BERSERK: A SIMPLE AND FLEXIBLE ACCESS CONTROL SOLUTION FOR SERVICE-ORIENTED ARCHITECTURES

Gonçalo Luiz
INESC-ID Lisboa / Technical University of Lisbon
R. Alves Redol, 9, 1000-029 Lisboa, Portugal
gedl@rnl.ist.utl.pt

André Zúquete
IEETA / UA / INESC ID Lisboa
IEETA, Campus Univ. de Santiago, 3810-193 Aveiro, Portugal
avz@det.ua.pt

António Rito Silva
INESC-ID Lisboa / Technical University of Lisbon
R. Alves Redol, 9, 1000-029 Lisboa, Portugal
rito.silva@inesc-id.pt

ABSTRACT

Nowadays many information systems are accessed using service-oriented architectures. Mapping traditional access control mechanisms, like objects ACLs or client's capabilities/privileges, to these architectures is not natural, being hard to deploy and to maintain. A more natural approach is to use service-oriented access control mechanisms. This paper describes BERSERK, a service-oriented generic access control solution based on the Intercepting Filter pattern. This solution is very flexible, being capable of expressing and enforcing security policies by composing security criteria from different domains. BERSERK maps filters to services and uses a simple logic composition language to build complex security criteria from filters implementing elementary security criteria. This enables the reuse of security-related code and facilitates the overall management of the information system security requirements.

KEYWORDS

Service-oriented security, reuse, expressiveness, flexibility, scalability.

1. INTRODUCTION

Most authorization mechanisms are centered either on principals or on objects accessed by principals. But with the growth of Service Oriented Architectures (SOA) [6] the use of the traditional access control mechanisms — lists (ACLs) for expressing accesses allowed/denied to objects or capability lists for expressing principals' privileges — is too inflexible and too hard to maintain. Instead, service-oriented access control and logging mechanisms are easier to use for expressing complex authorization and monitoring policies for SOAs.

In this paper we present a solution for a service-centered access control and logging. This solution, called BERSERK, is an implementation of the Intercepting Filter design pattern [5]. The BERSERK administrator has the power to create the intercepting entities (filters) capable of checking arbitrary authorization or logging criteria. BERSERK has the possibility to implement virtually any complex authorization criterion, namely supporting nowadays cross-domain security requirements. For instance, composing security policies that include network address validation, time validation and application's specific information are straightforward.

The paper is structured as follows. In the next section we give some motivation for this work. In section 3 we describe the generic architecture of our proposal. In section 4 we present the implementation of

BERSERK. In section 5 we analyze the performance impact of BERSERK's monitor. In section 6 we present some related work. In section 7 we present some of the future work for improving the functionality of the BERSERK's monitor. Section 8 describes how the BERSERK framework can be extended to be used on an example application. In section 9 we shortly describe the impact of BERSERK in a real scenario. Finally in section 10 we conclude the paper.

2. MOTIVATION

Traditional, monolithic applications require more effort to improve functionality than SOAs. Typical applications have a functional kernel, to which new features are added. To maintain a stable kernel is often hard because the new features may need some kernel mutation. Moreover some new features may clash with older code, resulting in more changes and therefore more spent time. These changes can be relieved using feature-oriented programming [3] but this technique requires some additional tools and consequently additional effort by the developers. SOAs don't need a confined kernel, because application's business logic is not necessarily layered. Instead, the invocation mechanism plays the kernel role. Adding functionality to a SOA does not require previous knowledge on previous application's features. This is particularly useful if the development team is large and/or if it suffers many changes over time.

SOAs have several benefits, including maintainability, reuse, development speed and parallelism, allowing junior developers to use old services as example [9,6]. However SOAs often need complex authorization mechanisms to assure a correct usage. Because small data objects can be used in many contexts, linking objects to security constrains is hard to maintain. An alternative is to associate constrains to activities using the objects instead of the objects themselves. In other words, security constrains should be linked to coarse-grained actions, like services, that access many small-grained, unprotected objects. This way it is possible to protect small objects and yet to allow different, though all correct, accesses to them. The security check is provided with each new service invocation, while the objects accessed by the service remain unaware of any changes. Adding such functionality separately in each object access would be time-consuming, error-prone and difficult to maintain.

HTTP-based interaction is a concrete case of a SOA where a response is transmitted after some processing which is triggered by a request. Both request and response potentially have security needs. BERSERK's approach uses the Intercepting Filter design pattern to provide the adequate service-centered security facilities, instead of the traditional object-centered mechanism.

