SECURING SYSTEMS AGAINST EXTERNAL PROGRAMS

BRANT HASHII, MANOJ LAL, RAJU PANDEY, AND STEVEN SAMORODIN
University of California, Davis

Although the practice of executing external programs is widespread, the security implications have yet to be systematically analyzed. The authors address this problem here, offering a resource-centric classification of security issues and solutions.

Internet users routinely and often unknowingly download and run programs, such as Java applets; and some Web servers let users upload external programs and run them on the server. Although the practice of executing these external programs has the sanction of widespread use, its security implications haven't yet been systematically addressed. In the brief, dynamic history of the Internet, such a situation is not unusual. New communication mechanisms and computing paradigms are often implemented before the security issues they engender have been rigorously analyzed.

Our goal here is to address this problem in the subdomain of external programs by systematically outlining security issues and classifying current solutions. Our focus is solely on protecting a host from external programs. We do not address the problem of protecting the communication medium or protecting an external program from runtime systems. Furthermore, we do not address the problem of correctly identifying the source of an external program (authentication).

We start our inquiry by reviewing the relevant models of computation, followed by an overview of the security problems associated with them. We then classify both the problems and the existing solutions using a resource-centric model that distinguishes problems associated with resource access from those associated with resource consumption. Finally, we classify solutions to each problem according to how and when they are applied.

EXTENSIBLE COMPUTING MODELS

In computing models that support program migration, a host provides a set of services to an external program by loading it and executing it within a local execution environment such as an operating system or a run-

 1089-7801/98/$10.00 ©1998 IEEE

time system. We call such computing models *extensible computing models*. Two examples of execution environments that support program migration are extensible operating systems and mobile programming systems.

Extensible operating systems allow the collocation of external programs by loading them directly into the kernel's address space. Thus, the external program shares its address space with the kernel and all other loaded external programs. SPIN[1] is an example of an extensible operating system that lets users download user-level extensions into the kernel.

Mobile programming systems also support migration by letting users upload programs to a remote host. Further, external programs can stop in mid-execution and then migrate to another host while retaining their state and data.

Since external programs run within the same name space as the runtime system, many of the traditional protection mechanisms no longer apply.

In addition, Web browsers such as Netscape support extensibility by letting applets be downloaded and executed within the browser. All of these systems provide an execution environment that loads externally defined user programs and executes them within its local name space. We refer to these execution environments as runtime systems.

Extensible Computing Benefits

The extensible computing model is appealing, first, because it lets operating system kernels implement only basic, core functionality, which can then be extended through external programs. This facilitates customization and efficient implementation of specific services and policies, such as application-specific memory management or caching policies.

Second, external programs are sometimes far more efficient at utilizing network bandwidth than traditional programming paradigms, such as remote procedure call (RPC). For example, if the external program encodes an application that must filter huge amounts of data, it can migrate to the host with the data, execute there, and then return with its results to the originating host. In this case, the network load incurred by migrating the external program is insignificant compared with the cost of migrating data and processing it locally.

Security Risks

Although appealing from both system design and extensibility viewpoints, runtime systems are extremely vulnerable to misbehaving external programs. Since external programs run within the same name space as the runtime system, many of the traditional protection mechanisms, such as address-space containment, no longer apply. As a result, an external program can maliciously disrupt an extensible system by interfering with the runtime system's execution or with the execution of other programs within the name space. It might also access unauthorized resources, use more than its fair share of resources, and even deny resources to other programs.

CLASSIFYING RUNTIME SERVICES

To help classify security issues, we distinguish two types of resources that a runtime system provides to external programs:

- *System resources* are those implicitly allocated to external programs, such as memory and CPU.
- *Conceptual resources* are those explicitly defined and managed by a host. They have well-defined interfaces that external programs use to access resources and request services. For example, a host might provide an interface to a database repository.

This distinction highlights fundamental differences in the mechanisms used to control resource access:

- Because system resources are implicitly allocated and managed by runtime systems, the mechanisms for access control are usually implemented within the runtime system. They also depend on the resource model that runtime systems create. For example, CPU resource access control is traditionally implemented in runtime systems through CPU scheduling algorithms.
- Because conceptual resources are accessed by explicit calls from external programs, access control is generally based on trapping these calls in software. For example, in the Java runtime system (JRTS),[2] calls to protected resources are

trapped when these resources call a reference monitor, which the host uses to track and control accesses.

We use this distinction in our classification of security approaches. As we describe below, there are three phases during which security can be implemented: *program-development*, *migration*, and *execution*. Each security approach varies both by phase and by resource type.

