A System Architecture for Flexible Control of Downloaded Executable Content

Trent Jaeger and Atul Prakash
Software Systems Research Lab
EECS Department
University of Michigan
Ann Arbor, MI 48109
Emails: {jaegert|aprakash}@eecs.umich.edu

Aviel D. Rubin
Security Research Group
Bellcore
445 South Street
Morristown, NJ 07960
rubin@bellcore.com

Abstract

We present an architecture that enables developers to build applications that can flexibly control downloaded executable content. The architecture includes an access control model for representing security requirements and a browser service for deriving application requirements from signed content messages and executing content in limited domains.

1 Introduction

The ability to download executable content is emerging as a key technology in a number of applications, including collaborative systems, electronic commerce, and web information services. Executable content is a message that contains a program that is executed upon receipt. Examples of executable content include Java applets [1], Tcl scripts [7], computational e-mail messages [2], and replicated process messages [5]. In most cases, executable content is implemented using a powerful language that can display user interfaces, engage users in a dialogue, return the results to the content provider, etc. The key features of these languages are that the content messages can be automatically downloaded to a wide variety of platforms and executed without recompilation. Thus, custom content located remotely (i.e., programs that are only to be used once) can be used effectively in applications for the first time.

Downloaded executable content provides a simple mechanism for users to execute custom applications. In the past, to execute content located on a remote site, users had to download the content (e.g., using `ftp`) and then install it properly on their machines. Therefore, users would have to manually locate the program and often compile the content. Therefore, the download process could take several minutes, so only competent users who really want the content would download it. With the advent of the World-wide Web (WWW) and these content languages, executable content can be downloaded simply by selecting the appropriate URL. The download process is now accomplished automatically in seconds, so users can easily use remotely located applications. For example, a workflow application may involve posting each activity on a web page associated with the appropriate user. The user can then select an activity's URL and the execution of that activity is initiated automatically. Since activities are generated dynamically and are unique, the activity's content must be downloaded each time the activity is run. Clearly, manual download is not sufficient for this application.

Unfortunately, there are dangers in executing content downloaded dynamically from an untrusted network, such as the Internet. For example, consider the downloaded executable content system in Figure 1. In this system, a *content provider* composes a content message containing a program. Through some mechanism (e.g., `http` or e-mail) the content is downloaded to another principal, called the *downloading principal.* The downloading principals use an interpreter process running on their machine to execute the content (number 2 in the figure). This process is owned by the downloading principals, so the content is executed with their access rights. A malicious content provider can use these access rights to: (1) read and write the downloading principal's private objects; (2) execute applications, such as `mail` or cryptographic software, to masquerade as the downloading principal to other users; and (3) read the password file on the downloading principal's machine.

To prevent these attacks, interpreters for executing

1063-5351/96 $5.00 © 1996 IEEE
Proceedings of IWOOOS '96

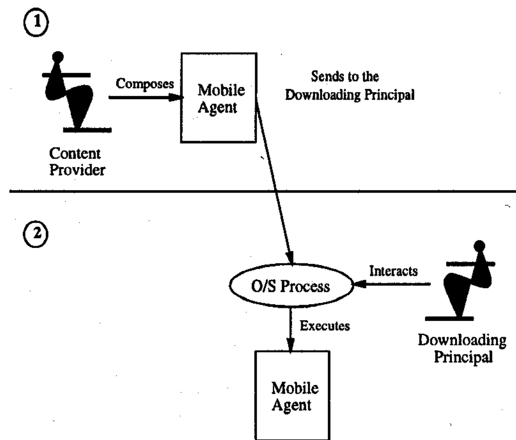

Figure 1: Downloaded content architecture

downloaded content, such as Java-enabled Netscape, Java's appletviewer [1], and Tcl's "safe" interpreter [2, 6], limit the access rights of the downloaded content so strictly that potentially useful actions are not possible. For example, content run using Java-enabled Netscape is prevented from performing any file system I/O and communicating with third parties (in principle anyway, see [3]). The Java appletviewer permits some access rights to be granted to content. Unfortunately, these rights must apply to content from all providers, so only rights for the most untrusted providers can be specified. Tcl's "safe" interpreter precludes all I/O by default. In addition, all these interpreters (as well as the Telescript engine [9]) prevent the execution of external applications. Therefore, an application can be constructed using these interpreters only if: (1) all of the downloading principal's data can be entered manually; (2) only communication with the content provider is required; and (3) no native software is needed.

