THINK: A Secure Distributed Systems Architecture

Christophe Rippert Jean-Bernard Stefani
LSR-IMAG laboratory, SARDES project, CNRS-INPG-INRIA-UJF
{Christophe.Rippert, Jean-Bernard.Stefani}@inria.fr

1 Introduction

In this paper, we present THINK, our distributed systems architecture, and the research we have conducted to provide the system programmer with an architecture he can use to build efficient and secure operating systems. By specifying and implementing elementary tools that can be used by the system programmer to implement a chosen security policy, we prove that flexibility can be guaranteed in an operating system without compromising security. Our work focuses on protection against denial of service attacks which compromise the system fairness in resource multiplexing and can cause the system to stall due to resource starvation.

We first briefly describing the THINK architecture before positioning our contribution against related work. We then present the elementary tools we have specified to ensure quality of service in THINK, before detailing the software memory isolation tool we have implemented and tested. We conclude by a concrete example of the utilisation of these tools.

2 The THINK architecture

2.1 Presentation

The distributed systems architecture THINK is a platform for the development of distributed operating systems kernels. The goal of the THINK architecture is to ease the development of efficient, flexible, and secure operating systems. THINK provides the system programmer with interfaces that reify the underlying hardware, and optional system abstractions proposed as libraries. The development of a kernel with THINK is made easier by its object-oriented framework, since in THINK, all resources (both hardware and software) are considered as objects. These objects export interfaces which define their behaviour and make them accessible to other objects. Each interface has a name in a given naming context, and is linked to other interfaces by bindings. A binding is essentially a communication channel between two objects. These bindings can take many forms, as simple as the association between a variable name and its value, or more complex like a binding over a network between two objects on different machines. Bindings are created by dedicated objects called binding factories, those main function (i.e. creating bindings) can be freely extended to enforce a chosen behaviour. Finally, objects can be grouped in domains according to a common property (e.g. security domains, fault-domains, etc). A more detailed presentation of THINK can be found in [1].

2.2 Related work

Compared to the OSKit [2] framework and set of libraries, THINK provides a better flexibility since all components are independent. The KaffeOS [3] project proposes some techniques to preserve quality of service in a Java environment, using the type safety properties of the language. Similarly, the SPIN [4] extensible operating system uses the properties of the Modula-3 language to permit the safe binding of modules in the operating system. In THINK, we aim to remain independent of the component development language. In the DTOS [5] project, policy-neutral security is enforced by way of security servers, which check that inter-component calls are allowed. This requires a modification of the component code, whereas in THINK, binding factories can make the security checks, which ensures a complete independence of the component code. The Scout/Escort [6] project focuses on protection against denial of service attacks by defining the I/O path abstraction. However, this does not take into account the resources allocated in the operating system kernel, whereas in THINK, we aim toward a global view of the resources which enable the system to associate each allocated resource with the benefiting user. This point of view is close to the resource containers abstraction [7], although this work has been conducted in monolithic kernels whereas in THINK we advocate a more modular architecture. The exo-kernel [8] advocates the same philosophy of a minimalist kernel, although it does not propose an object framework for the components as in THINK. Finally, the memory isolation tool presented below is inspired from software fault isolation [9], but in THINK we are able to isolate a single process whereas soft-

ware fault isolation works with binary object files representing a whole module.

3 Preserving quality of service

The THINK architecture offers an unparalleled flexibility for the construction of operating system kernels, since it allows access to the hardware through policy-neutral interfaces and provides all system abstractions as optional libraries. However, this flexibility must not imply a lack of security in the system. Denial of service attacks are a serious threat to operating systems since they can compromise the fairness of resource multiplexing, thus favouring some users at the detriment of others, or even preventing the system from functioning at all. Therefore our goal in THINK is to provide the system programmer the elementary tools he needs to build a system in which quality of service can be ensured. However, these tools must remain completely independent of the multiplexing policy chosen by the system programmer. More precisely, the tools must not restrict the programmer's freedom to implement the quality of service policy he deems best for his system. To achieve that, we base our analysis on four hardware resources which should be present in most devices: a processor, some kind of volatile memory (e.g. random access memory), some kind of persistent memory (e.g. a hard drive), and a network. We propose some elementary tools for each resource before detailing the software memory-isolation technique implemented in THINK and presenting a concrete example of the use of these tools.

3.1 Analysis of the four key hardware resources

The processor: Fair sharing of the processor time requires a way to preempt processes, usually by way of interrupts. Since most processors provide some kind of interrupt management, all there is to do is to reify this manager with fitting interfaces. It is also necessary to protect the methods used to register interrupt handlers, to prevent a misbehaving application from illegally installing its own interrupt handler. This can easily be achieved in the THINK framework by securing the bindings and the binding factories. For instance, bindings can be made unforgeable by using cryptography and cyclic redundancy checking, and binding factories can use a standard capability system to check which process can create a binding with a given resource.