RESOURCE-CENTRIC SECURITY PROBLEMS

Security problems occur when an external program tries to access and consume unauthorized resources at the host site.

- *Access control* refers to restricting a program's access to system and conceptual resources. For system resources we focus on limiting a program's ability to access other programs' memory resources; for conceptual resources, we focus on limiting access to resources with explicit interfaces.
- *Consumption control* refers to restricting how much of a given resource a program can consume.

Our primary focus here is on those aspects of resource-access and consumption control that are specific to the extensible computing model.

Controlling Resource Access

Access control prevents an external program from using unauthorized resources. Because external programs execute within the runtime system's name space, they can directly access any resource by naming it. Naming involves getting a resource handle and using it to invoke operations on the resource. An external program can use the handle to read sensitive files and send the information to remote hosts by accessing network resources. It can also disrupt the operations of a computer system by accessing local resources in an unintended manner. For example, the "Ghost of Zealand" Java applet misuses the ability to write to the screen: It turns areas of the desktop white, making the machine practically useless until it is rebooted (for full details, see http://www.finjan.com/applet_alert.cfm).

Limiting an external program's ability to read and modify the memory resources of other programs, including the runtime system, is an important part of access control. There are two aspects of memory access control:

- *Safety* limits an external program's ability to write into another program's name space, and thus limits opportunities to corrupt system-dependent data, crash a program by forcing it into an inconsistent state, or rewrite other programs to behave in an unintended manner.
- *Privacy* prevents an external program from reading another program's address space and thus limits opportunities to learn passwords and other sensitive information.

Access to conceptual resources poses problems similar to those posed by access to system resources. For example, once downloaded on a machine, external programs can read private files from local disks or copy proprietary information by accessing databases. Indeed, Hamburg's Chaos Computer Club demonstrated on German television how to use ActiveX, Microsoft's external programming system, to steal funds. In this exploit, the victim uses Internet Explorer to visit a Web page that downloads an ActiveX control. The control checks to see if Quicken, a financial management software, is installed. If it is, the control adds a monetary transfer order to Quicken's batch of transfer orders. When the victim next pays the bills, the additional transfer order is performed. All of this goes unnoticed by the victim (for more on this, see http://www.iks-jena.de/mitarb/lutz/security/activex.hip97.html or http://www.iksjena.de/mitarb/lutz/security/activex.en.html).

Although the system and conceptual resource access control problems have been studied within the context of traditional systems, the problems are different for extensible systems in several ways.

First, in extensible systems, authorization is more complex. In traditional operating systems, programs run on behalf of principals who are given certain rights. Once a program attains these rights, they usually remain valid during the program's execution. In extensible systems, however, an external program contains individual components that might have different rights and permissions. Hence, the level of granularity at which access rights must be checked and enforced is much finer— sometimes at the level of individual objects and functions.

Second, security mechanisms should be independent of the site's resources. Most traditional operating systems manage a fixed set of resources such as memory, CPU, files, and the network. Extensible systems, however, must manage resources that can vary from site to site.

Finally, most traditional operating systems implement an access control model that either allows or denies access. Extensible systems can allow conditional resource access based on runtime or program state. For example, a database vendor can specify that if there are more than 20 external programs in the system, each external program can only access its database up to 10 times.

Controlling Resource Consumption

When external programs use more than their share of resources, they leave the system vulnerable to attack. For example, external programs can stage denial-of-service attacks by intentionally over-using CPU, thereby denying CPU to other programs and the runtime system. Consumption control is required not only for system resources, but also for conceptual resources. For example, by opening multiple socket connections and flooding the network with data, an external program can deny network resources to other programs.

The resource-consumption problem is similar to the CPU-scheduling problem in that both require the runtime system to control the resources allocated to requesting programs. The two problems differ in the type of control that runtime systems need to exercise. The primary goal in CPU scheduling algorithms is to allocate CPU so it provides some quality of service (QoS) to executing programs. The QoS requirements can be specified as constraints, such as optimal resource utilization, response time, lower bounds on resource allocation, and deadline. Runtime systems, in addition to providing CPU scheduling, must control resource allocation in a way that satisfies the host-defined consumption constraints. Thus, the resource-allocation problem for extensible systems includes additional variables:

- *Combination of QoS constraints.* External programs can originate from different sites and so might have different kinds of QoS requirements.
- *Upper bounds*. To prevent denial-of-service attacks, runtime systems specify and enforce upper bounds on external programs' resource consumption.
- *Lifetime constraints.* External programs can circumvent a host's resource-consumption constraints by using resources, migrating to a different host, and then returning to the target host to use more resources. This engenders the need for what we call lifetime constraints on all executions of the external program.
- *Trust and preference.* The host's trust will vary for different external programs and might depend, for example, on the program's site of origin. The host has preferences on resource allocation based on trust or other factors, such as the kinds of activities the external programs are performing.
- *Dynamic nature of trust and preference.* Trust level and preferences can be dynamic, and can change with the host's system state. For example, if a particular site sends numerous external programs to the host, it could be attempting to stall the system and thus its trust level might be reduced.

