RZ 3630 (# 99640) 09/12/2005
Computer Science 17 pages

Research Report

A Vulnerability Taxonomy Methodology applied to Web Services

C. Vanden Berghe, J. Riordan

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

A Vulnerability Taxonomy Methodology applied to Web Services

July 2005

Abstract

We present a methodology for taxonifying vulnerabilities based on the likelihood that they will be present in a certain system. It attempts to capture and formalize the intuition that allows security professionals to make predictions about likely security problems. The method exploits the realization that the vulnerabilities present in a system are related to the set of properties that define the system. By modeling it using a selection of relevant properties and correlating this with the body of knowledge on historic vulnerabilities and the systems in which they lived, we obtain a heuristic of the likelihood that these vulnerabilities will reappear in a new system. The predictive nature of this methodology serves as an early warning for systems before they are widely deployed. As an example we apply our methodology to Web Services, thereby providing a tool to focus efforts in securing Web Services.

1 Introduction

Service-oriented architectures are a major evolutionary step in the creation of distributed systems. J2EE [25] and .NET [18] have embraced this trend and provide tools and support for Web Services (WS), which are practical implementations of the principles of a Service-Oriented Architecture (SOA). The architecture introduces a service layer around software components and can be regarded as the logical next level of abstraction after the componentization of software. Such an abstraction allows for the creation of rich, highly-distributed applications by enabling dynamic discovery and loose-coupling of heterogeneous components that span organizational boundaries.

The flexibility of these highly-distributed applications enables new and interesting possibilities. At the same time it creates numerous security challenges that need to be addressed for the ultimate success of the technology. This need was quickly recognized and has resulted in numerous standards addressing various aspects of SOA security. Examples include the WS-Security family of standards [22], SAML [20], XACML [19] and the Liberty Alliance specifications [14]. Each of these specifications targets some (not necessarily disjoint) collection of the security needs in the service layer.

By contrast, security concerns regarding implementations and operation have received far less attention. This disparity is largely rooted in the youth of WS rather than in the relative importance of the two classes of security failure. There are simply too few deployed and active instances of WS to generate an adequate body of knowledge about real-world security failures and the vulnerabilities behind them.

Our methodology is predicated upon two principal assumptions: The first is that one can usefully think of security vulnerabilities as being present in the properties of a system rather than in the system itself. The second is that we will not see fundamentally new vulnerabilities in SOA but rather variations of those already present in existing systems. Based on these assumptions, we develop a general-purpose methodology for generating predictive vulnerability taxonomies and apply it to WS. Some vulnerability types are more likely to arise than others in such architectures, depending upon the properties of the individual components.

As a testcase example, we apply our methodology to WS which illustrates how the resulting taxonomy can used to reason about practical security in WS. It indicates the degree to which the WS components are prone to several classes of vulnerability, thereby providing insight into an optimal use of finite resources for securing WS.

The paper is structured as follows. In Section 2 we present the aspects of SOA and WS that are the most important ones to our work. In Section 3 we give a brief overview of practical security, which is our security layer of our focus. Section 4 provides some insights into taxonomies in general and vulnerability taxonomies in particular. Section 5 lays out and explains our methodology for creating vulnerability taxonomies, using the example of WS. Section 6 details the actual results of this application to WS. Section 7 presents related work, and Section 8 concludes with a summary of our findings.

2 Web Services

In this section we present an overview of security-relevant aspects of WS. We start by outlining the principles of a Service-Oriented Architecture (SOA). This is followed by a description of WS and its relation to SOA. Finally we discuss the implementation-related aspects of WS.

2.1 Service-oriented architecture

SOA is a software development methodology aimed at improving the manageability of today's increasingly complex systems. A good understanding of the methodology's principles, and the reasoning behind them, is essential for understanding WS security, because WS is a practical implementation of the more general SOA concept.

