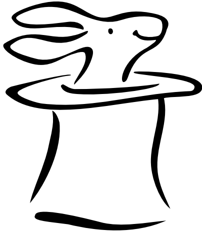


An Introduction to Safety and Security in Telescript

Telescript is a software technology for building distributed applications using *mobile agents* [Wh95a, Wh95b, STAN95, IBM95a]. Telescript extends the concept of remote programming, the ability to upload and execute programs to a remote processor, with migrating processes. A Telescript mobile agent is a migrating process that is able to move autonomously during its execution to a different processor and continue executing there. Mobile agents conceptually move the client to the server, where stationary processes, or *places*, service their requests. When it is done in a place, an agent might choose to move itself to a different processor, carry results back to where it originated, or simply terminate.

Clearly, security is a major concern in this scenario. The operator of a Telescript processor wants some assurance that nothing bad will come of its decision to admit an incoming agent. The host platform wants to know who is responsible for the agent. The agent, on the other hand, would like to trust that private information it is carrying will not be disclosed arbitrarily. It needs to trust the operator of the platform. The server application responsible for the place might also like to set its own policies.

Telescript provides the necessary security through a combination of mechanisms:

1. Basic Runtime Safety
2. Process Safety and Security
3. System Safety and Security
4. Network Security

This document provides an introduction to Telescript's safety and security features. As is true of any security mechanism, Telescript's are only effective if properly used. A number of examples of their use are provided.

This document is organized as follows. The next section gives a brief introduction to Telescript language and network concepts. The following section introduces general safety and security concepts and terminology. The next four sections describe each of the above mechanism groups, in turn. The last section gives direction for future security work.

Table of Contents

Telescript Concepts 1
- Language and Engine 1
- Region and Network 2

Safety and Security Concepts 2
- Safety 2
- Security 2

Object Runtime Safety 3

Process Safety and Security 3
- Protected References 4
- Encapsulation and Access Control 4
- Permits 6
- Security-Oriented Mix-in Classes 7
- Entering and Meeting 7

System Safety and Security 9

Network Security 9
- Privileged Region Authority 9
- Administrative Policy in the Engine Place 10
- Secure Channels 10

Future Security Directions 10

References 11

Telescript Concepts

What is Telescript? This section provides a brief introduction to Telescript concepts and terminology. For details, see [WH95c, WH95a, WH95d]. Familiarity with object programming concepts is assumed [DAG95].

Language and Engine

Telescript is an interpreted, dynamic, object-oriented programming language for writing mobile agents. A Telescript "program," or script, consists of a collection of classes. Classes are hierarchically organized by sub-classing, and Telescript supports a form of multiple inheritance using *mix-ins* . Classes have *features* which are their externally observable operations and attributes.

Features may be public or private. In contrast to the object models in some other languages, Telescript's private features are visible in sub-classes as well as in the base class. Features, or entire classes, can be *sealed* so that they cannot be overridden in sub-classes or sub-classed, respectively. Telescript does not permit incompatible signatures (feature overloading) in sub-classes.

The Telescript language reference specifies a number of predefined classes. These must be supported in every implementation of a Telescript interpreter, or *engine*.

Objects are always instances of some class, and always inherit from the class Object, which is at the top of the object hierarchy. An object encapsulates its *properties*, specified by the class and directly accessible only to code "inside" the object (and not to sub-classes). Properties reference other objects. Agents and places are also objects, instances of various sub-classes of the predefined abstract class Process.

Process objects provide Telescipt's multi-tasking functionality. Processes are pre-emptively multi-tasked, and scheduled according to priority. Process classes have a *live* operation which is invoked directly by the engine, on a new thread of control, to animate new process objects.

All objects must be *owned* by at most one process. Objects that are not owned are subject to being garbage collected. A process "owns" itself. A process can transfer ownership of an object to another process, provided it owns the object and all objects referenced by that object's properties, and, recursively, all objects referenced by objects in that object's *closure*.

