The Security Architecture of the Java Operating System JX – A Security Architecture for Distributed Parallel Computing

Christian Wawersich[1], Meik Felser[1], Michael Golm[2], and Jürgen Kleinöder[1]

[1] Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr.1, 91058 Erlangen, Germany
{felser,wawersich,kleinoeder}@informatik.uni-erlangen.de
[2] Siemens AG - Corporate Technology (CT SE 2)
Otto-Hahn-Ring 6, D-81730 Munich, Germany
michael.golm@siemens.com

Abstract. Using the unneeded computation power in the internet for distributed computing is getting more and more eligible. To increase the willingness to provide unneeded computing power, a secure platform is needed for the execution of untrusted code. We present the architecture of the JX operating system, which can be used to safely execute untrusted code. The problem of erroneous agents crashing the system is solved by using Java – a typesafe language – as implementation language. The resource consumption of the agents is controlled by a security manager, that inspects every interaction between an agent and a system service. If the security policy does not approve the use of a system service, the access can be denied. An agent execution system build upon JX is presented to illustrate the security problems occurring and the solutions provided by the operating system JX.

1 Introduction

More and more workstations are connected to the internet. The typical applications do not use the full capacity of theses machines. This effect is increased by the improvements in computer hardware. Some projects (e.g. seti@home) try to use these unneeded computing power for distributed computing. In a generalized way a personal computer may provide its unused computing power to the net. Everybody who wants to use it can assign a work package (called *agent*) to this computer. The owner or administrator of the machine does not need to trust the agent and must, nevertheless, be sure that the agent does not interfere with the normal operation of the system. For wider acceptance of this model of distributed computing we need a secure execution platform. This platform must assure that erroneous or malicious agents do not crash the system or consume all available resources. If this can be guaranteed the field of application can be even extended to more critical machines in the net, e.g. web servers that do not use their full capacity.

We provide JX a java-based operating system as an execution platform for untrusted code. Java allows developing applications using a modern object-oriented style, emphasizing abstraction and reusability. On the other hand many security problems have been detected in Java systems in the past [7]. The main contribution of this paper is an architecture for a secure Java operating system that avoids Java security problems, such as the huge trusted class library and reliance on stack inspection.

We follow Rushby [23] in his reasoning that a secure system should be structured as if it were a distributed system. With such an architecture a security problem in one part of the system does not automatically lead to a collapse of the whole system's security. Microkernel systems, especially systems that adhere to the multi-server approach, such as SawMill [13], and mediate communication between the servers [17] are able to limit the effect of security violations.

The JX system combines the advantages of a multi-server structure with the advantages of type-safety. It uses type-safety to provide an efficient communication mechanism that completely isolates the servers with respect to data access and resource usage. Code of different trustworthiness or code that belongs to different principals can be separated into isolated domains. Sharing of information or resources between domains can be completely controlled by the security kernel.

The paper is structured as follows. Section 2 gives an overview about Java security. The JX system architecture is described in Section 3. Section 4 completes the architectural overview with focus on security relevant aspect. An agent execution server is presented in Section 5 to illustrate the use of the previously presented security mechanisms. Section 6 describes related work and Section 7 concludes the paper.

2 Java Security

Java security is based on the concept of a sandbox, which relies on the type-safety of the executed code. Untrusted but verified code can run in the sandbox and can not leave the sandbox to do any harm. Every sandbox must have a kind of exit or hole, otherwise the code running in the sandbox can not communicate results or interact with the environment in a suitable way. These holes must be clearly defined and thoroughly controlled. The holes of the Java sandbox are the native methods.

To control these holes, the Java runtime system first controls which classes are allowed to load a library that contains native code. These classes must be trusted to guard access to their native methods. The native methods of these classes should be non-public and the public non-native methods are expected to invoke the SecurityManager before invoking a native method. The SecurityManager inspects the runtime call stack and checks whether the caller of the trusted method is trusted. The stack inspection mechanism only is concerned with access control. It completely ignores the availability aspect of security. This lack was addressed in JRes [6]. Also, Java is perceived as inherently insecure due to the complexity of its class libraries and runtime system [9].