Object-oriented and component-based software development methodologies have successively increased our ability to develop and manage complex applications. They are based upon the abstraction of *components*, which increases re-usability of software functionality. While this notion is very powerful, interfaces between components tend to be rather tightly coupled and inflexible. It therefore does not scale to distributed applications spanning organizational boundaries, as they require that developers understand and control each component in the application. The integration of components coming from a wide variety of organizations is therefore a daunting task.

SOA can be seen as an extension of the ideas of component-based software development aimed at increasing the flexibility of application integration. It introduces the concept of a service layer, which provides a framework to address the complexity of integration in highly-distributed heterogeneous systems. Conceptually, SOA represents a model in which loosely coupled pieces of application functionality are published, located, consumed, and combined over a network. Such exposed functionality is called a service. Services adhere to the following principles [17]:

- They deliver some *self-contained* and *composable* functionality that is advertised in terms of the task performed by the service. The focus is on *what* is performed instead of *how* it is performed. In addition, it should be possible to use the service as part of a larger application.
- They are *loosely-coupled* and can be *discovered and invoked dynamically.* Traditional applications depend on tight interconnections of all subsidiary components, therefore changing existing systems is exceedingly difficult. Loosely coupled systems require less coordination and allow for more flexible reconfiguration, thereby enabling the use of services across organizational boundaries.
- They have *well-defined interfaces* and *coarse-grained interactions.* Coupling systems is generally difficult, so it is important to keep interfaces simple and the number of interactions minimal. This keeps the dependencies between services manageable. In addition, it facilitates more thorough testing of the service and makes the interactions easier to understand.
- They are *network-addressable* and *location-transparent.* The ability to invoke the service over a network is key to the concept of SOA. It allows applications to use the services best suited to their needs, independently of their location.
- They are *interoperable.* This is a necessary characteristic of heterogeneous systems that communicate and cooperate. Towards this end, the service consumer and provider must agree on a mediating protocol and data-format on to which they map their platform-specific characteristics.

2.2 Web Services

WS are practical implementations of SOA principles. The term itself is not very well defined and is used both in a conceptual and in a technical sense. Conceptually, WS are XML-based SOA that use standard Internet transport protocols[1]. In the technical sense, WS refers to specific collections of standards, tools and practices to implement the WS concept. Different styles of WS exist such as SOAP Web Services [28], REST Web Services [10] and XML-RPC [27].

SOAP WS are the most common form. Specific to SOAP WS is the use of WSDL [29] as the service

[1]The term "transport protocol" now includes higher level protocols such as HTTP and BEEP

descriptor, SOAP [28] as the messaging protocol and UDDI [21] as the directory protocol. WS-I [31], an industry organization created with the mission to promote interoperability among WS, recommends this style of WS (with some additional restrictions) for interoperability reasons. Many of the higher-layer WS protocols, addressing issues such as workflow, business processes and security, are built atop these core specifications. The main advantage of this particular flavor of WS is its wide adoption and availability of supporting tools. The complexity of the protocol stack is its major disadvantage.

REST Web Services and XML-RPC provide a less complex alternative to SOAP Web Services, but are also less widespread. REST Web Services adhere to the concepts of the Representational State Transfer architectural style [10]. In this model, services are seen as resources addressable by URI's. Accessing and managing these services is done exclusively through the HTTP verbs GET, POST, DELETE and PUT, where each one has well-specified semantics. Most of the data transmitted is in the form of XML.

REST Web Services are very light-weight by design because they only use URI's, HTTP and standard resource representations, such as XML, jpeg, etc. They possess additional desirable properties such as improved performance due to better caching opportunities. XML-RPC, although often considered as another light-weight protocol for WS, does not adhere to all of the aforementioned properties of a SOA. For example, XML-RPC does not publish application functionality with a service abstraction, but is merely a convenient way to open functionality to the Web.

In this paper we focus on SOAP WS because of their wide acceptance. Many of the conclusions also hold for other WS flavors, as the fundamental concepts are largely shared.