The Agent class, a sub-class of Process, provides the sealed, private `go` operation. An agent can request the `go` operation on itself, providing a *ticket* argument. The ticket object describes the place where the agent is trying to go, with varying levels of specificity. If the engine can figure out where and how to route the agent, and the trip is ultimately successful, the agent "wakes up" executing its next line of code in the destination place.

Agent processes always execute in the context of one or more enclosing place processes. Places usually provide a service API for agents to interact with. Places can be nested within other places, and every engine has an *engine place* as its outermost place. As one of the last steps in processing an agent's `go`, the destination engine requests the `entering` operation on the destination place to give that place the opportunity to deny occupancy. If no place satisfying the ticket will admit the agent, the trip fails with a trip exception.

Telescript engines from General Magic include, in addition to the Telescript interpreter, supporting runtime components such as additional built-in classes, facilities for object persistance, and communications functions for moving agents around. A Telescript engine thus has a number of significant differences from other, more conventional "load and execute" remote programming runtime environments, for example, Java [JAVA95a], Safe-Tcl [SAFE-TCL], or Oblique [OBLIQUE]. A Telescript engine needs to support a multi-user, multi-process operating environment, not unlike a mini operating system unto itself.

Region and Network

What is a Telescript network?

A Telescript network consists basically of a set of interconnected Telescript engines. Telescript imposes no limit on the particular network technologies that might be used to transfer encoded agents between engines. Currently, engines from General Magic support agent transfer over TCP/IP, as well as using UDP over PPP for dialup links. An engine provides the platform-specific code that enables agent portability.

Each process has a name, an instance of class Telename, that has two OctetString-valued attributes, its *authority* and its identity. An authority uniquely identifies a Telescript user. If two processes have Telenames with the same authority, they are running on behalf of the same user.

An engine place runs under what is known as the *region* authority, analogous to the root account on a UNIX(tm) system. A region can consist of multiple engines with engine places running under the same authority. An agent may travel between places on a single engine, on different engines in the same region, or to an engine in a different region. For an agent to travel between engines, it must be serialized by the sending engine's *encoder* function, and *decoded* by the destination engine. The encoding process takes care of encoding the agent's current runtime state, including in it all objects owned by that agent, along with all referenced classes, unless a class is built-in or otherwise known to already exist at the destination. An agent travels self-contained, so to speak.

Anybody who wants to can put up a region. A Telescript network potentially includes regions that not everyone particularly trusts, and which certain agents might want to avoid.

Safety and Security Concepts

What is meant by safety and security, and what's the difference?

Safety

Safety is a term generally used in the context of programming languages. In a "safe" language, programs either do what they are supposed to do or fail gracefully [DAG95]. Safety features mainly promote robustness and prevent accidents.

A typical C program can count on relatively few safety guarantees. Only extreme diligence by the programmer catches hazards like accessing uninitialized pointers, referencing discarded objects, referencing beyond the end of an array, pointer arithmetic errors, or copying in a string that is longer than the destination buffer. Fortunately, the damage that such programs do is usually limited by the operating system [DAG95]. However, such hazards in critical programs, such as mail daemons, represent major opportunities for system penetrators.

Security

Security features, on the other hand, are intended to provide protection and integrity in the presence of malicious users. Security features enforce security policies. Without a security policy, even a default one, security can be a difficult thing to assess.

There is significant conceptual overlap between safety and security. For example, Telescript uses encapsulation, which is usually considered a safety mechanism, as a protection mechanism.

Very little has been written on general techniques for protecting against attacks such as Trojan horses [KAR87]. Generally, people are justifiably reluctant to let anyone run whatever programs they wish on their

systems, for fear of viruses or worms, for example. One common approach is to run untrusted programs in a "safe" or restricted environment. This is used, for example, by the Safe-Tcl interpreter, which has the "dangerous" primitives of Tcl removed [BOR93]. It is also usedby the HotJava(tm) class loader [JAVA95b], which partitions the runtime environment into different namespaces, with "safer" built-in classes available in less-trusted namespaces. Telescript also provides a form of restricted environment, using places and local permits.

