The Safe-Tcl Security Model

John K. Ousterhout
Jacob Y. Levy
Brent B. Welch

Sun Microsystems Laboratories
2550 Garcia Avenue, MS UMTV-29-232
Mountain View, CA 94043-1100
john.ousterhout@eng.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 the safe interpreter is controlled by trusted scripts running in a *master interpreter*. Safe-Tcl provides an *alias* mechanism that allows applets to request services from the master interpreter in a controlled fashion. Safe-Tcl allows a variety of security policies to be implemented even within a single application, and it supports both policies that authenticate incoming scripts and those that do not.

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 workstation.

The traditional "solution" to the security problem has been for people not to run 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 floppy 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 allows programs

Sun, Sun Microsystems, Java, and the Sun logo are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries.

to be associated with Web pages: when such a page is viewed, the program is executed to provide special interactive effects such as animations. As a result of these trends, it is becoming more and more difficult for users to tell when they are running a program or who wrote the program.

Safe-Tcl makes it possible for people to run programs written in the Tcl scripting language[9][12] without knowing their origin or trustworthiness. Safe-Tcl deals with potential security problems by restricting the behavior of programs so that they have fewer capabilities than the users who invoke them. The privileges granted to a program can be adjusted to match the program's trustworthiness. Programs of unknown origin should not be trusted at all, so they run with very few privileges. If the author of a program can be authenticated, and if that author is partially or fully trusted, the program 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.

Safe-Tcl is based on *padded cells*. In the padded cell approach, untrusted scripts (applets) are executed in separate environments that are isolated from the trusted portions of the application. The features available to the applet can be controlled by the trusted portions of the application. The implementation of Safe-Tcl is based on two key 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. Different security policies may be implemented by providing different sets of aliases in a safe interpreter.

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 two particular strengths:

- It 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, but rather provides mechanisms for implementing a variety of security policies. Different organizations can implement different security policies depending on their needs, and a single application can use different policies for different applets. In particular, it is 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.

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 and aliases. Section 5 discusses the issues in writing security policies and describes a few sample policies. Section 6 shows how authentication mechanisms can be used in Safe-Tcl, and Section 7 describes how Safe-Tcl allows a single appli-

cation to support multiple applets at the same time. Section 8 describes how Safe-Tcl deals with denial-of-service attacks. Section 9 discusses 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[9][12]. It has a 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, as shown in Figure 1. Several dozen basic commands are implemented in C as part of the Tcl interpreter. Each application can define additional Tcl commands in C or C++ 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 primitives with the builtin commands.

It is also possible to create packages containing useful sets of Tcl commands implemented in C or C++ 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.

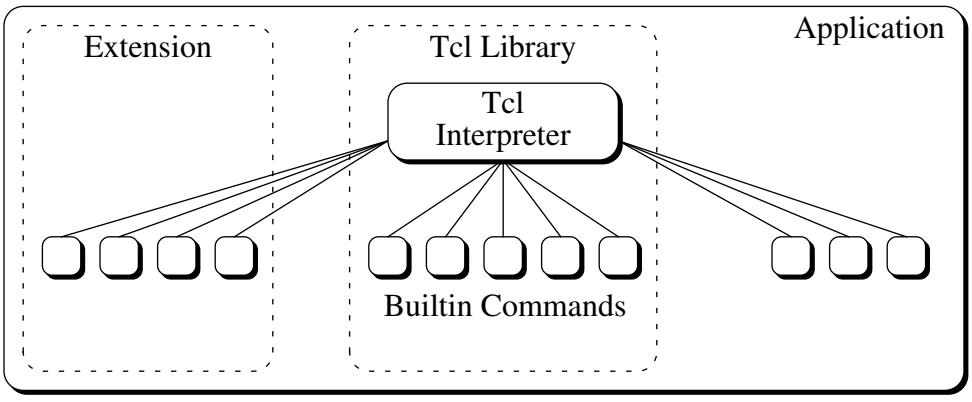

Figure 1. When an application uses Tcl it incorporates application-specific functionality into the Tcl interpreter as additional Tcl commands. The application can also load extensions dynamically to provide yet more Tcl commands.