Several recent platforms provide multi-model security policies, allowing to conjugate different needs from the various organization branches. However, the supported criteria are often limited to the most common criteria, such as users, groups and roles as principals and read/write access rights. For example, while allowing to simultaneously specifying open and closed policies, MACS [1] has a pre-defined set of actions, limited to database privileges like *select* and *update*, that can be controlled. Services can be used in various, and potentially heterogeneous, environments and thus they need arbitrary security criteria. BERSERK was designed for implementing virtually any security criterion.

In addition to the execution model, BERSERK provides a simple, yet expressive, logic language that configures the intercepting filter mechanism, defining **which filters** intercept each **service** and **how** should the interception behave, that is, the success conditions.

BERSERK is a **reusable** framework. That is, it allows the described generic model to be instantiated to a large number of existing products as well as future products. In section 8 we present a usage example of our solution.

2.1 The Fénix Project

BERSERK was originally developed within the Fénix Project. Fénix is an open-source university management system for incorporating all on-line campus activities and related management services. The project goals include reducing the gap between the information and the users, performing a consistent integration of data and functionality, integrating diverse application domains, achieve sustainability, reproducibility and alignment with University's organizational strategies. This project is also a case study for Software Engineering classes, enabling all students to be potential developers of Fénix. On a yearly basis, development and maintenance know-how is passed to near 200 students, building a considerable developer base. Fénix development team has about twenty developers, half of them change every year.

Fénix follows the SOA approach and provides HTTP-based services from start. It also had a custom implementation of the intercepting filter design patterns. Before introducing BERSERK its developing method was revealing severe scalability and reuse problems concerning security issues. As Fénix was growing more features were requested and therefore more security issues emerged. BERSERK main contribution to Fénix is the creation of a language capable of composing the existing filters. However, BERSERK is totally independent from Fénix and can easily be used in other projects.

3. ARCHITECTURE

To provide the proposed functionality we decided to design a flexible framework which gives administrators several options, in order to address as many situations as possible. To what this paper concerns, we decided to describe three main characteristics of BERSERK: the **intervention timing**, the **filters** and the **exceptions**.

As stated before, HTTP interaction is a good example of service-oriented interaction. Because HTTP is so widely spread — and also used in Fénix — we will often refer to HTTP as concrete examples for service-oriented situations.

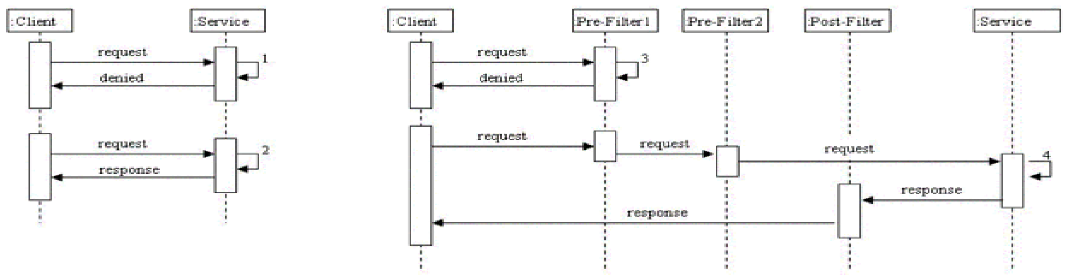
3.1 Intervention timing

SOAs have two key steps: the **request** and the **response**. The request contains the **principal**, the **service identifier** and the **service arguments**. The response is the service invocation **result**. These two key steps allow two intervention timings: (i) first the request must be verified, analyzing whether the invocation is legitimate; (ii) after request has been processed a response is produced and it may also be evaluated, changed or logged by security criteria. In BERSERK we called the first intervention timing the **pre-filtering** and the second **post-filtering**. Figure 1 shows the differences between a direct service invocation and a filtered approach, using both pre-filtering and post-filtering.

Pre-filtering assures the legitimacy and correctness of the services' requests. The decision is made taking into account both the request and the context. For example, consider the simple case of a user requesting to login. The principal is undefined, the service identifier should be login and the parameters the user's name and password. Despite the correctness of the authentication information, context-aware pre-filtering policies may reject login requests at particular hours, or restrict logins to certain subnets.