2.3 Web Services implementation

There is no universally accepted set of tools for developing WS or platform for their deployment; this would indeed defeat much of the purpose of WS. Instead, toolkits exist for many platforms. Nevertheless, it is safe to say that most WS will be developed in higher-level languages and deployed in managed-execution environments. Two concrete examples of WS implementations are J2EE [25] application servers and the .NET framework [18]. They are already widespread and provide good tool support for SOAP Web Services.

Development toolkits for J2EE and .NET provide wizard-based tools that take care of deploying the selected functionality as a WS, and even publish the WS with one or more directory services. The application servers handle message parsing, cryptographic transformations, data-type mapping and transparent dispatching. Although the specifics differ, both platforms abstract the WS details away from the applications.

Subsequent discussion of WS implementation-related properties is limited to properties shared by common implementations, as opposed to aspects of a specific implementation (thus rendering the results more widely applicable).

3 Practical Security

Our work focuses on the layer of security that addresses real-world vulnerabilities caused by a system's implementation, deployment and management. We refer to this layer as "practical security". It occupies a position complementary, yet related to, security specifications (such as those of WS). Specifications determine the way in which a system *should* behave, whereas the implementation determines the way in which it *does* behave.

Although we are concerned with such concrete and specific vulnerabilities, it makes sense to speak of them as abstractions. Any reader of a security vulnerability announcement list, such as Bugtraq [11], will be eventually struck with a certain sense of *déjà-vu*. On an average day several new vulnerabilities are published, but only very seldomly something fundamentally new is discovered. Instead, they are variations that differ only in the specifics, but not in the *implicitly-adjudged underlying reason* why the system is vulnerable.

The foundation of our subsequent analysis is that one can think of vulnerabilities as being manifest in the implicitly-adjudged underlying reason for their existence rather than in the systems themselves. We therefore propose a model that describes these reasons in terms of a collection of properties. This is analogous to the epidemiological practice of ascribing a disease to a population rather than an individual. This epidemiology applies to our work in three distinct manners.

- Code analysis and testing resources are limited by both time and cost. As such, they should be utilized towards maximum effect. Knowledge of what sorts of problem are likely to occur in which places is thus quite valuable.

- Properties should be seen as characteristics of the complete process of design, development, use and maintenance of the system. Determination of the property sets is both important and potentially difficult. The properties that influence the security of a system are not limited to technical or implementation-oriented properties, such as which development tools are used. They also include nontechnical issues such as security-awareness of the users of the system. These properties should ultimately describe the system as completely as possible.
- The influence-map from properties to vulnerabilities is not necessarily injective: indeed, a group of properties may be required to influence the presence of a vulnerability. For example, buffer-overflows are commonly attributed to the use of a low-level programming language, whereas they actually result from the combination of low-level programming languages, lack of testing, poor development processes, et cetera. Correspondingly, we do not assume strict causality but merely correlation: the C programming language does not *create* buffer-overflows itself, it is just particularly well-suited towards creating them and, therefore, programs written in C should be checked for buffer overflows.

To predict vulnerabilities of a new technology, we assume that their recurrent nature will continue with minor variations. For example, we anticipate that code using the standard selection and predicate language for data in the hierarchical model of XML (XPath) [30] will present problems similar to those in code using the standard selection and predicate language for data in relational databases (SQL): XPath-injection is very likely to be a problem in WS.

Based on these considerations regarding practical security, we developed a methodology for predicting vulnerabilities in systems through taxonification. Before detailing this methodology in Section 5, we introduce general concepts related to vulnerability taxonomies.

4 Vulnerability taxonomies

Although classification of items and events is a commonplace activity, there is no universal agreement on the semantics of terms surrounding classification and taxonification. Therefore we commence by defining these concepts and by providing some general observations about classifications. We then discuss vulnerability taxonomies in general and vulnerability taxonomies for WS in particular.

4.1 Classifications and taxonomies

Classification refers to both the operation and its result. Marradi [16] defines the operation as being one of three possibilities:

- an intellectual operation, whereby the extension of a concept at a given level of generality is subdivided into several narrower extensions corresponding to as many concepts at a lower level of generality;
- an operation whereby the objects or events in a given set are divided into two or more subsets according to the perceived similarities of one or several properties; or
- an operation whereby objects or events are assigned to classes or types that have been previously defined.