Much of Telescript's security is in the form of features whose use is discretionary. Security features permit a place or agent to protect itself against untrusted (and potentially malicious) agents or places. Programs can defend themselves, but they don't necessarily have to. Some Telescript security features are specified in the basic language. Others, especially relating to specific security policies, are implemented by Telescript applications themselves.

The remainder of this document describes the particular safety and security features of Telescript.

Object Runtime Safety

Because of its polymorphic structure and interpreted nature, Telescript can intrinsically provide basic runtime safety in the above sense of shielding the programmer from hazards that would cause the engine to crash. There are no pointers or pointer arithmetic. Instead, Telescript only provides structured object access using references.

All Telescript engines provide:

- runtime type checking with dynamic feature binding. The engine raises an exception if a program attempts to follow a null reference or access an unavailable or non-existent feature.
- automatic memory management. The engine allocates storage when objects are created and does automatic garbage collection on unreferenced objects. There is no explicit "free" operation to discard an object.
- exception processing. Programs have the opportunity to, and should make every effort to, catch exceptions. Otherwise, uncaught exceptions usually result in process termination.

Telescript defines a number of built-in data structure classes as sub-classes of the Collection class. These are accessed using "safe" operations. For example, the List class permits elements to be extracted by index position. An exception is raised if an invalid position is specified.

Process Safety and Security

Since processes are objects, and an object interacts with another object by invoking its features, how can one process's objects interact safely with another process's objects, if those objects have unknown and potentially untrusted class implementations? What if an untrusted operation contains a "Trojan horse" that makes unauthorized (and unexpected) copies of sensitive objects, or contains a "logic bomb" that causes the calling place or agent to die? [IBM95b]

Basically, all interprocess access is mediated by the Telescript engine. As with a domain (rings of protection) machine, the Telescript engine is like an operation system kernel, isolated and protected from the programs running on it. The engine may be considered part of a "trusted computing base" (TCB) that provides mechanisms for controlled sharing and safe process interaction[TCSEC, SALZ75].

All Telescript engines provide:

- an authenticated, unforgeable identity for each process, in the form of an authority. Authentication mechanisms are discussed under Network Security, below.
- protected references
- protection by encapsulation of private properties and features. This forms the basis for object-enforced access controls.
- quotas and process privileges using permits, including control over creation of new processes.
- security-oriented mix-in classes (Copyrighted, Unmoved, Protected, Uncopied)
- mediated protocols for process rendezvous (for example, entering a place, and Meeting Agents)

Protected References

A protected reference to an object cannot be used to modify that object. The object is, as far as the holder of that reference is concerned, read only. Objects can be passed by protected reference as arguments to operations, or returned by protected reference as results, according to the class definition. A protected reference cannot be used to transfer ownership of an object.

A protected reference to an object can be created at any time using the `protect` operation, which is sealed and defined on the class Object.