A basic post-filtering utility is for providing logging facilities, recording the activity of the system. But post-filtering can also be used to inspect results in order to determine whether the requesting principals should have access to them. For example, consider that a web page is requested. The principal may get through the pre-filtering but he/she is only allowed to view some of the page's components. The second intervention timing is responsible for filtering out all the disallowed components.

Figure 1. Direct and filtered requests



Consider now the example of Figure 1. On the left the client directly invokes the service; on the right the client invokes the same service but a filter chain intercepts the request. The business logic is the following:

(i) in the first invocation the request is denied (activities 1 and 3); (ii) in the second invocation the service is called successfully after checking the same security constrains (activities 2 and 4).

When the client directly invokes the service, the security logic is embedded in activities 1 and 2, being implemented by the service itself. In the first invocation the service is activated but it doesn't return anything.

When the intercepting filter approach is used, the client invokes the service but a filter chain intercepts the request. However, the client is totally unaware of this situation. In the first invocation Pre-Filter1 denies the access to the service after some processing (activity 3). Note that the service is totally unaffected by the disallowed request. The second invocation completes successfully. The activity 4 represents only the service logic, which can be totally free of security logic. The response is intercepted by a post-filter which also has the power to deny the client to view the response. However, in this particular case that does not happen.

3.2 Filters

Filters are the base security elements of BERSERK. A **filter** is an independent object that contains arbitrary logic, which can implement a given security criterion.

Filters intercept services, as previously described in Section 3. BERSERK provides a highly flexible way to configure the intercepting filter patterns, i.e. to configure which filter(s) intercept which service(s). The set of filters intercepting a given service is a **filter chain**. Thus, each service has an associated filter chain. Each filter chain contains a set of filters, eventually empty, composed using a logic language. A filter can belong to several chains, allowing a high **filter reuse**. Each service may be associated with multiple chains, which are sequentially activated, and each chain may be associated with multiple services as well, providing high **code** and **policy reuse**.

The **filter composing language** implements a very simple logic language which allows the basic connectors **AND** and **OR** and the unary operator **NOT**. The execution order can be changed using parenthesis. The semantics for this language is intuitive: each filter on a filter chain has an associated boolean value which tells if it succeeded or not. BERSERK evaluates the resulting boolean value for the expression and determines if the invocation should either stop or proceed. Even though the expression value can often be calculated by its partial evaluation (for example the expression 'A || B || C' evaluates to true if A is true, despite the values of B and C) BERSERK always interprets the complete expression. This allows relying on filter's side effects and avoids temporal analysis of try-and-error adaptive attacks to the system (like for PAM [8]). Plus it is guaranteed that the chain will consume the same processor time whether it succeeds or fails.

As stated before, filters contain arbitrary logic. This logic can be **reused** wherever it is needed. It is possible to implement all the security logic in a single filter. However this would be hard to maintain and would reduce the **reuse** possibilities. Instead, filters should concern on a single criterion, which should ideally be reflected on its name. For example a given filter should only check the principal IP address instead of checking both the IP and the password. The password should be checked by another separate filter. This separation allows flexible and arbitrary composition, providing a greater reuse of filters. Additionally, the separation increases the readability of the overall security policy implemented with filters and filter chains.

Depending on the needed criteria, filters can access application's domain data for determining whether it succeeds or not. Filters can also access network information (e.g. IPs), if the developer decides he/she needs to use that information for security criteria. The filter possibilities are thus virtually infinite, because a filter can process crossed-domain information to conclude on the legitimacy of an access.

3.3 Exceptions

Some systems may be compromised if something unpredicted happens. For example, some systems can be exploited if a hacker deliberately forces a runtime error.

BERSERK implements a deny-by-default policy. That is, if anything goes wrong in the invocation workflow (pre-filtering / service invocation / post-filtering) all the actions are rolled back and the access is denied. When BERSERK is running with an underlying transactional system, it rolls back all the modifications made to the database. These include the modifications made both by filters and the service itself. When a chain fails none of the subsequent chains will be executed. This is BERSERK's main limitation: the absence of mandatory chains which are executed independently of others chains success or failure.

4. IMPLEMENTATION

The current implementation of BERSERK is written in Java. This concrete implementation is released on the sourceforge website under LGPL license. The address is http://berserk.sourceforge.net.

BERSERK follows a two-layered architecture: the data storage tier, and the business logic tier.