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

- The language is interpreted. Since every action is already mediated, there is a natural place to add security controls.
- The language is safe with respect to memory usage: it has no pointers, array references are bounds-checked, and storage is managed automatically by the Tcl interpreter. This prevents scripts from making wild references to storage.
- Interpreters are first-class objects. An interpreter consists of a set of Tcl commands, a set of variable values, and an execution stack; it completely encapsulates the execution of a Tcl script. A single application can contain multiple interpreters that are totally disjoint from each other. This makes it possible to isolate scripts with different security properties.
- The language is command-oriented, in that the facilities available to a Tcl script are determined by the set of commands defined in its interpreter. Different interpreters can have different command sets with different security properties.

Although we will show in the remainder of this paper how to manage untrusted Tcl scripts, our work does not address security issues in C. Thus we assume that all of the C and C++ code associated with a Tcl application is trustworthy (e.g.,that users have properly authenticated it and/or analyzed its security properties).

3 Security issues

This paper envisions a security environment based on applications and applets. In this environment an untrusted program does not execute by itself; instead, it executes in conjunction with some trusted application. We use the term *applet* to refer to the untrusted program, and *application* to refer to the trusted environment in which it runs. The application provides a security model that restricts the execution of the applet. For example, the application might be a Web browser and the applet might be a program that animates the content of a Web page. Or, an application might consist of only a security model with no other functionality, and it might be used to run large “applets” that implement major applications such as spreadsheets or word processors.

The security issues associated with applets fall into three major groups: integrity attacks, privacy attacks, and denial of service attacks. These are discussed individually in the subsections below.

3.1 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 a few

limited modifications; for example, if the applet is an editor, it might be given write access to the file being edited.

3.2 Privacy attacks

The second form of attack consists of information theft or leakage; a malicious applet may try to read private information from the host environment and transmit it to a co-conspirator outside the environment. Information disclosed in this way may have direct value to the recipient, such as business information that could affect the value 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.

One approach to the privacy problem is to prevent applets from accessing sensitive information at all. However, this approach would also prevent applets from performing many useful functions. For example, this approach would prevent applets from helping to display, analyze, and edit sensitive information.

A more desirable approach is to give applets access to sensitive information but prevent them from transmitting the information outside the host environment. This approach is called *flow control*. In principle, it might seem possible for the security model to analyze the flow of information through the applet and prevent information read from sensitive sources from being written to insecure I/O ports. This might be done by analyzing the application statically to detect illegal flows. Or, it might be done using a technique called data tainting [8], in which data is tagged with information about its sensitivity. Data read from a sensitive source is tagged with a high sensitivity level, and attempts to write that data to an insecure I/O port are denied. Unfortunately these forms of flow control are hard to implement and use. Static analysis shares many of the difficulties of program verification . Data tainting has difficulties when data from different sources is combined arithmetically or when data is used to control conditional operations. Under these conditions it is difficult to determine whether information has really flowed from one variable to another, so the result must be tainted with the highest sensitivity level of the inputs; this can lead to overclassification, where all of an application's data ends up with a high sensitivity level[3].

Safe-Tcl's approach to flow control is to implement it via access control; rather than allowing access to all objects and restricting flow, Safe-Tcl disallows combinations of accesses that could result in unsafe flow. This means that either an applet can read sensitive information *or* it can open external I/O ports, but not both. A given applet must choose in advance which kind of access it wishes to have, and the other form will be totally denied to it.

Privacy is often defined in terms of a *firewall*. For example, many companies have special firewall machines that separate their internal networks from the Internet. Information inside the firewall is considered private, and the company's security policies are designed to prevent private information from leaking outside the firewall. In the rest of this paper we will assume the existence of a firewall; an environment with no firewall can be treated as if there were a separate firewall around each computer.

3.3 Denial of service

The third form of attack consists of denial of service, where the applet attempts to interfere with the normal operation of the host system. For example, it 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 application.

Denial-of-service attacks are less severe from a security standpoint than integrity or privacy attacks, yet they are harder to prevent. They are less severe because they don't do lasting damage; in the worst case, the effects can be eliminated by killing the application and freeing any extraneous resources that it allocated. In contrast, the effects of an integrity or privacy attack may be difficult to undo (e.g., if sensitive information has been leaked to a large audience its privacy can never be restored). On the other hand, it is difficult to distinguish a denial-of-service attack from acceptable behavior. For example, a perfectly legitimate attempt to write a file could consume most or all of the available file space. Or, a legitimate applet might attempt to create windows that exceed the space available on the screen (e.g., the applet may have been designed for a large screen yet be executing on a laptop with a much smaller screen). If such applets are treated as hostile then many useful forms of behavior may be prohibited.

3.4 Managing risk

It is unlikely that any security policy can completely eliminate all of the 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 impossible to implement perfect flow control for privacy [5]. 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 security models like Safe-Tcl do not try to eliminate security risks entirely, but rather to reduce them to a manageable level (e.g. so that the benefits provided by applets are greater then the costs incurred by security attacks). For example, Safe-Tcl makes it possible to reduce the rate at which sensitive information can be transmitted outside a firewall, even though it cannot completely eliminate the threat.

4 Safe interpreters, aliases, and hidden commands

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 *hidden commands*. The rest of this section describes these three techniques in more detail.

Tcl applications that don't need to execute applets use a single Tcl interpreter that has complete access to all the capabilities of Tcl, the application, and its user. All scripts run-

Commands	Functionality
`open`, `socket`	Open files and network connections.
`file`, `glob`	Create, copy, rename, and delete files and directories; query file attributes and file name space.
`exec`	Invoke subprocesses
`cd`, `pwd`	Manipulate current working directory.
`load`	Load shared library binary into application from file.
`exit`	Terminate application.
`source`	Evaluate Tcl script file.
`toplevel`	Create toplevel window.
`send`	Invoke Tcl script in another application on the same display.
`clipboard`, `selection`	Read and write selection/clipboard information.

Table 1. The Tcl and Tk commands that are not available to scripts executing in safe interpreters.

ning in this interpreter must be trusted. If a Tcl application wishes to execute an applet, it will use two interpreters: a master interpreter and a slave interpreter (see Figure 2). 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 made inaccessible in the safe interpreter. The disabled commands include those for accessing the file system, executing subprocesses, opening sockets, and many more (see Table 1). We refer to the commands that are left as the *safe base*; these commands will be available to virtually all applets in all applications.

The master interpreter in a Safe-Tcl application is much like kernel space in an operating system: it has complete access to all of the system's facilities. The safe interpreter in a Safe-Tcl application is similar to user space in an operating system. From user space it is not possible to access the kernel's memory, manipulate I/O devices, or have any direct

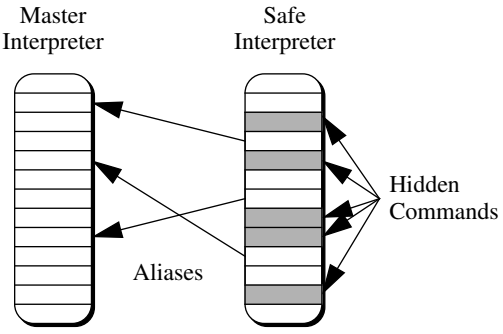

Figure 2. The basic Safe-Tcl mechanisms. Trusted scripts execute in the master interpreter while untrusted applets execute in the safe interpreter. All unsafe commands in the safe interpreter are hidden so that they cannot be be invoked from the safe interpreter. Aliases provide a mechanism for the applet to request protected operations from the master. The master interpreter can invoke the hidden commands in the slave.

communication with the outside world. Similarly, a safe interpreter is isolated from its master and cannot communicate directly with the rest of the application.

The safe base produces an interpreter that 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 of the useful things that programs do involve activities that are unsafe in the general case. In order for applets to carry out 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 the script create a single new file of limited size containing the results of its computation.

A similar situation exists in operating systems. By itself a process executing in user space cannot do anything interesting; for example, it cannot write to disk or communicate with the user. To solve this problem operating systems provide system calls, which give user processes restricted access to unsafe features. For example, a user process cannot directly write the disk, but it can invoke a system call that will write the portions of disk containing files owned by the process's user.

The alias mechanism in Safe-Tcl is analogous to system calls in operating systems. An 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 read and write variables in the safe interpreter and initiate the execution of scripts in the safe interpreter. The master manages the aliases for safe interpreter; it can create and delete aliases at any time and it defines the names of the source and destination 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 just hidden so that they cannot be invoked by the safe interpreter. However, the master may invoke the hidden commands of the safe interpreter. This allows the master to use the commands in restricted ways. For example, Figure 3 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 automatically hidden 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 hidden `socket` command in the safe interpreter. To the applet the `open` command appears to work in the normal fashion except that only certain file names may be used. Note that two versions of `open` exist in the safe interpreter: the hidden command and the alias.

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

interp alias $slave socket {} AliasSocket $slave

proc AliasSocket {slave 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 $slave socket $host $port]
}
```

Figure 3. 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. The first group of statements creates an array `safeSockets`; the names of the array elements are names of hosts that may be connected to, and the value of each element is a list of acceptable ports on that host. The `interp alias` command creates an alias, so that whenever the `socket` command is invoked in interpreter `$slave` the `AliasSocket` command will be invoked in the master interpreter with the name of the slave as its first argument. Thus if the value of `$slave` 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 last group of statements defines the `AliasSocket` procedure: it 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.

Hidden 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 object for use in transferring data to and from 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. The hidden command mechanism allows the master to invoke unsafe commands in the context of the safe interpreter, so that their side effects will occur there.

5 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. One of the strengths of Safe-Tcl is that it permits a variety of security policies. The simplest security policy consists of just the safe base with no aliases at all. Most security policies will probably have a small fixed set of aliases. If the applet is trusted, it might be given a security policy that restores the full set of (unsafe) Tcl/Tk commands. At the other extreme, highly sensitive environments might use security policies that hide some of the commands in the safe base (such as those that provide information about the platform on which the application is running).

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 poli-

cies 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, if an applet has access to both of these features then it 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. Thus Safe-Tcl encourages the development of many security policies, each tailored to support a different class of applets. For example, many policies fall into categories we refer to as Outside and Inside. An Outside policy gives an applet access to information outside the firewall, e.g. by fetching Web pages or opening sockets. However, an Outside policy must prohibit all access to sensitive information inside the firewall; otherwise, the script could leak that information outside the firewall. An Inside policy is roughly the opposite of Outside; it allows the applet to access sensitive information inside the firewall, but it must make sure that the applet cannot communicate outside the firewall. Inside policies must also limit access to read-only, so that the applet cannot corrupt data. Each one of these policies is fairly restrictive, yet each supports an interesting class of applets: Outside enables an applet to navigate the Internet on behalf of the user, while Inside allows an applet to safely search and analyze information inside a corporate firewall.

Another advantage of having more restrictive security policies is simplicity. If a security policy includes a large number of features, it will be difficult to analyze all of the interactions between its features to uncover security loopholes. If a policy includes only a small number of features, it will be easier to determine whether it is truly secure. Designing security policies is likely to be complicated in any case, but simpler, more restrictive policies are likely to be easier to manage than larger, more feature-rich policies.

Safe-Tcl security policies are intended to be independent from applications. Each security policy is packaged as a collection of Tcl scripts that implement the policy's aliases. Policies can be distributed independently of applications and it should be possible to create policies that are useful in a variety of applications. Of course, some policies may take advantage of application-specific features that limit their use to a particular application, but it should also be possible to design application-independent policies. This makes it possible for applets to be used in a variety of applications; if security policies were associated with particular applications, then an applet would have to be coded for the security policy of a particular application and the applet would not be usable with other applications.

