The Safe-Tcl Security Model

Jacob Y. Levy
Laurent Demailly
John K. Ousterhout
Brent B. Welch
Sun Microsystems Laboratories
`sunscript-plugin@sunscript.sun.com`

Abstract

Safe-Tcl is a mechanism for controlling the execution of programs written in the Tcl scripting language. It allows untrusted scripts (applets) to be executed while preventing damage to the environment or leakage of private information. Safe-Tcl uses a *padded cell* approach: each applet is isolated in a *safe interpreter* where it cannot interact directly with the rest of the application. The execution environment of an applet is controlled by a trusted script running in a *master interpreter*. Safe-Tcl supports applets using multiple *security policies* within an application. These policies determine what an applet can do, based on the degree to which the applet is trusted. Safe-Tcl separates security management into well-defined phases that are geared towards the party responsible for each aspect of security.

1 Introduction

Security issues arise whenever one person invokes a program written by another person. A program usually executes with all the privileges of the user who invoked it, so the program can read and write the user's files, send electronic mail on behalf of the user, open network connections, and run other programs. If a program is malicious, it can harm the user in many ways, such as by modifying the user's files, leaking sensitive information, or crashing the user's computer.

The traditional "solution" to the security problem has been for people to avoid programs written by people they don't trust. Unfortunately, two trends are making this approach less and less practical. The first trend is an increase in information sharing between people, for example via the World Wide Web; in many cases, the creator of the information is unknown to the recipient of the information. The second trend is a blurring of the distinction between programs and data, so that the act of retrieving and viewing information can cause a program associated with the data to be executed. For example, many systems allow a CD ROM disk to contain a start-up program that is run silently whenever the disk is inserted into a drive. Another example is the Java™ language [1], which, when used in conjunction with web browsers, allows programs to be downloaded and executed locally. When a page containing a Java applet is viewed, the applet is executed locally to provide functionality that is not implemented by the browser itself. As a result of these trends, it is becoming more difficult for users to tell when they are running a program or who wrote the program.

Safe-Tcl makes it safe for people to run applets written in the Tcl scripting language [7][10] without necessarily knowing their origin or trustworthiness. Safe-Tcl avoids potential security problems by restricting the behavior of applets so that they have fewer capabilities than the users who invoke them. The privileges granted to an applet can be adjusted to match the applet's trustworthiness. Applets of unknown origin should not be

---

trusted at all, so they run with very few privileges. If the author of an applet can be authenticated, and if that author is partially or fully trusted, the applet can execute with greater privileges. The mechanisms for authentication and granting of privilege are automated, so applications such as Web browsers can use Safe-Tcl without involving the user.

In Safe-Tcl, untrusted applets are executed in separate environments that are isolated from the application. The features available to an applet are selected by the trusted portions of the application. The implementation of Safe-Tcl is based on two basic facilities: *safe interpreters*, which provide restricted virtual machines for executing applets, and *aliases*, which are used by applets to request services from the trusted portions of the application in a controlled fashion. The alias mechanism makes it possible to provide restricted access to features that are essentially unsafe, such as file or socket access.

Safe interpreters and aliases function much like the kernel space/user space mechanism that has been used for protection in operating systems for several decades. Safe interpreters correspond to the address spaces for user-level programs, and aliases correspond to kernel calls.

The Safe-Tcl security model has three particular strengths:

- Safe-Tcl separates untrusted code from trusted code, with clear and simple boundaries between environments having different security properties.
- Safe-Tcl does not prescribe any particular security policy and supports varying levels of trust. Instead, it provides a mechanism for implementing a variety of security policies and levels of trust. Organizations can implement different policies based on their needs, and a single application can use different security policies for different applets.
- Safe-Tcl gains power and flexibility by using Tcl throughout as the scripting language. All configuration information is expressed as Tcl scripts, and the mechanisms for verifying trust, checking permissions and implementing policies are also expressed in Tcl.

The rest of this paper is organized as follows. Section 2 provides background information on the Tcl scripting language. Section 3 introduces the security issues associated with executing applets. Section 4 describes the basic mechanisms of the Safe-Tcl security model, including safe interpreters, aliases and protected commands. Section 5 discusses security policies and security packages. Section 6 explains how the master interpreter decides whether to let an applet use a given policy. Section 7 describes the overall Safe-Tcl security model. Section 8 discusses how Safe-Tcl deals with denial-of-service and privacy attacks. Section 9 gives the implementation status of Safe-Tcl, and Section 10 compares Safe-Tcl with other security models.

