Architectures for secure portable executable content

Stefanos Gritzalis
George Aggelis and
Diomidis Spinellis

The authors

Stefanos Gritzalis is Assistant Professor in the Department of Informatics at the Technological Educational Institute (TEI)of Athens, Aegaleo, Greece, E-mail: sgritz@acm.org and also at the Department of Information and Communication Systems, University of the Aegean Research Unit, Athens, Greece, E-mail: sgritz@aegean.gr
George Aggelis is a Postgraduate Student in Information Systems in the Department of Informatics at Athens University of Economics and Business (AUEB), Athens, Greece. E-mail: gangel@hermes.aueb.gr
Diomidis Spinellis is Visiting Professor in the Department of Information and Communication Systems, University of the Aegean Research Unit, Athens, Greece. E-mail: dspin@aegean.gr

Keywords

Computer architectures, Computer programming, Distributed data processing, Internet, Security

Abstract

The Java programming language supports the concept of downloadable executable content; a key technology in a wide range of emerging applications including collaborative systems, electronic commerce, and Web information services. Java enables the execution of a program, on almost any modern computer regardless of hardware configuration and operating system. Safe-Tcl was proposed as an executable content type of MIME and thus as the standard language for executable contents within e-mail messages. However, the ability to download, integrate, and execute code from a remote computer, provided by both Java and Safe-Tcl, introduces serious security risks since it enables a malicious remote program to obtain unauthorised access to the downloading system's resources. In this paper, the two proposed security models are described in detail and the efficiency and flexibility of current implementations are evaluated in a comparative manner. Finally, upcoming extensions are discussed.

Internet Research: Electronic Networking Applications and Policy
Volume 9 · Number 1 · 1999 · pp. 16–24
© MCB University Press · ISSN 1066-2243

Introduction

Downloadable executable content (or mobile code) are based on the idea of transmitting data that are actually codes to be executed. Use of the World Wide Web has exploded over the past few years, and this growth of popularity has had a significant impact on the number of Web-based document authors using mobile code. An increasing number of authors are including Java applets or other forms of mobile code in their pages, as well as Safe-Tcl scripts (tclets) as executable contents within their e-mail messages to be executed on arrival at the reader's machine. Although the majority of these are not intended to be hostile, some may be, and more may simply be prone to errors leading to potential dangers for the unknown user.

The most important capability provided by the Java platform and not found in most traditional programming languages is executable program portability, which means that Java programs written on one type of hardware or operating system can be executed without recompilation on almost any other type of computer. However, security concerns become especially important in such an environment since the presence of downloaded executable content makes the local computer vulnerable to a potential attack from the possibly untrusted source of the executable (Thompson, 1984). Java proponents promise that through the new security architecture delivered in the upcoming Java Development Kit (JDK 1.2) Java provides a secure environment for downloadable executable content that under specific circumstances can make use of a system's resources without compromising their availability and integrity (McGraw and Felten, 1996).

Not surprisingly, Java is not the only platform providing support for portable executable programs (Thorn, 1997). The Safe-Tcl language, which is based on Tcl (a procedural, high-level, scripting language designed to be simple, portable, easily embeddable, and extensible), attempts to provide for "enabled mail" which would allow users to send e-mail with embedded Safe-Tcl executable programs (scripts). Safe-Tcl is designed as a language that satisfies strong security and portability constraints. It deals with potential security problems by restricting the behaviour of programs so that they have fewer capabilities than the users who invoked them. The Safe-Tcl security model makes it possible to implement highly restrictive security policies for scripts of unknown origin as well as less restrictive policies for scripts whose authors are known and trusted.

This paper evaluates the security features offered by the Java and Safe-Tcl programming languages and describes the basic mechanisms of each of the proposed security models. We present and compare the current implementations as well as upcoming extensions of the two security models, and evaluate their efficiency and flexibility. Although Microsoft's Active-X technology also supports downloadable executable content and is based on an interesting security architecture it is not examined in this article because, in its current implementation, it is operating system and hardware specific.