When an applet starts execution, its interpreter contains only the safe base, which includes an alias for loading security policies. The applet requests a specific security policy using the Tcl package mechanism, which invokes the alias. The master interpreter decides whether or not to permit that policy for the applet and, if the policy is permitted, creates the aliases associated with the policy. If the security policy is denied then an error is returned to the applet, which will cause it to abort in most cases. However, if the applet

chooses, it can catch the error and request alternate policies or even attempt to execute with no security policy, using only the commands in the safe base.

An applet may use only a single security policy over its lifetime. Once it has successfully loaded one policy it may not load any other policy, even if it gives up the aliases of the first policy. Changing the security policy for an applet would effectively compose the features of the security policies, which is not safe (see Section 7).

6 Using authentication

If the author of an applet is unknown, the receiving application must assume the worst and restrict what the applet can do. However, if the application can deduce something about the author of an applet, then it may be able to grant more privileges to the applet. For example, a company might produce an officially approved set of applets for internal functions such as travel authorization and salary adjustments. If an application can determine that an applet is one of the approved ones, it can make additional security policies available to the applet. The additional policies might provide access to corporate databases associated with travel expenses and salaries, which could not safely be permitted to applets of unknown origin.

There exist a variety of authentication mechanisms for verifying the origin of a particular piece of information (e.g., see [6]). Most techniques involve encryption of some sort. For example, one approach using public key encryption is for the creator of an applet to encrypt the applet with his or her private key and distribute the encrypted applet along with the name of the author. Before executing the applet, an application can retrieve the public key of the author whose name was attached to the applet; if the applet can be decrypted with the public key then it must have been written by the owner of the private key.

Another approach is for an organization to compute a cryptographic checksum for each of its authorized applications. This can be done using algorithms such as MD5 [10], which produce a unique identifier for a string of text. Before an application executes an applet it can recompute the checksum for the applet and compare it against a database of trusted checksums. If it matches, then the applet can be given additional privileges.

The current version of Safe-Tcl does not have builtin support for authentication, but it can be added as an extension and we plan to provide builtin support in a future release.

Authentication mechanisms can also be used to distribute new security policies. For example, the following mechanism allows an untrusted applet to carry a trusted security policy with it; when an application executes the applet, it can safely load the security policy even though it doesn't trust the applet:

1. When the security policy is created, its creator generates a cryptographic certificate for the security policy. For example, an MD5 checksum can be computed for the policy's Tcl script and encrypted with the creator's private key; the encrypted checksum plus the name of the creator form a certificate.

2. The security policy can then be distributed freely along with the certificate.
3. If an applet wishes to use that security policy, it includes the Tcl script and certificate for the policy.
4. When the applet starts executing in an application, it invokes an alias, passing it the Tcl script for the security policy and the certificate.
5. The alias verifies the authenticity of the policy by decrypting the certificate's checksum with the creator's public key, then recomputing the checksum of the Tcl script and verifying that it matches the checksum from the certificate. If the policy has been authenticated, and if the application trusts the policy's creator, the application can safely load the policy even though it doesn't trust the applet that delivered the policy.

The advantage of using authentication for security policies is that it makes it easier to distribute new security policies. For example, a corporate security authority can create new policies, generate certificates for them, and distribute the policies to applet developers. Applet developers can then use those policies and incorporate them with the applets as described above. Any application that trusts the corporate authority can immediately run applets containing the new security policies, even though it doesn't trust the applets. No changes need to be made to applications, and individual users need not install or even understand the new policies.

Even under the best of conditions, security policies are likely to be difficult to write and analyze. New policies are likely to be developed only by experts with special skills. At the same time, it is important to have a rich and constantly improving set of security policies available for use by applets. Authentication makes it possible for a variety of applets and applications to take advantage of new policies developed by experts.