2 Overview of Tcl

Tcl is an interpreted scripting language [7][10]. Its simple syntax is based on *commands* made up of *words*, much like Unix shell programs such as `sh`. For example, the command

```
set a 45
```

contains three words. The first word of each command, such as `set` in the example, selects a C *command procedure* that will carry out the command, and the other words are passed to the command procedure as arguments. The Tcl language syntax consists only of a few simple substitution and quoting rules used to parse commands. Most of the behavior of Tcl is defined by the command procedures, which are free to interpret their arguments however they like.

Tcl is *embeddable* and *extensible*. The Tcl interpreter is a C library package that can be incorporated in a variety of applications. Several dozen basic commands are implemented in C as part of the Tcl interpreter. Each application can define additional Tcl commands in C ,C++, or Java to augment the basic facilities provided by Tcl. Typically, an application will implement just a few Tcl commands that provide primitive access to its facilities; more complex features are created by writing Tcl scripts that combine the application's primitive features with the built-in commands.

It is also possible to create packages containing useful sets of Tcl commands implemented in C, C++, or Java and then load these packages into any Tcl application on the fly. Tk is one such extension; it provides a collection of commands for creating graphical user interfaces.

Tcl has four properties that make it attractive as a vehicle for executing untrusted scripts:

- The language is interpreted. All actions are already mediated, so this is a natural place to add security controls.
- The language is safe with respect to memory references: it has no pointers, array references are bounds-checked, and storage is managed automatically by the Tcl interpreter.
- Interpreters are first-class objects. An interpreter consists of a set of Tcl commands, a set of variable values, and an execution stack. An application can contain several interpreters that are disjoint from each other. This makes it possible to isolate scripts with different security properties in different interpreters.
- The language is command-oriented; the facilities available to a Tcl script are determined by the set of commands defined in its interpreter. Access to unsafe features is controlled by curtailing access to specific commands in an interpreter executing an untrusted script.

Our work addresses security issues in Tcl scripts and assumes that the implementation of the Tcl interpreter is trustworthy. Extensions written in C, C++, or Java that are made available to applets are also assumed to be trustworthy.

3 Security Issues

The security issues associated with applets fall into four major groups: integrity attacks, privacy attacks, impersonation attacks and denial of service attacks:

- Integrity Attacks: a malicious applet may try to modify or delete information in the environment in unauthorized ways. For example, it might attempt to transfer funds from one account to another in a bank, or it might attempt to delete files on a user's personal machine. In order to prevent this kind of attack, applets must be denied almost all operations that modify the state of the host environment. Occasionally, it may be desirable to permit the applet to make modifications; for example, if the applet is an editor, it might be allowed to write to a file if approved by a user through a file selection dialog.
- Privacy Attacks: these attacks try to steal or leak information belonging to the user. A malicious applet may try to read private information from the host environment and transmit it to a conspirator outside the environment. Information disclosed in this way

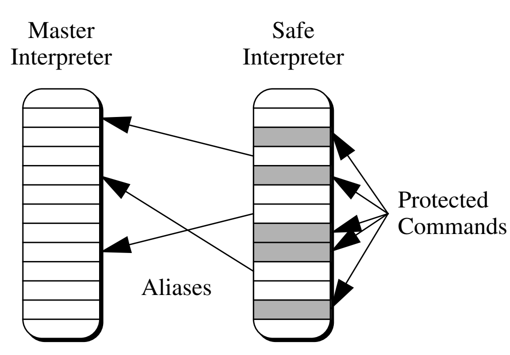

Figure 1. The basic Safe-Tcl mechanisms. Trusted scripts execute in the master interpreter while untrusted applets execute in the safe interpreter. All unsafe commands in the safe interpreter are protected so that they cannot be invoked from the safe interpreter. Aliases provide a mechanism for the applet to request mediated operations from the master. The master interpreter can invoke the protected commands in the safe interpreter.

may have direct value to the recipient, such as business information that could affect the price of a company's stock, or its disclosure could damage the party from which it was taken, for example, if it describes an individual's treatment for substance abuse.