The Java security model

Java was created to enable the development of programs in a heterogeneous network-wide environment. It allows Java-compatible Web browsers to download code fragments dynamically and then execute those code fragments on the local machine. Executable portability, meaning that a Java program (or applet) is portable not only in source code but also in compiled binary code, was therefore one of the major design goals of Java. The Java Virtual Machine (JVM), a system that simulates an abstract machine, is the part of the Java-compatible Web browser that provides this portability layer (Sun Microsystems, 1997a;1997b). The JVM architecture defines an instruction set, a register set, a stack, a garbage-collected heap, and a memory area. This architecture allows a single executable to run unmodified on many different systems. To achieve this, the Java compiler compiles Java code to an architecture independent object file format containing JVM code (or bytecodes), which is then interpreted by a processor-specific JVM implementation or compiled on the fly into the machine code of the particular processor.

The aim of the Java security model is to protect users from malicious applets originating from untrusted sources across a network. Java provides a customisable "sandbox", which is a dedicated area of the Web browser within which the actions of the applet are restricted. Within its sandbox the applet may do anything but access the user's files, network connections, and other sensitive resources. The basic idea of the sandbox model is that programs loaded from the local file system are executed with full access to vital system resources, whereas executable content downloaded from a remote source is considered untrusted, and can therefore access only the limited resources provided inside the sandbox. The first release of the Java Development Kit (JDK 1.0) was based on the above described mechanism of the sandbox model.

Overall security is enhanced through a number of mechanisms. First, the language itself was designed with security in mind so that every program that conforms to the language specification, automatically obeys basic low level security restrictions (Yellin, 1995). The most important features that make the Java language attractive as an environment to write safe code are the lack of pointer arithmetic, mandatory array bounds-check at runtime, the prohibition of casts of primitive types into reference types, and the automatic garbage collection.

Security is provided by the JVM during the loading and verification of the JVM code. Applets are loaded from the network by the applet Class Loader which receives the bytecode instruction stream and converts it into internal data structures that represent the applet's classes. The class loader, apart from fetching an applet's executable content from the network, also enforces the name space hierarchy. By maintaining a separate name space for trusted code which was loaded from the local disk, the Class Loader prevents untrusted applets from gaining access to more privileged, trusted parts of the system.

The bytecode verifier is invoked by the Class Loader, and before the execution of the newly imported applet, ensures that the applet conforms to the specifications of the Java language, and that there are no violations of name space restrictions or of memory accesses. The bytecode verifier, along with the properties of the JVM, guarantee language safety at runtime. The third component of the Java security model is the Security Manager, which restricts the way in which an applet can use visible interfaces by performing run-time checks on dangerous methods such as those for file or network access. The JVM consults the Security Manager whenever such a dangerous operation is about to be attempted. The Security Manager then has a chance to veto the operation by generating a security exception. Actions of a piece of untrusted code are therefore restricted to the minimum thanks to these checks performed in advance by the Security Manager.

The concept of digital signatures was adopted by the Java security model with the second release of the Java Development Kit (JDK 1.1). Until then, downloaded executable content was considered to be untrusted, unless it was downloaded from the file system of the local disk. Consequently, all applets obtained from the open network could access only the limited resources provided inside the sandbox. It was not until in JDK 1.1 that JVM became capable to distinguish between untrusted and trusted remote executable code. The concept of correctly digitally "signed applet" was then introduced that allowed a remote applet to be treated as if it is trusted local code. The Java Archive (JAR) file format, which consists of the widely used ZIP format plus some meta-data files, is used to deliver the signed applets along with their signatures. Unsigned applets are treated as untrusted applets so their execution is still encapsulated by the sandbox.

With the upcoming JDK 1.2, Java will move away from the sharp distinction between applets that run in a browser's "sandbox" and are thus denied all access to the resources of the host operating system versus applications that have unrestricted access (Gong *et al.*, 1997; Lindhorn and Yellin, 1997). The security enhancements that are introduced by the JDK 1.2, including a simpler policy configuration, an extensible access control structure, and an extension of the security checks performed over all forms of Java programs (applications and applets), were designed to give developers more control over their applications. The security policy introduced consists of a mapping between properties of the running code (the URL of the code and the code signature) and a set of permissions granted to the code. The permissions a piece of code is entitled to are computed as the sum of permissions each signature that the piece of code carries. To accomplish this goal a simple configuration language for statement of policy constraints has been defined and can be used. Before a controlled resource is to be accessed, an access control decision is made, based on the permissions of the executing code.

JDK 1.2 also introduces the concept of a protection domain. This is defined to be a set of all the objects that correspond to a principal; where a principal is an entity in the computer system to which authorisations are granted (Gong and Scemers, 1998). In JDK 1.2 permissions may no longer be granted to classes but to protection domains, with every class belonging to one domain only. If a thread transverses more than one domain while executing, the permissions it is entitled to are computed based on the principle of least privilege. In case there is a need for communication between different domains it may be performed either indirectly through system code, or directly, provided that all participating domains allow it.

JDK 1.2 introduces a new class called AccessController which makes it easier for the code to learn the status of all its callers and perform access controls. The only thing the programmer has to do is call the checkPermission method of this class, having the system itself perform the access control. For backward compatibility reasons, the SecurityManager usage is still allowed. In order to distinguish between system classes and remote classes and impose the proper security policy, the ClasssLoader class has been replaced by the SecureClassLoader. Furthermore, a new class called Java.security.Main has been introduced in order to impose the security policy to locally installed applications.

The Safe-Tcl security model

Safe-Tcl is an extension of the Tool Command Language (Tcl) (Ousterhout, 1994). Developed by Marshall Rose and Nathaniel Borenstein, Safe-Tcl is a secure version of Tcl used for executing scripts on the Internet. Tcl is a scripting language which is typically used to glue together building blocks written in system programming languages like C, C++, and Java. It is easy to embed Tcl into a legacy program to add scripting features or a GUI interface.

Safe-Tcl's syntax is identical to the syntax of Tcl, since the former is in essence an extended subset of the latter. Specifically, the features in Tcl considered to endanger the local system have been removed, while several new features considered as trusted under all security policies have been added.

Safe-Tcl, like several other languages including Java, deals with address-space protection by doing without C-style pointers and by enforcing bounds checking in array references. Moreover, storage management is handled automatically by the Tcl interpreter. An interpreter encapsulates completely the execution of a Tcl script. Consequently, the facilities available to a Tcl script are determined by the set of commands that its interpreter contains.

Safe-Tcl deals with security by controlling the execution of Tcl scripts (tclets) using a padded cell approach. This control is accomplished using safe interpreters, which restrict the commands available to an applet, and aliases, which allow controlled access to unsafe commands. An application such as a Web browser can have more than one Tcl interpreter. A script can only invoke the commands and use the variables available in the interpreter it runs on, each interpreter having its own set of commands and variables. The scripts that are considered to be trusted, either because they are residing on the local host, or because they originate from a trusted source, are executed in a Tcl interpreter called the "master" (or "trusted") interpreter, which contains the full set of all Tcl's commands. Scripts that are thought to be untrusted, such as the ones on a Web page downloaded from an unknown host across the network, are executed in a new separate Tcl interpreter created by the application. This interpreter is called the "slave", or "untrusted" interpreter, or padded cell. All commands that could result in endangering the system's security, such as those for reading and writing files, are made inaccessible to those scripts (they are removed from the slave interpreter), and thus only a limited set of safe commands (the safe base) is available. The above mechanism is also called the dual-interpreter mechanism.

The padded-cell approach is similar to the kernel and user space distinction in modern operating systems. Programs running under the slave interpreter are similar to user space processes in an operating system which cannot access the disk directly. The master interpreter, having unrestricted power to do anything, is much like the operating system kernel; code running under its privileges has to be trustworthy.