4.1 Storage Layer

BERSERK has a flexible and fully configurable and customizable storage layer, giving developers to use their own storage flavor. By implementing the Data Access Object design pattern [4], BERSERK can delegate the storage tasks to user-defined mechanisms. These options are out of the scope of this paper.

The storage components for BERSERK are the configuration for **filters**, **chains**, and **services**. Each of these components contains the information BERSERK needs to properly run the intercepting filter mechanism. Definitions of filters and services include a transactional flag meaning whenever they must be executed in a transactional environment. Chain configuration includes the filter expression that is interpreted by BERSERK's parser.

4.2 Logic Layer

BERSERK's logic core components are the *ServiceManager* and the *FilterBrokerFactory*. *ServiceManager* is responsible by the intercepting mechanism, running the service encompassed with the proper filters. *FilterBrokerFactory* is responsible be reading the stored information about filters and build an instance of *FilterBroker* responsible by actually executing the filter chain.

ServiceManager has a cache of all filters and services used in the client application avoiding the reading overhead. Because it implements the Singleton design pattern [2] the cache is never replicated in memory.

When BERSERK is accessed for the first time, all the singletons are instantiated and the cache is built. This may affect the first service invocation performance, if the BERSERK is not initiated before. More detailed discussion on performance is made later on this article. For each service a *FilterBroker* is instantiated and kept on *ServiceManager*'s service cache. When the corresponding service is invoked, *FilterBroker* is executed and processes the service's filter chains.

5. PERFORMANCE

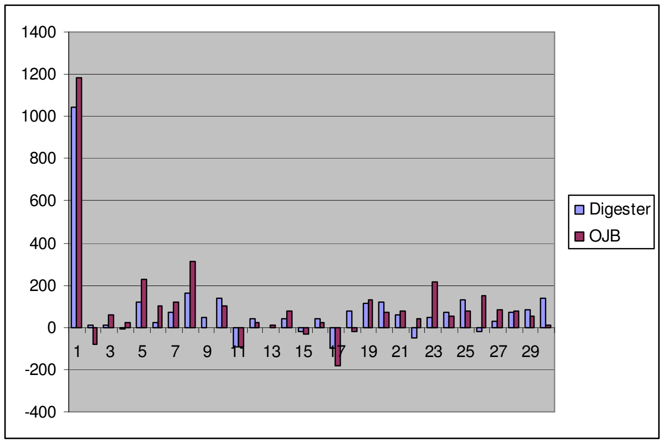
BERSERK is a monitor and therefore it introduces an overhead to the execution of services. To conclude about the loss of performance we ran a test, by configuring BERSERK to execute matrix multiplication service. The service is pre-filtered by an access control filter and post-filtered by a logger. To give a better idea of the loss of performance, the service is executed 30 times. Note that the underlying storage mechanism can also affect the execution time of the first invocation. The results using XML (Digester) and OJB configuration are display below; the comparison is made against direct java invocation:

Table 1. Total execution time

Digester (ms)	OJB (ms)	Direct (ms)
35981	36483	33588

To distinguish the first invocation, a graphic is presented bellow, showing the difference of time spent in each invocation when compared to direct invocation:

Table 2. Execution times



Like it was predictable direct invocation is often faster, since no introspection is needed. In the first invocation both Digester and OJB brokers need plenty more time to multiply the matrixes. The extra time is not spent doing the calculations but initializing BERSERK framework.

6. RELATED WORK

There are several examples of pre-filtering and post-filtering security-related actions that are applied to services. Because service invocation traditionally required some sort of client-server communication, most filtering actions where added at some level of the protocols' stack. Examples of such filters are TCP wrappers, packet filtering firewalls and application gateway firewalls.

A TCP wrapper is an application (or library) that is very helpful to decide if a given client may interact with a server (that implements a service). TCP wrappers are pre-filters that are launched before actually establishing the client-server interaction. The TCP wrapper checks the client identification (DNS host name/IP address, user name) against the required service rules and decides upon the acceptability of the interaction. If it is allowed, the client thereafter interacts directly with the server; if not allowed the server is not contacted. The connecting client is unaware that TCP wrappers are in use. Legitimate clients will not notice anything different; other clients never receive any additional information about why their attempted connections failed.

Like for BERSERK, TCP wrappers enable to detach some protection policies, implemented by the wrapper, from the service application. This allows many applications to share a common set of configuration files for simpler management. But TCP wrappers are very simple pre-filters, they do not analyze client-server application-level protocols. With BERSERK pre-filtering can be more powerful, as all service requests, which implement the application-level protocol, can be checked in detail.