- Impersonation Attacks: a malicious applet might perform actions on behalf of the user of the hosting application without his or her authorization. For example, it may send e-mail in the user's name containing damaging statements, or it may make it appear that the user is attempting to mount some form of attack against a remote resource. The purpose of an impersonation attack can be to damage the impersonated user's reputation.
- Denial of Service Attacks: these attacks interfere with the normal operation of the host system. For example, an applet might consume all the available file space, cover the screen with windows so that the user cannot interact with any other applications, or exercise a bug to crash its hosting application.

It is unlikely that any security policy can completely eliminate all security threats. For example, any bug in an application gives a malicious applet the opportunity to deny service by crashing the application. In addition, there exist subtle techniques for signaling information that make it nearly impossible to implement perfect flow control for privacy [4]. Attempts to completely eliminate the risks would restrict applets to such a degree that they would not be able to perform any useful functions.

Thus, Safe-Tcl does not try to eliminate security risks entirely. Instead, it attempts to reduce the risks to a manageable level, so that the benefits provided by applets are greater than the costs incurred by security attacks. Safe-Tcl concentrates on preventing integrity attacks, privacy attacks and impersonation attacks. It is not geared towards preventing all denial of service attacks. Denial of service attacks generally do not permanently impact the user's ability to perform useful work or the integrity of her information, and thus, while annoying, are less damaging.

4 Safe Tcl

Safe-Tcl uses a *padded cell* approach to security: applets are executed in isolated environments where their capabilities can be restricted. Padded cells are implemented using three mechanisms. First, Safe-Tcl uses *safe interpreters* to isolate applets and prevent them from using any of the unsafe features of the language. Then it restores access to a restricted subset of the unsafe features using *aliases* and *protected commands*.

If a Tcl application wishes to execute an applet, it uses two interpreters: a master interpreter and a safe interpreter (see Figure 1). The master interpreter retains full functionality, so only trusted scripts such as those written by the user or the application designer may execute there. The safe interpreter is used for executing the applet. All of the unsafe commands (those that could result in security compromises if misused) are disabled in the safe interpreter. These commands include those for accessing the file system, executing subprocesses, opening sockets, and many more.

The set of commands left in the safe interpreter, called the *safe base*, allows the applet to perform only safe actions. With only this set of commands an interpreter is indeed safe for executing applets, but in this state the interpreter is not very interesting because scripts running in it are completely isolated. If a script cannot access files, open sockets, or communicate with other processes, then there aren't many useful things that it can do. In fact, most useful programs involve activities that are unsafe in the general case. In order for applets to perform useful activities, they must have restricted access to unsafe functions. For example, it is not safe to let an applet write arbitrary files, but it probably is safe to let an applet create a single new file of limited size containing the results of its computation.

This separation of trusted code from untrusted code is similar to the separation in operating systems between user-space code and kernel-space code. Kernel-space code can write directly on every location on the disk, but user-space code has no direcy access to the disk. Instead, it must use a system call to write data to portions of the disk as permitted by checks enforced by code executing in kernel-space.

The alias mechanism in Safe-Tcl is analogous to system calls in operating systems. An

```
# Create an array in which the names of elements are host
# names and the values are lists of acceptable port numbers.

set safeSockets(sage.eng) 1024
set safeSockets(sunlabs.eng) 80
set safeSockets(www.sun.com) {80 8015}
set safeSockets(bisque.eng) {3000 4000 5000}

# Create an alias that causes the AliasSocket command to be
# invoked in the master whenever socket is invoked in the safe
# interpreter.

interp alias $safe socket {} AliasSocket $safe

# Define the procedure that implements the alias.

proc AliasSocket {safe host port} {
    global safeSockets
    if {![info exists safeSockets($host)]} {
        error "Unknown host: $host"
    }
    if {[lsearch -exact $safeSockets($host) $port] < 0} {
        error "Bad port: $port"
    }
    return [interp invokehidden $safe socket $host $port]
}
```