COMPONENTS OF RESOURCE-CENTRIC SECURITY

There are two components to security solutions: policy specification and policy enforcement. An organization must be able to specify the behavior allowed on its systems. The mechanism for precisely enforcing the policy must also be available. A policy enforcement mechanism without a well-thought out policy might end up enforcing constraints that are too restrictive or too lenient. Likewise, a security policy without the ability to enforce it is useless.

Policy Specification: Implicit or Explicit

A security policy is a set of access constraints that specifies who can access resources, which resources they can access, and how much of a given resource they can use.

We can divide security policies into two kinds: host-independent and host-specific. Host-independent policies are the fundamental and primitive security policies that all extensible systems must enforce. Foremost among them are safety and privacy. A runtime system must provide mechanisms for isolating different external programs to prevent them from reading and writing other programs. Host-specific policies, on the other hand, can vary from site to site.

Security policies can be specified implicitly or explicitly. Implicit mechanisms are typically part of the system implementation. For example, CPU scheduling algorithms implicitly implement CPU consumption control. Another example is the sandbox policy implemented in Web browsers using early versions of JRTS.[3] In this policy, remote applets were not allowed to access local files or to open network connections to hosts other than their originating site.

The problem with implicit policy specification is that it creates specific, inflexible, and uncus-

tomizable systems. However, implicit policies can express host-independent security policies (such as safety and privacy) more naturally since they do not require any additional effort. They also capture certain security guarantees that all systems must provide. For example, the JRTS version 1.2 implements the policy of least privileges, which ensures that programs with less privilege do not acquire more access rights by making calls to privileged components.

Systems that let users specify their policies explicitly use a policy language for specification purposes. For example, explicit access control policies in SPIN are specified using the domain-type-enforcement language. Similarly, JRTS version 1.2 uses a simple language for associating various principals and permission rights with different applets.

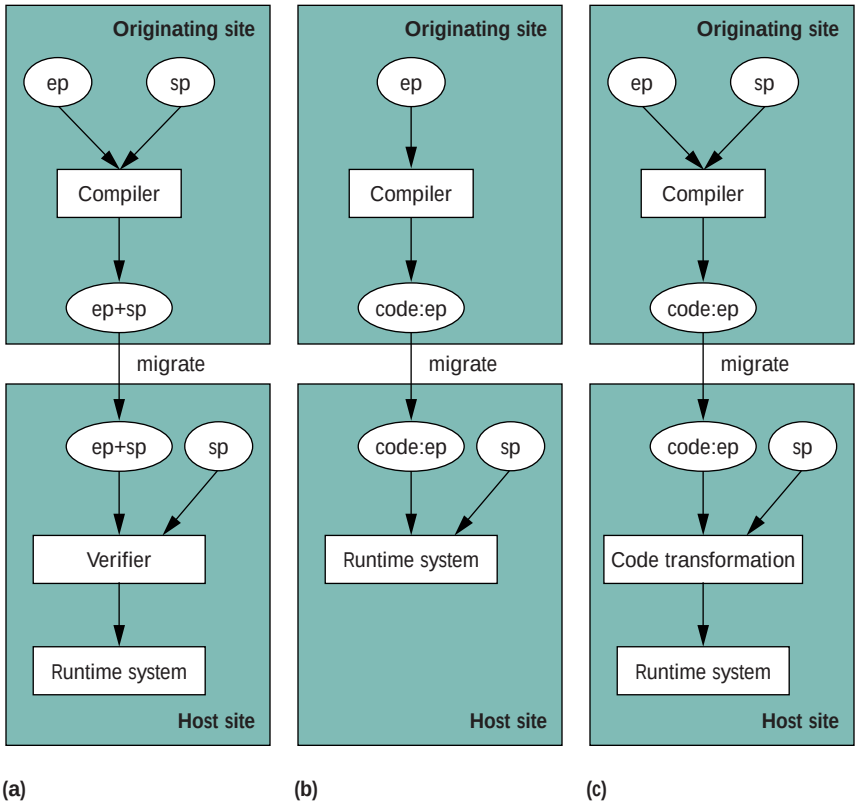

Figure 1: Security enforcement during different phases: (a) program development phase, (b) program execution phase, and (c) program migration phase. (ep) denotes an external program and (sp) denotes the host site's security policies.