The basic mechanism provided by Safe-Tcl which allows a slave interpreter to make requests from its master is called aliases. Through this mechanism, Safe-Tcl provides restricted access to features that are essentially unsafe; untrusted code is allowed to communicate outside its padded cell in a carefully controlled way. As an example, a slave interpreter wanting to access a limited number of files within a single directory may be provided with an alias (or "safe call") enabling it to make this access in a controlled way. An alias is, in fact, the association between a command in the untrusted interpreter and another in the trusted interpreter. Whenever the former command is invoked by a script running within the untrusted interpreter, the latter is the one that is actually executed instead. The master interpreter has complete control over the safe calls in a slave interpreter, and can provide different sets of safe calls depending on what it knows about the script. As an example, if one script is more trusted than another (originating from a more trusted source), then it may be given a more comprehensive set of safe calls than the untrusted one. Furthermore, it is the master interpreter's responsibility to create and delete aliases as well as any other variable of the slave interpreter, and to define the source and destination of each safe call (i.e. which command of the slave is going to be substituted, and which one from the master interpreter will be invoked instead).

The commands that are made inaccessible to the safe interpreter are not actually removed from it. Instead, they are hidden, making it impossible for the untrusted interpreter to invoke them. As the master interpreter has complete control over the instruction set of the slave interpreter it is able to invoke its hidden commands. This architecture ensures that restricted commands are executed within the correct environment, i.e. that of the untrusted interpreter. The fact that the Tcl code needed to implement the security restrictions usually consists of just a few lines makes it easy for security analysts to test the code and fix any bugs or "holes" they may find.

One of the strengths of Safe-Tcl is that it permits a variety of security policies. A security policy in Safe-Tcl consists of the commands available in safe interpreters using the policy, including both the safe base and any aliases. The simplest security policy consists of the safe base with no aliases at all. If the Tcl script is trusted, it might be given a security policy that restores the full set of Tcl commands. At the other extreme, highly sensitive environments might use security policies that hide some of the commands of the safe base.

Java security extensions and implementations

The new security model proposed by the JDK 1.2 provides fine-grained access control, easily configurable security policies, a more extensible but simplified access control structure, and enforced security checks to any kind of Java software. The capability of fine-grained access control existed in previous releases of JDK, but the application programmer could not use it unless he/she did substantial programming (mainly by subclassing and configuring the SecurityManager and ClassLoader classes which implement security restrictions and class loading over the network). The problem with such code is that it is extremely security-sensitive and requires sophisticated skills and in-depth knowledge of computer security. The new mechanism makes this process far simpler and thereby safer.

The capability of a configurable security policy existed in previous versions of JDK, but it was not as easy to use as it is under JDK 1.2. Being aware that writing security code is not straightforward, the JDK 1.2 implementers allow application builders and users to configure security policies without having to develop specific programs.

Before JDK 1.2, in order to create and use a new access permission one had to add a new check method to the subclassed Security-Manager class. The new architecture introduces an easily extensible access control structure allowing customised, typed permissions (each representing an access to a system resource) and the automatic managing of all permissions of the correct type. With the new security mechanism, in most cases, no new method in the SecurityManager needs to be defined.

Until now, one of the most common security concerns about the Java security model had to do with the concept that all local code was considered to be trusted. In the new security model, local code (e.g. non-system-code, application packages installed on the local file system) is subjected to the same security control as applets, although it is still possible to declare the policy on local code to be less restrictive, thus enabling such code to be trusted. The same extensions of security checks are now applied to all Java programs including applications as well as applets.

A protection domain is a vital component of the new model forming the basis for making access control decisions. It serves as a convenient mechanism for grouping and isolation between units of protection. There are two distinct categories of protection domains:
(1) the system domain; and
(2) the application domain.

Only system domains can access all the protected external resources (file system, network connections, and input/output functions). It is through the system's security policy that the user or the system administrator specifies which new protection domains should be created and what permissions should be granted to them. In JDK 1.2 permissions are granted to protection domains, and classes and objects are granted the permissions of the domain to which they belong. The Java runtime maintains such a mapping from code (classes and objects) to their protection domains and to their permissions. It is important to ensure that at any time the application domain does not gain additional permissions while calling the system domain (or any other domain that is granted more permissions than the calling one). When access to a critical system resource is requested at runtime, a special AccessController is directly or indirectly invoked by the resource-managing code that evaluates the request and decides whether the request should be served or thrown away. Evaluating the request means examining the call history and the permissions granted to the relevant protection domains.