Figure 2. When this code is executed in a master interpreter, it creates an alias that allows a safe interpreter to open sockets to a restricted set of addresses. Whenever the `socket` command is invoked in interpreter `$safe` the `AliasSocket` command will be invoked in the master interpreter with the name of the safe interpreter as its first argument. Thus, if the value of `$safe` is `child`, and the command "`socket bisque.eng 4000`" is invoked in the safe interpreter, then the command "`AliasSocket child bisque.eng 4000`" will be invoked in the master. The `AliasSocket` procedure checks to see if the host and port are among those that are allowed. If so, it invokes the hidden `socket` command in the safe interpreter to actually open the network connection.

alias is an association between a command in the safe interpreter, called the *source command* for the alias, and a command in the master interpreter, called the *target*. Whenever the source command is invoked by a script in the safe interpreter, the target command is invoked instead. The target command is typically a Tcl procedure. It receives all of the arguments from the source command and its result is returned to the safe interpreter as the result of the source command.

The master interpreter has complete control over the safe interpreter. It can initiate the execution of scripts, create and delete aliases, controlling the names of the source and target commands for each alias. The safe interpreter cannot create new aliases on its own. During the execution of an alias, the master can access the state of the safe interpreter and invoke additional scripts in the safe interpreter to carry out the functions of the alias.

The commands that are disabled in the safe base are not actually removed from the safe interpreter; they are *protected* so that they can be invoked only by the master interpreter. This allows the master interpreter to ensure that only a subset of the command's features are used. To use a protected command, an applet must use an alias which checks that the arguments and intended usage are safe, and then the alias invokes the protected command. Figure 2 shows an alias that allows sockets to be opened only to a pre-specified list of hosts and ports. The `socket` command, which is used to create network connections, is unsafe so it is protected in the safe interpreter; the code in the figure creates a new `socket` command that is an alias. The alias validates the host and port, then invokes the protected `socket` command in the safe interpreter. To the applet the `socket` command appears to work in the normal fashion except that only certain network addresses may be used. Note that two versions of `socket` exist in the safe interpreter: the protected command and the alias.

Protected commands are needed because many Tcl commands implicitly modify the interpreter in which they are invoked. For example, the `socket` command creates a new I/O channel for use in communicating over the socket. The channel is created in the interpreter where the `socket` command executes, so if the alias invoked `socket` in its own interpreter (the master) then the safe interpreter wouldn't be able to use the resulting channel.

5 Security Policies and Security Packages

In Safe-Tcl, aliases are grouped into security policies. Each security policy has a name and contains one or more aliases that are known to be safe for certain kinds of applets. An applet chooses which policy it uses, subject to the checks described in Section 6; a single applet may only use one policy over its lifetime. Many different policies are possible, each imposing a different set of restrictions on applets controlled by the policy. Some policies are safe for all applets to use, while other policies are only safe for applets that can be verified to originate from a trusted source. We designed Safe-Tcl to encourage the development of many different policies, and to allow the reuse of policies in many applications

Why is it important to allow multiple security policies? Wouldn't it be better to have just one policy that includes all of the features that are safe for applets? Multiple security policies are needed because safe features do not compose: if feature A is safe and feature B is safe, the combination of A and B is not necessarily safe. For example, it is safe for an applet to open network connections outside the firewall as long as the applet cannot communicate with hosts inside the firewall. It is also safe for an applet to read local files, as long as this is the only communication the applet makes outside its interpreter. However, an applet that has access to both of these features can transmit local files outside the firewall, which is a breach of privacy.

Since safe features do not compose, no single security policy can include all of the features that are safe in isolation. Safe-Tcl encourages the development of many security policies, each tailored to support a different class of applets. The simplest security policy consists of just the safe base with no additional features enabled. Most security policies will probably enable a small set of additional features.. In an extreme case where the applet is completely trusted, it can be given a security policy that restores the full set of unsafe Tcl commands and enables all the features provided by the hosting application.

An applet obtains a security policy by invoking the `policy` alias which is installed as part of the safe base. The alias checks whether the applet is allowed to use the requested policy. If allowed, it installs the aliases specified by the requested policy and records information that will be used later to control how these aliases are used by the applet. An applet may obtain at most a single security policy over its lifetime; once it has successfully obtained one policy it may not obtain any other policy. Changing the security policy for an applet or allowing it to use multiple policies composes the features of the security policies, which is not safe.