Policy Enforcement: Static or Dynamic

Policy enforcement consists of identifying and responding to actions that might violate the security policy. Violations can be caught either statically or dynamically. Static enforcement mechanisms examine a program's code to ensure that the program does not enter a state where a security violation can occur. This verification occurs before the program executes. Runtime systems can also statically schedule or allocate resources so that resource-consumption constraints are met. Static checking is incomplete in that it cannot be used to identify all security violations, especially those that are based on dynamic properties. For such cases, dynamic checking is needed.

In dynamic checking, a wrapper is placed between a program's request code and the resource access code to determine if the request should be satisfied at runtime. A wrapper is interposition code that can be inserted by the programmer, generated by the compiler, or defined by the runtime system via special system calls. Also, a separate tool (such as software to isolate faults[4]) can examine a program statically and insert dynamic checks where needed. Because static checking reduces runtime checks, thereby improving execution time, it should be used whenever possible. Static checking also catches problems prior to execution, which prevents the runtime system from having to deal with policy violations in mid-execution.

SECURITY SOLUTIONS FRAMEWORK

As Figure 1 shows, we classify solutions according to when they are applied to an external program—during the program development, migration, and execution phases.

Program-Development Phase

In the program-development phase, developers at the originating site create different program components, such as classes and methods. The external program is then compiled to create a transportable representation. Security policies are enforced in this phase using a tool, such as a compiler. As Figure 1a shows, an external program (ep) is compiled along with a security policy (sp). The compiler not only generates an intermediate form of the program, but

it also checks for security policy violations. Further, it generates code for checking and enforcing the policy during an execution.

The program (ep+sp) is migrated to the host site and executed there. Because a level of distrust exists between the originating and host sites, the program (ep+sp) cannot be trusted to enforce the security policy. Thus, the runtime system must include a verifier to ensure that the migrated code enforces the policy. A partial solution is to cryptographically sign the code at compile time and include a checksum. The runtime system can then authenticate the code by checking its signature and verify its integrity via the checksum before executing it.

The runtime system must include a verifier to ensure that the migrated code enforces the security policy.

The problem with this solution is that it only asserts that the code is from a trusted site. It does not guarantee that the external program will not cause security problems by inadvertently reading or writing other programs' memory locations or going into an infinite loop and consuming all CPU resources.

Another solution is to rigorously verify that the external program does not violate the security policy. For example, the JRTS uses a bytecode verifier to ensure that the migrated code satisfies all safety policies as specified in the Java programming language. However, this solution also has limitations. Since the bytecode verifier is extremely complex, it is unclear how to ascertain if a particular implementation is correct; implementations of the Java bytecode verifier could be tricked into executing unsafe code.[3]

Language-based solutions are appealing for many reasons. Because the compiler can examine the source code, it can statically check for many kinds of security violations prior to execution. Further, the compiler can use the program semantics to generate efficient code for dynamic checks.

The main problem with language-based approaches is that they assume the availability of security policies. However, policies are often unavailable for several reasons. First, external program destinations are sometimes unknown and thus it is not always possible to compile them with the right security checks. Second, even if the destination sites are known, external programs can visit many different hosts with different security policies. Although multiple sites could send their policies to the originating site, this solution is too cumbersome as it encounters problems that rival those seen with cryptographic key distribution. However, for host-independent policies, language-based approaches work well—provided that the host site can verify that the code follows a security policy.

Program-Migration Phase

Figure 1b shows how security policies are enforced after an external program arrives at a site and before it executes. In this approach, a security-enforcement tool examines the external program's intermediate form to determine if there are any possible security policy violations. It then inserts additional security check-code in the migrated program such that its execution doesn't violate the host's security policies.

Security enforcement mechanisms applied during this phase are appealing in that they can enforce many security policies independent of existing tools such as compilers or runtime systems. The approach can therefore be used to add a security layer to legacy systems. The separation of policy specification from policy enforcement also allows different components (programs, policies, and resources) to be easily modified without affecting each other.

The problem with these mechanisms is that they have additional processing and code-modification costs. In addition, if you use them in a mobile programming system to enforce host-specific policies, you incur additional post-processing costs to undo the modifications so that the external program can migrate to another host. This performance cost is unexplored and is a topic of ongoing research.

Program-Execution Phase

Solutions in this phase enforce security policies as the program is executing. As Figure 1c shows, the runtime system checks for policy violations by monitoring the external program's behavior. Monitoring may involve intercepting different function or system calls, or tracking the external programs' resource usage.

Monitoring can be done in two ways. In the first method, the runtime system can monitor a fixed number of system calls. Here, each call performs the check. This method, primarily used in traditional operating systems, is not extensible because to change policy you must change the system call itself. Also, adding new system calls or resources is difficult.

The second method forces each protected resource to call a reference monitor. In this approach, you can change policy by changing the reference monitor. You can also easily add new protected resources. JRTS uses this approach for supporting customizable security policy specification and enforcement.

The main advantage of this phase is that enforcement mechanisms have access to the runtime state of the system and the program. This allows the enforcement of policies based on state information. For example, the JRTS uses protection domain information stored in the execution stack to enforce the policy of "least privilege."

The disadvantage of this phase, in the case of access control, is that enforcement is by necessity a runtime mechanism. Since it generally doesn't use the program's semantics, it might perform unnecessary access checks.

EXISTING SOLUTIONS

We now present some solutions for the security problem. Many of these techniques have been widely studied in relation to general security; here we focus on techniques as applied to external-program security.

Memory Access Control

A memory access control mechanism has two components: the memory model and the safety check. The memory model specifies how an extensible system's name space is partitioned into safe and unsafe regions. The safety checks ensure that every read or write refers to a memory location in a safe region. Given that every memory access must be verified, access control mechanisms must implement the safety checks efficiently. There are several different approaches to defining memory models and executing safety checks.

Development-phase solutions. Solutions employed in this phase rely on the features of safe programming languages such as Java and Modula-3. The features include strong typing, restricted memory-reference manipulations, language runtime-supported memory allocation and deallocation, and dynamic checking.

In the language-based approach, the memory model is derived from the language-type system. All memory locations created using rules for instantiating variables denote safe regions. All other memory locations are unsafe. Safety checks in these approaches take place in two steps. First, a compiler statically analyzes a program to ensure that most accesses are safe. It does this by applying the language semantic rules; these ensure that a program cannot create a reference to a memory address in an unsafe region. Second, in cases where the compiler cannot statically guarantee the safety of a memory access (such as array accesses), the compiler generates runtime checks.

SFI provides protection against untrusted programs by restricting the memory locations they can access.

Many extensible operating systems and mobile-code runtime systems use the language-based approach for memory protection. For example, the SPIN extensible operating system relies on Modula-3's safety mechanisms to load application-specific kernel extensions into the kernel's address space.[1] Similarly, the JRTS relies on the Java programming language's safety mechanisms when downloading Java applets.

Language-based solutions support a fine-grained safety model in that they can implement memory access control for individual data structures. Also, because many safety and privacy policies can be enforced at compile time, safety checks can be employed efficiently. A limitation in language-based solutions is that external programs can be written only in a specific language. Further, the approach requires a verifier to ensure that external programs follow type safety rules.

Migration-phase solutions. Software fault isolation (SFI)[4] supports memory access control among trusted and untrusted extensible system components during the migration phase. SFI provides protection against untrusted programs by restricting the memory locations they can access.

SFI creates a memory model by partitioning the address space of the extensible system into logically separate address spaces, called *fault domains*. Each untrusted component, including its code and data segments, belongs to a unique fault domain. This fault domain is the untrusted component's safe region; all addresses outside it are its unsafe region. The safety checks ensure that component instructions can jump or write only to addresses inside the fault domain. They do this by statically verifying the targets of most store and jump

instructions. For instructions whose targets cannot be verified statically, SFI modifies the program code by either inserting wrapper code around the instructions or by sandboxing them with modified target addresses.

SFI includes a very efficient mechanism for supporting memory protection and RPC among trusted and untrusted modules. Further, it does not depend on the language used to specify external programs. SFI is used in VINO[5] for supporting memory access control. Vino is an extensible operating system that uses SFI to ensure that user-level kernel extensions do not interfere with the kernel's operation. While SFI does support mechanisms for sharing and communication among trusted and untrusted components, complex sharing and memory access relationships among fine-grained components are difficult and expensive to implement.

Execution-phase solutions. For memory access control in the execution phase, solutions rely mainly on the underlying hardware and are implemented within operating systems.

Traditional operating systems enforce memory access control by encapsulating each program within an address space and ensuring that the program's access is limited to that space. In these solutions, the memory model partitions the logical address space of a program into a set of pages. Safety checks come via hardware mechanisms that check every read and write instruction for validity. Any access to an unsafe region causes a trap in the kernel and raises an exception.

Hardware-based solutions are appealing for their simplicity and the safety net they provide. Also, unlike the language-based approaches, hardware-based approaches have a small trusted computing base (TCB) that can be verified for correctness with relative ease and they are independent of the external program's specification language.

Although hardware-based approaches provide adequate and efficient support for safety at a coarse-grain level, it is not yet clear if they can support efficient fine-grained memory access control.

Access Control for Conceptual Resources

Solutions for controlling access to conceptual resources involve identifying resource use, determining if access is allowed, and possibly denying it. Solutions differ in how these checks are performed and the flexibility with which checks can be added or changed.

Development-phase solutions. Proof-Carrying Code[6] is one of the most promising techniques for controlling conceptual resource access. In PCC, the host site publishes its security policy and the originating site combines proof of the external program's compliance with the executable to create a PCC binary. When the program arrives, the host site validates the proof.

Two properties of PCC make it a very appealing approach. First, the PCC binary is tamper proof. Changes in the binary often result in a validation error. In cases when there is no validation error, the code is guaranteed to be safe. Second, proof checkers are essentially type checkers, which are efficient and easy to implement.

Although PCC has been used mainly to prove memory safety policies, it can be used for conceptual resource access constraints as well. However, it is not practical for host-dependent policies, where a different proof is required for each site's security policy. Also, it is not clear if the approach is scalable with respect to the size of external programs and security policies. Finally, most safety proofs must be generated manually at this point, which limits the method's practicality.

Migration-phase solutions. The Ariel Project[7] uses program transformation to control access to conceptual resources in this phase. The Ariel approach has two components. The first is a policy language that lets a host site specify the conditions under which requests for resources can be accepted or denied. The second is a program transformation tool that generates code to ensure that access constraints are satisfied. This code is then patched into the external program and the resource that is accessed.

The advantage of this approach is that it can be used to complement security components implemented by compilers and runtime systems.

Execution-phase solutions. There are several approaches to enforcing access control security during the execution phase, including Safe-Tcl[8] and JRTS.

Safe-Tcl supports a flexible and extensible access control mechanism. It requires at least two interpreters: a regular (or master) interpreter for trusted code and a limited (or safe) interpreter for untrusted code. When untrusted code requires access to a system resource, it traps into the regular interpreter, which then decides whether to allow the access. The Safe-Tcl designers classified a set of instructions as unsafe and then disabled those instructions

in the safe interpreter. A security policy is specified by aliasing the safe interpreter's disabled instructions to the master interpreter's procedures. These procedures can then check arguments and, if the security policy allows, call the masked instruction in the master interpreter.

In the JRTS version 1.1, the site manager defines resource access constraints by implementing a SecurityManager that enforces local access control policies. In addition, each protectable resource includes explicit calls to the security manager. In this security model, you must implement the SecurityManager correctly. Furthermore, JRTS does not include the notion of protection domains. If you want to allow different constraints for different external programs, you must build an addition security infrastructure. As a result, early Web browsers did not use protection domains and instead implemented a sandbox policy that prevented all Java applets from accessing certain local resources, like files.

The new Java security model[2] addresses many of these concerns by providing runtime support for fine-grained access control and configurable security policies. The new model augments the SecurityManager with an AccessController, which is primarily responsible for deciding if a program can access a resource. Each site specifies its access control policy by creating protection domains and a set of permitted actions for programs belonging to each domain. Each resource includes a call to the AccessController to check if a specific action is allowed. An important component of the new Java security model is the *policy of least privileges*, which prevents external programs from gaining privileges by crossing protection domains. To accomplish this, the new JRTS checks the activation-record privileges of the current thread's stack. This ensures that protection domains do not acquire additional access rights by invoking calls to privileged protection domains. The new JRTS also supports least-privilege overrides in that you can let a domain assume responsibility for whatever it calls and ignore its callers' privileges. There are several other extensions to the JRTS security model, described in the box, "Extending JRTS."

Resource Consumption

With system resources, consumption control focuses on memory and CPU usage. For conceptual resources, the consumption control model depends on the resources involved. For example, mechanisms for controlling network bandwidth will differ from those for controlling disk space. Much of the work in resource consumption has focused on CPU resource scheduling. We briefly describe this work and discuss what changes need to be made to CPU scheduling algorithms so they can be applied to enforce consumption control for various resources.

EXTENDING JRTS

Versions of the Java runtime system (JRTS) prior to 1.2 did not contain the mechanisms necessary to easily implement complex access control policies. As a result a number of projects have expanded the JRTS's security infrastructure.

The *J-Kernel project*[1] extends the JRTS security model by implementing multiple protection domains within a single Java virtual machine. Resource capabilities are stored in a system-wide repository; domains access it to look up capabilities. These capabilities are implemented as wrappers that provide the bookkeeping associated with changing protection domains. The J-Kernel is implemented entirely as a Java library. J-Kernel also provides additional security functionality through the ability to revoke capabilities. In it, resources can be deleted without leaving dangling references and thus, when a domain terminates, all of its capabilities can be revoked and its memory freed.

Another capability system, used by Netscape Communicator version 4, is *stack introspection*.[2] In this approach, capabilities are shared not by passing them as parameters, but by passing them through the call-stack. Here an external program calls a method enablePrivilege(resource), which consults a policy engine and, if access to the resource is allowed, makes an annotation on the call-stack. The resource then calls the method checkPrivilege(resource), which searches the call-stack for the enable privilege annotation. Programs can disable and revoke privileges, which lets an application dynamically change access control policies.

Another way to extend JRTS's flexibility is to change how it views classes. Type hiding[2] prevents an external program from knowing that protected resources exist by modifying the dynamic linking process to hide classes, such as the file class. Type hiding also allows a class to be replaced with a proxy class that can check the arguments of the invoked method and conditionally throw an exception. It can also modify the arguments to fit the necessary condition. For example, if the policy says that all file writes must occur in the "tmp" directory, the proxy class could prefix the appropriate path to the file name.

REFERENCES

1. C. Hawblitzel et al., "Implementing Multiple Protection Domains in Java," *1998 USENIX Tech. Conference*, Usenix Assoc., Berkeley, Calif., 1998; available to Usenix members at http://www.usenix.org/publications/library/proceedings/usenix98/hawblitzel.html.
2. D. Wallach et al., "Extensible Security Architectures for Java," *Proc. 16th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1997; available at http://www.cs.princeton.edu/sip/pub/sosp97.html.

CPU algorithms and schemes. CPU scheduling consists of two parts: an algorithm and a scheme. A scheduling algorithm is a specific solution to a scheduling problem. A scheduling problem consists of a set of applications with similar constraints, such as deadlines for real-time applications. The scheduling algorithm's aim is to schedule a set of applications in a way that meets their constraints. An example here is the earliest-deadline-first algorithm (EDF).[9] To schedule applications with different constraints—and thus different scheduling algorithms—you need a scheduling scheme to combine the different algorithms. For example, to schedule real-time and interactive applications, the scheme[10] combines two algorithms: EDF for real-time applications and multi-priority based round robin for interactive applications.

CPU scheduling algorithms and schemes can be used to control resource consumption for external programs.

Extending CPU algorithms. Because external programs originate from different locations, their resource requirements differ. Further, host sites export different kinds of resources with different resource-usage constraints. Thus, you need a scheduling scheme to control resource allocation in extensible systems. The CPU scheduling algorithms and schemes can be used to control resource consumption for external programs. However, they do not provide mechanisms for limiting resource usage, and must be extended to take care of additional variables such as upper bounds, lifetime constraints, trust, and preference.

Controlling resource consumption. Market-based resource control and Vino both provide mechanisms for controlling allocation of resources to external programs.

Market-based resource control[11] is modeled on the market-based economy: External programs carry currency with them, and resource owners set prices on resource use. To buy resources, external programs interact with the resource owners and a network of banks. While such a scheme takes care of lifetime constraints, an external program can cause denial-of-service attacks if it has a lot of cash. Also, because the cost for resources is uniform for all external programs, the host cannot control the allocation based on its preference or trust of a particular external program.

VINO[5] supports resource-consumption control by preventing external programs from hoarding resources. It encapsulates each external program's invocation in a transaction. By recording all changes to the system state, an external program can be aborted at any time by undoing the transactions. Vino aborts external programs when they use too much of a resource or hold it too long.

Future work in resource-consumption control might involve developing ways to extend the existing scheduling schemes and the algorithms to control external programs' resource consumption.

CONCLUSION

Our framework classifies solutions based on when a security policy is enforced. Language-based approaches are applied during the program's development phase. They represent the best method for implementing host-independent security policies such as memory access control. Migration-phase approaches enforce security by inserting interposition code in the migrated program before it is downloaded into a runtime system. These approaches separate policy specification from policy enforcement and thus are useful for adding security to legacy systems.

Approaches applied during the execution phase enforce security by trapping resource accesses through hardware or software mechanisms, and by checking if these accesses are allowed. Approaches based in the execution phase appear to be best suited for implementing dynamic policies, especially those that depend on the execution state of extensible systems.

We have reviewed the various techniques currently used to ameliorate the security issues inherent in extensible systems. They demonstrate a growing understanding of the issues involved and effectively address specific security problems, but they do not encompass the general problem, except in a piecemeal, ad hoc fashion. In particular, it is not clear how different techniques can be composed to create a robust, efficient, and secure extensible system. Unraveling this composition problem yields another research challenge: to develop a rigorous understanding of the role each phase plays in constructing a general solution to the problem of mobile programming security. ■

ACKNOWLEDGMENTS

We thank Earl Barr, Fritz Barnes, and Scott Malabarba for their invaluable assistance in writing this article. We also thank the reviewers for their excellent comments. Our work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The US government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, or the US government.

REFERENCES

1. B.N. Bershad et al., "Extensibility, Safety and Performance in the SPIN Operating System," *Proc. 15th ACM Symp. Operating System Principles*, No. 15, ACM Press, New York, 1995, pp. 267-284; available at http://www.acm.org/pubs/articles/proceedings/ops/224056/p267bershad/p267-bershad.pdf.
2. L. Gong et al., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," *Proc. Usenix Symp. Internet Technologies and Systems*, USENIX Assoc., Berkeley, Calif., 1997; available to Usenix members at http://www.usenix.org/publications/library/proceedings/usits97/gong.html.
3. G. McGraw and E.W. Felten, "Java Security: Hostile Applets, Holes, and Antidotes," John Wiley & Sons, New York, 1997.
4. R. Wahbe et al., "Efficient Software-based Fault Isolation," *Proc. 14th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1993, pp. 203-216; available at http://www.acm.org/pubs/articles/proceedings/ops/168619/p203-wahbe/p203-wahbe.pdf.
5. M.I. Seltzer et al., "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," *Second Symp. Operating System Design and Implementation*, Usenix Assoc., Berkeley, Calif., 1996; available at http://www.eecs.harvard.edu/~vino/vino/osdi-96/.
6. G.C. Necula and P. Lee, "Safe Kernel Extensions without Run-Time Checking," *Second Symp. Operating System Design and Implementations*, Usenix Assoc., Berkeley, Calif., 1996; available at http://www.usenix.org/publications/library/proceedings/osdi96/necula.html.
7. R. Pandey and B. Hashii, "Providing Fine-Grained Access Control for Mobile Programs Through Binary Editing," Tech. Report CSE-98-8, Univ. of California, Davis, 1998; available at http://www.cs.ucdavis.edu/~pandey/Ariel/Papers/tr9809.ps.
8. J.K. Ousterhout, J.Y. Levy, and B.B. Welch, "The Safe-Tcl Security Model," Tech. Report TR-97-60, Sun Microsystems Laboratories, 1997; available at http://research.sun.com/technical-reports/1997/abstract-60.html.
9. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, Vol. 20, No. 1, Jan. 1973.
10. B. Ford and S. Susarla, "CPU Inheritance Scheduling," *Second Symp. Operating Systems Design and Implementation*, Usenix Assoc., Berkeley, Calif., 1996; available at http://www.usenix.org/publications/library/proceedings/osdi96/ford.html.
11. J. Bredin, D. Kotz, and D. Rus, "Market-Based Resource Control for Mobile Agents," *Proc. Autonomous Agents*, May 1998, ACM Press, New York, pp. 197-204; available at http://www.acm.org/pubs/articles/proceedings/ai/280765/p197-bredin/p197-bredin.pdf.

Brant Hashii is a PhD student in the Department of Computer Science at UC Davis. His research interests include computer security, security policies, auditing, and mobile programming. Currently, he is exploring ways to apply access constraints to mobile programming systems. He is a member of IEEE and the ACM.

Manoj Lal he is a graduate student in the Department of Computer Science at UC Davis. His research interests include computer security, particularly as it applies to mobile programs and runtime systems. He is working on developing resource control techniques that enable secure execution of mobile programs. Lal received a BS in computer science and engineering from the Indian Institute of Technology, Delhi, India.

Raju Pandey is an assistant professor in the Department of Computer Science at UC Davis. His research interests include Web-based computing, parallel and distributed programming, operating systems, and software engineering. He leads the Ariel project, which is developing novel techniques for supporting secure and efficient mobile-program execution. Pandey received a PhD from the University of Texas at Austin. He is a member of the IEEE and the ACM.

Steven Samorodin is a graduate student in the Department of Computer Science at UC Davis and is a software engineer at Marimba, Inc. His research interests include computer security, particularly as it applies to Internet computing. He is a member of Usenix and the ACM.

Readers can contact Hashii, Lal, Pandey, Samorodin at the Parallel and Distributed Computing Laboratory, Computer Science Department, University of California, Davis, CA 95616; {hashii, lal, pandey, samorodin}@cs.ucdavis.edu.