Volatile memory: To implement the notion of security domains, we need a way to isolate components or groups of components from each others. Since hardware isolation is usually deemed inflexible and very costly for context switches, we have implemented in THINK a software memory isolation tool, based on segment matching and code splicing techniques. This tool is implemented as an optional library so as to enable the system programmer to use the hardware isolation provided by the memory management unit of the processor if he prefers so. To permit cross-domains calls, the binding factories can be used to ensure that a component will call (i.e. create a binding with) only authorized methods from other components.

Persistent memory: A fair sharing of the storage space can be easily implemented using classical disk quotas. However, this does not protect against a malicious program which would make repetitive reads and writes of big files to slow down the disk accesses of other applications. Lots of work has been conducted in the field of disk scheduling, basing reordering of disk accesses on criteria like the order of arrival of the request, the localisation of the requested sector on the disk, or the real-time constraints of the request. To protect the system against disk access flooding attacks, these algorithms can be modified to take into account security constraints. For example, a scheduling algorithm can collect statistics on the processes accessing the disk and reorganize the requests so as to treat those coming from identified monopolizing processes after those from normal processes. In THINK, this statistic gathering disk scheduler can easily be programmed as a library component which the programmer can link with his own component in which the disk scheduling policy is defined. Thus, we ensure a complete independence of the policy (implemented by the programmer in his component) and the tool (provided by THINK as an optional library).

The network: Fair sharing of the network bandwidth between the processes is a basic quality of service tool which can be easily parameterized to enforce any chosen bandwidth multiplexing policy and can evolve dynamically as process priorities change. However, the major threat remains the SYN flooding attacks, which have become a common danger for Internet services and especially Web servers. No complete solution has yet been found to protect systems against these attacks but some steps can be taken to lessen the risk of flooding the backlog queue. First, the server can filter packets with source IP addresses obviously forged. A sentry can also count the number of connections in SYN_RECEIVED state and empty the backlog queue before it is saturated. This does not prevent legitimate connections from failing since they will be cancelled when the queue is emptied, but it protects against a backlog queue overflow which might results in a system crash. It is also possible for the server to learn from attacks: if more than a fixed number of SYN packets coming from the same IP address have been discarded because the corresponding ACK packet did not came in time, the server can consider that address to be spoofed and drop all subse-

quent SYN packets coming from it. Finally, SYN cookies [10] can be implemented too, as an optional tool considering the TCP protocol incompatibilities that they induce. In THINK, these tools should be implemented in the binding factories, since flooding a server with connection requests is equivalent with flooding a binding factory with requests for bindings with a remote component.

3.2 A software isolation mechanism

We present here the software isolation mechanism we have developed in THINK to implement the notion of security domains. This mechanism was implemented on a PowerPC processor, but it should be easy to port to any RISC machine.

The algorithm we use to enforce process isolation is based on code-splicing and segment-matching. The goal is to generate code for every memory-access to verify that the access is in an allowed area. This code generation takes place when a process is created at runtime and its code loaded in memory, so as to hide the delay induced by the code parsing and code generation within the process creation delay. When the process code is loaded into memory by the system, the algorithm parses it and generates code to check the address points to an allowed area (this area is identified by its lower and upper limits, which are here held in two dedicated registers). The checking code cannot be directly inserted in the initial code since we are working on raw binary code which would make address translation very difficult and costly. Therefore the generated code is stored in a dedicated memory area and the initial code is modified to replace the memory access by a branch to this generated code. This algorithm can be optimized for instructions with the address encoded in them, like some branches on the PowerPC for example. In that case, the check is made when the initial code is parsed and a process containing an illegal access will not be executed at all.

All the benchmarks we have conducted were executed on a PowerPC G4 866 MHz with 384 MB of SDRAM PC100 memory. All test programs were compiled with `gcc` using the `-O` option for optimizing the memory accesses by putting local variables in registers.

Memory consumption: On the PowerPC, the average size of the generated code for a memory access is 5 instructions. Obviously, not all instructions in a programme make memory access, so we monitored the algorithm for a simple test program to find the average increase. The chosen test algorithm is a bubble sort program, which can be coded in 29 assembly instructions, including 9 memory accesses. Amongst those 9 accesses, 4 are branches those destinations can be statically checked. This results in a generated code size of 25 instructions, which is almost has much as the original code, therefore doubling the program size in memory. This result can appear to be rather prohibitive, but one must consider that this increase of memory space required only applies to the code of the application, not to its data. Since most applications include much more data than code, doubling the code size is not so costly as it seems.

