Singularity Design Note 26

Security: Permissions

In which we describe what access-control permissions are, and how they are implemented.

1. Introduction

For each possible principal/object (or actor/subject) combination, an access-control system specifies a set of ***permissions*** (aka rights).

Most existing ACL systems only support a static set of possible permissions, such as r/w/x. Other ACL systems support an extensible, general set of permissions, but leave those permissions unenforced by the system mechanisms. In this latter type of extensible ACL systems, application programs must interpret permissions—a cumbersome and error-prone task. Auditing which applications use which extended permissions, and how, is near impossible.

Singularity gives us the opportunity to reconsider our aspirations for access control. In particular, we might hope to provide static guarantees about access requests. For example, in Singularity, a service might expose two distinct, isolated contracts, say ReadContract and a ReadWriteContract, and be guaranteed that only code that refers to the ReadWriteContract can do certain operations. Unprivileged code would be allowed to instantiate and refer to the ReadContract, but not the more powerful ReadWriteContract. (This can be enforced, for example, by setting ACLs on the contracts themselves, checked during code installation, perhaps such that only code signed by certain publishers can refer to the extended contract.) At the very least this will help the static auditing of the system: i.e., what code can do what things.

Contracts (and their channels and endpoints) seem like a natural place for access control in Singularity, since they are the means of regulating communication. However, creating and maintaining different contracts for different access permissions is both cumbersome and hard-to-get-correct—at the very least programmers should be supported by languages, tools, or other mechanisms. Even then, it is not clear how permissions, such as read or write (or fast-forward, play, and pause), map onto contracts.

Our thesis is that

(A) a simple set of program annotations, as well as small extensions to the system runtime mechanisms, can allow programmers to easily define permissions for contracts, and regulate the permissions granted to each user of a contract,

(B) permissions can be contract-specific, i.e., defined in terms of a contract's messages and state, and yet be enforced by the system mechanisms, with close to no access-control-related imperative code having to be written by those who implement contracts.

The possibility of static reasoning about the system should not be reduced by any mechanisms that are introduced to help the programmers. Therefore, we believe that any proposals should pass the following litmus tests:

(C) the same type of static reasoning should be possible in the system, as if programmers had created separate, distinct contracts (such as ReadContract and ReadWriteContract in the example used previously), and

(D) the same mechanism that supports static reasoning about the system—i.e., what code can use what permissions on what contracts—should also support the dynamic assignment of permissions based on dynamic principals (e.g. those determined at runtime such as users and restricted program instances).

Below, we propose an implementation that we believe achieves the above goals, and proves our thesis.

2. Defining permissions based on contracts

To make things concrete, below is a simple example of a contract that might come in both read and read/write flavors.

```
contract StringDictionaryContract {
    in message Lookup(String key);
    in message Insert(String key, String value);

    out message LookupAck(String value);
    out message LookupNack;
    out message InsertAck;
    out message InsertNack;

    START: one {
      Lookup ?  -> LOOKUP-ACK ;
      Insert ? -> INSERT-ACK  ;
    }
    LOOKUP-ACK: one {
      LookupAck! -> START;
      LookupNack! -> START;
    }
    INSERT-ACK: one {
      InsertAck !  -> START;
      InsertNack ! -> START;
    }
}
```

Our proposal is that each message in the contract be annotated with a [permission:name], where the name is an arbitrary string identifier, tied to the contract, that identifies a set of messages. For technical reasons we discuss later, permissions are only associated with in-messages sent from client to server (Imp to Exp). Below we see our example contract with annotations for read and write.

```
contract StringDictionaryContract {
    permission read;
    permission write;

    [permission:read]
    in message Lookup(String key);
    [permission:write]
    in message Insert(String key, String value);

    out message LookupAck(String value);
    out message LookupNack;

    out message InsertAck;
    out message InsertNack;

    START: one {
      Lookup ?  -> LOOKUP-ACK ;
      Insert ? -> INSERT-ACK  ;
    }
    LOOKUP-ACK: one {
      LookupAck! -> START;
      LookupNack! -> START;
    }
    INSERT-ACK: one {
      InsertAck !  -> START;
      InsertNack ! -> START;
    }
}
```

As mentioned above, a permission is named by an arbitrary string and declared at the beginning of a contract. This name is scoped by its containing contract. It is likely that the same permission names (such as read and write) might be used in different contracts, and it seems potentially worthwhile to have a convention—at least for the system contracts—about the names of permissions. This might, for example, help in the creation of simpler, more general ACLs on system objects.