From the above description it becomes apparent that there is a clear intention to move from a black-and-white security model to a new configurable one that promises to give developers more control over their applications. Developers and network administrators will be able to assign varying degrees of access to any kind of Java software based on code signatures. Consequently, Java will move away from the sharp distinction between completely restricted applets and applications free to do whatever they want.

To accomplish this goal, JDK 1.2 features additional security tools, providing support for digital signatures to ensure authentication and integrity, message digests, key management, certificate management, and access control (Sun Microsystems, 1997c). The new keytool and jarsigner tools replace javakey, which was used in JDK 1.1 applications for key and certificate generation and management. The new tools provide more features than javakey, including the ability not only to generate digital signatures but also to verify them. The identity database that javakey created and managed is replaced by the new keystore architecture. A keystore is a protected database that holds keys and certificates for an enterprise. For backward compatibility reasons, it is possible to use the keytool command to import the information from an identity database into a keystore.

Keytool can be used to create public/private key pairs and self-signed X.509v1 certificates used to digitally sign Java applications and applets and to manage keystores. Access to a keystore is guarded by a password defined at the time the keystore is created by the person who creates the keystore and altered only when providing the current password. In addition, each private key kept in a keystore can be guarded by its own password.

The jar tool, which was also available in JDK 1.1, is used to create JAR files. To sign an applet the producer first creates a JAR file and a digital signature based on the contents of the JAR. The jarsigner tool, used to sign JAR files or to verify signatures on signed JAR files, accesses the keystore when it needs to find the private key to use when signing a JAR file. Since accesses to the keystore and to private keys kept there are guarded by passwords, only users knowing the passwords can access a key and use it to sign a JAR file. One other new tool available in JDK 1.2, called policytool, is a graphical user interface used to create and modify the external policy configuration files that define the system's Java security policy.

The Java cryptography architecture refers to the framework for accessing and developing cryptographic functionality for the Java platform. The official Java implementation by Sun includes an implementation of the NIST DSA algorithm, the MD5, and SHA message digest algorithms (Schneier, 1996). In addition, since the ability to encrypt data before being transferred is critical, APIs for data encryption are contained in a Java cryptography extension as an additional package to JDK 1.2.

An important enhancement to the proposed security model would be a configurable audit system allowing system administrators to study the circumstances under which security breaches occurred.

The methods and tools needed to define a system-wide policy have not been developed yet. It is up to the user to define the policy used against a certain piece of Java code based on its digital signatures. The end-user should be able to see and define only part of the policy enforced on a piece of Java code. In an organisation, an enterprise-wide policy should be designed by the network administrators. Within the environment of such an organisation, a central repository could be used to store the security policy. The latter should be formed by three entities:

(1) an organisation-wide policy made by the network administrator;
(2) a local policy by the administrators of the local networks of the organisation; and
(3) the end user.

The policy each of these entities will define, will be based on the signer of Java code and on the certificate authority which certifies the signer's signature.

Safe-Tcl security extensions and implementations

The intent of the Safe-Tcl language design is that it should be essentially harmless to evaluate a Safe-Tcl program that comes from an unknown or hostile sender. In Safe-Tcl, an untrusted script is isolated in its interpreter context, much like a Java applet is isolated in its sandbox, having the capability to invoke a few extra commands that are carefully implemented by another interpreter to ensure safety. The set of these extra commands, the exposed aliases, and their implementation make up a security policy in Safe-Tcl. Rather than adopting a single security policy, Safe-Tcl allows different security policies for different applets, resulting in added flexibility in function and levels of trust, but at the same time increasing the possibility for additional loopholes from the additional complexity. Added flexibility means that an application can choose exactly how much trust to place in the applet by choosing from a variety of security policies. Completely untrusted code might be executed in the safe base without the added functionality of any aliases at all, whereas fully trusted applets could be executed with unrestricted access.

The presence of multiple security policies with their inevitable flaws and interactions introduces additional complexity which can be a source of loopholes that may be both hard to predict and hard to prevent. However, careful design and implementation of such a system can often reduce security risks to an acceptable level.