JX avoids this problem by not trusting the JDK class library. No user defined native methods are allowed and the system services are implemented in Java accessible with a fast RMI-like communication mechanism as described in the following Section.

3 JX System Architecture

3.1 Domains

JX is a single address space system and protection is based on the type-safety of the Java bytecode instruction set. A small microkernel contains low-level hardware initialization code and a minimal Java Virtual Machine (JVM).

The JX system is structured into domains (see figure 1). Each domain represents the illusion of an independent JVM. A domain has a unique ID, its own heap including its own garbage collector, and its own threads. Thus domains are isolated with respect to CPU and memory consumption. They can be terminated independently from each other and the memory that is reserved for the heap, the stack and domain control structures can be released immediately when the domain is terminated. The microkernel represents itself also as a domain. This domain has the ID 0 and is called DomainZero. DomainZero contains all C and assembler code that is used in the system, all other domains execute 100% Java code. JX does not support native methods and there is no trusted Java code that must be loaded into a domain. There is no trust boundary within a domain which eases administration. Because the domain contains no trusted code it is a sandbox that is completely closed. For communication with other Domains we create new holes by introducing *portals*.

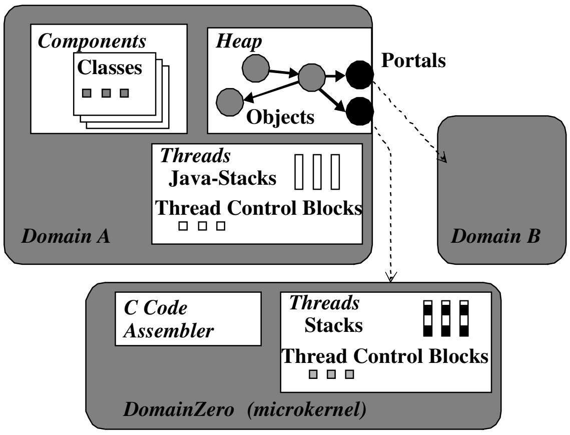

Fig. 1. Structure of the JX system

3.2 Portals

The portal mechanism is used for communication between different Domains, similar to RMI which is used for communication between different JVMs. Portals are proxies

[25] for a service that runs in another domain. Portals look like ordinary objects and are located on a domains heap, but the invocation of a method synchronously transfers control to the service that runs in another domain. Parameters are copied from the client to the server domain.

Portals and services are automatically generated during portal communication. When a domain wants to provide a service it can define a portal interface, which must be a subinterface of jx.zero.Portal, and a class that implements this interface. When an instance of such a class is passed to another domain the portal invocation mechanism creates a service in the source domain and a portal in the destination domain. Each domain possesses an initial portal: a portal to a naming service. Using this portal the domain can obtain other portals to access more services. When a domain is created, the creating domain can pass the naming portal as a parameter of the domain creation call.

3.3 Scheduler

CPU scheduling in JX is handled in two levels, which both can be implemented in Java. The first level is the global scheduler that decides which domain is allowed to use the CPU. Since this scheduler controls the CPU, it is a critical part of the system and it must be trusted. When a domain is selected by the global scheduler, a domain-local scheduler is activated. The task of this scheduler is to distribute the allocated CPU time among the threads of its domain. Being only responsible for one domain, the schedulers of the second scheduling level need not to be trusted. If they are malicious, they can only harm their own domain, other domains are uninfluenced concerning CPU time.

4 JX Security Architecture

4.1 JX as a Capability System

The portals, used in JX for inter-domain communication, are capabilities [8]. A domain can only access other domains when it possesses a portal to a service of the other domain. The operations that can be performed with the portal are listed in the portal interface. Although the capability concept is very flexible and solves many security problems, such as the confused deputy [14], in a very natural way, it has well known limitations. The major concern is that a capability can be used to obtain other capabilities, which makes it difficult, if not impossible, to enforce confinement [3].