We expect that many policies will be similar in the set of features they provide. To encourage reuse of the implementation of aliases, Safe-Tcl has the concept of *security packages*, named sets of aliases that are installed as a unit. The aliases are implemented by one Tcl script and reused by name in multiple policies.

Policies are written in Tcl using a style that is easily parsed by the configuration management package provided with Safe-Tcl. Each policy is organized into a number of sections, and each section contains permissions and restrictions referring to a set of features. Below is a snippet of the home policy which allows an applet to communicate with servers on the host from which it was loaded, and to fetch web pages from that host:

```
section features
allow url
allow network
allow persist
```

```
section urls
allow $originHomeDirURL*
```

The *home* policy enables the *persist*, *network* and *url* security packages in the *features* section. The *persist* security package allows an applet to store a limited amount of information persistently on the user's machine, *network* allows the applet to open sockets to a restricted set of remote servers, and *url* allows access to a limited set of web pages and remote web services. The *urls* section allows URLs to be fetched from the subtree of the web site rooted at the directory containing the applet.

The *outside* policy is similar to the *home* policy and reuses the persist, network and url security packages. Its *url* section allows URLs to be fetched from the C|Net web site:

```
section features
allow persist
allow url
allow network

section urls
allow http://www.cnet.com/*
```

Aliases installed into the safe interpreter housing the applet allow the applet mediated access to features provided by the hosting application. The target command checks the arguments according to the restrictions imposed by the applet's policy and decides whether to allow the call. In the above example, if a URL fetch is requested, it is only allowed if it refers to a web page on the C|Net web site. If the operation is allowed, the call is forwarded to the actual implementation.

The Tcl web browser plugin comes with several policies, including *home* and *outside*, mentioned above. Each of these policies is fairly restrictive, yet supports a large class of interesting applets. An advantage of having more restrictive security policies is simplicity. If a security policy included a large number of features, it would be difficult to analyze all of the interactions between its features to uncover security loopholes. A policy that includes only a small number of features is more amenable to analysis and increases our confidence that it is really secure.

6 Controlling the Use of Security Policies

Safe-Tcl is designed to support a large range of security policies. Some policies can be used by all applets, without requiring that the applet be trusted. Other policies, especially those that provide access to features that can be used to mount security attacks, require that the applet be trusted to some degree before they can be used by that applet. Safe-Tcl check whether an applet is allowed to use a policy when the applet first requests to use the policy. If the applet is allowed to use the policy, the security packages enabled by the requested policy are installed.

Safe-Tcl provides the concept of a *trust map* to allow site and application administrators to control which applets can use each security policy. A trust map is a Tcl script, organized into sections similarly to a policy, that specifies under what circumstances each policy can be used; the map also defines the names of all security packages provided by the application and controls various other application level resources. Each application using Safe-Tcl has its own trust map; here is part of the policies section in the trust map for the Tcl web browser plugin:

```
section policies
disallow trusted
allow home
allow javascript \
ifallowed javascriptTrustedURLs $originURL
```

A trust map can contain statements that disable a policy for all applets, as is the case for the *trusted* policy, above. Similarly, a policy can be enabled for all applets, as for the *home* policy. The trust map also provides a place to insert authenticators that decide to allow or disallow the use of a policy based on some property of the applet. We see an example of this in the staement allowing the use of the *javascript* policy if the URL from which the applet was loaded is allowed by the section *javascriptTrustedURLs*. Authenticators can use attributes of the applet such as its MD5 checksum [9], an attached signed certificate [5], or the URL from which it was loaded.

Each application and site This separation of trust and authentication from the actual policies is important, because otherwise policies can not be shared between applications.

7 Security Model and Security Roles

Safe-Tcl cleanly factors into three distinct parts that reflect the roles of three human participants in the management of security:

- Security packages encapsulate features provided by applications and implement constrained access to these features. Security packages are provided by the application's author.
- Security policies determine which security packages an applet can use and what resources it can access using the provided features. A security expert designs each security policy.
- The trust map determines whether an applet can use a given security policy. Trust maps are edited by site and application administrators.

Much of the power and flexibility of Safe-Tcl stems from its use of Tcl throughout to implement the security model. Tcl is used to implement the configuration mechanism that controls access to resources by applets. Tcl scripts specify whether a policy can be used by an applet and under what conditions. Finally, Tcl is used to implement the security packages and the aliases used by applets to access features provided by security packages.