Code generation delay: We monitored the delay induced by the generation of the spliced code when the process is created in memory. We copied/pasted 1000 times the code of the bubble sort algorithm, therefore obtaining a code of 29000 instructions and found that the algorithm takes 3.90 ms to complete. Since each PowerPC instruction is 4 byte long, we obtain a processing time of 28 MB/s (i.e. the algorithm would take 1 second to modify the original code and generate the spliced code for a 28 MB long original code). Considering that the most consuming part of the process creation task is when the code loader in charge of creating the process accesses the hard drive to read the ELF file, and that the average read time on a standard hard drive is 15 MB/s, we believe that the processing time of the segment matching algorithm is acceptable and will not affect the system performances.

Runtime penalty: We first performed benchmarks for basic operations. First, we monitored the runtime of a single memory access (i.e loading a 32-bit integer from memory in a register) and found that the runtime is multiplied by 3.5 with segment matching. This prohibitive cost is easily understandable, since we add to the runtime of the memory access the runtime of an addition, two comparisons, and two branches. For a local branch, there is no increase since the destination address can be staticaly checked when the process is created and therefore no code is generated. Finally, for an absolute branch (such are used in interprocess calls), the segment matching induce an increase of +16.67%, which is very competitive compared to the cost of IPC through hardware isolation. We compared this interprocess call with an optimised LRPC [11] and found that the LRPC is more than 25 times slower than our mechanism.

We then monitored the runtime for two significant algorithms. We first sorted 100000 integers with the bubble sort algorithm and found 56970 ms without segment matching and 116280 ms with it, resulting in multiplying the runtime by 2. We then implemented Heron of Alexandria's square root computing algorithm (a classical iterative algorithm), and monitored its performances for 10^6 computations of the square root of 10^6. The runtime was 6655 ms without segment matching, and 6758 ms with it, thus an increase of 1.55%. We chose these two standard algorithms because the bubble sort is representative of algorithms making lots of memory accesses whereas the square root computing represents algorithms making very few memory accesses. Thus,

we can conclude that the runtime penalty for the software memory isolation algorithm will be below +100%, depending on the frequency of memory access of the application. As an exemple of a real application, we ported the `gzip` data compression algorithm and found an increase of the runtime of approximately +100%, which is logical since the LZW algorithm makes heavy use of memory access to manage its string table.

Analysis: The prohibitive cost of segment matching memory accesses clearly reduces the interest of our mecanism for inforcing data confidentiality. On the other hand, this mecanism is very interesting for IPC, especially compared to hardware isolation. Thus, combined with a secure framework based on secure binding factories, this mecanism can be used to implement the notion of security domains proposed in the component model on which `Think` framework is based. By isolating component with software memory isolation, we prevent the programmer from directly calling a remore method with a forged pointer for exemple, thus forcing him to pass through the binding factories where security checks can be made.

3.3 Example

We present here the example of a fair scheduler implemented using the elementary security tools presented above. Considering an application composed of several processes, the programmer wants to ensure that each process will be allocated the same amount of time as the others. The first tool needed is the software memory isolation mechanism, which is used to define a different protection domain for each process, and another one for the scheduler. To be able to interrupt a process execution, the scheduler needs to use the system clock. So it registers a new interrupt handler for that hardware resource using the `TrapRegister` method provided by the interface reifying the interrupts. By doing that, a binding is created between the scheduler and the clock object by the binding factory managing the interrupt handlers. But before creating the binding, the binding factory authenticates the object calling the `bind` method to ensure that it is an allowed object (i.e. the scheduler object and not an application process). Thus, using the tools proposed by the THINK architecture, the programmer can ensure that his policy of fair scheduling between the processes will be enforced.

4 Conclusion

As we have shown in this paper, security can be enforced in an operating system without sacrificing flexibility. By providing elementary security tools, the THINK architecture does not restrict the system programmer's liberty to implement whichever security policy suits him best. Combined with a secure framework based on protected binding factories, these tools provide the kernel prorammer with all he needs to build a secure customised system.

Bibliography

1. Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall and Gilles Muller. THINK: A Software Framework for Component-based Operating System Kernels. In Proceedings of the USENIX Annual Technical Conference, 2002.
2. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.
3. Godmar Back, Wilson C. Hsieh, Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000.
4. Przemyslaw Pardyak, Brian N. Bershad. Dynamic Bindings for an Extensible System. In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, 1996.
5. Duane Olawsky, Todd Fine, Edward Schneider, Ray Spencer. Developing and Using a "Policy Neutral" Access Control Policy. In Proceedings of the New Security Paradigms Workshop, 1996.
6. Olivier Spatscheck, Larry L. Peterson. Defending Against Denial of Service Attacks in Scout. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.
7. Gaurav Banga, Peter Druschel, Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.
8. Dawson R. Engler, M. Frans Kasshoek, James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.
9. Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. Efficient Software-Based Fault Isolation. In Proceedings of the ACM SIGOPS'1993.
10. Jonathan Lemon. Resisting SYN flood DoS attacks with a SYN cache. In Proceedings of the BSDCon 2002 Conference.
11. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. Lightweight Remote Procedure Call. In ACM Transactions on Computer Systems, Vol. 8, No. 1, February 1990, pages 37-55.