The idea is that any particular client endpoint of the contract (.Imp side) is constrained to a set of named permissions, which in turn limits the set of messages the client can send in each state. Below is our example contract when constrained to the read permission.

```
contract StringDictionaryContract {
    [permission:read]
    in message Lookup(String key);

    out message LookupAck(String value);
    out message LookupNack;

    out message InsertAck;
    out message InsertNack;

    START: one {
      Lookup ?  -> LOOKUP-ACK ;
    }

    LOOKUP-ACK: one {
      LookupAck! -> START;
      LookupNack! -> START;
    }

    INSERT-ACK: one {
      InsertAck !  -> START;
      InsertNack ! -> START;
    }
}
```

In the above example, note how in the START state transitions for non-permitted messages have been elided. The INSERT-ACK state is still part of the contract, although it is unreachable from the START state. The reason for this is subtle. As we will see later, the client can at any point reduce its view of the permissions on the client endpoint. Since such a reduction might happen in the INSERT-ACK state that was previously reachable, it is possible to have a StringDictionary.Imp endpoint in state INSERT-ACK with statically known [read] permission only.

The above permission framework can be extended in various ways. One simple (and probably important) extension is to permit annotations such as [permission:X,Y] on messages, specifying a disjunction of permissions, namely that the message can be sent either with permission X or Y.

Orthogonally, it would be useful to associate a partial order among permissions. In our example contract, this would allow the definition of two permissions: `read`, and `readwrite`, where `readwrite <= read`, ensuring that the `readwrite` permission always implies the `read` permission.

3. Using channels based on contracts with permissions

Contracts annotated with permissions give rise to a family of sub-contracts for each subset of possible permissions. For each such contract, there are corresponding types for the importing endpoint. Exporting endpoint types do not carry static type information about permissions.

```
StringDictionaryContract.Imp;       // No permissions, just the base contract
StringDictionaryContract[read].Imp; // Read permissions only
StringDictionaryContract[write].Imp; // Write permissions only
StringDictionaryContract[read,write].Imp;  // Read and write permissions
```

A new channel can be created for each combination of endpoint types.

```
StringDictionaryContract.Imp[read,write]! imp;
StringDictionaryContract.Exp! exp;

StringDictionaryContract[read,write].NewChannel (out imp, out exp);
```

Each channel is created under a specific set of permissions associated with the client side of the channel. These initial permissions act like the dynamic type of an object in that they cannot be changed at runtime. However, just as one can upcast an object pointer and view the object under a super-type, client endpoints may be viewed under fewer permissions than those associated with the channel at creation.

Note that we assume that creating a channel without specifying a permission set corresponds to a channel with the empty permission set.

If syntax is needed for specifying all permissions one can define partial orders on permissions as follows:

```
contract StringDictionaryContract {

  permission all;
  permission read <= all;
  permission write <= all;
  …
```

To create a contract with both read and write permissions, one can then use the `all` permission.

```
StringDictionaryContract.Imp[all]! imp;
StringDictionaryContract.Exp! exp;

StringDictionaryContract[all].NewChannel (out imp, out exp);
```

Message argument types that are themselves endpoints specify the permissions associated with the endpoint by the simple means of using the endpoint types qualified by permissions as shown above.

```
contract FooService {

  message Connect(StringDictionaryContract[read].Imp! arg);
  …
```

It might be possible to provide an API to allow applications (either client or server) to restrict permissions dynamically, for example since channels are bi-directional, a client might choose to dynamically reduce permissions when binding to a specific server. We currently have no plans to support such an API.

It should not, in general, be necessary for applications to manipulate and query permissions on channels. The type system should enforce that all interaction sequences according to the endpoint types are permissible and safe. The Singularity runtime will check that communications over channels comply with the protocol (including permissions) associated with a channel. In the event that a permission fault is detected, an error should be thrown. It's important to note that any error constitutes a violation of the underlying contract and must be handled in the same manner as if the process holding the other endpoint had terminated.

3.1. Sub-typing induced by permissions

Contract inheritance provides sub-typing for exporting endpoints. Permissions add a sub-typing hierarchy to importing endpoints as follows: Given two endpoint types C[P].Imp and C[Q].Imp, where P and Q are permission sets, then C[P].Imp is a subtype of C[Q].Imp if Q is a subset of P.

In our StringDictionary example, we thus obtain the following sub-type relations:

```
  StringDictionary[read].Imp <= StringDictionary.Imp
  StringDictionary[write].Imp <= StringDictionary.Imp
  StringDictionary[read,write].Imp <= StringDictionary[read].Imp
  StringDictionary[read,write].Imp <= StringDictionary[write].Imp
```

These relations are natural in that they allow a client to forget some permissions and thus use only a subset of the functionality of a channel.

In order for these sub-type relations and the corresponding up-casts to be sound, we require the following restrictions on contracts with permissions

1) Permissions are only associated with messages sent from the Imp to the Exp side. Messages sent by the Exp side are visible under all permission sets.

The restriction guarantees that clients can up-cast the Imp endpoint to fewer permissions in any state. If the state is a send state for the client, the upcast may result in fewer possible messages the client can send. If no messages can be sent from a state under a set of permissions, then the state is terminal. If the upcast of the client side endpoint happens in a client receive state, then the client still needs to handle all possible messages sent by the server, since messages in the server-client direction do not have associated permissions.

Downcasts on Imp endpoints (gaining more permissions) are not permitted, even if the exporter that originally granted permissions allowed equal or greater permissions and the endpoint was subsequently upcast. This is especially important in an environment where endpoints can be passed to other processes. We want it to be impossible for a client to move an upcast endpoint to another process where it might be unexpectedly downcast.

3.2. Service providers

A service provider implementing the StringDictionaryContract.Exp service should be implementable without the need for separate code paths for each subset of permissions (i.e., [], [read], [write], [read,write]). Instead, the service provider should be able to implement the service in a permission agnostic way, where it simply provides the full service [read, write]. Any particular endpoint it serves might only exercise part of the protocol. It is the responsibility of the type system and contract inheritance rules to guarantee that no client/service ever sees a message that is not expected.

Thus, server side endpoint types (.Exp types) do not have associated permissions, since the server cannot be sure as to the set of permissions assumed by the client (the client can reduce its statically known permissions at any point). Receives on .Exp endpoints therefore always assume that all messages (under all sets of permissions) could be received.

4. Enforcement of permissions

Principals in Singularity are fundamentally determined at runtime since it is difficult to know a priori which programs will run and on whose behalf. It is the subject of future research to determine if a static declaration of system behavior can give security guarantees that are also useful. Nevertheless, given the existence of dynamic principals, permissions checks must also be dynamic.

We describe below a mechanism for dynamic access control at binding time. By binding time, we mean dynamic binding, when a service accepts a new connection for a particular contract. If and when we determine how to declare system security policy more completely in manifests, we might be able to perform some binding time checks when an application is installed rather than at runtime.

It is at binding time that ACLs and the security subsystem are consulted to control the acceptability of new endpoints and their implied permissions. Once binding is accomplished, permissions need not be checked again by application code (e.g. on each message send). Instead, the enforcement of permissions is now a matter of protocol conformance to the specified contract. The Singularity runtime currently checks conformance with the protocol. Ultimately, we expect that protocol conformance can be enforced statically.