8 Denial-of-Service and Privacy Attacks

Although Safe-Tcl was designed primarily to address issues of integrity, impersonation and privacy, its mechanisms can also be used to prevent denial-of-service attacks. For example, an applet can be prevented from consuming all the disk space by protecting the `puts` command, which writes data to files. In its place an alias can be created to count the bytes that are output and enforce a limit.

However, many denial-of-service attacks, particularly those associated with graphical user interfaces, are hard to prevent. For example, that an applet attempts to create a window that covers the whole screen and prevent the user from interacting with any other applications. Aliases and hidden commands could be used to restrict the sizes of windows, but the applet could then create several smaller windows that together cover the whole area of the screen. Furthermore, in some situations (such as laptop computers with small screens) it may be desirable to let an applet use the entire screen.

Safe-Tcl currently does not prevent most denial-of-service attacks. We will address this in the future with a combination of resource controls and a kill-key that lets a user intervene when an applet misbehaves.

Lampson [4] shows that it is generally impossible to prevent information from being communicated from one applet to another through covert channels such as manipulation of the scheduler or other finite accessible resources. While the rate of communication is limited, it is still possible to leak a significant number of bits per second through this form of attack. For example, if one applet has access to file stored on a server within a firewall, while another applet has the ability to communicate over the network with a server outside the firewall, it is possible for the two applets to collaborate and disclose information stored in these files to outside parties. Safe-Tcl by default disallows untrusted applets to use the `inside` policy which is intended to allow access to resources on an Intranet. Thus, to a large extent, information leakage through covert channels is prevented because applets can by default not gain access to private information stored on hosts on an Intranet.

9 Status

Safe-Tcl has been available in public Tcl releases since the Tcl 7.5 release in April 1996. Safe-Tcl integration with Tk is implemented as part of a Tcl/Tk plugin module for web browsers which was released in July 1996 [6]. The plugin allows Tcl/Tk scripts to be included in Web pages with embedded custom GUIs. As of this writing (December 1997) Safe-Tcl is in 2.0 beta release with full support for safe interpreters, aliases, mechanisms for creating and installing security policies, and a trust map implementation. Safe-Tcl does not yet support a kill key, CPU usage limits, or authentication.

10 Related Work

10.1 The Borenstein/Rose prototype

Nathaniel Borenstein and Marshall Rose implemented a prototype of Safe-Tcl in 1992 that pioneered most of the ideas, including safe interpreters and aliases [2]. The Borenstein/Rose prototype was used for active email messages and later as part of the First Virtual Holdings Internet payment system.

Our implementation generalizes the Borenstein/Rose prototype in several ways. The prototype only allowed one safe interpreter, while our work allows any number of safe interpreters. There was no concept of security policies and security packages in the Borenstein/Rose prototype, and there were no mechanisms to specify configuration information. Their implementation was specific for one problem domain, how to send scripts via electronic mail messages safely, while our approach can be applied to any problem domain. Finally, our work introduces the concept of trust maps to allow or disallow applets to use specific policies.

10.2 Object Oriented Systems.

Most other security models for executing untrusted code, such as Java [12] and Telescript [11], are based on object systems. These models are similar to Safe-Tcl in that they use safe languages that control pointers and memory allocation. However, they differ from Safe-Tcl in that they provide only a single virtual machine that contains all of the objects and classes. Security properties are associated with individual objects or classes; for example, one class may be marked as coming from an untrusted source while another may be marked as trusted. This information is used when deciding whether or not to allow a particular operation. For example, before allowing a file to be opened, Java checks to see if there are any untrusted classes on the current call stack; if so, the open operation is denied. In contrast, Safe-Tcl's padded cell approach uses multiple virtual machines (interpreters) and the security properties are associated with the virtual machine, not individual pieces of data or code. Security decisions are made based on the virtual machine that is currently executing; thus, while executing in a safe interpreter it is not possible to open a file, but it is possible to open a file if control is first transferred to a trusted interpreter using an alias.

In Java the use of a single virtual machine for both trusted and untrusted code requires security checks at a low level within the system, and this make it difficult to implement sophisticated security policies. Security manager calls must be made at low levels, right before a native method call into the operating system, otherwise untrusted code could call the method directly and bypass the security check. The security check can easily test if untrusted code is on the call stack and deny access. However, it is more difficult to implement a policy that denies some accesses by untrusted code but allows other accesses. For example, a policy could provide a method that displays a file selection dialog and returns an open I/O channel. The goal is that the file open operation is only done via the user interface dialog so the user knows what file is being accessed. With one low-level check, it would be difficult to allow accesses from the trusted dialog without allowing other, more direct accesses from the untrusted code.The use of a single virtual machine makes it difficult to hide dangerous operations, which forces a low-level security check. The low-level check limits the flexibility of policy code and adds overhead to dangerous operations for both trusted and untrusted code.

In contrast, in Safe-Tcl security checks are done at a very high-level, when an alias transfers control to a trusted virtual machine, which leaves room for flexible policy implementations. (See Figure 3.) Safe-Tcl can replace the open command with an alias that displays a user interface dialog or uses policy code that limits untrusted code to a private directory and a limited number of files. The implementation of these aliases may look through the file system and open various configuration files that specify the directory location and limits on files. Eventually the alias will open a file for use by the untrusted code. The alias is implemented in a trusted virtual machine that can do anything on behalf of untrusted code. This provides a very flexible environment for wraping policy code around dangerous operations, and the policies only add overhead to untrusted code that must use aliases.

There are more advantages to the Safe-Tcl implementation:

- Aggressive security policies can hide more commands from untrusted virtual machines, such as those that gave clock and timing information. This only affects the untrusted virtual machine, and it does not require modification of the existing clock and timer implementations. In Java, adding a security manager check to the clock subsystem requires modification of trusted code.
- Tcl allows multiple virtual machines (i.e., interpreters) within the same application, and different interpreters can have different security policies through different sets of aliases. This is more awkward in Java because there is a single security manager that would have to manage different policies for different kinds of untrusted classes. Our understanding of Java development is that they plan to support multiple security managers in the future.

10.3 Pure authentication: ActiveX

An approach proposed by Microsoft authenticates downloaded applets and asks the user of the hosting machine to assign trust to the identified principal. ActiveX prevents untrusted programs from being executed. In this approach all security decisions are delegated to the user of the machine. This is the only approach that works for compiled programs, because there is no practical mechanism for restricting what machine code can do.

But ActiveX only verifies identity, and trust involves more than just authentication.

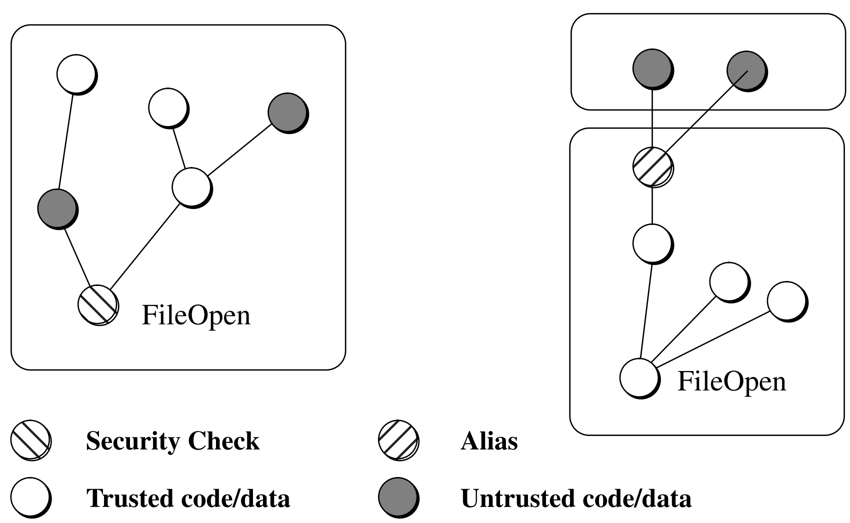

Figure 3. With an object-oriented approach to security (left) security checks must be done at a low level, guarding the call to the operating system, so they cannot be by-passed. A low-level check makes it difficult to determine if trusted code called by untrusted code should be allowed to perform the operation. With the padded cell approach (left) security checks can be done at a high-level when control transfers from code in a separate virtual machine. Edges in the graph represent method or procedure calls, and boxes represent virtual machine boundaries.