4.2 The Reference Monitor

JX as described up to now can not enforce confinement. Thus an additional mechanism is needed: a reference monitor that is able to check all portal invocations and can thereby ensure a user defined security policy. A reference monitor must be tamper-proof, mediate all accesses, and be small enough to be verified. A reference monitor for JX must at least control incoming and outgoing portal calls.

We modified the microkernel to invoke a reference monitor when a portal call invokes a service of the monitored domain (inbound) and when a service of another domain is invoked via a portal (outbound). The internal activity of a domain is not controlled. The same reference monitor must control inbound and outbound calls of a domain, but different domains can use different monitors. A monitor is attached to a domain when the domain is created. When a domain creates a new domain, the reference monitor of the creating domain is asked to attach a reference monitor to the created domain. Usually, it will attach itself to the new domain but it can – depending on the security policy – attach another reference monitor or no reference monitor at all.

It must be guaranteed, that while the access check is performed, the state to be checked can only be modified by the reference monitor. For these reasons the access check is performed in a separate domain, not in the caller or callee domain. The reference monitor must have a a consistent view of the past parameters. One way is to freeze the whole system by disabling interrupts during the portal call. This would work only on a uniprocessor, would interfere with scheduling, and allow a denial-of-service attack. Therefore, our current implementation copies all parameters from the client domain to the server domain up to a certain per-call quota. These objects are not immediately available to the server domain, but are first checked by the security manager. When the security manager approves the call the normal portal invocation sequence proceeds.

4.3 Access Decision Based on Intercepted IPC

Spencer et al. [26] argue that basing an access decision only on the intercepted IPC between servers forces the security server to duplicate part of the object server's state or functionality. We found two examples of this problem. In UNIX-like systems access to files in a filesystem is checked when the file is opened. The security manager must analyze the file name to make the access decision, which is difficult without knowing details of the filesystem implementation and without information that is only accessible to the filesystem implementation. The problem is even more obvious in a database system that is accessed using SQL statements. To make an access decision the reference monitor must parse the SQL statement. This is inefficient and duplicates functionality of the database server.

There are three solutions for these problems:

1. The reference monitor lets the server proceed and only checks the returned portal (the file portal for example).
2. The server explicitly communicates with the security manager when an access decision is needed.
3. Design a different interface that simplifies the access decision.

Approach (1) may be too late, especially in cases where the call modified the state of the server. Approach (2) is the most flexible solution. It is used in Flask with the intention of separating security policy and enforcement mechanism [26]. The main problem of this solution is, that it pollutes the server implementation with calls to the security manager. This makes approach (3) the most promising approach. Our two example problems would be solved by parsing the path in the client domain. In an analogous manner the SQL parser is located in the client domain and a parsed representation is passed to the server domain and intercepted by the security manager.

This has the additional advantage of moving code to an untrusted client, eliminating the need to verify this code.

4.4 Controlling Portal Propagation

In [18] Lampson envisioned a system in which the client can determine all communication channels that are available to the server *before* talking to the server. In JX this is equivalent to a security manager that knows all portals that are owned by a domain. As we can not enforce a domain to be *memoryless* [18], we must also control the future communication behaviour of a domain to guarantee the confinement of information passed to the domain.

Several alternative implementations can be used to find all portals of a domain or to keep track of them, respectively:

1. A simple approach is to scan the complete heap of the domain for portal objects. Besides the expensive scanning operation, the security manager is not sure, that the domain will not obtain portals in the future.
2. Another approach is to install an outbound intercepter to observe all outgoing communication of the domain. Thus a domain is allowed to posses a critical portal but the reference monitor can reject it's use. The performance disadvantage is that the complete communication must be checked, even if the security policy allows unrestricted communication with a subset of all domains.
3. The best approach is a security manager that checks all portals transferred to a domain. This can be achieved by installing an inbound interceptor which inspects all data given to a domain and traverses the parameter object graph to find portals. But this operation is expensive if a parameter object is the root of a large object graph. During copying of the parameters to the destination domain, the microkernel already traverses the whole object graph. Therefore it is easy to find portals during this copying operation and the kernel is able to inform the security manager, that there is a portal passed to the domain. Then the security manager decides whether the portal will be created or not accordingly to the security policy. The security manager must also be informed if a portal is released. This way the reference monitor is able to keep track of the portals a domain actually possesses.

