

Improving Interoperation Security through Instrumentation and Analysis

A framework and case study

David Llewellyn-Jones, Madjid Merabti, Qi Shi, Bob Askwith, Denis Reilly

School of Computing and Mathematical Sciences

Liverpool John Moores University

James Parsons Building

Byrom Street

Liverpool, L3 3AF

UK

{D.Llewellyn-Jones, M.Merabti, Q.Shi, B.Askwith, D.Reilly}@ljmu.ac.uk

ABSTRACT: Interoperation between heterogeneous services results in a variety of serious security concerns, from privacy through to authentication and policy enforcement. We look at composition analysis techniques, enabled using instrumentation, as a means of improving security in interoperating systems. The techniques described harness the system of systems nature inherent in all interoperating system configurations. We present the ongoing development of a framework for combining instrumentation and composition analysis capabilities in a novel manner and discuss a case study involving the prevention of data leakage through access control analysis.

KEY WORDS: Interoperation, Security, Component composition, Instrumentation, Access Control, middleware.

1. Introduction

The implications for security of interoperation between systems are both broad and potentially serious. The interfacing of heterogeneous systems can cause particular difficulties, especially where individual services are used that were not designed with security in mind.

A number of theories and architectures exist concerned with the security of data and services between systems. For example, CORBASec offers a framework intended for use with CORBA (Lang, *et al.* '02). This provides a very practical solution to some of the security difficulties involved in interoperation between systems. A number of theoretical composition results may also apply where interoperability plays a role, such as Non-interference (Focardi, *et al.* '97) and Composable Assurance (Shi, *et al.* '98), however to our knowledge no system exists that draws on both practical instrumentation techniques and theoretical composition results to provide an applicable analysis framework. The novelty of our approach therefore derives from this use of instrumentation techniques to allow the practical application of security composition analysis, and has particular application where interoperability is concerned. A framework for such analysis is presented using instrumentation to provide the input parameters needed for analysis. An example scenario will be considered, and we will derive the functionality needed of a service for the instrumentation to perform effectively, thus allowing such security techniques to be used.

In general, there are a range of security issues that affect or are affected by interoperation – such as policy reconciliation and access control – and a number of solutions have been proposed to tackle them (for example, Dawson *et al.* (Dawson, *et al.* '00) and Kokolakis and Kiountouzis (Kokolakis, *et al.* '00)). There are numerous others for which specific solutions often exist, including buffer overruns, authorisation issues and insecure protocols. Rather than concentrate on a framework or methodology to tackle particular issues, we present a novel approach that relies on dynamic analysis of an interoperating federation, and which is extensible so that future security concerns may be addressed if they are amenable to the method of analysis employed. To demonstrate the use of the framework we have concentrated primarily on access control difficulties in this paper. The leakage of data through unenforced access mechanisms is a serious concern, since it opens the potential for private data to be read or amended by unauthorised people or services. In a distributed environment this difficulty is exacerbated by the fact that having control in one part of the system does not provide any guarantee as to the effectiveness of the access control

mechanisms in another part of the system. The result is a real potential for data being passed to unauthorised services.

The structure of this paper is as follows. In the next section we discuss an overview of the framework for analysis of security properties in system of systems. This is broken down into its two major parts: section 3 considers instrumentation and section 4 considers composition analysis. Both are essential elements in the overall framework. Whilst the framework we present is extensible, it is instructive to consider a specific example of its use in greater detail. We do this in section 5, where the case of access control and data leakage is investigated. Finally we conclude and discuss future work in section 6.

2. A Framework for System of System Analysis

The crucial feature that we harness for the application of our security framework is the componentised nature of interoperable systems. Such systems will by definition be comprised of more than one smaller systems or services working together to form a larger application, or federation, potentially with additional interface components used to allow successful interoperation.

For example, services accessed via a website may be based on multiple lower-level services distributed across a variety of systems and organisations. Figure 1 shows an example scenario where a web service provider offers a service based on data provided by a number of organisations (*e.g.* it could be map data and traffic data from two separate organisations).

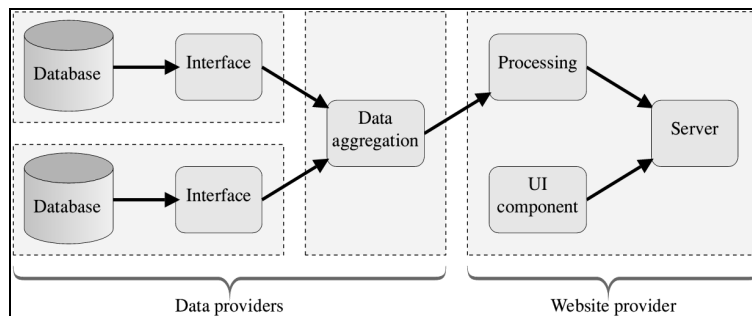


Figure 1. Example web service interoperation

Composition analysis provides a method for establishing security properties based on information concerning the properties of the individual components and the topological configuration of the network or interfaces between those components.

