used, attention must be paid to preserving Capability semantics at layers below the framework. Saving an object instance or Capability in a persistent store requires that no subject that is not authorized for the Capability be able to access that store. SSL or another means of protecting traffic and authenticating the correspondent must be used for exchanging Capabilities over the network. Clients must be protected from code on the network that would misappropriate Capabilities. Note that enforcement stops short of explicit client action to share Capabilities, just as it is impractical to prevent a person from sharing his or her password.

5.4. Capabilities on the Web

Most of the discussion of distributed object frameworks in this paper covers server-tier and middle-tier components, because most distributed object technology is deployed in those components. But the Web is an end-user medium, with the Web browser the most familiar software artifact.

5.4.1. Client Framework Components

A Web page that is part of a Capability-based application can dynamically download a software component to execute within the browser. Such component may be a Java applet, a plug-in, or a compiled ActiveX control that is coded to the framework API and uses the techniques discussed. However there are well-known tradeoffs to this design, such as trust issues with accepting downloaded code to run in the user's security context (or some narrower context); and user experience issues with latency and errors in the actual download or instantiation.

5.4.2. HTTP Cookies and Object References

Probably the bulk of Web pages delivered to browsers is coded in the HTML and Javascript scripting languages. These languages fall well short of supporting basic object-oriented properties of encapsulation, inheritance, polymorphism, etc. The challenge for a Capability application designer is to model the system so that Web pages can exchange some form of object references without an object-oriented framework.

The widely-used HTTP Cookie construct [22] specifies a format for exchanging state information between client and server. A Web page can dispense Capability "equivalents" via Cookies to the user's browser. (The Cookie string would be an "equivalent" because it would not be an object reference in the framework.) If the Cookie string contains a Swiss number that designates a particular object in the server, and the Cookie is exchanged via SSL, then the Capability requirements of 3.4.1 are met. (The session ID Cookie exchanged by many sites is one type of object designator.) The completeness requirement here intends that the Cookie strings act as Capabilities for objects covered by the security policy. "Objects" not known to the distributed framework, such as HTML blocks and Javascript variables, should contain no Capability

equivalents.

Note that some sites use URL parameters rather than Cookies to exchange state information. The problem here is that misbehaving Javascript could lift a Capability equivalent from such a parameter and POST it anywhere, whereas Cookie exchange can be restricted to a particular, SSL-authenticated server trusted not to introduce such code [3].

6. Conclusion

A distributed system is subject to myriad threats. The Capability security model is simple, consistent, flexible, and expresses a wide range of policy, which are all major countermeasures to the threats. There are production-quality tools becoming available that support end-to-end implementation.

In contrast, an application security model composed from the various security features of the implementation platforms is vulnerable to platform-specific attacks and changing behavior of the security features. An end-to-end, platform-independent Capability-based design should be considered at the outset for any substantial system to be exposed to the Web.

7.

References

1 Steigler, M., Capabilities As Security Solution, *Introduction to Capability-Based Security*, http://www.skyhunter.com/marcs/capabilityIntro/solmodel.html.

2 Lampson, B. W., A Note on the Confinement Problem, *Comm. ACM 16*, 10 (Oct. 1973), 613-615.

3 Hardy, N., Factories, http://www.cap-lore.com/CapTheory/KK/Factory.html.

4 Fabry, R., Capability-Based Addressing, *Comm. ACM 17*, 7 (July 1974), 403-412.

5 Bayko, J., CPU History, *General Technical Information*, Toronto Microelectronics, Inc., http://www.tme-inc.com/html/service/general.htm#432.

6 Hardy, N., The Confused Deputy (or why capabilities might have been invented), *Operating Systems Review* (Oct. 1988), pp. 36:38. Also http://cap-lore.com/CapTheory/ConfusedDeputy.html.

7 Hardy, N., The KeyKOS Architecture, *Operating Systems Review*, September 1985, pp. 8-25. Updated at http://www.cis.upenn.edu/~KeyKOS/OSRpaper.html.

8 Shapiro, J., Hardy, N., EROS: A Principle-Driven Operating System from the Ground Up, *IEEE Software* (Jan. 2002), p. 26-33. Also http://www.eros-os.org/papers/IEEE-Software-Jan-2002.pdf.

9 Shapiro, J., *EROS: The Extremely Reliable Operating System*, http://www.eros-os.org.

10 Microsoft Corp., Internet Explorer Critical Updates, http://www.microsoft.com/windows/ie/downloads/critical/default.asp.

11 Miller, M. et al, An Ode to the Granovetter Diagram, *Proceedings, 4th Int'l Financial Cryptography Conference* (Feb. 2000) ISBN 3-540-42700-7. Also http://www.erights.org/elib/capability/ode/overview.html.

12 Netscape Corp., Object Signing, *Introduction to the Capabilities Classes,* http://developer.netscape.com/docs/manuals/signedobj/capabilities/index.html.

13 Microsoft Corp., Erroneous Verisign-Issued Digital Certificates Pose Spoofing Hazard, http://www.microsoft.com/technet/security/bulletin/MS01-017.asp.

14 Miller, M., ERights Home Page, http://www.erights.org.

15 Stiegler, M., *E in a Walnut*, http://www.skyhunter.com/marcs/ewalnut.html.

16 Sun Microsystems, Inc., API for Privileged Blocks, http://java.sun.com/products/jdk/1.2/docs/guide/security/doprivileged.html.

17 Stiegler, M., private correspondence with the author, Mar. 2002.

18 Hardy, N., Objects and Facets, http://www.cap-lore.com/CapTheory/ObExp.html.

19 Gamma, E. et al, *Design Patterns*. Addison-Wesley, Reading, MA, 1995, p.185-193.

20 Borland Corp., *Programmer's Guide to Visibroker*, Version 4.5. Borland Corp., Scotts' Valley, CA, 2001. http://www.inprise.com/techpubs/books/vbcpp/vbcpp45/programmers-guide/vbcpp45programmers-guide.zip, http://www.inprise.com/techpubs/books/vbj/vbj45/programmers-guide/vbj45programmers-guide.zip.

21 Wollrath, A., Waldo, J., Trail: RMI, *The Java Tutorial.* Sun Microsystems, Inc., Mountain View, CA. http://java.sun.com/docs/books/tutorial/rmi/index.html.

22 Netscape Corp., Netscape Cookies, http://www.netscape.com/newsref/std/cookie_spec.html. Also http://developer.netscape.com:80/docs/manuals/js/client/jsref/cookies.htm.