Confinement can now be guaranteed with two mechanisms that can be used separately or in combination: 1. the control of portal communication and 2. the control of portal propagation.

4.5 Principals

A security policy uses the concept of a *principal* [8] to name the subject that is responsible for an operation. The principal concept is not known to the JX microkernel. It is an abstraction that is implemented by the security system outside the microkernel, while the microkernel only operates with domains. Mapping a domain ID to a principal is the responsibility of the security manager. We implemented a security manager which uses a hash table to map the domain ID to the principal object. Once the principal is known, the security manager can use several policies for the access decision, for example based on a simple identity or based on roles [11].

The principal information can also be used for the local scheduling decision of a service domain. In our capability based file system interface, for example, each open file is represented by a portal. Each request (reading from or writing to a file) results in the activation of a service thread in the file system domain. But these threads do not immediately get CPU time. The local scheduler of each service domain decides which thread is allowed to run and thus controls which request is executed first. As described in the previous paragraph, each portal call and therefore also each service thread is associated with the principal information of the calling domain. The scheduling decision can be based on this information.

5 Example of Use: Agent Execution Server

We configure the system to act as an agent execution platform to illustrate how our security architecture works in a real system. A agent execution server should be able to execute code that was previously sent to him by a client. In detail the scenario is shown in figure 2.

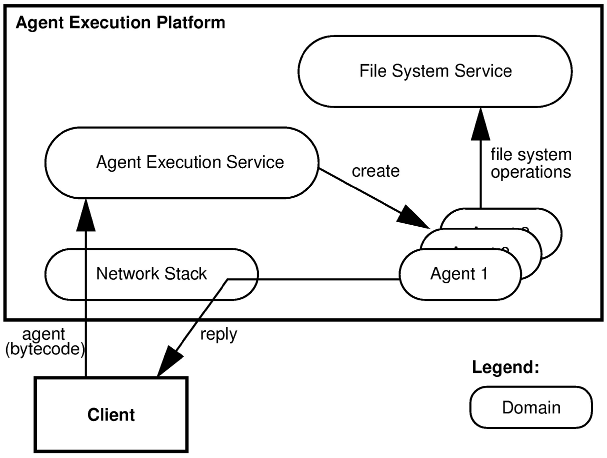

Fig. 2. Agent Execution Platform

A client transfers a component containing bytecode to the server. There the code is verified and compiled to nativecode. Afterwards a new domain is created and the received component is installed inside. Executing the component in a new domain is necessary to protect the system from erroneous or malicious code. Additionally an appropriate security manager must be attached to the domain to restrict the usage of resources and services.

In the following we use an agent implementation, that searches data on the file system. In a remote file system this requires that each file that should be examined must be transferred over the network. Using an agent reduces the network traffic, because it can move to the host that contains the file system and perform local file system operations. In this scenario we have to use a reference monitor that checks each file system operation. At the same time we have to control the CPU time each client is allowed to use.

Security Manger. We use a security manager, that controls the access to the file system service. Since our file system implementation is accessed using a capability based interface, we only have to check whether a domain (agent) is allowed to get a file capability or not. To accomplish this, the reference monitor is activated when a portal is passed between two domains (see Section 4.4). It reads the principal ID of the receiving domain and determines the portal type to know whether it is a file portal or not. If it is a file portal, the portal type also informs about whether it is for read-only access or for modifying access. On the basis of this information the security policy can be applied. In our example file operations of agents were restricted to a predefined group of files whereas local applications were allowed to access all files. Similar to this scenario any other resource, provided by a service of the agent execution platform, can be controlled.