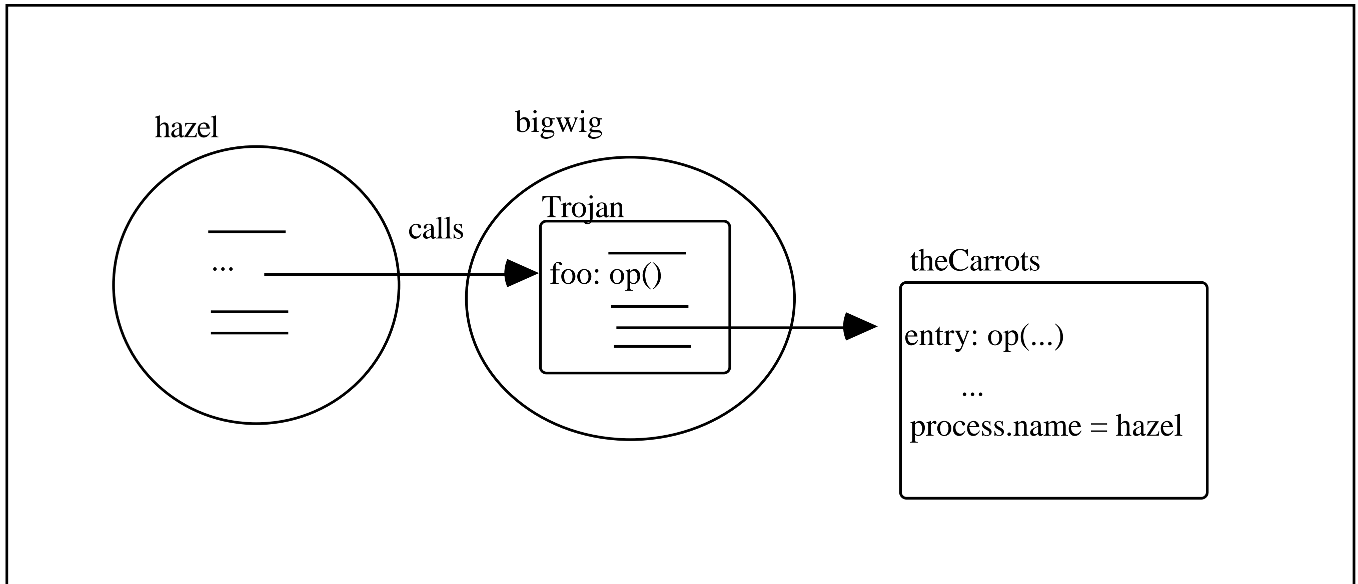
Encapsulation and Access Control

Typical object oriented languages use encapsulation and information hiding for software engineering purposes. Telescript uses encapsulation to provide an engine-enforced protection perimeter around objects. In particular, the only access to an object's properties is through its public features, which can (and should) enforce their own protection policies [LAMP71].

By providing public operation wrappers that make access checks around the private features that actually perform the function, programmers can take advantage of encapsulation to implement their own access controls. Telescript provides different ways for identifying the authority and class of the caller (i.e., the *requester*, or *client,* from the point of view of the executing code) that are useful in making identity-based access checks. These are obtained directly from the engine in global variables, and include:

- the current **process**. An unprotected reference to the process that owns the computation thread that requested the operation.
- the current **owner**. An unprotected reference to the process that will own (retain owner references to) any objects created. The owner is usually the current process, but can be temporarliy changed, for code executed within an `own` block, to one's own owner. That is, to the owner of the object that actually supplies the code being executed.
- the current **sponsor**. An unprotected reference to the process whose authority will get attached to new process objects, and who will be charged for them. Processes own themselves, so for new process creation, it doesn't make any difference who the current owner is. The engine uses the authority (and permit, see below) of the current sponsor, usually the current process, to determine responsibility for new agents and places.
- the **client.** This is the object whose code requested the current operation. The client's owner might be yet another process.

Why are so many different identities needed? It depends on the particular situation that might arise. Consider the following scenario:



Here, hazel and bigwig are both agents that have entered theWarren. Hazel gets a reference to bigwig's Trojan object and naively requests the foo operation on it. However, Trojan, now in control on hazel's thread, tries to get service from theWarren attempting to masquerade as hazel. Bigwig might get away with it if theCarrots did access control checks on the basis of process alone.

The following code fragment provides an example of making an access check:

```
MyAccessViolation: class(Exception) = ();
MySafeClass : class ( Object ) = (
   public
      mySafeOp:op ( theArg: Object )  = {
        theSponsor:OctetString = sponsor.name.authority;
        theClass:Class = client.class;
        if (friends.examine(theSponsor) == nil) ||
           (okClasses.examine(theClass) == nil) ) {
              throw MyAccessViolation() ;
           };
        myRealOp ( theArg );  // could check the arg, too...
        };
   private
      friends: List[OctetString,Equal] = ... ;
      okClasses: List[Class,Equal] = ... ;
      myRealOp:op (argument: Object) = { ... };
);
```

Here, the program maintains a simple private list of references to authorized classes along with a list of authorized authorities to implement an access policy that checks the class and authority of both the requester and the current sponsor (process). It's not expecting a call from a Trojan object .

Had `foo` been a sponsored operation, "sponsorship" would have passed by the call, so that mySafeOp would sense the sponsor as bigwig. Hazel's class could have been written knowing that `foo` has to be sponsored, owing to the fact that Trojan was sub-classed from a class where `foo` was sponsored.