The Safe-Tcl environment is easily extensible. However, this should be done with great caution, since the introduction of a new Safe-Tcl command can have serious security implications. Whenever a new command is added, the author should consider whether hostile parties could use this command to cause any harm. In order to make an extension to the Safe-Tcl, one writes a procedure in full Tcl, to be interpreted by the trusted Tcl interpreter. This particular command may then become available in the untrusted interpreter using the declareharmless primitive. Furthermore, expressions may be evaluated in the untrusted interpreter by using the restrictedeval primitive.

Writing a security policy is a complex effort that should not be undertaken lightly. It involves careful design, exhaustive testing, public review and analysis, and continuous debugging. Implementers have to consider what features a security policy should provide while balancing the security risks to which an application using the policy will be exposed. A security policy is a Tcl script or a shared library that is loaded into an unsafe master interpreter. It consists of two parts:

(1) a management part, concerned with installing the policy into safe slaves and cleaning up any associated state when a slave is destroyed; and
(2) a runtime part, concerned with actually implementing the features of the policy.

Safe-Tcl uses a platform-dependent mechanism for obtaining the initial setting for the search path for finding security policies.

Safe-Tcl scripts start executing on the safe base. If they need access to unsafe features, tclets can request to use a named security policy by invoking the package with the policy name required. If the request is denied by the application's trusted interpreter an error is returned. The tclet can catch the error and request to use a different named policy, granting less permissions, until a request is accepted. A tclet can only use one security policy during its lifetime. Once an invocation of the package required to load a security policy succeeds, Safe-Tcl prevents subsequent invocations of a security policy. These restrictions are designed to prevent a tclet from composing security policies either concurrently or sequentially in ways not supported or foreseen by the authors of the policies. Allowing such composition would expose the application to unknown security risks.

The Safe-Tcl extension has been available as part of the Tcl since the Tcl 7.5 release. Forthcoming releases will include standard authentication and encryption mechanisms to prevent denial-of-service attacks. A variety of authentication mechanisms exist for verifying the origin of a mobile code segment, most of which involve encryption of some sort. The same mechanisms can also be used to distribute new security policies. For example, an untrusted tclet may carry an encrypted trusted security policy with it so that when an application executes the tclet it can safely load the security policy even though it does not trust the tclet. Safe-Tcl mechanisms can also be used to prevent denial-of-service attacks. Safe-Tcl's approach to CPU resource usage involves the invocation of a scheduling function in the trusted interpreter once the untrusted interpreter has executed a predefined number of commands which can abort the tclet.

For interactive applets the scheduling function can check to see if the kill key has been pressed, whereas for non-interactive applets the scheduling function can implement an upper limit on CPU usage and on memory allocation.

Conclusions

The popularity of the World Wide Web has had a significant impact on the usage of downloadable executable content attracting considerable interest throughout the Internet community. The two languages that have been described seem to have an edge over other programming languages for portable mobile code in terms of security, since they both make it possible to create environments where applets with different levels of trustworthiness can be executed with an acceptable level of risk. The security features of both languages function independently of trust placed on the imported code. In the anarchic environment of the Internet this is an important advantage over languages, such as O'Caml and Limbo, and technologies, such as Active-X, whose security model depends on object code signed by a, presumably, trusted party.

The flexible security policy introduced in JDK 1.2 provides an integrated method in order to grant specific permissions to applets based on the signatures carried by them. The JDK 1.2, with the new protection mechanisms of security policy, access permissions, protection domains, and access control checking introduces a flexible security model. Java is definitely moving away from the restrictive black-and-white security model of the "sandbox" towards a configurable mechanism which gives developers and administrators fine-grained control over their systems and applications. The Safe-Tcl security model is also introducing an approach whose main benefit is the added flexibility in function and levels of trust: rather than adopting a single security policy, Safe-Tcl allows different security policies for different applets.

As far as security is concerned, Java work has concentrated on providing a granular and complete security framework which can be applied in a variety of contexts, whereas Safe-Tcl has focused on security in the specialised context of e-mail. Moreover, those who vote for Java claim that the security model it offers is more complete than the one proposed by Safe-Tcl, since mechanisms like namespace protection and bytecode compilation go a long way towards safety and efficiency (Weiss *et al.*, 1996).

