Split Capabilities for Access Control

Alan H. Karp, Rajiv Gupta[1], Guillermo Rozas[2], Arindam Banerji[3]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-164 (R.1)
May 22nd , 2002*

access control, computer security, capability systems

Split capabilities represent a new way to control access to resources that has advantages over traditional approaches in both scalability and revocation of privileges. We present the basic idea of split capabilities and describe how they were used in a commercial system. We also explain why a variety of common attacks against a system using split capabilities fail.

[1] Present address: CorporateOxygen, Cupertino, California
[2] Present address: Transmeta Corporation, Santa Clara, California
[3] Present address: Chimewell, Alviso, California

Split Capabilities for Access Control

Alan H. Karp, Rajiv Gupta[*], Guillermo Rozas[†], Arindam Banerji[‡]
Hewlett-Packard Labs
Palo Alto, California

Keywords: Access control, computer security, capability systems

Abstract

Split capabilities represent a new way to control access to resources that has advantages over traditional approaches in both scalability and revocation of privileges. We present the basic idea of split capabilities and describe how they were used in a commercial system. We also explain why a variety of common attacks against a system using split capabilities fail.

1. Introduction

The fundamental problem of access control is to limit what a process can do to an object and when that process can do it.[1] For example, whether or not to honor a request to read or write a particular file is a question that any access control mechanism[2] must be able to answer. Unfortunately, the access control mechanisms we use when sharing resources over the Internet were designed in the days when networking computers was a rarity. Many of the security breakdowns occurring today come from the resulting mismatch between today's realities and the assumptions made in designing those mechanisms

Specifying the access policy is conventionally described as one of populating an *access control matrix*, which has a row for each resource and a column for each user. An element of this matrix is the set of access rights on that resource being granted to that user. Consider the simple example with four users and four files shown below. Bob can read and write his own file, while Carol can read Bob's file. Note in particular the broad set of privileges granted to the user denoted *root*. Of course, real systems have many users and many thousands of resources.

[*] Present address: CorporateOxygen, Cupertino, California
[†] Present address: Transmeta Corporation, Santa Clara, California
[‡] Present address: Chimewell, Alviso, California
[1] We often say incorrectly "*who* can do what to whom when". Not properly understanding this distinction leads to a variety of security lapses.
[2] We use the term *access control mechanism* to denote the way control of access to resources is enforced. *Security policy* is a term we use to describe who gets what access rights. We address only the former issue.

	alice	bob	carol	root
/u/alice/file	R,W			R,W
/u/bob/file		R,W	R	R,W
/u/carol/file	W		R,W	R,W
/sys/log				R,W

The access control matrix is sparsely populated, so people developed two representations, compressing horizontally to get *access control lists* (ACLs) and compressing vertically to get *capability lists* (CLs). An ACL has an entry for each resource containing a list of the access rights for each user. In our example, the ACLs would be

/u/alice/file	alice(R,W), root(R,W)
/u/bob/file	bob(R,W), carol(R), root(R,W)
/u/carol/file	alice(W), carol(R,W), root(R,W)
/sys/log	root(R,W)

When a user starts a process, that process is assigned the identity of the user, granting the process all of the user's privileges.

Each capability in a CL specifies access rights to a resource being granted to the process holding the capability. Traditionally, each capability is an unforgeable, indivisible pair containing a unique identifier for the resource and a list of access rights that are authorized by that capability [1, page3]. In our example, Alice's capability list would be

(/u/alice/file,R), (/u/alice/file,W), (/u/carol/file,W),

with similar lists for the other three users. When Alice starts a process, she can decide which of these capabilities to make available to that process.

ACLs are held by the computing infrastructure, called the Trusted Computing Base (TCB), often the operating system kernel but sometimes the entity controlling the resources. Some systems keep the capability in the TCB, in which case the CL is a list of handles to the capabilities. In other systems, the capability is held by the user's process, in which case it is cryptographically protected against tampering and forgery. CLs are held by the user's process in both cases.

There are several problems when using ACLs, especially in a networked environment [10,11]. First of all, access control is based on the identity of the user who started the process. The unfortunate effect is that every request carries the user's full privileges. That's why opening an email attachment can unleash a virus; the virus, running in the user's process, has all the privileges of that user. Imagine the damage that could be done by a virus running as root in our example. A second problem is one of scalability. If there is no entry for a user in an ACL, access must be denied. That means that the ACL must be modified every time a potential user becomes an actual user. However, the ACL is a resource critical to the security of the system, so only a limited number of people should be able to modify it. These people can become overwhelmed in a dynamic environment like the Internet.