Scheduling. We also have to restrict the time an agent is allowed to use the CPU. Since we installed each agent in a separate domain this is the task of the global scheduler. We implemented a stride scheduler [28].The stride scheduler distributes the available CPU time to the domains accordingly to the amount of *tickets* they have. The tickets are described as the right to use a resource. The more tickets a domain has the more often it can use the CPU. Every time a new agent is installed, a predefined amount of tickets is assigned to the agent. If there are no tickets left, we can either stop the execution of new agents or we can redistribute the available tickets between the running agents, thus the maximal CPU time for each agent is reduced.

If an agent uses a service it is desirable that the time a service spends for an agent is charged to the agent's CPU time account. This demands a cooperation between the global scheduler and the scheduler of the service domain hence we have to trust the local scheduler of the service. When the local scheduler is activated, it extracts a list of domains that have called his service from the list of runnable service threads. Then it asks the global scheduler which of these domains should be activated accordingly to the stride scheduling strategy. The service thread which handles the request of this domain is then activated. The time used by this thread can be charged to the client domain's account.

For most services this does not allow an exact accounting of this time because there are often resources that are shared between all service clients. For example the buffer cache management in a file system service. A buffer cache is necessary for an efficient service but the time used to manage it can not be accounted to a single service thread. If we want an exact accounting we have to use a separate file system service instance for each agent, but in most situations it will be enough to have a separate service instances for a groups of agents.

6 Related Work

Capability-Based Systems. Several operating systems are based on capabilities and use three different implementation techniques: partitioned memory, tagged memory

[10], and password capabilities. Early capability systems used a tagged memory architecture (Burroughs 5000 [22], Symbolics Lisp Machine [21]), or partitioned memory in data-containing and capability-containing segments (KeyKOS [12] and EROS [24]). To become hardware-independent, password capabilities [1] have been invented and are used in several systems (Mungi [16], Opal [4], Amoeba [27]).

Type-safe instruction sets, such as the Java intermediate bytecode, are a fourth way of implementing capabilities. The main advantages of this technique are that it is hardware-independent, capability verification is performed at load time, access rights are encoded in the capability type and not stored as a bitmap in the capability, and capabilities can not be transferred over uncontrolled channels.

Virtual Machines. Virtual machines can be used to isolate systems that share the same hardware. The classic architecture is the IBM OS/360 [20]. Virtual machines experienced a recent revival with the VMWare PC emulator [29]. VMs only work at a large granularity. VMWare instances consume a lot of resources to emulate a complete PC which makes it impossible to create fine-grained domains. Most applications require controlled information flow between classification levels; that is between VMWare instances. A virtual machine realizes a sandbox. The holes of the VMWare sandbox are the emulated devices. Thus, communication is rather expensive and stating a security policy in terms of an emulated device may be a difficult task.

Java Security. The J-Kernel [15] implements a capability architecture for Java. It is layered on top of a JVM, with the problems of limited means of resource control. It uses classloaders to separate types. The capability system is not orthogonal to application code which makes reuse in a different context difficult.

The MVM [5], and KaffeOS [2] are systems that isolate applications that run in the same JVM. The MVM is an extension of Sun's HotSpot JVM that allows running many Java applications in one JVM and give the applications the illusion of having a JVM of their own. There are no means for resource control and no fast communication mechanisms for applications inside one MVM. KaffeOS is an extension of the Kaffe JVM. KaffeOS uses a process abstraction that is similar to UNIX, with kernel-mode code and user-mode code, whereas JX is more structured like a multi-server microkernel system. There needs to be no trusted Java code in JX. Communication between processes in KaffeOS is done using a shared heap. Our goal was to avoid sharing between domains as much as possible and we, therefore, use RPC for inter-domain communication.

7 Conclusion

We described the security architecture of the Java operating system JX, which can be used as a secure agent execution platform for distributed computation. The security concept of JX consists of language-based protection and operating system protection. Typical Java security problems, such as native methods, execution of code of different trustworthiness in the same thread, and a huge trusted class library are avoided.