The first is an *a priori classification*, in which one starts from a concept and then further refines it into subconcepts. In this definition "the extension of a concept" refers to its most direct or specific meaning. The second is an *a posteriori classification*, in which one starts from the actual objects or events and proceeds by grouping them according to properties, or taxonomic characters. Each of these two types of classification produces a hierarchy of classes. The final possibility is the actual process of assigning the objects or events to these pre-defined classes.

A *taxonomy* is a "classification, including bases, principles, procedures and rules". This definition, introduced by Simpson in his seminal work on the classification of animals [24], is widely accepted, and is also used in other vulnerability classifications [13]. The definition suggests that a taxonomy is more than a classification, in the sense that it also describes the principles according to which the classification is done and the procedures to be followed in order to classify new objects. A resulting class of a taxonomy is called a taxon (Pl. taxa).

The goal of classification is to turn chaos into regularity. Systematization is necessary to handle the large amount of information humans are confronted with. The first known attempt to perform systematic classification dates back to Aristotle (322-287 BC), who worked on the first classification of animal species. For more than two millennia systematic classification was the exclusive domain of biological sciences. This changed at the end of the 19th century, when classifications started to appear in diverse branches of science [13].

The key to building a good taxonomy is the choice of the taxonomic characters according to which the objects or events will be classified. These taxonomic characters should be relevant and readily and objectively observable in the objects to be classified [24, 13]. Lough [15] gives an overview of desirable properties when building a taxonomy. Some of the most widely accepted are: deterministic, exhaustive, mutually exclusive, objective, and useful. Some of these properties are partially contradictory, e.g., a taxonomy can be made exhaustive by adding the catch-all category *other*, but doing so generally renders it less useful.

An important, yet often overlooked, issue is that there is no such thing as the ultimate taxonomy. Rather, the form of the taxonomy should be adapted to the intended usage. In practice this means that the intended usage as well as the *scope* and *viewpoint* of the taxonomy should be stated explicitly. We define scope as the part of the universe to be included in the taxonomy. This determines what should be classified. An example of scope is *all flowers indigenous to a certain country*. Equally important is the viewpoint, which determines how one looks at the things within the scope and consequently which properties will be relevant. For example, the viewpoints of a botanist and a florist yield totally different relevant classification properties in a taxonomy of flowers.

4.2 Vulnerability taxonomies

The perceived similarities between vulnerabilities published on security mailing lists such as Bugtraq [11] are an indication for the utility of taxonifying vulnerabilities. Although ideally every report features a distinct vulnerability instance, one can naturally abstract them into classes of vulnerability instances with similar properties. This abstraction facilitates understanding of the origins of the vulnerabilities, and allows for the development of avoidance and mitigation methods for such vulnerabilities. The key challenge in the production of a good taxonomy is the selection of properties that optimally contribute to this understanding.

As discussed, these properties depend on the scope and the viewpoint. A vulnerability taxonomy implies an implicit limitation of the scope, namely to vulnerabilities. Some examples of scopes that are further limited are:

- Vulnerabilities in UNIX operating systems [3, 6]
- Cryptographic vulnerabilities [26, 2]
- Application-level vulnerabilities

By setting the scope and the viewpoint of the taxonomy, one also determines which vulnerabilities to classify as well as how to look at them. For instance, the viewpoint of a developer, a system administrator or an attacker on a particular vulnerability are quite different.

Several vulnerability taxonomies have been proposed. Those proposed by [1, 5] had the scope of "vulnerabilities in operating systems". Subsequent taxonomies differed in scope and viewpoint [3, 4, 6, 9, 13]. A shortcoming of the aforementioned taxonomies is that they do not make their intended usage explicit, but instead aim at being general-purpose.