7 Multiple applets

The Safe-Tcl model permits more than one safe interpreter in a single application, each with its own security policy. A single master interpreter can provide different sets of aliases for each safe interpreter, and it can use the alias mechanism to implement communication between the safe interpreters. This mechanism could be used to provide meeting places for applets, where many independent applets could exist simultaneously and communicate with each other. One possible example is an electronic marketplace, where individuals could send agents (in the form of applets) to buy and sell goods.

Special care is required when implementing communication between applets, since this effectively composes their security policies. For example, if one applet is using an Inside security policy and another is using an Outside security policy, it is not safe to allow them to communicate. If they could communicate, the Inside applet could read internal information and pass it to the Outside applet, which could then leak the information outside the firewall. Even communication among applets with the same security policy may not be safe. For example, consider a security policy that allows an applet to open a single socket connection anywhere. This policy produces either an Inside applet or an Outside applet, depending on whether the socket's target is inside the firewall or outside the firewall, but

in either case information cannot flow from inside the firewall to outside as long as only one socket is open. However, if two different applets using this policy are allowed to communicate, they can collude to pass information through the firewall.

8 Denial-of-service attacks

Although Safe-Tcl was designed primarily to address issues of integrity 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 hiding 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, suppose 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. Another example is grabs, which force the user to interact with a single window before doing anything else. Grabs are a key part of the look and feel of most windowing systems, yet they can result in system lockup if misused.

We plan to handle many of the denial-of-service attacks, particularly those related to GUIs, with a “kill key” that the user can press at any time to destroy the applet under the mouse. With this approach an application need not worry at all about many denial-of-service attacks; if they occur, the user will notice and kill the offending applet. The kill-key approach is relatively simple to implement and also accommodates a large range of legitimate applet behavior.

The kill-key approach only works if there is a user present and if denial-of-service attacks will be noticed by the user. These conditions are met for most of the attacks that can occur in interactive applications, such as consuming too much CPU time, misusing windows, etc. However, the kill-key approach will not work in non-interactive applications such as an electronic marketplace, since there is no user to notice the problem.

In any case, we plan to implement mechanisms to monitor CPU usage in Safe-Tcl. Otherwise an applet can hang up its application by entering an infinite loop. Safe-Tcl’s approach to CPU controls is to invoke a scheduling function in the master interpreter once the safe interpreter has executed a predefined number of commands. The scheduling function can either abort the applet or give it a new “time slice”, after which the scheduling function will be invoked again. For interactive applets the scheduling function can check to see if the kill key has been pressed; for non-interactive applets the scheduling function can implement an upper limit on CPU usage.

9 Status

Safe-Tcl has been available in the 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 Netscape Navigator, which was released in July 1996[7]. The plugin allows Tcl/Tk scripts to be included in Web pages; when the pages are viewed, the scripts run in a safe interpreter to display custom GUIs. As of this writing (October 1996), Safe-Tcl supports safe interpreters and aliases but it allows only a single security policy consisting of the safe base. We plan to make a broader variety of policies available in the fall of 1996, and users can already define their own policies if they wish. 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. The main contribution of our implementation is to generalize the ideas in the Borenstein/Rose prototype. For example, the Borenstein/Rose prototype allowed only a single safe interpreter per application and did not separate specific security policies from the overall security model. In addition, it did not provide hidden commands so several of the security controls had to be hard-wired in the C code of Tcl and Tk; our approach allows them to be implemented as Tcl scripts that are part of a security policy.

10.2 Object-oriented approaches

Most other security models for executing untrusted code, such as Java and Telescript[13], 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 (see Figure 4). 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; for example, while executing in a safe interpreter it is not possible to open a file.

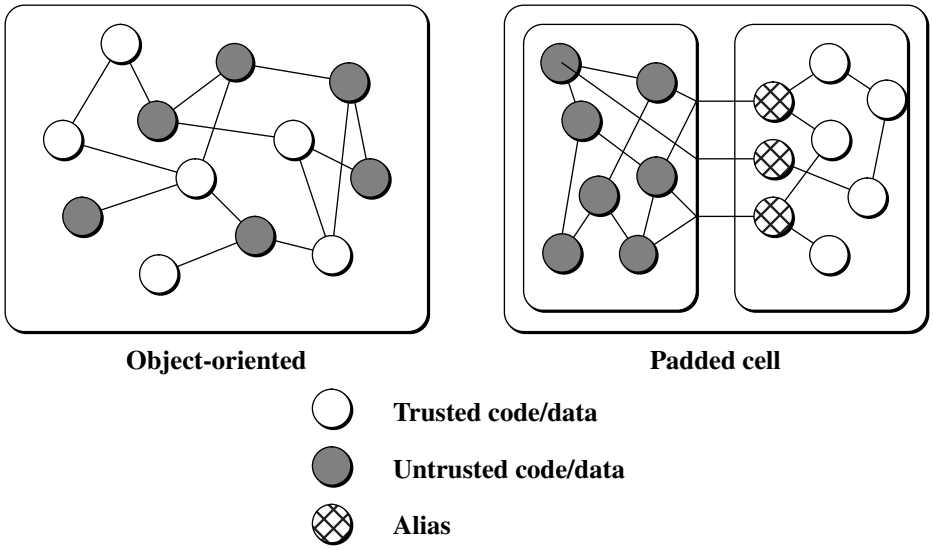