A variety of properties relevant to security can be established in this way. For example, our previous work has looked at buffer overrun vulnerabilities where composition analysis can be used to establish whether such vulnerabilities are likely to be triggered, or can be circumvented using carefully chosen intermediary components (Llewellyn-Jones, *et al.* '05a).

We present the ongoing development of our framework based on instrumentation and composition analysis below. An implementation of the framework using Jini (Jini '01) and the MATTS composition analysis engine (Llewellyn-Jones, *et al.* '05b) will be discussed, followed by an example application enforcing access control policies to prevent data leakage across interoperating systems.

The framework incorporates two phases, as shown in Figure 2. The first establishes dynamic dependencies between application services from a federation using instrumentation services at run time. The second undertakes a composition analysis based on these dependencies. For this second stage, during the analysis process, additional information regarding individual components is generally required for an accurate result to be obtained. For example, in the access control case study, information concerning the access rights of individual components may be needed, along with information concerning the internal dependencies that exist between interface hooks (*e.g.* input and output channels). Such information depends on the property being tested and the dependencies established during the first phase.

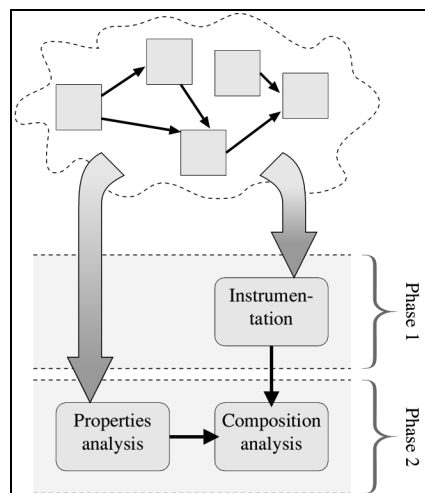


Figure 2. *Analysis process*

We have deliberately chosen to separate instrumentation from the composition engine so as to improve flexibility and allow either to be re-implemented or swapped if an alternative method is desired.

Composition analysis and property analysis are both undertaken in parallel during the second phase. The reason for this is that the properties required are liable to change throughout the analysis process, based on information already obtained. For example, properties of interface hooks that are not bound to may not be needed for the analysis to be performed, but the fact that the property will not be needed cannot be established before we know the dependencies. In the case of access control, we can avoid the need to check access rights for components that are found not to access any external data.

We will now consider the two phases in more detail, along with details about the current implementation.

3. Instrumentation Phase

Instrumentation works in conjunction with middleware technologies (*e.g.* CORBA, Jini, *etc.*) to unobtrusively provide dynamic information about the behaviour and performance of distributed application services. Although in its generic form instrumentation is intended to refer to the class of instruments that may be capable of detecting various properties, for our current purposes we are interested only in using them to establish *dependencies* between services. For a more complete exposition, see Reilly (Reilly, *et al.* '03).

Our current implementation is based on Jini Middleware Technology, a Java middleware developed by Sun Microsystems (Arnold, *et al.* '99). In theory any middleware solution satisfying the minimal requirements set out below could have been used.

For the purposes of instrumenting a service, we utilise a design pattern. This allows us to create services that can be interposed between communicating services with interfaces implemented at run-time to relay calls between the communicating services. In addition to relaying calls, the instrumentation is also able to record and forward properties and details about the services and their communications. Interface creation relies on reflection for its dynamic capability. However, in the case that reflection is not available for a particular middleware solution (*e.g.* in the case of CORBA services) or for time critical applications where the delay caused by reflection is unacceptable, a static interface may also be used, although such interfaces would need explicit generation for each distinct service being monitored.