Note that, since Telescript always provides unprotected references to the requester, process, place, and so forth, it is important for agents and places to always make the appropriate access control checks.

Also, attribute features are not simply "passive," but are always implemented using getter and setter operations. Normally, most class definitions simply use the default getters and setters. However, unless sealed, nothing prevents overriding, for example, to perform access control checks. Unfortunately, an untrusted sub-class could put malicious code in there, too.

With regard to installing bogus classes, the Telescript engine won't admit an agent carrying a class that has the same name as one that's already in the engine, unless it's the same class. In other words, within an engine, class names must be unique. Without going into excessive detail, new classes (the ones normally used by the engine during class search) are "installed" in what is known as the process's public and private *packages* . These private process attributes are collections of "write once" objects, so that only a process can add new class packages to itself . The engine searches for a class either in the private packages of the process instantiating the new object, or in the public packages of that process and its enclosing places. Usually, class definitions are found in the engine place.

Permits

Permits provide a mechanism for limiting resource consumption and controlling the capabilities of executing code. A permit is an object (of the built-in class Permit) whose attributes include, among others:

- **age**: maximum age in seconds
- **extent**: maximum size in octets
- **priority**: maximum priority
- **canCreate**: true if new processes can be created
- **canGo**: true if the affected code can request the go operation
- **canGrant**: true if the permit of other processes can be "increased"
- **canDeny**: true if the permit of other processes can be "decreased"

Permits are reminiscent of the quotas and privilege vectors of VMS. Consult the Telescript Reference Manual for the full Permit description.

Telescript uses four kinds of permits:

- **native** permits are assigned by the process creator.
- **local** permits can be imposed by a place on an entering agent or on a process created in that place. Local permits only apply in that place.
- **regional** permits are like local permits but imposed by the engine place. Regional permits only apply within a particular engine or set of engines comprising a region.
- **temporary** permits, which are imposed on a block of code using the Telescript `restrict` statement.

The actual age and priority of a process are sealed, read-only public attributes of class Process. The native, local, and regional permits of a process are also sealed public attributes.

The *effective* permit is computed by the engine by the logical intersection of all applicable permits in a particular situation. For example, in the case of a Boolean-valued capability, the effective permit value is true only if that capability is true in all permits intersected. For an integer-valued attribute, the effective permit value is the minimum specified in any of the permits intersected.

Temporary permits are useful for bounding the damage that suspect code might do, as the following illustrates:

```
paranoid := Permit();       // create a new instance
```

```
paranoid.canCharge = false; // No capabilities
paranoid.canCreate = false;
paranoid.canDeny = false;
paranoid.canGo = false;
paranoid.canGrant = false;
paranoid.canRestart = false;
paranoid.canSend = false;

paranoid.age = *.age + 2;   // give it two seconds
paranoid.extent = *.size + 1000;
paranoid.charges = *.charges + 1000;

try {
   restrict paranoid {              // restrict statement
      yourObject.yourSuspectCall(); // the suspect call
      };
   catch failed: PermitViolated {
      ...
      };
   catch ...    // other catch clauses
```

If the suspect call fails because of a permit violation, the engine raises a Permit Violated exception that can be caught by the calling procedure. If any of the native, local, or regional permits is violated, the engine terminates the process with a Permit Exhausted exception.

Note that is `yourSuspectCall` were sponsored, it would be executed under yourObject's permit. Not only would yourObject's owner be charged for resources consumed, but the `paranoid` permit would not be in effect during that call, nor be needed.

Security-Oriented Mix-in Classes

Telescript provides some useful Mix-in classes that associate security-relevant attributes with objects of those classes, where the associated functionality is enforced by the engine. These include:

- **Unmoved**. An agent cannot take such an object along with it when it does a `go`. Places, for example, are Unmoved.
- **Uncopied**. An attempt to make a copy of such an object returns a reference to the original object rather than creating a copy.
- **Copyrighted**. This class is provided as a language extension rather than part of the language. Nonetheless, it is built into engines. An attempt to instantiate such an object will fail during initialization if it is not properly authorized by a suitable Copyright Enforcer object.
- **Protected**. Such an object cannot be modified once created, and any reference to such an object is like having a protected reference, except that ownership can be transferred. Packages are Protected.