Figure 4. With an object-oriented approach to security (left) trusted and untrusted classes intermingle in a single virtual machine, which results in complex security interactions. With the padded cell approach (right) untrusted code is isolated in a separate virtual machine; interactions between untrusted and trusted code occur only through well-defined aliases, which reduces the complexity of the security issues.

The object-oriented and padded cell approaches are equivalent in that either one can be used to emulate the other. However, we believe that the padded cell approach is substantially less complicated in the common case; because of this it makes security problems easier to manage. The reason for Safe-Tcl's simplicity is that it has more clearly defined boundaries between domains with different security properties. In Safe-Tcl, all the data and code with the same security properties are colocated within the same interpreter. As long as execution stays within a single interpreter, no security issues arise. Security issues occur only when execution switches from one interpreter to another via aliases, so the security policy can be encapsulated in the code that implements aliases. In the object-oriented approach, however, the boundaries between different security domains are more complex: any method invocation could cause a change in security domain, and it may be hard for either the caller or the callee to know that there has been a change in security domain. Thus more code will have to be aware of security issues in the object-oriented approach.

Said another way, the complexity of security issues in the padded cell approach grows with the number of virtual machines, which is no more than the number of principals. In most applications, this is small (typically two: the user, who is trusted, an an incoming applet, which is not). In the object-oriented approach, the security complexity grows with the number of classes, which is related to the functionality of the system.

Fortunately object-oriented systems can emulate the padded cell approach by grouping objects and classes according to their security properties and controlling invocations between these groups to simulate multiple virtual machines. We think that such an approach will simplify the security issues in object-oriented systems.

10.3 Pure authentication: ActiveX

Another approach to security is to require authentication of all applets, so that untrusted programs are never executed. In this approach no security mechanisms need to be built into the execution environment since all applets are assumed to be trusted; all that is needed is the initial authentication. The best example of this approach is Microsoft's ActiveX, which is based on pre-existing technology where it is not possible to restrict the execution of applets. It is difficult for ActiveX to implement a security model like Safe-Tcl so total authentication is the only option.

The problem with the ActiveX approach is that trust involves more than just authentication. 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. If programs are to be attached to the information exchanged on the Web, it is important to allow the programs to be executed without requiring trust.

10.4 Software-based fault isolation

Software-based fault isolation (SFI) is a technique whereby untrusted programs written in unsafe languages such as C can be executed safely[11]. SFI modifies the binary output of the compiler to insert additional instructions that check all loads, stores, and jumps and ensure that the program lives within a restricted region of memory. It also provides protected jumps into the environment, which are analogous to aliases. SFI is sometimes referred to as "sandboxing" because it forces the program to "play only in its sandbox".

Software-based fault isolation provides safety and multiple virtual machines for languages like C that do not provide these features inherently. This represents the lowest-level set of facilities on which one could build a security model like Safe-Tcl for C programs.

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 policies into well-defined modules that are distinct from both the host applications and the untrusted applets. This makes it easier to analyze the properties of a security policy, and to reuse policies.

Our experiences with Safe-Tcl have taught us two 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. In addition, authentication provides a powerful way to distribute security policies, even as part of unauthenticated applets.

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. Li Gong provided useful comments that improved the presentation in this paper. If you read this paper and give us helpful comments, your name can appear here too.

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] S. Fritzinger and M. Mueller, “Java Security”, ???

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

[6] 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.

[7] J. Levy, *Welcome To The Tcl Plugin*, `http://www.sunlabs.com/research/tcl/plugin/`.

[8] Netscape Inc., “JavaScript in Navigator 3.0”, `http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/atlas.html#taint_dg`.

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

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

[11] 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.

[12] B. Welch, *Practical Programming in Tcl and Tk*, Prentice-Hall, ISBN 0-13-182007-9, 1995.

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