The approach used to represent dependencies is based on extensions to the work conducted by Hasselmeyer (Hasselmeyer '01) and makes use of Jini's `Administrable` interface (Jini '01). The approach does impose one condition in order to access service bindings, namely that the services are required to implement

the `Administrable` interface. Any service that implements the `Administrable` interface must define the `getAdmin` method, which itself returns an object that implements the `Dependable` interface. Through this the bindings associated with an application-level service can be discovered by accessing the `getBindings` method, as follows.

```
public interface Dependable {
    public Class getDeclaringClass ();
    public Object[] getBindings ();
}
```

Through querying this method, the instrumentation is able to establish service dependencies. Details of all dependencies within a federation are accumulated by the instrumentation, for the dynamic generation of a digraph that can then be passed on to the composition analysis engine in the second phase. Figure 3 shows a graphical representation of the dependencies between a selection of Jini services, as established through this method of instrumentation.

Of course it will be apparent that services not implementing the `Administrable` interface cannot be queried in this way. In the future we aim to explore alternative instrumentation methods that do not impose this requirement.

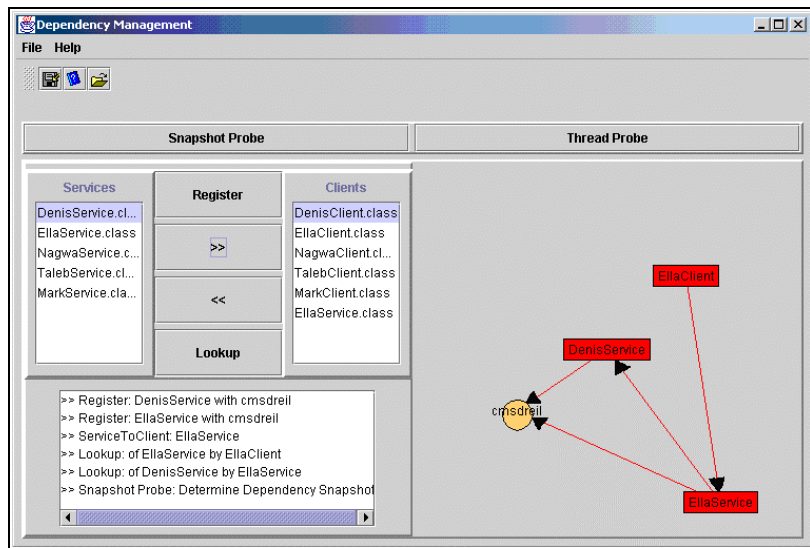


Figure 3. Graphical representation of dependencies between services

4. Composition Analysis Phase

The second phase makes use of the MATTS composition analysis engine. The engine is still being developed, but in essence, it provides a means of analysing systems of systems based on their dependencies. The analysis is directed by a script file in a simple XML format. The script file describes a process that should be followed for each property that is to be tested for. MATTS is able to interpret this script and follow the processes, ultimately resulting in a verification or refutation of the property as it applies to the configuration of the services.

Assuming a suitable script file is being adhered to, the most important information needed by MATTS to undertake its analysis is an overview of the dependencies or links between the components of a system. As has already been explained in the previous section, this information is provided by instrumentation that dynamically reads it during the life span of a federation. As dependencies change, so the federation is re-evaluated based on its new configuration.

During the analysis, it often transpires that additional information is needed to complete the analysis and provide an accurate result. This is because composition results invariably depend on both the composition structure and the specific properties of individual services. These latter properties may be queried at any time during the analysis.

The result of the composition analysis stage yields whether the security property being tested for is satisfied or not. As previously mentioned, this may be whether a particular configuration is liable to trigger a buffer overrun vulnerability, or as we will see later, if it is liable to cause an access violation. Further properties may be tested for through the application of additional scripts.

Such results are important for interoperability security, since they can tell us whether a particular configuration of services will be safe to deploy and use.

At present, the system can be used as a ‘warning’ indicator, to tell whether a potential security vulnerability is present in a system. In future development we hope to produce a ‘pro-active’ version that not only establishes possible problems, but also provides dynamic solutions, *e.g.* through the generation of intermediary services that marshal data safely between otherwise potentially vulnerable services.

The process can be understood most clearly by considering an example.

5. Case study: access control

Access control provides an important security mechanism for preventing unauthorised access to data and retaining data privacy. It can, however, present problems for interoperating systems, since access control mechanisms may not be available for all services and even when available, may not provide uniform access control methods to support interoperation. An access control mechanism that can be layered on top of existing services may therefore provide an effective way to impose access control without the need to significantly modify underlying systems.

Moreover, access control mechanisms often fail to account for potential data leakage that can occur as information is passed between distributed, interoperating systems. If an authorised service passes private data to a second receiving service on a different platform, how can the sending service be certain that the receiving service will not pass the data onwards to a further unauthorised actor, even if the receiving service is itself authorised to have access to the data? Using the framework described in the preceding sections we can easily enforce access control whilst simultaneously preventing data leakage of the nature just described.