CLs don't have these problems. A user can specify the exact set of capabilities available to a process. Thus, the "execute" capability can be withheld from the email process, and the virus program won't run. Scalability is enhanced because capabilities are easily copied (but not forged, of course), so one user can pass the capability to others who are trusted.[3]

Traditional capabilities, such as those shown in Alice's capability list above, have two problems. It is hard to revoke a capability [1, page 149] because the system has no control over the passing of capabilities from one user to another. The number of capabilities in a system is also a problem because a capability is needed for each separately controlled access right on each resource. This means that each time a user joins the system, many thousands of capabilities need to be issued; each time a user leaves, they must be revoked. Combined, these two problems present a challenge for system designers. Split capabilities have the advantages of traditional capabilities without their limitations.

Four different types of capability structures have been used. Representatives of these types, described in Section 6, are

1. Traditional capabilities [1]
2. Simple Public Key Infrastructure (SPKI) capability certificates [7]
3. E-language capabilities [2]
4. Split capabilities

This paper compares traditional capabilities with split capabilities, but many of the points made apply to SPKI capabilities, too.

Capabilities have been used to control access to hardware resources, such as memory segments; ephemeral resources, such as processor time; and software resources, such as files. This paper concentrates on the last of these.

The basic idea of split capabilities is presented in Section 2. Split capabilities were used in the commercial, open source product described in Section 3. That implementation allows us to enforce visibility rules to control naming as described in Section 4. Various attacks against the system are considered in Section 5. Other capability systems are described in Section 6, and Section 7 summarizes the key points.

[3] At first glance, this feature appears to defeat security policy. Carol trusts Alice and passes her the capability to read Bob's file. What if we don't want Alice to read Bob's file? We gain no real security by trying to prevent such sharing. Carol can always act as Alice's agent, forwarding her requests to Bob and sending Alice the results. As far as Bob is concerned, Bob is making the request. It's best not to try to enforce the unenforceable.

2. Split Capabilities

The basic idea is to divide the capability into two parts, a handle to the resource being accessed and a handle to a separate resource representing the access rights being requested. While such separation of name from authority is potentially problematic [8], in our system these two elements are brought together in the TCB of the resource.

Our example illustrates these points. Alice would hold the resource handles

/u/alice/file, /u/carol/file.

The TCB associates these handles with a resource handler, which in turn maps them to data structures relevant to the specific resources. In this example, the file system would map the resource handles to specific file identifiers, inode numbers in Unix, for example.

In addition, Alice would have handles to resources that we call *keys* (as in keys that open locks, not cryptographic keys). Operating on a resource requires handles for both the resource and a key that unlocks the requested permission. These two pieces of the split capability come together in the TCB, which, for our example, maintains a table containing the entries

Name	Type	Value	Permissions
/u/alice/file	file	939438	(821,4493:R) (821,4493:W)
/u/carol/file	file	2831AB	(821,138B:R)(821,138B,8923:W)
/u/bob/file	file	329BF5	(821,3324,5AF3:R)(821,3324:W)
/sys/log	file	A93ED	(821:R)(821:W)
alicefiles	key	4493	(821,4493:Destroy)
bobfiles	key	3324	(821,3324:Destroy)
carolfiles	key	138B	(821,138B:Destory)
rootfiles	key	821	(821:Destroy)
bobread	key	5AF3	(821,3324:Destroy)
carolwrite	key	8923	(821,138B:Destroy)

The value is type specific. The value of a file might be an inode number, while for a key the value is the identity of the lock it opens. Each key opens exactly one lock, but different keys can open the same lock.

The permissions associate locks with access rights. Any lock that gets opened by a key results in the corresponding permission being authorized. When Alice asks to write ”/u/carol/file”, the file system verifies that the write permission, “W”, was unlocked before honoring the request.

In order to enforce the access rules in our example, we need to distribute the handles properly. If we give Alice handles to the files denoted “/u/alice/file” and “/u/carol/file” and to the keys labeled “alicefiles” and “carolwrite”, she will be able to unlock read and write permissions on "/u/alice/file" and write permission on "/u/carol/file". The other users also get the appropriate resource handles.