JX provides a number of security mechanisms of different invasiveness. The capability mechanism is inherent in the architecture and guarantees a minimal level of security. On a per-domain basis this mechanism can be supplemented by a monitor that controls propagation of capabilities between domains and, if necessary, a reference monitor that mediates access to these capabilities.

References

1. M. Anderson, R. Pose, C. S. Wallace. A password-capability system. In: The Computer Journal, 29, (1986) 1–8.
2. G. Back, W. C. Hsieh, J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In: Proc. of 4th Symposium on Operating Systems Design & Implementation, (2000).
3. W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In: Proc. of the 7th DoD/NBS Computer Security Conference, (1984) 291–293.
4. J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. In: ACM Trans. on Computer Systems, 12(4), (1994) 271–307.
5. G. Czajkowski, L. Daynes. Multitasking without Compromise: A Virtual Machine Evolution. In: Proc. of the OOPSLA01,(2001)125–138.'
6. G. Czajkowski, T. von Eicken. JRes: A Resource Accounting Interface for Java. In: Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications 98,ACMPress,(1998)21–35.'
7. D. Dean, E. W. Felten, D. S. Wallach, D. Balfanz, P. J. Denning. Java security: Web browsers and beyond. In: D. E. Denning (ed.) Internet Beseiged: Countering Cyberspace Scofflaws. ACM Press, (1998) 241–269.
8. J. B. Dennis, E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In: Communications of the ACM, 9(3), (1966) 143–155.
9. L. v. Doorn. A Secure Java Virtual Machine . In: Proc. of the 9th USENIX Security Symposium , (2000) 19–34.
10. R. S. Fabry. Capability-based addressing . In: Communications of the ACM, 17(7), (1974) 403–412 .
11. D. Ferraiolo, R. Kuhn. Role-based access controls. In: Proc. of the 15th National Computer Security Conference, (1992) 554–563.
12. B. Frantz. KeyKOS – a secure, high-performance environment for S/370. In: Proc. of SHARE 70, (1988) 465–471.
13. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, L. Reuther. The SawMill Multiserver Approach. In: Proc. of the 9th SIGOPS European Workshop, (2000).
14. N. Hardy. The confused deputy. In: Operating Systems Review, 22(4), (1988) 36–38.
15. C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, T. v. Eicken. Implementing Multiple Protection Domains in Java. In: Proc. of the USENIX Annual Technical Conference, (1998) 259–270.
16. G. Heiser, K. Elphinstone, S. Russel, J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. In: 17th Australiasion Computer Science Conference, (1994) 271–280.
17. T. Jaeger, J. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, K. Elphinstone. Synchronous IPC over Transparent Monitors. In: 9th SIGOPS European Workshop, (2000).
18. B. W. Lampson. A Note on the Confinement Problem. In: Communications of the ACM, 16(10), (1973) 613–615.

19. P. Loscocco, S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In: Usenix 2001 Freenix Track, (2001).
20. G. Mealy, B. Witt, W. Clark. The Functional Structure of OS/360. In: IBM Systems Journal, 5(1), (1966) 3–51.
21. D. A. Moon. Symbolics Architecture. In: IEEE Computer, 20(1), IEEE, (1987) 43–52.
22. E. I. Organick. Computer System Organization: The B5700/B6700 Series. Academic Press, Inc., New York, (1973).
23. J. Rushby. Design and Verification of Secure Systems. In: Proc. of the 8th Symposium on Operating System Principles, (1981) 12–21.
24. J. S. Shapiro, J. M. Smith, D. J. Farber. EROS: a fast capability system. In: Symposium on Operating Systems Principles, (1999) 170–185.
25. M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In: ICDCS 1986, (1986) 198–204.
26. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In: Proc. of the 8th USENIX Security Symposium , (1999).
27. Tanenbaum. Chapter 7. In: Distributed Operating Systems. Prentice Hall, (1995).
28. A. Waldspurger, W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, (1995).
29. Webpage of VMWare, http://www.vmware.com/.