The engine searches for a Copyright Enforcer using a similar algorithm to that used during class search. The Copyrighted mix-in can specify a Telename indicating a package or set of packages, by supplying just an authority, which should contain the Copyright Enforcer object. It also can specify a key. If no Telename is given, the engine does the usual search, starting with the current sponsor, trying the first one it finds that matches the key, or throwing Class Unavailable if none is found.

Entering and Meeting

Before an agent can enter or be created in a place, the engine gives that place the opportunity to deny occupancy. It does this by requesting the place's `entering` operation with arguments that include the identity of the agent, its class, and its permit. Note that this operation is executed on a new thread of

control. A place can choose to deny the agent entry, to impose a more restrictive local permit on it, or do simply do nothing. Once the agent enters, the place is provided an (unprotected) reference to the agent, and vice-versa, by the engine.

Here's an illustration:

```
MyParanoidPlace : class ( Place ) = (
   public
            ...                              // initialize, live, etc.
      entering: sponsored op( who: protected Telename;
                              theClass: Class;
                              permit: unprotected Permit;
                                // intersection of native, regional, requested
                              ticket: protected Ticket | Nil) Object
           throws DestinationUnknown, OccupancyDenied =
      {
         *.myRestrictEntry(who.copy(), theClass); // local access check
                                                // throws OccupancyDenied for us
      // we'll let this agent in, but with restrictions...
        try {    // could fail in subtle ways
         permit.canCreate = false;   // can't make new agents in here.
         permit.canSend = false;     // we will let it go out eventually
         permit.canRestart = false;
         permit.canCharge = false;   // nada...
         permit.canDeny = false;
         permit.canGrant = false;
         permit.age = 120;           // live free for two minutes or die!
         if ((permit.priority == nil)||(permit.priority > *.priority)) {
                permit.priority = *.priority - 5 ; // make sure lower priority
         };
         permit.authenticity = *.myAuthPolicy(who, class, permit);
                                     // set how much we trust this guy
         // don't care about limiting size or charges
         return(nil);
         }
        catch Exception {
            throw OccupancyDenied;   // that's it, no fooling around.
        };
      };
   private
      myRestrictEntry: op(name: Telename; theClass: Class)
          throws OccupancyDenied = { ... };
           ...                    // other stuff here...
);
```

`Entering` is a sponsored operation that is called by the engine. Inside the `entering` operation, `client` evaluates to nil.

Analogous to `entering`, the engine mediates a meeting protocol between two agents (of classes that have the built-in mix-in Meeting Agent) that gives them the opportunity to accept or reject a meeting, according to their own policy. Agents in a place can't discover the other occupants on their own, but can find out from a cooperating place. Meeting agents meet in places whose classes have the built-in mix-in Meeting Place. Consult the language reference manual for specifics.

An example of a restricted place, normally used in production Telescript regions, is a place called *purgatory*. The engine place can be programmed to route incoming agents whose trips fail to purgtory. On entering, agents are given a very restricted permit with a relatively short time to live. The intent is to enable them to catch a trip exception and recover on their own.

System Safety and Security

Telescript provides built-in classes that allow programs to access underlying system resources. In contrast to the restricted environment approach previously mentioned, these classes are visible to all processes. Instead, they are access controlled using the Copyrighted, Uncopied, and Unmoved mix-ins.

The engine requires that there be an appropriate Copyright Enforcer for these classes in a region process on the package search path. Usually, the Copyright Enforcer is located in the engine place. This prevents an arbitrary agent from merely supplying its own Copyright Enforcer. Each of these Copyright Enforcers is looked up under a different key, so that different policies can be imposed by the region for each class.