Network packet filtering firewalls, like IPtables [7], follow a similar approach: when a network packet arrives it may be subject to pre-filtering rules and before a packet being transmitted it may be subject to post-filtering rules. Pre-filtering and post-filtering rules may decide upon the acceptability of the packet or even change its contents. IPtables rules are based on packet characteristics, for example its source and destination network addresses, transport protocol, connection state, etc. For each packet that arrives, IPtables searches for a matching rule. If a match is found, IPtables provides pre-defined actions (like drop, accept, etc) to decide upon the packet's acceptability.

All this activities are likewise possible with BERSERK, but this is more generic and more flexible. BERSERK must be more generic as it to be used in a wider range of situations. It can infer on any data source it has access to (for example, the application's domain database) while IPtables' domain of analysis is mainly limited to network packets and communication protocols. BERSERK is more flexible because it allows the actions (filters) to contain arbitrary code and to take arbitrary decisions. There is not a pre-defined set of actions, like in IPtables, instead a wide range of actions that may be encoded in filters.

Furthermore, BERSERK provides a more powerful mechanism to implement security policies: the **filter composing language**. Expressing conjunction or disjunction cases in IPtables is usually a complex and error prone task.

In the conjunction case the packet is forwarded from chain to chain; each chain adds a condition. In the disjunction case, the package may fail in a condition, but a subsequent check will match the packet. This is clearly hard to set up and even harder to maintain.

Application gateway firewalls are application-level servers that stand between clients and servers (services). Application gateway firewalls are more powerful than the packet filtering ones because they may apply security criteria to protocol contents (requests and replies), while that is very hard to do at network level. But for each application-level protocol there must be a different gateway firewall, there is not a generic, parameterized solution. And, even worse, gateway firewalls usually do not share code between them. With BERSERK we can solve both problems. BERSERK is a generic platform that provides the basic mechanisms for building complex, service invocation oriented security policies. And, additionally, those security policies use filters and filter chains that can be easily reused because they may be totally independent of the application-level protocol.

7. FUTURE WORK

Several features may be added to improve BERSERK functionality. Below is a list of proposed features, as well as a short description and motivation.

<u>Parameterized Filters</u>

By the release of BERSERK V1.0 filters only have access to service's parameters. That is, there is no way to explicitly pass filters its own parameters. Adding expression power to filter composing language, enabling first order predicates is a possible solution to this limitation. With this feature filters become more reusable. For instance, a filter for checking if the client's IP address belongs to a set of authorized networks could receive as parameter a list of authorized network masks.

<u>Service Groups</u>

Services with similar semantics should be grouped. This would allow the macro-management of security requirements for services groups. for example, delegate entire services groups' access on users. To achieve this functionality the service type concept and associations between services and service types should be created

<u>Mandatory Chains</u>

As stated before, BERSERK is not capable of running subsequent chains if a give chain fails. Therefore some pre-filtering chains will not be executed and the post-filtering chains will not be executed at all. The solution is the creation of mandatory chains, that is, chains that are executed independently of the success of previous chains. Consider a post-filter that checks if the principal is accessing the system from a certain IP. Independently of the success of the access it may be desirable to take a certain action on post-filtering. This filter chain should be mandatory.

8. USAGE

As stated before, BERSERK is a reusable framework. This chapter presents a very simple usage for our solution. The focus of this section is not on building complex services but to exemplify how to integrate an existing business-logic into BERSERK's filter and services.

8.1 Service

In BERSERK's perspective any class which implements the IService interface and has a run() method is a service. Developers can implement services with arbitrary logic and then execute them via the monitor entry point (execute() method). Suppose the service is called "A". In order to get BERSERK to execute it the execute() method should be called like execute(principal,"A",arguments), where arguments are the parameters for the class' run() method.