However, there are many applications that have greater I/O requirements than are permitted by these interpreters. Consider the workflow example discussed above. It is likely that an activity will need data that is stored on the downloading principal's machine or will need to communicate with a principal other than the content provider. For example, assume that a downloading principal's activity is to generate a purchase order from the following forms: a credit check, a sales statement, and a delivery statement. Assume that only the downloading principal is authorized to read all these forms. The only scenario that can be implemented using present technology requires that the content provider (or the content provider's server) obtain all these forms and provide them to the downloading principal. Suppose a form already resides on the downloading principal's machine. Despite the fact the downloading principal already has the form, the content provider must execute a protocol similar to the following: (1) locate the form on the downloading principal's system; (2) encrypt it to protect its privacy; (3) download it to the content provider's machine; and (4) send it with the content back to the downloading principal. This is not only a waste of time, but it still doesn't protect the form's privacy because Java downloaded content can send the form back to the content provider once it's decrypted. Therefore, current interpreters execute this activity more slowly and without satisfying its security requirements.

In this paper, we describe a system architecture for flexible control of downloaded executable content. The architecture solves two problems: (1) determination of the access domains of content and (2) enforcement of this domain. We define an access control model to represent the access rights of *principals* (e.g., content providers, groups, services, etc.) to perform *operations* (e.g., read, write, delete, etc.) on *objects* (e.g., files, sockets, environment variables, etc.). A principal's access rights are referred as its *domain*. Access rights of content to any type of object can be defined (including application objects which are not discussed further in this paper) using this model. Derivation of a content domain involves little, direct end user specification, so spoofing attacks become less likely. However, end users can still modify a content's domain by performing actions using trusted interfaces that indirectly add or remove access rights. Content domains are enforced by a trusted interpreter. In addition, trusted systems that can represent the rights in the domain can enforce the domain on external software.

2 Problem Definition

Unfortunately, by expanding the access rights of content, new, subtle attacks become possible that can enable a malicious content provider to obtain many of the unauthorized rights we listed above. For example, if flexible communication and file I/O are permitted, then the use of unauthenticated content or poorly specified rights may lead to the leakage of information to an unauthorized principal. Worse yet, if content can write to an object that is executed (or interpreted) by the downloading principal, then a content provider can get malicious commands executed (e.g., that may inject a virus) by processes which have the downloading principals full rights. The execution of

native software presents additional problems because this software is outside of the control of the interpreter and is traditionally executed with the rights of the downloading principal.

Current content and operating systems lack the tools to flexibly control content and prevent such attacks. Below, we list our design goals for a flexible downloaded content execution system.

- **Authentication**: The source of all content that is to be granted any significant rights must be authenticated.
- **Flexible Specification**: The access control model must be flexible enough to specify any subset of the downloading principals access rights.
- **User Specification**: The domains of content should be derivable with as little end user specification as possible. However, end user actions may result in the delegation of rights.
- **Use Existing Rights**: Traditional file system access rights should be used wherever possible to reduce end user specification.
- **Comprehensive Control**: Access to all system objects must be controlled. These objects include: files, communication channels, URLs, environment variables, interface objects, and CPUs.
- **Dynamic**: Content access rights may evolve as the downloading principal performs actions that result in the delegation or restriction of rights.
- **Transferrable**: The rights may be transferred to principals in other systems (i.e., operating systems) that are trusted to enforce these rights on other processes on those systems.

3 Solution Description

We make the following assumptions about the system architecture. First, we assume the existence of a public key infrastructure that be used to securely obtain the public key of any principal. Thus, any principal can verify the source and integrity of a message signed with a private key. Next, we assume that we can identify any I/O commands in the content language. This is necessary to control access to system objects. Also, we assume that the operating system has an unmodified trusted computing base, protects process domains, and provides authentication of principals. This ensures that system software, such as cryptographic software, can be trusted, processes can only interact in controllable ways, and authentication of principals is possible. Without trust in the operating system, it is not possible to build trusted applications that run on that operating system. A secure operating system, such as Trusted Mach [8], is designed to satisfy these requirements.

Given these assumptions, our system architecture is shown in Figure 2. The architecture consists of a hierarchy of logical processes each with a successively smaller domain of access rights (from bottom to top in Figure 2): (1) operating system; (2) browsers; (3) application-specific interpreters; and (4) content interpreters. Starting at the bottom the processes with the most rights is the *operating system*. The operating system is a privileged principal that can perform principal authentication and system object operations, including process execution, remote communication, and file I/O. The *browser* is a process trusted by the downloading principal to execute content safely. The browser gets content authenticated, assigns content to an appropriately limited interpreter, and authorizes content interpreter actions on system objects, and transfers access control information to the operating system. *Application-specific interpreters* are designed to implement a specific application. These interpreters have a predefined domain that is assigned (typically by system administrators) based on the trust in the developer of the interpreter and the domain requirements of the interpreter. However, these interpreters may be granted additional rights by the downloading principal. *Content interpreters* execute the downloaded content in a limited domain.