For each Singularity application, the contracts, and sets of requested permissions on those contracts, can be statically determined from the application manifest, or from the types of endpoints manipulated by the program.

4.1. Bind-time access control

In Singularity, channel binding can occur in two situations. In both cases a channel endpoint is passed over an existing channel. However, in one case the resulting channel is bound to a named object in the namespace, and in the other case the channel is anonymous. (Although new processes receive initial channels that don't arise from endpoint marshalling, these channels fall into the anonymous category.) In our model, anonymous channels are much like capabilities in that the access control system does not constrain the passage of endpoints between cooperating processes, except as we discuss below under policy constraints on endpoint movement.

When the exporter endpoint is presented to a service provider for binding to a named object, an access control decision must be made. The code that implements name service binding must be able to query the incoming endpoint to determine what dynamic (maximal) permissions are associated with the channel. This can be implemented by adding a new endpoint method call, perhaps returning a bitmap of requested permissions. For this purpose, it might be useful to associate an integer value with a permission so as to indicate a bitmap position and to facilitate storage of ACLs as indexed by permission. The representation, evaluation, and storage of ACLs for objects in the namespace is considered by other SDNs, and is outside the scope of this document. However, one possible mechanism for using ACLs in the context of permissions is that for each namespace object and contract type, there is a set of ACLs: one per permission. In order for achieve a successful binding, the requesting principal must be granted access by ACLs corresponding to all requested permissions. It also might be possible to represent permissions within ACLs and thereby combine the two abstractions.

It would be preferable if the manipulation of the actual permissions on endpoints can be abstracted away and implemented in the security subsystem, so that service providers do not have to handle them explicitly. For example, a service provider would ideally manipulate an abstract policy object that was built in part from declarative specifications and in part from parameters such as file-meta-data (for the file system). The service provider would then ask the security subsystem whether an endpoint satisfies the policy, given the endpoint's permission and peer principal. In that way, the service provider would not need to know the actual permissions on the endpoint, only the security subsystem would inspect them.

4.2. Per-message permissions enforcement

In the abstract, per-message permission enforcement is accomplished implicitly through contract conformance and the type system. The runtime only instantiates code for sending messages that are permitted by the active permission set. In practice, however, handling all possible combinations of permissions may produce too many types to represent statically at runtime. Therefore, we imagine that the dynamic permissions for a channel (those set at creation) can be stored in the channel (or endpoint) data structure, and that this can be used to drive runtime enforcement. The required data can probably be as simple as a bitmap.

The precise implementation details for runtime permission checking are TBD.

4.3. Policy Constraints on Endpoint Movement

It may prove useful to allow system policy to constrain movement of endpoints (and their permissions) between processes. The mechanisms that we develop should be designed to make security-related programming more fool-proof. There are lots of possibilities, some of which we describe here.

One option might be to restrict "switch receive" to only be allowed on BoundExp endpoints, with "bind" being a polymorphic operation that transforms C.Exp to C.BoundExp for all C, and which might also be the point of ACL checks and might also prevent further movement of the BoundExp endpoint. This would enforce the uniform application of security checks at bind-time.

So, for example:

```
      C.BoundExp = System.BindWithACLs( C.Exp, "\foo\bar" );
```

might make an access control decision based on the dynamic principal of the corresponding C.Imp at that point in time.

Also, we might restrict C.Imp movement after a bind on its C.Exp counterpart. For instance, C.Imp might not be movable at all (this is probably much too restrictive), or C.Imp might only be movable to dynamic principals that extend the bind-time principal. For instance, if the bind-time principal is "/sing/login@ulfar + /bin/word", then C.Imp could only be moved to processes whose dynamic principals start with that prefix (i.e., children or roles of that principal).

A variant on this idea might allow endpoint arguments to be marked with a [RequiresAuthentication] attribute. The receiver of such an endpoint would have to perform and ACL check as above before the endpoint becomes useable.