However, Safe-Tcl has some advantages over Java that simplify the creation of safe environments. One advantage lies in the simplicity of the proposed model which may be considered as a generalisation of the user space-kernel space model that has been used successfully in operating systems for several decades (Gritzalis, 1991). Furthermore, security policies are separated into well-defined modules that do not depend on host applications or on untrusted applets, making it easier to analyse the properties of a security policy and to reuse policies.

Ultimately, a choice between the two languages will depend more on the application domain and the respective inherent features of each language and less on its approach towards security. Java's execution environments featuring aggressive optimisation techniques such as just-in-time compilation and native method interfaces provide Java with a distinct efficiency advantage over Safe-Tcl. In addition, Java's object-orientation, rich class libraries, components framework, and support for concurrency and internationalisation make it the language of choice for large mission-critical or retail-market applications. Its safety-model blends nicely with the requirements of those applications since the configurable security policies applied to respective protection domains correspond to the requirements of enterprise-wide security management. Java's heavyweight language and security architecture provides Safe-Tcl with a clear ecological niche: application extensions, scripting, rapid prototyping, and user interfaces. Safe-Tcl's small footprint, flexibility, and expressiveness are the exact features required in the above named areas. The ability to write in Safe-Tcl specialised security policies in the same language as the application is the appropriate approach for the respective application areas.

It is interesting that the Sunscript group is working on Tcl-Java integration, since Tcl has several properties that nicely complement Java. Two new products were recently released named Jacl and Tcl Blend (Stanton, 1998; SunMicrosystems, 1997d). The first is a new

Java implementation of Tcl 8.0, that can be used to write extensions for Tcl in Java code that will run on UNIX, Windows, and the Macintosh platforms. Taking advantage of the capabilities that the reflection classes in JDK 1.2 offer, TclBlend provides a dynamic interface to Java. The second is a new package for Tcl 8.0 that allows loading and interacting with the JVM (running in a Java only environment). It is very possible that this kind of integration will lead to a security integration as well, since both languages have a built-in security model, the combination of which may be used to have more complete control over mobile code.

References

Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R. (1997), "Going beyond the sandbox: an overview of the new security architecture in the Java Development Kit 1.2", *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, USENIX Association, Monterey, CA, pp. 103-12.

Gong, L. and Scemers, R. (1998), "Implementing protection domains in the Java Development Kit 1.2", *Proceedings of the Symposium on Network and Distributed System Security*, online, http://isoc/NDSS98/

Gritzalis, D. (1991), *Information Systems Security*, GCS Publications, Athens.

Lindhorn, T., and Yellin, F. (1997), *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA and Wokingham.

McGraw, G. and Felten, E. (1996), *Java Security Hostile Applets, Holes and Antidotes*, J. Wiley & Sons Inc., New York, NY.

Ousterhout, J. (1994), *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA and Wokingham.

Schneier, B. (1996), *Applied Cryptography*, J.Wiley & Sons, New York, NY.

Stanton, S. (1998), "TclBlend: Blending Tcl and Java", *Dr. Dobb's Journal*, Vol. 23 No. 2, pp. 50-4.

Sun Microsystems, (1997a), "Frequently asked questions – applet security", online, http://Java.sun.com/sfaq/

Sun Microsystems, (1997b), "Secure computing with Java: now and the future", online, http://Java.sun.com/marketing/collateral/security.html

Sun Microsystems, (1997c), "Security in JDK 1.2", online, http://Java.sun.com/docs/books/tutorial/security 1.2/

Sun Microsystems, (1997d), "Jacl and Tcl blend", online, http://sunscript.sun.com/Java/

Thompson, K. (1984), "Reflections on trusting trust", *Communications of the ACM*, Vol. 27 No. 8, pp. 761-3.

Thorn, T. (1997), "Programming languages for mobile code", *ACM Computing Surveys*, Vol. 29 No. 3, pp. 213-39.

Weiss, M., Johnson, A. and Kiniry, J. (1996), "Security features of Java and HotJava", Open Software Foundation Research Institute, online. htpp://user/cs.tu-berlin.de/~majo/java/security.htm

Yellin, F. (1995), "Low level security in Java", online. http://Java.sun.com/sfaq/verifier.html