Below is an example of a service which prints a string to stdout:

```
public class StdoutWriter implements IService
{ public void run (String string)  {
      System.out.println(string);  } }
```

8.2 Filters

Filters must implement the IFilter interface which declares the execute() method. Filters have access to the principal and to service's arguments in order to analyze the invocation legitimacy. The following code implements a filter which verifies if the principal is an object which can be successfully converted into an integer:

```
public class RequesterIsIntegerFilter extends AccessControlFilter
  { public void execute(ServiceRequest request, ServiceResponse response) throws FilterException {
     if (request.getRequester() instanceof String)
     try {
      new Integer((String) request.getRequester()); }
      catch (NumberFormatException e)    {
        throw new AccessDeniedException("could not convert " + request.getRequester() + " to integer"); }
      else
    throw new AccessDeniedException("could not convert " + request.getRequester() + " to integer"); }}
```

8.3 Configuration

A XML configuration for this example would be:

```
<berserk><filterDefinitions><filter>
                    <name> RequesterIsIntegerFilter </name>
                    <implementationClass>filters.RequesterIsIntegerFilter</implementationClass>
                    <description>Checks if the requester can be converted to a java.lang.Integer</description>
                    <isTransactional>false</isTransactional></filter></filterDefinitions>
    <filterChainsDefinitions><filterChain>
                    <name>AccessControl</name> <expression>OnlyIntegers</expression>
                    <description>Access Control Chain</description>
                    <invocationTiming>1</invocationTiming>
                    <filterClass>filters.AccessControlFilter</filterClass>
          </filterChain></filterChainsDefinitions>
    <serviceDefinitions><service>
                    <name>StdoutWriter</name>
                    <implementationClass>services.StdoutWriter</implementationClass>
                    <isTransactional>false</isTransactional>
                    <filterChains><chain name=" AccessControl "/></filterChains>
    </service></serviceDefinitions></berserk>
```

Note that in this simple example the power of the filter composing language is not used.

9. EXPERIENCE OF BERSERK IN FÉNIX

Fénix original security monitor had a strict declarative language, only allowing conjugation of filters. If a disjunction was needed developers should create a new filter to explicitly allow both criteria. With the introduction of BERSERK the number of needed filters drastically decreased because with few filters one can compose a large number of filter chains. Developers seldom develop new filters, since the existing ones already address a large number of security criteria. Instead they search for the filter(s) they need and use the filter composing language (and therefore a filter chain) to express the needed security policy. Fénix is using BERSERK as an external library allowing easier and frequent upgrades.

10. CONCLUSION

We presented BERSERK, a service-oriented security mechanism that implements the Intercepting Filter design pattern. This design pattern is especially adequate for SOA, and namely for Fénix, because it was design deal with a pure service-oriented and stateless architecture: the HTTP protocol. We consider that BERSERK is applicable to any application which needs a flexible and cross-domain security mechanism.

BERSERK is an evolution of the original Fénix implementation for the intercepting filter design pattern. Its main contribution is the introduction of a filter composing language, allowing higher filter reuse and the possibility to run chains in two different timings — before the service invocation and after the invocation of the service and before returning the results to the invoker).

BERSERK's main limitation is the lack of mandatory chains, that is, chains that will be executed independently of the success of other chains. This is particularly useful when using a BERSERK based logging system. Currently BERSERK is deployed on a service-oriented architecture with about 2.000 users, allowing fast, coherent and scalable authorization configuration, evolution and extension.

References

[1] E. Bertino, S. Jadodia, and P.Samarati. Supporting multiple access control policies in database systems. In Proc. of the 1996 IEEE Symp. on Security and Privacy, pages 94–109, Oakland, CA, USA, 1996. IEEE Computer Society, IEEE Computer Society Press.

[2] Brian d Foy. The singleton design pattern. The Perl Review (http://www.theperlreview.com/Articles/v0i1/singletons.pdf).

[3] Product-Line Architecture Research Group. AHEAD - Algebraic Hierarchical Equations for Application Design. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[4] Sun Microsystems. JavaTM BluePrints - Guidelines, Patterns, and code for end-to-end Java: Data Access Object. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html, 2003. Consulted on October the 25th, 2003.

[5] Sun Microsystems. JavaTM BluePrints - Guidelines, Patterns, and code for end-to-end Java: Intercepting Filter. http://java.sun.com/blueprints/patterns/InterceptingFilter.html, 2003. Consulted on October the 25th, 2003.

[6] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing: Introduction. The ACM Digital Library, 2003.

[7] Rusty Russel. The iptables manpage, 1998. Consulted on December the 25th, 2003.

[8] Vipin Samar. PAM - Pluggable Authentication Modules. http://www.opengroup.org/tech/rfc/rfc86.0.html, 2003. Consulted on January the 3rd, 2004.

[9] Michael Stevens. The benefits of a Service-Oriented Architecture. http://www.developer.com/tech/article.php/1041191, 2002. Consulted on December the 23rd, 2003.