The keys in our example have only one operation, removal from the table, which is denoted "Destroy".[4] In our example, Carol is given the handle to key "carolfiles" that grants permission to destroy itself and the key denoted "carolwrite". The user we call root can remove any key from the table.

The TCB doesn't need to understand the semantics of the request; it is only responsible for unlocking permissions. The resource handler, the file system in our example, does not need to know what process made the request, what key was used, or how that process got the key; it only needs to verify that the proper permission was unlocked. This separation of responsibility makes the system quite flexible. For example, you can introduce a new type of resource or a new access method on an existing type without modifying the TCB.

Splitting the capability lets one access rights resource represent different permissions on different resources at the discretion of the resource owner, which makes it surprisingly easy to enforce some complex security policies. For example, we can emulate Unix file permissions by giving Alice three keys for her user permissions (read, write, execute), three for each group she belongs to, and three for world permissions. Note that the number of keys Alice needs is independent of the number of files she has access to.

We can describe more complex security policies quite easily. Military style (multi-level) security [4] assigns security levels to people and resources and specifies the access rules. The **-property* says that you can read documents at the same or lower security level and write documents at the same or higher level. We enforce this property by giving someone with a Secret clearance a handle to a "Secret" key. This key can unlock read permission for unclassified documents, read and write permissions for secret documents, and write permission for Top Secret documents. One key per user is all that's needed.

Another advantage of split capabilities over conventional capabilities is revocation. All that's needed to stop Alice from writing Carol's file is to remove the lock 8923 from the table entry for Carol's file. If Alice leaves the system, all that need be done is to remove Alice's keys from the resource table. It doesn't matter with whom Alice shared the keys. Once a key's entry is removed from the table, all uses of that key are immediately invalidated.

As described here, the system has a fatal flaw. There is nothing to prevent an attacker from guessing the handle for a resource or, more importantly, a key. The Client Utility system described in the next section shows one way of addressing this problem.

[4] A real system should also have a "Clone" permission that allows creation of a new key that opens the same lock as its parent.

3. Commercial Implementation

The basic work on split capabilities was done at HP Labs in 1996 as part of the Client Utility project, and split capabilities appeared as the access control mechanism in the first open source releases of e-speak [6]. Subsequent versions of e-speak used SPKI capability certificates [7], as they were deemed more appropriate for business-to-business platforms. In order to avoid confusion with this latter system, we'll use the term *Client Utility* or *CU* when describing the platform using split capabilities. Here we'll describe only those parts of CU relevant to the discussion of split capabilities.

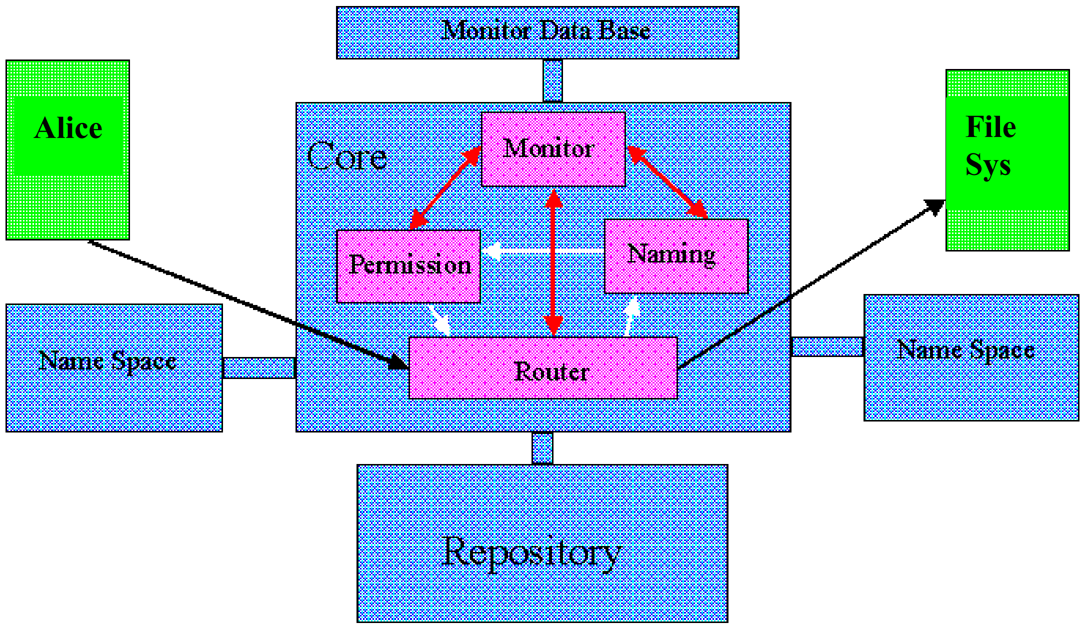

Figure 1: Single machine view.

In order to explain how CU uses split capabilities, it is necessary to first describe the architecture. Figure 1 shows the basic operations within a single *Logical Machine* (the TCB), which consists of a *core* and a *repository*, the latter holding information on resources managed by the core. The core is typically a process running on top of a conventional operating system, such as Unix or Windows. Processes running in their own address spaces that interact using the CU protocol are called *clients* of the core.

The basic unit of control is a *resource*. Before the core can manage a resource, the resource must be *registered* in the repository. Each repository entry contains a designation of which client will act as *handler* for the resource, *e.g.*, the file system, a field of data delivered only to the resource handler, *e.g.*, the inode number, and fields containing locks and permissions.

The fact that you can't hurt what you can't name is central to CU's use of split capabilities. Name visibility is controlled by a *protection domain* the core maintains in its address space on behalf of each client. Clients interact by sending messages to the core consisting of an envelope, which contains information used by the core, and a payload, which contains application specific information. The core never looks at the payload, which means that end-to-end security can be implemented by encrypting and/or digitally signing the payload.

Each protection domain includes a name space for the use of the client. When Carol wants read Bob's file, she constructs a message using her names for the file and the key to be used to unlock read permission, *e.g.*, "bobFile" and "readBobFile". The core looks up these names in Carol's name space to find the repository handles bound to the names. The core uses the information in the corresponding repository entries to construct a message to be delivered to the file system for processing. In this example, the core forwards a message to the file system containing the resource value 329BF5, representing the file's inode number, and the unlocked permission, "R". The file system then interprets the payload to determine whether the request should be honored and what operations to perform.

Should Carol use a name that doesn't appear in her name space, say "aliceFile", she is told that the resource does not exist. Note that the names in each client's name space are specific to the client. Hence, there is no way to guess the name of a resource or for names to be communicated out of band, say by email. A name has meaning only within the CU system and is specific to the binding in a particular client's name space.

Thus far we've talked about access control within a machine. CU would be quite limited unless it permitted access to resources on other machines. Extending the model across logical machines involves three steps - connection, export/import, and invocation.

When two machines first connect, they each start a client to act as a proxy for users on the other machine. These proxies are clients of their respective cores. The connection protocol used by the proxies includes protocol negotiation, mutual authentication, and the exchange of encryption information to be used on the link between machines.

Exporting a resource involves creating an *export form* for the resource's registry entry. The export form of the resources representing permission to read Bob's file might look like

(Resource:resource1),(Permissions:key1).
(Resource:key1),(Permissions:)

Here, the repository handle for the resource we called "/u/bob/file" earlier is bound to the name "resource1" in the exporting proxy's name space. Similarly, the repository entry for the resource we earlier called "bobread" is bound to the name "key1". Since CU does not rely on the importing machine to enforce permissions, only the name of the key that unlocks read permission on Bob's file is communicated.

This export form is passed to the importing proxy, which registers the resources with its core, listing itself as the handler and setting the permissions to the names of the imported keys. Thus, Bob's file and the key representing read permission, registered with Carol's logical machine, would have repository entries

Repository Handle	Type	Value	Permissions
A33BE5	proxy	resource1	(733D:key1)
A33BE4	key	733D	(3822:Destroy)

The type of *proxy* indicates that the request is to be forwarded to the proxy. The value assigned to Bob's file is the name appearing in the exporting proxy's name space, and the permission is the name of the key in the exporting proxy's name space that unlocks the read permission.

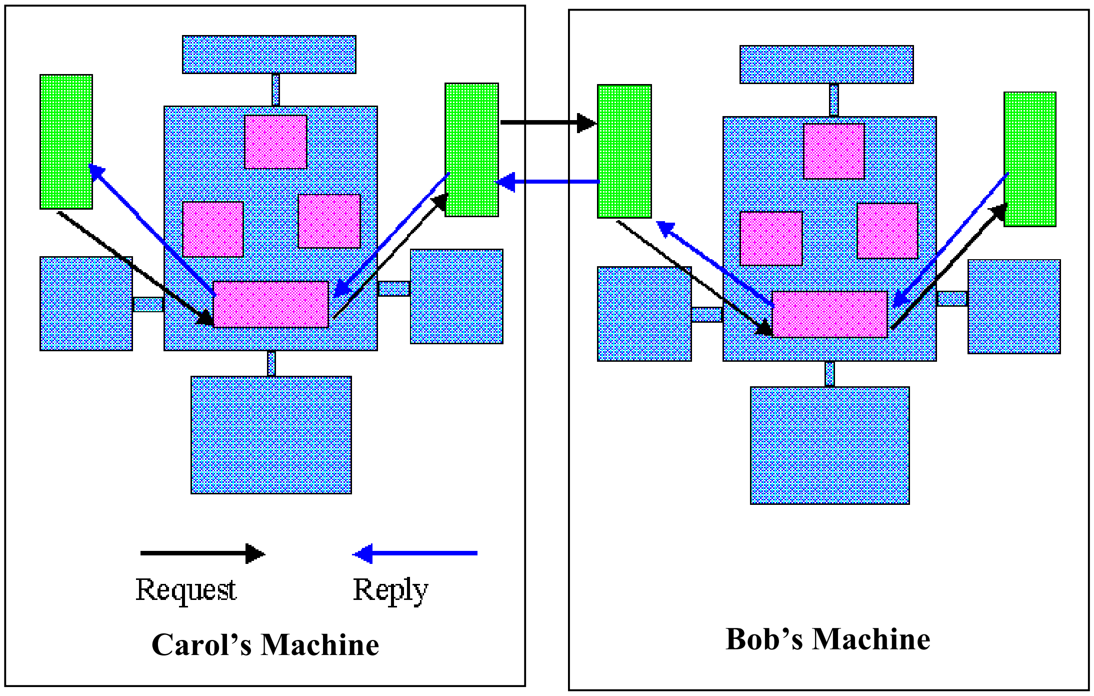

Figure 2: Accessing a remote resource.

A request from a client on one logical machine for a resource on another follows the path shown in Figure 2. Carol's core does exactly what it does in the single machine case; it forwards the request to the handler, which is the proxy in this case. If Carol's request unlocks read permission on Bob's file, the proxy will see a permission of "key1" on a resource with a value "resource1". The proxy sends this request to its counterpart on Bob's machine. That proxy repeats the request with the names "resource1" and "key1". The file system sees a request for inode number 329BF5 with permission "R", the same request it would have seen had Carol's request been local. Thus, neither Carol's application nor the file system nor either core needs modification to deal with remote requests.

4. Visibility

Control of what a client can name is an important part of CU access control. In this section we'll see how CU can make it difficult for someone to obtain certain name bindings or use them should they be made available. Actually, the CU architecture requires something better than described so far for a reason that has not been explained yet. So far, the only method described for getting name bindings was to have them available as part of the client's initial environment. There are two other methods.

1. When a message is received, name bindings for all the resources named in the original message, except for keys submitted to unlock permissions, are put into the name space of the recipient.
2. A field in the repository entry describes the resource so it can be discovered with a look-up request to the core.

Security rules could be violated if one client inadvertently sent a name binding to a client who should not be allowed access to the resource or if a client discovered such a resource. This situation is a particular problem with bindings to keys.

Our solution is to add to the repository entry of each resource two lists of locks that we call *allow* and *deny*. Each look-up and each attempt to use a name in a message is checked by comparing these lists to the keys accompanying the request. If any of the deny locks is opened, or if none of the allow locks is opened, the system acts as if the resource does not exist. These keys can be the same ones used to grant access rights.

Of course, it's natural to ask why any client would submit keys that might deny access to some resources. There are several reasons. The request might be part of a test run of a program, and the user doesn't want to risk damage to the resource. Also, the key that denies access to one resource may be needed because it grants a desired access right or appears in the allow field of another resource that the client wants to access. Finally, CU includes *mandatory keys* in each protection domain that are implicitly included in each request. This last feature allows a system administrator to enforce a variety of policies by putting certain keys in each client's protection domain. We saw one such example in Section 1, where we wanted to enforce multi-level security [4]. Putting the "Secret" key in the client's protection domain guarantees that this key is included with every request. Clients do not have names bound to their own mandatory keys, so they can't remove them from their protection domains nor can they share them with other users.

The visibility tests allow us to enforce some common security policies in a rather straightforward manner. One is compartmentalization, in which a client in one compartment may not access resources in another compartment. An example might be a consulting company that wants to assure a customer from company ABC that people working on its contract can't accidentally mix resources with those working on the contract for a competitor, say company XYZ. The administrator can put a lock corresponding to one key in the allow field of resources associated with company ABC and the same lock in the deny field of those resources associated with company XYZ. If the corresponding key is put in the protection domain of one of the consultant's employees working on a project for ABC, that person will not even be able to find out that resources associated with company XYZ exist.

Compartmentalization is an example of the problem of *rights amplification*, the ability to do something only if two, separate access rights are available. For example, if I give Alice access to the can opener and Bob access to the can, my tuna is safe as long as the capabilities cannot be presented together. Normally, we don't care if a client gives a capability to another client. After all, Alice could read a file and send Bob the results, so it doesn't matter if Alice gives Bob the capability to read the file. However, if Alice can give Bob access to the can opener, they can share the tuna. Split capabilities with visibility control can prevent such rights amplification. The key that grants access to the can opener also denies access to the can, and *vice versa*.

Visibility controls can also be used to implement the *restricted *-property* of multi-level security, a policy that allows a user to read and/or write resources only at the client's security level. Putting a lock in the allow field of every confidential resource, a different one in the allow field of every secret resource, and a third in the allow field of every top secret resource allows this level of control. Putting the key corresponding to the user's security level in the protection domain of a client guarantees that the client can only access resources at its level. Although less flexible than the full *-property, the restricted form is enforced by the TCB, while the full form is enforced by the resource handler.

5. Attacks

We don't know how well the system will stand up to real attacks, but we can look at a number of possibilities to see if we can identify weaknesses. Here, we'll look only at attempts to perform unauthorized actions. Many denial of service attacks are also dealt with by the CU architecture, but they are not relevant to the present discussion. We won't worry about some social problems, such as poorly chosen passwords or people who write their PINs on their ATM cards. There is also nothing we can do about attacks against the underlying operating system or its components.

Social engineering is always a concern. An attacker might get a user to reveal the names of certain resources, particularly keys. This information does the attacker no good, since the user's names have meaning only through the bindings in the user's name space. Tricking a user into transferring name bindings is possible, but visibility rules can mitigate the damage. Also, since keys are rarely transferred this way, such a request should make the user suspicious.[5]

The attacker might also go directly after the core and attempt to learn the repository handles of certain resources. Again, the information does the attacker no good, since the core does not accept repository handles from clients. What if the attacker learned the

[5] The CU design provides an additional level of protection; the right to transfer a name binding can be controlled.

value associated with one of the locks? No problem, since the core only accepts names bound to the repository entries of keys, not references to locks.

The attacker could also try to get the name of a resource exported to someone else. This information is useless, because the proxy acting on behalf of the attacker has name bindings only for resources exported to the attacker. The attacker could induce the user to start a process running malicious code, but the default behavior should be to start processes with minimal privileges. Hence, the user will be warned of suspicious activity when the process asks for unexpected access rights.

Next, consider a malicious user on a single machine who would like to do some unauthorized operation. Since any user has access to a process with enough resources to complete a valid login attempt, the attack could start there. This task will have a protection domain with some named resources. The attacker has no names for anything else, so the initial attack will be against those resources. We can make sure that the attack does no harm by giving this process access to keys that would allow it to modify only resources needed by the login procedure.

Our attacker now attempts to get additional resources into the protection domain of this login process. However, without access to keys granting permission to add these resources, this attack fails. For example, the initial set of keys made available to anonymous clients need not include permission to add bindings to the client's name space. Additionally, the mandatory keys in this protection domain can include a key corresponding to a lock in the deny field of every resource not needed to complete the login process.

Say that the attacker has logged in, either by guessing a password or because the attacker is an authorized user. The attacker can now modify any resources the active account can. The attacker can also add to its task's name space bindings to any resources made available to it by others. However, there is no way for the attacker to even name a resource that doesn't have a binding in its name space. Since the system administrator controls critical system resources, such as the system log files, visibility tests can prevent general users from getting access to them.