Each file in a filesystem is stored with access control data associated with it determining the situations in which the file may or may not be accessed. For example, Unix-like systems store UIDs, file permissions and related meta-data in a file's inode. Similar data is held by the operating system about the access rights of processes.

The aim of the case study is to ensure read access to data from a file is granted only to those services with sufficient access rights, and similarly for write access.

The benefit provided by the dependency information is that data flow across services can be determined, and a model constructed of the data flow throughout the federation of services. This model can be established at run time and re-analysis can take place whenever dependencies change.

A simple MATTS script, a fragment of which is shown below, describes the requirement that data may only flow into or out of a component if that service has sufficient access rights to read from or write to the originating or destination file respectively.

```
1. <sandbox id="s2" config="c1">Read access control check</sandbox>
2. <property id="idAuth">Level component is authorised to</property>
3. <configuration id="c1" init="1">
4.   <component id="c2">
5.     <input format="*"/>
6.     <input id="in1" init="0" format="*"/>
```

```

7.     <process config="check" action="check=@n"/>
8.     <process id="auth1" init="1" action="@a[@n][0][idAuth]"/>
9.     <process cond="result &lt; auth1" action="c1=0"/>
10.    <output format="*" cond="c1==1"/>
11.    <component>
12.        <process config="check" action="check=@n"/>
13.        <process id="auth2" init="1" action="@a[@n][0][idAuth]"/>
14.        <process cond="result &lt; auth2" action="c1=0"/>
15.        <input id="in2" init="0" format="*" />
16.        <output format="*" config="c2" follow="fresh" cond="c1==1"/>
17.        <output format="" cond="c1==1"/>
18.    </component>
19. </component>
20. </configuration>

```

The MATTs parser works by maintaining both a current position in the script and a current position in the component structure. Each element in the script determines how both of these positions should be updated at each step.

The script above tells the parser to follow the dependencies between components (lines 6, 10, 15 and 16). At each component a process is undertaken (indicated by the `action` attributes in lines 8 and 13, but not actually shown in full) to test the lowest read access required by the data flowing through the system and into that component. A test is then performed to establish if the component has sufficient access rights to read this data (lines 9 and 14). The script continues only if access is granted (lines 10 and 16). The script results in a true result only if it successfully completes, hence will return false if access would not be granted at any of the components in the system.

The script can be augmented by considering slicing properties of components that allow internal dependencies between inputs and outputs to be established in addition to the dependencies between interoperating services. Slicing is an established method of code analysis that determines all subsets of a piece of code that are, for example, affected by a given variable (Weiser '81). The process can therefore be used to establish if the data received on a particular input has an effect on the data sent on any given output. By introducing this into the script above we can refine the analysis with the consequence that access restrictions that would otherwise be imposed on interoperating systems may on a number of occasions be safely relaxed without affecting the overall security of the system. At present, however, it has only been possible to apply such techniques to a simple virtual machine and bytecode implementation. There is no theoretical reason why the same techniques could not be applied to Java bytecode and this is something we hope to look at in future work.

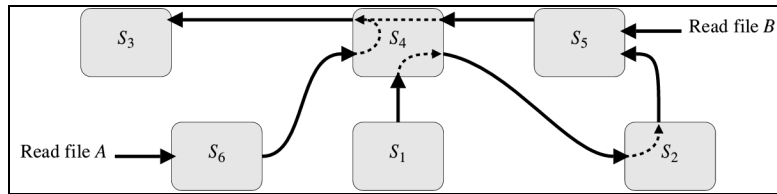


Figure 4. *Service interactions*

As a particular case, consider a number of services S_1, \dots, S_6 that interoperate via distributed middleware as shown in Figure 4. The dependencies (shown as solid arrows) between services can be established through instrumentation. We assume that different access rights are required in order to access the data contained in files A and B . The result of the analysis will tell us that service S_6 must have sufficient rights to access file A ; service S_5 must have sufficient rights for access to file B ; services S_3 and S_4 must have right to access *both* files A and B ; whilst the read access rights of services S_1 and S_2 do not matter, since no data is read into the service. Note the internal dependencies are marked with dotted arrows in the figure.

The framework described here is able to automatically establish these results and, based on the access rights information supplied about each component and file, establish whether any access violations or data leakage is likely to occur after deployment.