There will probably never be a vulnerability taxonomy as universally accepted as Linnæus's original classification is in biology for several reasons. The scope of vulnerabilities is highly dynamic, since the modes of daily computer usage are evolving rapidly. In addition, the computer vulnerabilities themselves are inherently difficult to describe, and most often negative terminology is used in describing them. This is related to the fact that they exist where conceptual models break down. Finally, the way in which one describes a vulnerability is strongly tied to the viewpoint.

There are at least three realizations common to vulnerability taxonomies:

- Generating a good taxonomy is difficult (see discussion in Section 4.1).
- A taxonomy depends not only on the vulnerabilities themselves but also on the viewpoint of the taxonomy creator; this viewpoint is generally determined by the intended use of the taxonomy.
- Related vulnerabilities often manifest themselves in packages that share some common property.

Generating a vulnerability taxonomy with the scope of WS incurs the additional difficultly that the WS themselves are not yet fully deployed. However, it is our central assumption that there is a sufficient body of knowledge regarding related systems to allow accurate prediction of the classes of vulnerability that are most likely to be problematic in WS. The key benefit of such a predictive taxonomy is that it can guide the investment of limited resources in securing WS.

5 Proposed Methodology

Our methodology produces predictive vulnerability taxonomies based on correlation of properties of system components and their adjudged influence on a selection of historical vulnerabilities. It is predicated on the two principal assumptions introduced in Section 3. Namely that it is meaningful to think of vulnerabilities as being present in the properties of the system in addition to being present in a particular version of a particular program and that vulnerabilities in new systems are mostly variants, in an abstract sense, of vulnerabilities existing in older systems.

We begin with a discussion of correlative properties and the process by which they are selected. Two inputs drive this process. The first is an architectural refinement of the system being analyzed into functional components. These components need to be of sufficiently fine granularity to express possible attack scenarios. They need also be uniform with respect to the selected properties for each component. The second is a representative collection of vulnerabilities, associated with the selected properties by the adjudged influence of the properties on the vulnerabilities. Naturally, the selection of properties, refinement of architecture, and collecting of vulnerabilities are iterative interdependent steps in the methodology. These inputs allow for the correlation of components with vulnerabilities through common properties. The result is a table detailing the likelihood of a vulnerability variant being present in a given component.

In the remainder of this section we detail each of the general steps of the methodology and illustrate them using the example of WS. Details that go beyond clarification of the methodology are provided in Section 6.

5.1 Architecture and selection of properties

The predictive capabilities of the produced taxonomy are determined by the accuracy with which the selected properties describe the system with respect to vulnerabilities. The selection of the properties is therefore paramount. While there is no predefined and exact means of selecting properties, we target those with influence on the security of the system. The task is facilitated by the fact that we merely need coverage rather than a perfect selection. Selection of irrelevant properties is not a problem, as they will have no effect in the correlation. Selection of strongly corelated properties is more problematic as it may lead to multicollinearity, resulting in an unproportional influence of these properties. In this work we endeavor to avoid the problem through utilization of the basis of this work: we attempt to use our security experience to select properties that are not causally related. While this heuristic does not have statistical rigor, it does have the advantage of being tractable.

A complex system is comprised of different components with diverse properties. In order to capture this with our methodology, it is necessary to refine the analyzed system into components for which the properties are well defined (e.g., true, false or irrelevant, rather than dependent upon the subcomponent). In a complex system, different components are prone to different vulnerabilities. The refinement offers the advantage that we capture not only the sorts of problems the complex system is apt to have, but also where within the system they are most likely to appear.

It is also useful to list pairs of components as connections, which indicate the composition or simultaneous presence of two components. These connections have security-relevant properties themselves, since properties on both the physical connection (e.g., use of encryption) and the logical connection (e.g., trust relationship) can render the system more or less secure.

We clarify the selection of properties using the example of WS. WS is a complex, highly-distributed architecture, but as an input into the methodology, a simple refinement suffices. As depicted in Figure 1, our WS refinement is comprised of a service requester, a directory service, a service provider, backend systems and the connections between these components. Intermediaries are not explicitly modeled, but are seen as a service provider acting as a service requester.