Unauthorized processes, or processes that are not running under the region's authority, cannot create instances of the following classes:

- **File.** A File object can create a handle to any file that the engine can access on the underlying operating system.
- **External Handle**. An External Handle object can open a TCP/IP port on the underlying operating system.
- **Control Manager.** A Control Manager object can be used to perform a number of management and control operations on an engine. For example, a Control Manager can be used to change attributes of processes, such as their authority, or to halt the engine.

In general, applications that use files or the network should be designed so that direct access is only given to known well-behaved processes, which use them in turn to provide services to untrusted processes.

Network Security

Telescript is a distributed processing environment that supports agent mobility across multiple hosts. Telescript networks provide:

- a privileged region authority for management and administration
- the ability to set region-based administrative policies
- inter-region secure channels

Privileged Region Authority

The region authority is privileged as is the root account on a UNIX system. However, an engine should never be run under the root account. The engine place runs under the region authority.

Processes running as region can, for example, modify the regional permit of any process, or instantiate a Control Manager. The default Copyright Enforcers in the engine place usually authorize only the region to create system sensitive objects.

Administrative Policy in the Engine Place

A Telescript engine in general will run with a customized engine place that is a sub-class of the built-in Engine Place class and instantiated by a "boot script." The main reason for doing this is to establish and enforce regional policies. Regional policies can provide, for example:

- controls on entering the region. For example, the engine place might deny entry to an agent of an unauthorized class.
- policies for granting regional permits. For example, based on the authority of the entering agent, or by the amount of trust in the authentication of that agent.
- copyright enforcers
- management and auditing. For example, using a Control Manager.

Secure Channels

Telescript engines ideally would always transfer encoded agents over secure channels between regions. A secure channel is an authenticated "opaque pipe," usually (but not always) created using cryptographic techniques. An engine authenticates the authority of the remote region, providing a basis for its policy of how much to trust the foreign region to properly authenticate the authority of the agents it sends.

A *security regime* specifies a set of security services that can be negotiated over a secure channel. Currently, engines only have security regimes for secure communication with Magic Cap communicators over the ATT PersonaLink service. More security regimes can be defined as they are needed.

The simplest security regime is the Identification regime, which only exchanges the authority information without any other protection. The Authentication regime, on the other hand, requires strong mutual authentication using RSA public key encryption [PKCS], session key negotiation using the Diffie-Hellman algorithm with perfect forward security [DIFFIE], and session encryption using RC4. Export quality encryption is used (e.g., 512-bit RSA keys, 40-bit RC4 keys).

An asymmetric protocol is used in the Authentication regime that permits a device to authenticate using an RSA public key pair that it generated during its initialization. The region uses a public key certificate based approach used to authenticate itself to a device. The certificate, in essence, associates the region's authority with its public key, and must be digitally signed (using the RSA algorithm) by one of the General Magic keys. The corresponding public keys are included in all devices.

Magic Cap devices register their key (and other information, for example, a credit card number) with the ATT PersonaLink service using the Registration security regime. Registration uses session encryption, but doesn't demand a pre-existing relationship with the device as the Authentication regime would. Region policies control which agents are allowed in during registration, for example.

Future Security Directions

Telescript security is evolving. There are a number of known limitations with the present version. For a list of language-specific safety limitations, refer to the Annex of the Telescript Language Reference Manual [WH95c]. Some issues related to the current implementation are mentioned in the Telescript Programming Guide [GUIDE].

Future versions based on work currently in progress may incorporate:

- computed class names. Currently, Telescript uses octet strings derived from programmer-supplied identifiers to name classes. In the future, a more rigorous naming scheme based on cryptographic hash functions will provide unique class names that are bound together with the actual class definition. This will permit flexible policies on imported classes, for example.
- digitally signatures on objects. This will permit end-to-end security mechanisms to supplant the current region-based policies. Note that while constant objects such as classes are good candidates for being signed in this way, agents and other objects with dynamically changing state are not .
- Internet security regimes. For example, the use of SSL [SSLREF] or some other standard with certificate-based region authentication. This is part of the plan for inter-region secure channels over TCP/IP, and would accommodate more flexible security regimes, for example, stronger encryption algorithms for domestic use.