The protocol for executing content using this architecture is shown in Figure 3. The content provider sends a content message to the downloading principal's browser. This message includes the content, the content type, and any authentication/encryption information. The browser verifies the identity of the content provider, the integrity of the content and content type, and the freshness of the message (if necessary). If the verification succeeds, the browser determines which interpreter to execute the content. If the type refers to a known application-specific interpreter and the content provider has the rights to execute content in that interpreter, then the content is executed in that interpreter. If there is no type specified, then the content is run in an interpreter for that provider. If content attempts to perform an operation on a system object, this operation is authorized by the browser using the content interpreter's domain. If the operation executes a new process or uses a network service, the browser communicates the limited rights to the oper-

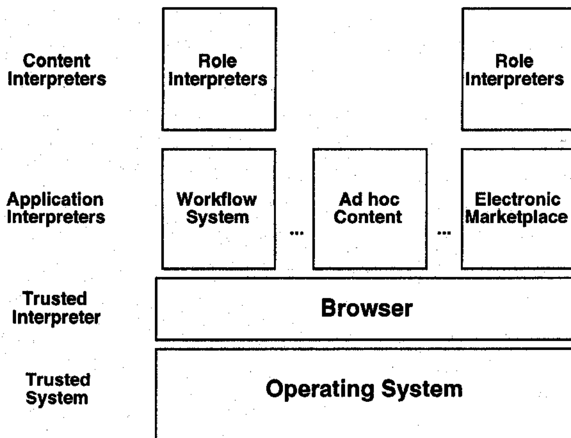

Figure 2: **Architecture**: (1) *Trusted System* contains trusted system services, such as authentication and system object operations, upon which the architecture is built; (2) *Trusted Interpreter* is a general-purpose browser that enforces access control on downloaded content; (3) *Application Interpreters* store shared state, control content's access to it, and define content provider roles; (4) *Content Interpreters* execute downloaded content. Note that the *ad hoc interpreter* executes content that is not associated with a known application, so its domain is based on the content provider's permission to execute arbitrary content.

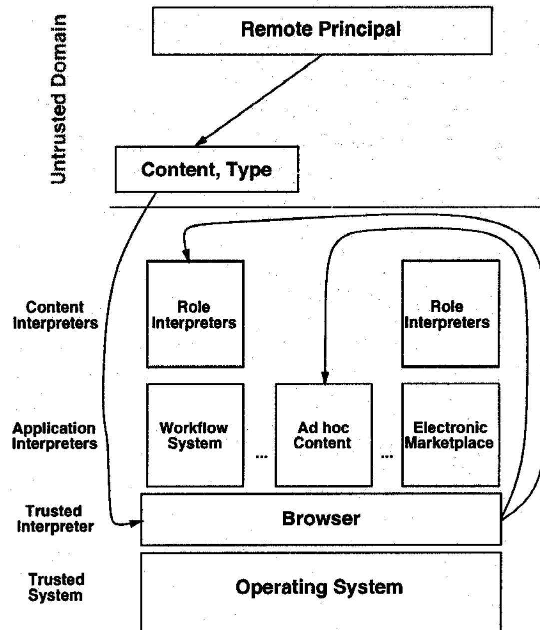

Figure 3: **Protocol**: Content is downloaded to the *browser* which authenticates the content provider and finds the appropriate *application-specific interpreter* to execute the content. If the content provider is authorized to execute content in that interpreter, then it executes the content in the content provider's *content interpreter* for that application.

ating system that can control the that process.

The access rights of interpreters are specified by domains. A principal's domain is specified by listing the operations that the principal can perform on object groups and exceptions to these rights [4]. The relationships in the access control model are shown in Figure 4. Object groups specify sets of objects, so a class hierarchy is useful in implicitly defining sets by objects of a type. For example, access to sockets can be limited to a host and port number whether the socket channel is authenticated or not. In addition, the access control model enables us to make statements as to whether a principal is ever allowed to execute a class's operation (or conversely , always). At present, rights to file system objects, environment variables, and URLs can be specified at present. In the future, we plan to specify domains for interface and CPU usage.