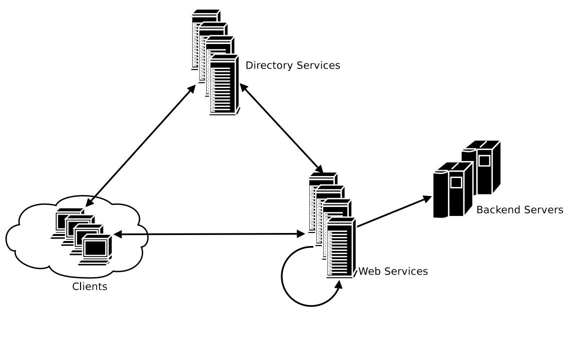


Figure 1: WS Architectural Refinement

The next step is the selection of the most relevant properties describing these components. We divide them into two categories: paradigm-related and implementation-related.

The first category contains properties related to the underlying concepts of WS and SOA. Examples of the properties in this category include support for dynamic discovery and binding of services, XML-based messaging, and cross-platform support.

The second category involves properties related to actual implementations. These properties are equally important but less uniformly applicable and more difficult to find; different implementations have different properties. In order to ensure general applicability of the produced taxonomy, it is important to choose properties common to most implementations. For example, most WS implementations use high-level languages and server application containers, which provide system-level services such as memory and transaction management. As such we consider "use of high-level languages" a shared property.

The result of this property selection step is a matrix [properties × architecture]. The rows contain the different properties and the columns contain the different components of the architecture. For every property and component there is a cell describing the degree to which the property is present in the component. The different values are "the property is present", "the property is not present" or "the opposite property is present".

5.2 Selection and assessment of vulnerabilities

The next major step of our methodology is the selection of vulnerabilities. Recalling the assumption that new vulnerabilities will largely be variants of existing vulnerabilities (as opposed to fundamentally new vulnerabilities), it is necessary to identify a representative base of existing vulnerabilities. With several "new" vulnerabilities discovered on an average day, it is necessary to discriminate. We set the following three criteria: coverage, relevance and availability of information.

Coverage in this context refers to the need for a broad and complete range of vulnerabilities. The goal is to estimate the likelihood that variants of the selected vulnerabilities will appear in the new system. Therefore, it is necessary to ensure that the selection is not too limited. While buffer-overflows account for a major part of the published vulnerabilities, there is no need for proportional representation of this class of vulnerabilities in the selection.

Our second criterion is *relevance*. The vulnerabilities should also be relevant to the security layer of interest. If the layer of interest is practical security for applications, the selected vulnerabilities will be centered around the application-level. Likewise, vulnerabilities from similar systems will have a higher chance of being relevant to the new system. For example, vulnerabilities in CORBA [12] bear greater relevance to DCOM [8] than those of an architecturally dissimilar system.

The final criterion is *availability of information* about the particular vulnerability. The best sources for obtaining vulnerability information are security databases and mailing lists, such as Bugtraq [11]. Nevertheless, the available information is often sparse or obscured, inhibiting understanding of the vulnerability. Hence, it is best to select the vulnerabilities with the most complete information available.

This selection step yields a list of vulnerabilities that needs to be assessed, in order to determine the influence that each property has on each vulnerability. Many properties will not influence the likelihood that a vulnerability appears whereas others will either increase it or decrease it.

This assessment is necessarily partially biased, as our method aims to capture the experience of a security professional. In many cases, the assessment will be trivial as the property is a prerequisite or conversely an inhibitor of the vulnerability. In other cases, there is no relation whatsoever between the property and the vulnerability. In cases where the influence is less clear, different professionals may come to different conclusions. This can be mitigated by having a more fine-grained scoring system and averaging results or a questionnaire resulting in a standardized score.

The result of this vulnerability selection and assessment step is a matrix [vulnerabilities × properties]. Each cell qualitatively describes the influence of the property on the vulnerability. Later, this qualitative description will be mapped onto a quantitative value in order to perform correlation.