If we're dealing with more than one machine, the attacker can try to get the other machine to do something unauthorized on its behalf. However, it doesn't matter if the attacker is running on a machine with a corrupted core, modified operating system, or customized hardware. Even if the attacker refuses to honor the permissions in the exported resources, the attack fails because these permissions are checked when the proxy on the machine that owns the resources attempts the access. Even if that proxy is corrupted, it can only do what is authorized through its protection domain.

6. Related Work

A complete review of access control mechanisms, or even only capabilities, is beyond the scope of this paper. The early history [1] and modern developments [3] of capabilities are well documented elsewhere. Here, we'll briefly describe the archetypes of the different capability systems mentioned in Section 1.

SPKI capability certificates [7] contain a name for the resource and one or more access rights. Wildcards can be used to grant access to a group of similarly named resources, such as files in a directory. SPKI certificates are open documents that are digitally signed to prevent forgery or tampering. These signatures are checked against the resource handler's trust assumptions before granting access. The submitter of the capability must prove knowledge of the private key associated with the public key assigned to the capability. Anyone with a SPKI certificate can create a new capability containing a subset of the access rights. Hence, SPKI capabilities don't have all the scalability problems of traditional capabilities, but revocation is a problem.

E-language capabilities [2] are quite different. Each is a handle to a facet of a particular bit of state. Several facets may have access to this state, each with a subset of the methods used to manipulate it. For example, one facet of a file may have read, write, and execute methods, but another facet of that file may have only a read method. The access rights are implicit in the facet being accessed, in contrast to a traditional capability in which they are listed explicitly.

There is a close analogy to split capabilities, namely virtual memory. Each "name" is a virtual address useful only to the process using it. In some systems [9], access to the pages of the virtual memory is controlled by *storage protection keys*. Split capabilities as we have used them are more general, applying to any kind of resource, and need not involve the operating system.

7. Summary

Capabilities have a number of advantages over access control lists [10], and our implementation of split capabilities has advantages over traditional capabilities, namely

1. Revocation is easy.
2. The number of elements to be managed scales only with the sum of the number of resources and the number of access rights instead of with their product.
3. Using split capabilities with visibility controls allows the simple specification of complex security policies.

We have only implemented split capabilities within the CU architecture. It may be possible to use them in other environments, for example, those with a global name space. Split capabilities could also be used as the basis for building a capability secure operating system such as Eros [11,12].

Acknowledgements

We'd like to thank Harumi Kuno, Nigel Edwards, Jonathan Shapiro, Chris Hibbert, and all three referees for helping us improve the presentation.

References

1. Henry M. Levy, *Capability-Based Computer Systems*, Digital Press, Bedford, MA (1984)
2. http://www.skyhunter.com/marcs/ewalnut.html
3. http://www.skyhunter.com/marcs/capabilityIntro/index.html
4. David E. Bell and Leonard La Padula, *Secure Computer System: Unified Exposition and Multics Interpretation*, ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA 01731 (1975) [DTIC AD-A023588] http://csrc.nist.gov/publications/history/bell76.pdf (1975)
5. Alan H. Karp, Rajiv Gupta, Guillermo Rozas, Arindam Banerji, "The Client Utility Architecture: The Precursor to E-speak", HP Labs Technical Report, HPL-2001-136, (2001) available at http://www.hpl.hp.com/techreports/2001/HPL-2001-136.html
6. "E-speak Architectural Specification: Beta 2.2", ftp://ftp.hp.com/linux/espeak/Architecture_2.2.pdf (1999)
7. ftp://ftp.hp.com/linux/espeak/Architecture_3.14.pdf (2001)
8. Norman Hardy, "The Confused Deputy", Operating Systems Reviews, **22**, #4, (1988)
9. *System/370 Principles of Operation,* GA22-7000-9, IBM (1983)
10. http://www.erights.org/elib/capability/consensus-9feb01.html (2001)
11. http://www.eros-os.org/essays/ACLSvCaps.html
12. J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A Fast Capability System", in *Proc. 17th ACM Symposium on Operating System Principles*, pp. 170-185, Kiawah Island, near Charleston, SC, December (1999), see also http://www.eros-os.org/