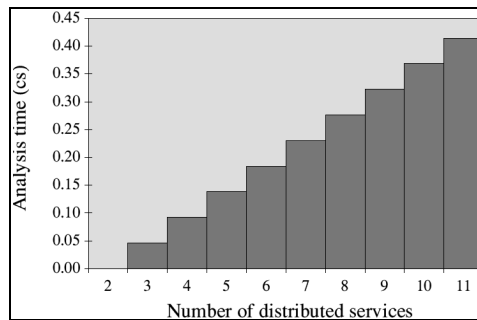


Figure 5. *Distributed service analysis timings*

We applied this technique to a linear configuration of minimal services designed to send small sequences of data to each other across a network. Timing results for this are in Figure 5. In this case, with simple dependencies between services, we can see that analysis time is linear in the number of components. These timings represent times for pure analysis. They do not take into account delays during instrumentation

readings or caused by network overheads. Although the times shown are relatively small, the question of delays introduced in larger systems is an issue that needs to be considered carefully.

6. Conclusion

In this paper we presented a framework intended to improve security for interoperating systems. The framework is built using techniques of instrumentation – to establish dependencies between services – and composition analysis – to assess the security implications of using services in conjunction with each other – in a two phase process.

We explained how the framework can be applied as a layer on top of existing services. Our current implementation – based around Jini middleware technology – requires services to implement the `Administrable` interface in order for the instrumentation to work effectively. This presents an obstacle to applying the technique to legacy systems; however we hope to overcome this limitation in future work by looking at alternative methods to instrument dependencies.

At present the framework is in an early phase of implementation and whilst both instrumentation and analysis parts have been implemented separately using Jini and MATTS respectively, there is still work to be completed in integrating the two effectively. As one aspect of this, the direct analysis engine used to establish service properties through direct inspection of bytecode currently applies only to a simple virtual machine bytecode and we hope that this might ultimately be extended to work with Java bytecode and other low level code.

Looking beyond this, a significant improvement would be to design a more proactive architecture, not only capable of detecting security breaches, but also of mitigating the problem by interposing additional services into the framework.

As with any security mechanism, we make no claims about providing a panacea for security issues. Nevertheless we do believe the combination of instrumentation and composition analysis is likely to prove a useful tool, especially in the field of interoperation, since it has the potential to provide a sophisticated means of establishing security properties in an automated manner, even in heterogeneous environments where the framework can be overlaid onto existing and legacy systems. The earlier study of an access control mechanism highlighted a case in point, and the extensible nature of the system using scripting allows for future advances to be incorporated into the system easily. Whilst scripts apply to any component configuration, a particular script is however specific to a particular problem scenario (such as access control in the case here).

Besides the general improvement of the framework to make it more integrated and universally applicable, the largest challenge for a system such as this is the establishment of new security scripts. We believe that the practical establishment of such composition analysis techniques provides the prospect of exciting potential in the area of security in solutions for composition and interoperability.

References

- Arnold K., Osullivan B., Scheifler R. W., Waldo J., Wollrath A., O'Sullivan B., Scheifler R., *The Jini Specification, Second Edition*: Addison-Wesley, 1999.
- Dawson S., Qian S., Samarati P., Providing security and interoperation of heterogeneous systems, *Distributed and Parallel Databases*, vol. 8, pp. 119-145, 2000.
- Focardi R., Gorrieri R., Non Interference: Past, Present and Future, *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, California, USA, 1997.
- Hasselmeyer P., Managing Dynamic Service Dependencies, *12th International Workshop on Distributed Systems: Operations & Management*, Nancy, France, 2001.
- Jini, Architecture Specification 1.2, Sun Microsystems, 2001.
- Kokolakis S. A., Kiountouzis E. A., Achieving interoperability in a multiple-security policies environment, *Computers & Security*, vol. 19, pp. 267-81, 2000.
- Lang U., Schreiner R., *Developing secure distributed systems with CORBA*. Norwood, MA, USA: Artech House, 2002.
- Llewellyn-Jones D., Merabti M., Shi Q., Askwith B., Buffer Overrun Prevention Through Component Composition Analysis, *COMPSAC 2005*, 2005a.
- Llewellyn-Jones D., Merabti M., Shi Q., Askwith B., MATTS Technical Manual, Liverpool John Moores University, Liverpool, 2005b.
- Reilly D., Taleb-Bendiab A., Dynamic instrumentation for Jini applications, *Software Engineering and Middleware. Third International Workshop, SEM. Revised Papers*, Orlando, FL, USA, 2003.
- Shi Q., Zhang N., An effective model for composition of secure systems, *Journal-of-Systems-and-Software*, vol. 43, pp. 233-44, 1998.
- Weiser M., Program Slicing, *5th International Conference on Software Engineering*, 1981.