Authentication identifies the principal (person or organization) who wrote something, but it doesn't indicate whether the principal is trustworthy. Trust can really only be placed in principals you are *familiar* with. The authentication approach works well for applets written by large companies that are known to be trustworthy (or that can be sued if their software is defective). However, authentication doesn't help when applets are written by individuals and smaller companies that are not well known. One of the reasons for the popularity of the World Wide Web is that it enables communication among large numbers of individuals and small organizations that have no prior knowledge of each other.

ActiveX is unsuitable for executing untrusted programs retrieved from the web, because it is based only on authentication and does not provide any restrictions on what a program can do once it is trusted. This does not scale well to the web's distrubuted and decentralized nature. In contrast, Safe-Tcl enables safe execution of code trusted to varying degrees, ranging from completely untrusted to completely trusted. For some operations, no knowledge about an applet author's identity is needed, while other operations may require full authentication.

11 Conclusions

There is no silver bullet that will make security trivial. Creating safe environments for executing applets will always be difficult, and no security model will ever be totally safe, since even a small bug in programming can open a huge security hole. However, we think it is possible to create environments where applets with varying degrees of trust can be executed with an acceptable level of risk. Safe-Tcl has several properties that simplify the creation of such environments:

- The padded cell model is simple. It generalizes the user space-kernel space model that has been used successfully in operating systems for several decades.
- Safe-Tcl groups data and code with similar security properties together, which reduces the amount of code that must be aware of security issues.
- Safe-Tcl separates security management into well-defined phases that are geared towards the party that is responsible for each aspect of security. It separates implementation of security policies, generally an activity for security experts, from the implementation of security packages, which is done by engineers creating an application, and from configuration management, an activity reserved for site and system administrators.

Our experiences with Safe-Tcl have taught us three important lessons about security. The first is that safe features do not necessarily compose. This makes it difficult to provide a single security policy with a large variety of features; instead, it encourages a large number of smaller, specialized security policies. The second lesson is that it is important to take advantage of authentication mechanisms yet not require them. If programs are to be intimately tied to information, and if information is to be freely distributed among strangers, then it is important to support the execution of totally untrusted programs. At the same time, authentication can be used to boost the power of applets when they come from known sources. The third lesson is that using a scripting language to implement a security model is both doable and adds unique value by allowing the resulting system to be very flexible and configurable.

12 Acknowledgments

This work would never have come about without the pioneering efforts of Nathaniel Borenstein and Marshall Rose, who designed and built the Safe-Tcl prototype. Nathaniel Borenstein, Wan-Teh Chang, Robert Drost, Clif Flynt, Li Gong, Mark Harrison, Ray Johnson, Anand Palaniswamy, Marshall Rose, Rich Salz, Juergen Schoenwaelder, and Glenn Vanderburg provided useful comments that improved the presentation of this paper.

13 References

[1] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, ISBN 0-201-63455-4, 1996.

[2] N. Borenstein, "EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail," *IFIP WG 6.5 Conference*, Barcelona, May, 1994, North Holland, Amsterdam, 1994.

[3] D. Denning and P. Denning, "Data Security," *Computing Surveys*, Vol. 11, No. 3, September 1979, pp. 227-249.

[4] B. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, Vol. 16, No. 10, October 1973, pp. 613-615.

[5] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Transactions on Computer Systems*, Vol. 10, No. 4, November 1992, pp. 265-310.

[6] L.Demailly, J. Levy, *Welcome to the Tcl Plugin*, `http://sunscript.sun.com/plugin/`.

[7] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, 1994.

[8] R. Rivest, *The MD5 Message Digest Algorithm*, RFC 1321, April 1992.

[9] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," Proc. 14th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 27, No. 5, December, 1993, pp. 203-216.

[10] B. Welch, *Practical Programming in Tcl and Tk*, Prentice-Hall, ISBN 0-13-616830-2, Second edition, 1997.

[11] J. White, *Telescript Technology: The Foundation for the Electronic Marketplace*, white paper, General Magic, Inc., 1994.

[12] F. Yellin, "Low Level Security in Java," *World-Wide Web Conference*, Boston MA, December 1995. Also available as http://www.javasoft.com/sfaq/verifier.html.