In our WS example, we begin with the values "positive", "none" and "negative", meaning respectively more likely, no influence and less likely. For instance, text-based communication, which facilitates understanding and manipulation of the transmitted data, increases the likelihood of input validation vulnerabilities; its influence is therefore positive. Similarly, the use of high-level programming languages and managed computing environments diminishes the likelihood of buffer-overflow vulnerabilities; their influence is therefore negative.

5.3 Correlation of properties and vulnerabilities

The purpose of this step is to compute an estimate of the likelihood that a variant of a vulnerability will be present in the different components of the system, by combining the information resulting from the previous steps; namely the matrices [vulnerabilities × properties] and [properties × architecture].

To obtain a quantitative estimate, we first assign numerical values to the two matrices. In our WS example, for the first matrix, we map the three values ("the property is present", "the property is not present" and "the opposite property is present"), to the values 1, 0 and -1 respectively. Similarly, for the second matrix, the three values ("positive", "none" and "negative") are mapped to 1, 0 and -1.

It is now possible to determine the likelihood that a particular vulnerability is present in a particular component. We do so by means of a simple linear composition with equal weights, obtained by the multiplication of the two matrices [vulnerabilities × properties] × [properties × architecture] (as shown in Figure 2). This results in the desired description matrix [vulnerabilities × architecture].

The values in this matrix are not mathematical probabilities (e.g., they are not bounded) but rather indications of likelihood, with higher values indicating higher likelihood. For simple components, the values indicate the most relevant vulnerabilities for each component. Recall that, the simultaneous presence of components are modeled as connections between components. In this case, the meaning of the values is different: they indicate the likelihood that a vulnerability may be exploited over the link between the components.

As an example, input received over a trusted link is less apt to be validated than that received over an untrusted link. Therefore the likelihood of an exploit of an input validation vulnerability over the trusted link is greater. This is especially important for WS, where the most obvious attack flow (user attacks WS) is not the only realistic attack scenario.

5.4 Leveraging existing taxonomies

We now use a similar construction to combine the results of the previous correlation step with existing vulnerability taxonomies. This combination may be viewed as:

- A summarization of results of the previous step with respect to the viewpoint of the existing vulnerability taxonomies.
- A specialization of existing vulnerability taxonomies into a vulnerability taxonomy for the system under study.

A taxonomy can again be represented by a matrix, namely [vulnerabilities × categories]. In this matrix the rows represent particular vulnerabilities and the columns represent the taxonomy's different classes. The cells of the matrix describe whether the vulnerability is an instance of the class; values are therefore "true" or "false". Vulnerabilities generally belong to one and only one class, although this is not uniformly the case.

By mapping the values "true" to "1" and "false" to "0" and normalizing with a weight vector, we can use a simple matrix multiplication to obtain the desired result: [architecture × vulnerabilities] × [vulnerabilities × categories] × [normalizing vector] = [architecture × categories] (as shown in Figure 3). The normalizing vector accounts for the different number of vulnerabilities per class.

For our WS example we used the Bugtraq classification which we selected following our use of Bugtraq as the source of vulnerability data. It is rather coarse-grained, having only 7 classes, but is reasonable for our purposes.

6 Applying our methodology

In this section, we detail application of our methodology to WS and the salient results obtained. We focus on the details, as the coarse-scale process was already explained in the previous section. We begin with the architectural model and properties, and continue with the selection of the vulnerabilities. We present the produced result matrix and touch on how these results can be validated. Finally, we highlight the most interesting findings.

The actual values for matrices used in our WS example are presented in Appendix A and Appendix B. The selected vulnerabilities (identified by their Bugtraq number) were classified according to their Bugtraq classification.