The domain of a content interpreter is the intersection of the application interpreter's domain, the content provider's domain, and the domain of the role that the content provider assumes in the application. The first two domains are specified by system administrators and the third by the application (it's a subset of the application's domain). An application role defines the rights the content provider needs and is trusted with in the application. For example, an active collaborator may be able to modify all shared state, but an observer may not be able to modify any shared state. The downloading principal determines what state is shared by performing application actions that "load" objects into the shared state. For example, the downloading principal uses the application interface to load a file into a domain shared by a role in the application. Since the browser controls interpreter access to system objects, it is asked to perform this action. The browser is executed with the downloading principal's domain, so it can engage the downloading principal in a dialogue (in a trusted window) that could result in any of the downloading principal's files being loaded (i.e., the rights being granted to the role and the application interpreter). Granting of rights normally outside the content provider's domain, such as $1000 of digital cash, are possible using this mechanism, but additional security mechanisms may be required to prevent users from leaking large amounts of money or extremely sensitive objects ac-

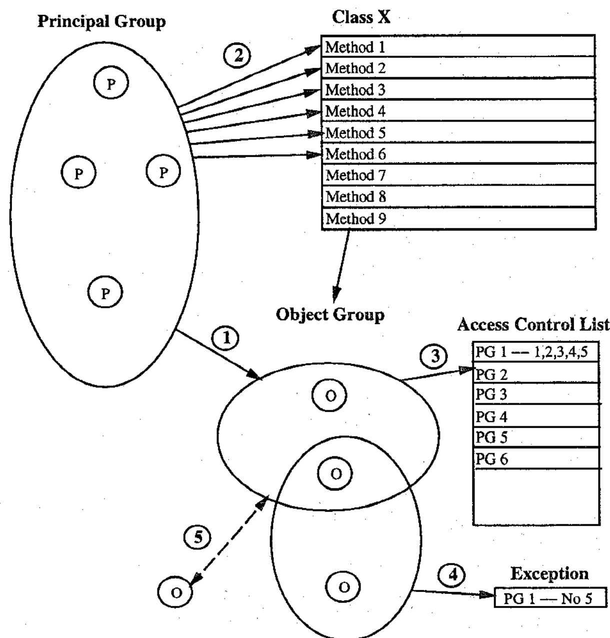

Figure 4: Access control relationships among entities: (1) principal groups can execute operations on objects that belong to classes (class operations); (2) principal groups can execute operations on objects in object groups (accessible objects); (3) ACLs specify principals' abilities to execute operations on the object group, but there can be exceptions (object access rights); and (4) an object can join or leave a group (transform).

cidentally. Writing executables or creating executable objects is prevented by domain specification and preventing content from writing objects that can be executed, respectively. Unauthorized leakage via I/O channels is detected by determining if the content provider's write domain intersects with an unauthorized principal's domain.

4 Conclusions

We defined a system architecture to flexibly control downloaded executable content (i.e., Tcl scripts, Java applets). Our goal is to enforce fairly arbitrary access control domains for content to remove limitations on the kinds of applications that can be constructed. Removal of these limitations means that a number of additional avenues of attack, such as user spoofing, information leakage, and software execution, must be closed. Our system architecture provides a flexible access control model to enable arbitrary and dynamic specification of content access rights. To prevent attacks the architecture uses content authentication to identify content providers, a trusted system administrators and application developers (within their domain) to specify content domains with limited end user involvement, a comprehensive access control model to ensure that all system object operations can be authorized, and leakage detection to ensure that objects are not leaked or additional access rights are not obtained by the content. Although efficiency is not proven, by providing concise domain specifications, authorization and intersection of rights should have reasonable performance, but further analysis is needed.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, pages 389–402, 1994. Available via anonymous ftp from ics.uci.edu in the file mrose/safe-tcl/safe-tcl.tar.Z.

[3] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[4] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, pages 131–148, July 1996.

[5] M. Knister and A. Prakash. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems – The Journal of the Usenix Association*, 6(2):135–166, 1993.

[6] J. Levy and J. Ousterhout. Safe Tcl: A toolbox for constructing electronic meeting places. In *The First USENIX Workshop on Electronic Commerce*, pages 133–135, 1995.

[7] J. Ousterhout. Tcl and the Tk toolkit, 1994.

[8] Trusted Information Systems, Inc. *Trusted Mach System Architecture*, TIS TMACH Edoc-0001-94A edition, Aug. 1994.

[9] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper.