Self-Protected System: an experiment

Noel Depalma, Benoit Claudel, Renaud Lachaize
Institut National Polytechnique de Grenoble, France
Noel.Depalma, Benoit.Claudel, Renaud.Lachaize@inria.fr

Sara Bouchenak
Universit Grenoble I, France
Sara.Bouchenak@inria.fr

Daniel Hagimont
Institut National Polytechnique de Toulouse, France
Daniel.Hagimont@enseeiht.fr

April 12, 2006

Abstract

The complexity of today's distributed computing environment is such that the presence of bugs and security holes is statistically unavoidable. A very promizing approach to this issue is to implement a self-protected system, similarly to a natural immune system which has the ability to detect the intrusion of foreign elements within the system.

We designed and implemented an autonomic system called Jade, which relies on software component architectures to reconfigure applications according to observed events. The knowledge of the application architecture can be used to detect foreign activities and to trigger counter-measures. We described how this approach can be applied the provide self-protection for a clustered J2EE application.

1 Introduction

Enforcing the security of a computing system lies on some key capacities. First, as preventive measures, it is important to define tight access control policies, so that hackers can hardly break into the system and hide their tracks. Second, one should be able to distinguish suspicious activities from the "normal" operation of the system. Third, once detected, the malicious processes must be stopped in a comprehensive an efficient way. In addition, it is desirable to log the system's activity with a good amount of details (and protection/redundancy to prevent attackers from completely destroying the logs), so that the full sequence and scope of malicious acts can be determined a posteriori, to launch the appropriate recovery procedures and take new measures against future attacks.

Unfortunately, these goals are very hard to meet in practice, for several reasons.

- It is notoriously complex to specify and maintain access control policies that are effective, globally consistent (across different programs and computers) and not overly restrictive for users.
- The complexity of today's software components (and their interactions) is such that the presence of bugs and security holes is statistically unavoidable. This leaves the opportunity for attackers to develop new hijacking techniques ("exploits") at a very high pace. Keeping up with the appropriate security patches requires a continuous vigilance.
- Detecting malicious activities within the system is, in general, far from trivial and relies almost exclusively on human expertise. For this reason, most intrusions are only noticed once much damage has been done.
- Careful logging of the system's activity under normal circumstances often leads to unacceptable performance and tremendous need in terms of storage. Besides, extracting crucial hints from the verbose logs is not obvious at all.

Overall, most problems stem from the fact that (human) administrators are unable to cope with the amount of work required to properly secure a computing infrastructure at the age of the Internet.

We propose to address the above problems through the design and implementation of a self-protection system. Our two main goals are to: (i) simplify the configuration (and reconfiguration) of security components according to the knowledge of the system's structure and operation and (ii) ease the development of automated counter measures to various classes of attacks.

We have designed and implemented a prototype of autonomic management system (called Jade) which has been successfully used to provide self-healing [2] and self-optimizing [11] capacities to a clustered J2EE architecture. We are currently investigating the use of Jade to provide self-protection capacities for the same application class.

The rest of the paper is structured as follows. Section 2 presents the self-protection approach. Section 3 provides an overview of Jade, the autonomic management system that we implemented, and it application to clustered J2EE applications. Section 4 describes a scenario which illustrates the use of Jade to implement self-protection in a J2EE application. We overview related works in Section 5 and then conclude the paper.

2 Self-protection

2.1 Common security tools

This section briefly reviews the main tools and techniques currently used by security experts to fight against intrusions. We make the distinction between

different functions (protection filters, detectors of suspicious activity, logging & backtracking tools) although many available solutions integrate several of them.

Protection filters are used to restrict interactions among machines (or, more generally, distributed processes/resources) to a given set of limited, well established set of patterns. For instance, a firewall acts as a network filter that checks if any given packet can be forwarded according to its related protocol, source/destination addresses and ports. Other examples include restricted rights for data access or program execution, through the use of capabilities or ACLs (access control lists).

Detectors (or scanners) examine elements of the system activity (contents/attributes of application-level or network-level requests, files, ...) and compare them against a library of patterns typical of known attacks. If an intrusion is detected, an alarm notification is sent to the administrator. In addition, scanners can also react themselves against the intrusion, but their action is usually limited in scope (block offending request/packet, quarantine suspect resource) and context (no coordination between the different servers). Thus, (quick) human intervention is generally required anyway for further study and containment of the problem, especially if the suspicious activity was detected within the internal network (suggesting that there is a breach in the main "frontier" with the outside world).

Loggers record detailed (and possibly protected/redundant) data about the system activity so that once an intrusion attempt has been detected, it is possible to determine the sequence of events that led to the intrusion and the potential extent of the damage (e.g. data theft/loss). Backtracking tools can help to automate parts of this process but human expertise is still required for an accurate understanding of the attack.

As a summary, protection filters enforce a set of preventive access control measures, scanners try to dynamically spot and block suspicious activity and logging tools allow an analysis of the system's recent life to initiate proper recovery and take new measures against future attacks of the same kind.

These tools are invaluable for system administrators. However, they are not powerful enough to ensure good levels of security, for several reasons. First of all, most detectors can only protect the system against known attacks. Therefore, pirates are always a length ahead with the resort to new "exploits", which are able bypass filters and scanners. Furthermore, human administrators are heavily solicited by the alarms produced by the scanners. In particular, they are usually in charge of initiating lots of actions, both for coordinated defense at the cluster scale (e.g. through reconfiguration of the filters and scanners) and investigation (e.g. with backtracking tools). As a consequence, the human resources still represent the main bottleneck of the security infrastructure, which tends to increase the vulnerability window of a system exposed to a new kind of attack.

The purpose of our work is not to replace the existing tools but rather to provide a systematic approach that allows more closely-coupled interactions between them, so that the cluster-wide, coordinated reaction against an attack can become automated, and thus, more efficient.

2.2 The Self-protection Approach

Research on self-protection systems is a recent initiative, still in its prospective stage, and has been emphasized by the more global calls for "Autonomic Computing" (AC), which also encompass concerns about other dimensions such as (self-) configuration, optimization and repair (after failures). This approach is notably inspired by the operation of the human body and has lead to the concept of computer immune system (CIS), in the mid 1990s.

The main goal of natural immune systems is to protect a live being from dangerous foreign pathogens. This mission relies on a key ability, the "sense of self" (SoS), that is, the capacity to detect the intrusion of foreign elements within the "system" (in this case, the body), though the distinction of self from nonself. Once an intruder is properly detected, measures can be taken to destroy it (or at least contain its damages and progression). In the context of a computing system, nonself may correspond to the activity of a malicious program or an unauthorized user.

Based on this analogy, Forrest et al. [5] determined the main design principles required to build computer immune systems, which are summarized below.

Autonomy: The immune system does not require (much) outside management or maintenance. It autonomously classifies and eliminates attacks, i.e. it is able to recognize previously seen attacks as well as new types of intrusions.

Distributability: There is no central coordination, and, as a result, no single point of failure within the immune system. This implies that no single component is essential and that the incorrect behavior or death of some security components can be compensated by repairing or creation of new components.

Multi-layered: multiple layers with different mechanisms are combined to provide robust and flexible facilities for security. Inspired by these principles, we propose architectural patterns to improve the coordination between the multiple elements which compose a security infrastructure. Our focus is not on the development of new specific techniques for access control, intrusion detection or backtracking but rather on the mechanisms that allow an efficient and flexible integration of these various tools within a global, automated control process.

3 An overview of the Jade self-management system

3.1 Design of jade

We have adopted the overall organization proposed for autonomic computing [10]. An Autonomic Manager implements a feedback control loop, which

regulates a part of a system, called a Managed Resource. In order to allow hierarchical control, an Autonomic Manager may itself play the part of a Managed Resource.

In order to be controllable, a Managed Resource needs to be equipped with a management interface, which provides entry points for an Autonomic Manager. This interface should allow the manager to observe and to change the state of the resource.

The management interface needs to provide the following functions.

- Inspecting the contents of the managed resource, i.e. consulting any readable parameters; reading the values of any probes attached to resource; if the managed resource is composite (made of an assembly of parts), retrieving information on the structure of this assembly.
- Deploying and (re)configuring the managed resource; if again the resource is composite, modifying the structure of the assembly, e.g. dynamically inserting, rebinding, or removing some of its parts (for instance inserting a probe, adding a node to a cluster, etc.).

We propose to use a component model as a base for the implementation of Managed Resources. The component model that we use is Fractal [4], which has the following benefits.

- It provides a uniform, adaptable, control interface that allows introspection (observing the properties of the component) and dynamic binding (reconfiguring an assembly of components).
- It defines a hierarchical composition model for components, allowing a sub-component to be shared between enclosing components, at any level of granularity.

We use Fractal to wrap any managed ressource in a component, thus providing a uniform management interface for all these resources. This provides a means to:

- Managing legacy entities using a uniform model, instead of relying on resource-specific, hand-managed, configuration files.
- Managing complex environments with different points of view. For instance, using appropriate wrapping components, it is possible to represent the network topology, the configuration of the J2EE middleware, or the configuration of an application on the J2EE middleware.
- Adding a control behavior to the encapsulated legacy entities (e.g. monitoring, interception and reconfiguration).

3.2 Self-management for J2EE applications

The above approach is illustrated in the case of a J2EE architecture. In this setting, an L5-switch balances the requests between two Apache server replicas. The Apache servers are connected to two Tomcat server replicas. The Tomcat servers are both connected to the same MySQL server.

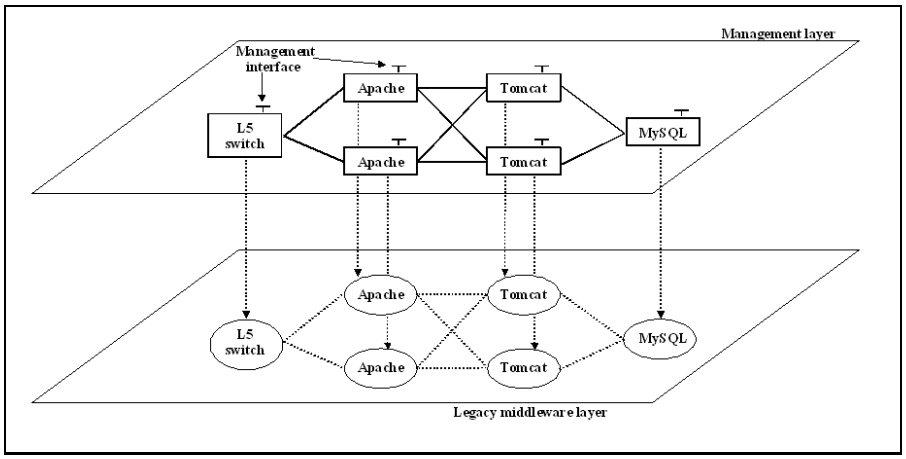

Figure 1: Component-based management

In figure 1, the vertical dashed arrows represent management relationships between components and the wrapped software entities. In the legacy layer, the dashed lines represent relationships (or bindings) between legacy entities, whose implementations are proprietary. These bindings are represented in the management layer by component bindings (full lines in the figure).

Wrapping managed ressources

In the management layer, all components provide the same (uniform) management interface for the encapsulated resources, and the corresponding implementation is specific to each resource (e.g., in the case of J2EE, Apache, Tomcat, MySQL, etc.). The interface allows managing the reource's attributes, bindings and lifecycle.

Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers. The management layer provides all the facilities required to implement such administration programs:

Introspection: The framework provides an introspection interface that allows observing managed resources (MR). For instance, an administration program can inspect an Apache MR (encapsulating the Apache server) to discover that this server runs on node1:port 80 and is bound to a Tomcat server running on node2:port 66. It can also inspect the overall J2EE

infrastructure, considered as a single MR, to discover that it is composed of two Apache servers interconnected with two Tomcat servers connected to the same MySQL server.

Reconfiguration: The framework provides a reconfiguration interface that allows control over the component architecture. In particular, this control interface allows changing component attributes or bindings between components. These configuration changes are reflected onto the legacy layer. For instance, an administration program can add or remove an Apache replica in the J2EE infrastructure to adapt to the current load.

The implementation of the management layer relies on Fractal components which wrap the administrated legacy software. More precisely, the Fractal component model allows management of:

Attribute: An attribute is a configurable property of a component. The component's interface exposes getter and setter methods for the attributes. A modification of a component attribute is reflected to the legacy software attribute. For instance apache's port is reflected to the legacy Apache configuration file (httpd.conf).

Binding: The component's interface exposes methods for controlling bindings between components. The configuration of bindings in the management layer is reflected to the legacy software layer. For instance, by configuring a binding between an apache and a tomcat component in the management layer, an Apache httpd may be connected to a Tomcat server to which it delegates dynamic requests. The implementation of this binding configures the worker.properties file in the legacy software.

Life cycle: The component's interface exposes methods for controlling the component's execution. The basic lifecycle operations of a legacy system can be performed through this lifecycle interface (e.g. starting and stopping the execution of a component). In the case of apache, it is implemented by calling the Apache commands for starting/stopping a server.

Self-repair and self-optimizing

One important autonomic administration behavior we consider in Jade is self-optimization. Self-optimization is an autonomic behavior which efficiently maximizes resource utilization to meet the end user needs with no human intervention required. Jade aims at autonomously increasing/decreasing the number of replicated resources used by the application when the load increases/decreases. This has the effect of efficiently maximizing resource utilization (i.e. no resource overbooking).

To this purpose, a QoS manager uses sensors to measure the load of the system. These sensors can probe the CPU usage or the response time of application-level requests. The QoS manager also uses actuators to reconfigure the system. Thanks to the generic design of Jade, the actuators used by the QoS manager are

themselves generic, since increasing/decreasing the number of resources of an application is implemented as adding/removing components in the application structure.

Besides sensors and actuators, the QoS manager makes use of an analysis/decision component which is responsible for the implementation of the QoS-oriented self-optimization algorithm. This component receives notifications from sensors and, if a reconfiguration is required, it increases the number of resources by allocating new necessary and available nodes. It then deploys those software resources on the new nodes and adds them to the existing application structure, just by creating the associated components on these nodes. Symmetrically, if the resources allocated to an application are under-utilized, the QoS manager performs a reconfiguration to remove some replicas and release their resources.

Another autonomic administration behavior we consider in Jade is self-repair. In a replication-based system, when a replicated resource fails, the service remains available thanks to replication. However, we aim at autonomously repairing the managed system by replacing the failed replica by a new one. Our current aim is to deal with fail-stop faults. The implemented repair policy rebuilds the failed managed system as it was prior to the occurrence of the failure. To this purpose, the failure manager uses sensors that monitor the health of the used resources through probes installed on the nodes hosting the managed system; these probes are implemented using heartbeat techniques. The failure manager also uses a specific component called the System Representation. The System Representation component maintains a representation of the current architectural structure of the managed system, and is used for failure recovery. One could state that the underlying component model could be used to dynamically introspect the current architecture of the managed system, and use that structure information to recover from failures. But if a node hosting a replica crashes, the component encapsulating that replica is lost; that is why a System Representation mechanism is necessary. This representation is reflects the current architectural structure of the system (which may evolve); and is reliable in the sense that it is itself replicated to tolerate faults. The System Representation is implemented as a snapshot of the whole component architecture.

Besides the system representation, the sensors and the actuators, the failure manager uses an analysis/decision component which implements the autonomic repair behavior. It receives notifications from the heartbeat sensors and, upon a node failure, makes use of the System Representation to retrieve the necessary information about the failed node (i.e., software resources that were running on that node prior to the failure and their bindings to other resources). It then allocates a new available node and redeploys those software resources on the new node. The System Representation is then updated according to this new configuration.

A more detailed description and evaluation of self-optimization and self-repair in Jade is available in [2].

4 Self-protection with Jade for J2EE applications

4.1 Architecture-based sense of self

As mentionned in Section 2.2, the "sense of self" (SoS) is the capacity to detect the intrusion of foreign elements within the administrated system though the distinction of self from nonself.

As we have seen in the overview of the Jade system, we aim at providing a component-based architectural representation of the administrated environment in order to enable observations and reconfigurations. The architecture of the application provides a notion of sense of self, as it defines the components which are supposed to be running on the machines and the communication channels which may be used by these components. Any execution which does not take place within these components or communication channels is considered nonself.

The design of Jade also follows the design principles of immune systems (presented in Section 2.2):

Autonomy: The self-protection policy which may be defined with Jade does not have to know much about the attacks it may have to face. It can detect abnormal behaviours, that is those which are nonself (which don't comply with the architecture of the application). In our scenario, we detect intrusions as nonself behaviors. For instance, if an application attempts to use a not-declared communication channel (binding), an alarm event is raised, which will triggers a counter-measure.

Distributability: There is no single point of failure to security. Any node can detect an abnormal execution, on the local machine or as an incoming request from another node. In our scenario, each node is responsible for the detection of nonself incoming communications.

Multi-layered: Jade allows for the combination of many security techniques. Many detection and counter-measure mechanisms can me added by wrapping existing tools, deploying and configuring them. In our scenario, we wrapp a firewall to detect nonself communications.

4.2 Scenario

We describe in this Section a simple scenario which aims at illustrating the implementation of sense of self protection policies on top of the Jade system. This scenario is currently under development.

We consider a security flaw which allows attackers to execute arbitray code in one tier of the deployed J2EE architecture. An example of such a flaw is the Apache Chunked Encoding Overflow as defined in [1]:

Apache Web Server contains a flaw that allows a remote attacker to execute arbitrary code. The issue is due to the mechanism that calculates the size of "chunked" encoding not properly interpreting the buffer size of data being

transferred. By sending a specially crafted chunk of data, an attacker can possibly execute arbitrary code or crash the server.

Therefore, exploiting this security flaw (or anyone such), an attacker can gain control on a machine running this Apache server and subsequently attempt to attack other machines. Notice that this flaw can be exploited even if the Apache server is placed behing a firewall. Our assumption is that attackers will always find a way to bypass statically defined protection barriers.

In order to detect nonself execution, we place a firewall on each machine involved in the J2EE architecture, as illustrated in Figure 2.

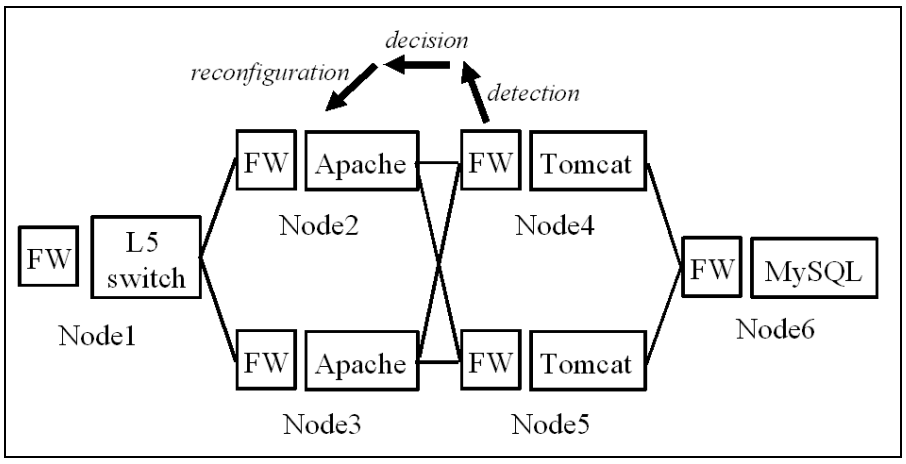

Figure 2: Self-protection scenario

The firewall software is wrapped in a component, so that it can be deployed and configured by Jade (similarly as the other software resources). Each firewall is configured so that it will only accept requests issued by machines which have a declared binding pointing to it. For instance, firewall on Node4 will only accept requests from Node2 and Node3. The firewalls are automatically configured according to the deployed J2EE architecture.

Moreover, each firewall is configured to behave as a intrusion detector. When the J2EE architecture is deployed, the communication ports used to connect the J2EE tiers are chosen randomly, which makes it more difficult for an attacker to exploit legal bindings. And an attempt to use/scan an unbound port is detected by the firewall to raise an alarm event.

Therefore, the configuration of the firewalls only enable requests which follows legal bindings and unlegal requests raise an alarm. Therefore, the deployment and configuration of firewalls allows detecting (some of) the nonself behaviours.

Whenever an alarm is raised, different counter-measures can be executed:

- the machine (in the cluster) which issued the request can be isolated from the J2EE architecture and replaced by another one (as in the repair al-

gorithm). The reconfiguration of the architecture also reconfigures the firewalls accordingly.

- a message can be sent to the human administrator to take any additional measure.

- an analyse of logs on the different tiers can eventually allow to find the remote machine which issued the web request (on the global J2EE web server), thus allows a reconfiguration of the firewall placed in front of the overall J2EE architecture (on Node1) to deny access from this remote machine.

5 Related work

Defining computer immune systems is a major trend in self-protected system. This approach has been described by Forrest [5] and Kephart [9]. The basic idea behind immune systems is to distinguish legal (self) behaviours from unlegal (nonself) ones (typically virus, worm, sql injection, ...).

Forest describes a sense of self in the case of unix processes by identifying sequences of system call which provide a compact signature for self, distinguishing self from non-self behaviours. This system requires to build up a database of normal behavior for each program of interest.

Kephart describes an anti-virus system where detectors are based on the immune system analogy and are able to find unkown virus. Furthermore, when a new virus is found, its signature is spraid accross the network to all other protected computers. Our work is in the same vein, but we propose to exploit the knowledge of the architecture of the application to detect non-self activities.

Self-cleansing system [8] is another solution to build self-protected software. This pessimist approach makes the assumption that all intrusions cannot be detected and blocked. Therefore, the system is considered to be compromised after a certain time. The control loop used by this system periodically re-installs a part of the system from a secure storage.

An important property for self-protected system is the ability to mask important knowledge such as the system's structure, software versions, user's data files... The Secure Distributed Storage [6] (SDS) is such a system that secures data by spreading and crypting them accross multiple computers. A file is sliced in multiple crypted data chunks. Thus, if a computer is compromised, the hacker can only get the incomplete data stored on the computer. In our scenario, the connection ports which are used to implement bindings between components are randomly chosen, thus making it more difficult for an attacker to exploit legal bindings. In our approach, hidding (as much as possible) the architecture of the application is also a crucial issue.

When a system is compromised, another important function is the ability to restore the system in a trusted state. File system snapshots can successfully restore system's data when an intrusion is detected. However this solution also rollbacks legal data modifications induced by users. The Taser system [7]

provides the file system with a selective self-recovery capability. Taser logs all file system access for each process. If a process is compromised, Taser computes illegal access for each file and is able to rollback illegal modification. However if a dependency is found between an illegal and a legal access (e.g: a legal read operation after a compromized write operation), Taser requires a human intervention. A similar approach could be followed to restore the database tier in J2EE application whenever a nonself activity performed database modifications.

6 Conclusion

Today's distributed computing environments are increasingly complex and difficult to administrate. This complexity is such that the presence of bugs and security holes is statistically unavoidable. Therefore, access control policies become very difficult to specify and to enforce.

A very promizing approach to deal with this issue is, following the autonomic computing vision, to design a self-protected system which is able to distinguish legal (self) from unlegal (nonself) behaviours. The detection of an unlegal behaviour triggers a counter-measure to limit the exploitation of the intrusion and prevent further intrusions.

In this vein, we have designed and implemented a system called Jade which allows the definition of autonomous administration programs. Jade relies on a component model for wrapping the administrated resources and provides support for the definition of autonomic managers which capture significant event and trigger relevant actions. Jade has been successfully used to implement self-optimization and self-healing autonomic policies for clustered J2EE applications.

In this paper, we investigated the application of Jade features to implement self-protection for J2EE applications. We showed how the knowledge of the (component-based) architecture of the administrated application can be exploited to implement a notion of self. In our scenario, firewalls are automatically deployed on every nodes and configured according to the deployed J2EE architecture. These firewalls allow detecting (some) non-self behaviours and taking adequate counter-measures.

This work is at a preliminary stage, but opens many perspectives. It only detects some of the potential non-self behaviours. For instance, we plan to investigate the definition of detectors for SQL injection attacks [3] based on an analysis of the logs generated by the administrated software components.

References

[1] Apache chunked encoding overflow. http://www.osvdb.org/838, June 2002.

[2] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. Depalma, V. Quema, and J.-B. Stefani. Architecture-based autonomous repair man-

agement: An application to j2ee clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, October 2005.

[3] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J.-B. Stefani. An open component model and its support in java. In *International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, may 2004.

[5] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[6] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Sciences*, 243(1-2):363–389, 2000.

[7] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *twentieth ACM symposium on Operating systems principles*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[8] Y. Huang and S. Arun. Self-cleansing systems for intrusion containment. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, 2002.

[9] J. O. Kephart. A biologically inspired immune system for computers. In *Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139, Cambridge, MA, US, 1994. MIT Press.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer Magazine*, 36(1), 2003.

[11] C. Taton, S. Bouchenak, F. Boyer, N. De Palma, D. Hagimont, and A. Mos. Self-manageable replicated servers. In *VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, Trondheim, Norway, August 2005.