One final caveat. Although every effort has been made to address any security-related bugs that have come up, General Magic makes no claims that the current engine is "bulletproof." In other words, the current engine implementation has not yet been through a rigorous security verification or certification program.

References

[WH95a] White, J.E., *Telescript Technology: An Introduction to the Language*, General Magic, Inc., Sunnyvale, CA., October, 1995 http://www.genmagic.com/WhitePapers

[WH95b] White, J.E., *Mobile Agents*, in *Software Agents,* Bradshaw, J. (ed.), AAAI Press and MIT Press, Menlo Park, CA.,1996

[WH95c] White, J.E., *Telescript Language Reference Manual*, "General Magic, Inc, Sunnyvale, CA., October, 1995 http://www.genmagic.com

[STAN95] Genesereth, M.R., and Ketchpel, S.P., "Software Agents," available at URL http://logic.stanford.edu/sharing/papers/agents.ps

[IBM95a] Harrison, C.G., Chess, D.M., and Kershenbaum, A., "Mobile Agents, Are they a good idea?," Research Report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, March, 1995 http://www.research.ibm.com/xw-d953-mobag-ps

[WH95d] White, J.E., High Telescript, General Magic, Inc., Sunnyvale, CA., September, 1995

[DAG95] Dagenais, M.R., Building Distributed OO Applications: Modula-3 Objects at Work, Draft Version, March, 1995. http://www.vlsi.polymtl.ca/dagenais/if515/main/main.html

[JAVA95a] The Java Language Specification, Release 1.0 Alpha3, Sun Microsystems, Mountain View, CA., May, 1995 http://www.javasoft.com/whitePaper/javawhitepaper_1.html

[JAVA95b] "HotJava(tm): The Security Story, " http://www.javasoft.com/1.0alpha3/security/security.html

[SAFE-TCL] Ousterhout, J.K., "Scripts and Agents, the New Software High Ground," Invited Talk, Winter, 1995, USENIX Conference, New Orleans, LA. http://www.smli.com/research/tcl

[BOR93] Borenstein, N.S., "EMail with a Mind of Its Own: The Safe-Tcl Language for Enabled Mail," available from ftp://ftp.fv.com/pub/code/other/safe-tcl.tar

[OBLIQUE] Cardelli, L., "A Language with Distributed Scope," Computing Systems, vol 8, no 1, pp. 27-59, MIT Press, 1995 http://www.research.digital.com/SRC/Obliq/Obliq.html

[IBM95b] IBM Corporation, "Things that go bump in the Net," http://www.research.ibm.com/xw-d953-bump

[TCSEC] Department of Defense Trusted Systems Evaluation Criteria, " DoD 5200.21 STD, National Computer Security Center, December, 1985

[SALZ75] Saltzer, M., and Schroeder, M., "The Protection of Information in Computer Systems," Proceedings of the IEEE, vol 63, no 9, pp. 1278-1308, September, 1975

[LAMP71] Lampson, B. W., "Protection," Operating Systems Review, vol 8, no. 1, pp. 18-24, January, 1974

[KAR87] Karger, P. A., "Limiting the Damage Potential of Discretionary Trojan Horses," Proceedings of the IEEE Symposium on Security and Privacy, pp. 32-37, Oakland, CA., April, 1987

[PKCS] The PKCS Public-Key Cryptography Standards, RSA Data Security, Inc., 1993. http://www.rsa.com/pub/pkcs

[DIFFIE] Diffie, W., Van Oorschot, P., and Wiener, M., "Authentication and Authenticated Key Exchanges," in Designs, Codes and Cryptography, volume 2, Kluwer Academic Publishers, pp. 107-125, 1992

[SSLREF] Hickman, K., and El Gamal, T., "The SSL Protocol," draft-hickman-netscape-ssl-01.txt, Netscape Corporation, Mountain View, CA., June, 1995 http://home.netscape.com/newsref/std/SSL.html

[GUIDE] Edighoffer, G., *Telescript Programming Guide* "General Magic, Inc, Sunnyvale, CA., October, 1995 http://www.genmagic.com