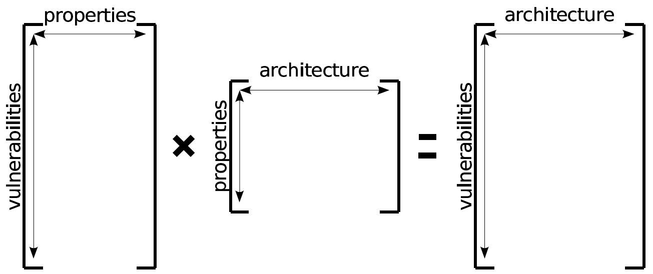


Figure 2: Correlation of vulnerabilities and architecture

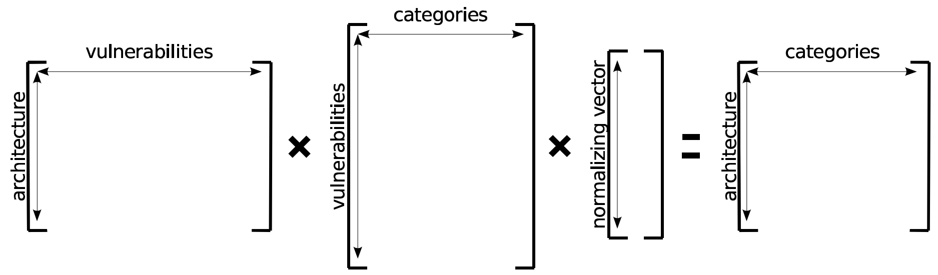


Figure 3: Correlation of architecture and vulnerability categories

6.1 Architecture and Properties

Our WS architecture model was introduced briefly in Section 5.1 and is depicted in Figure 1. It is a coarse-grained model by design, with four functional components and their connections:

- Clients: the service requester such as an ordering system.
- Directory services: the service locator such as UDDI or DNS.
- WS providers: the actual providers of a service (can be clients of other WS providers).
- Backend systems: the backend systems behind the WS providers such as a database.
- Links: components have logical links between them, e.g., WS requesters communicate directly to WS providers and directory services, but only indirectly to backend systems.

The loopback connection in Figure 1 represents the ability of a WS provider to make requests to other WS providers, aggregate the results and return them to the service requester.

The property selection follows from the architecture model. We selected 17 properties divided in two categories: properties related to the WS paradigm itself, and properties related to implementations of WS systems.

The properties of the first category, derived from the definition of SOA and WS (discussed in Section 2.1 and Section 2.2), are:

- WS are designed to enable interactions between diverse systems and are therefore considered *cross platform.*
- WS allow for the *dynamic discovery* of WS providers through directory services.
- WS support *dynamic binding* between service requester and provider.
- WS are an implementation of SOA and therefore are *service oriented.*
- WS use *XML-based messaging* for communication.
- WS, in contrast to web applications, are especially well suited for *machine to machine interaction.*
- WS are *message-centric*; messages describe what should be performed instead of how it should be performed.
- WS are not limited to a particular transport protocol and are therefore *transport agnostic.*

The second category of properties, related to the implementation of WS, is less clearly defined (as they are, by definition, implementation dependent). We selected the following properties shared by the most popular WS implementations:

- WS are typically implemented in systems with a *managed execution environment*, such as Java or .NET.
- WS have a highly *layered* structure.
- WS make heavy use of *general-purpose libraries and components*.
- Creation, configuration, and deployment of WS is often done via *wizard*.
- Interaction between the WS components typically takes place over *stateless synchronous transport protocols*, e.g., HTTP.
- WS are *not particularly efficient*, e.g., extensive XML parsing.
- The WS standards are *highly complex*.
- Different connections in the WS architecture have *different trust-levels*.

For every property and architectural component described above, we assessed the property's presence: present, not present, or the opposite property is present. The result of this selection and assessment architectural properties is shown in Figure 5 of Appendix A.

6.2 Vulnerabilities

The second step in the methodology is the selection of the vulnerabilities that are representative of the body of knowledge of historical vulnerabilities. We used Bugtraq as the source of the vulnerabilities as it is one of the largest publicly available vulnerability databases. We selected a suitable subset using the criteria set in Section 5.2: coverage, relevance and availability of information.

Availability of information was the first criterion used to create our subset. It is important as vulnerability descriptions are often incomplete and inconsistent. Therefore, we discarded vulnerabilities whose descriptions did not provide adequate information to understand root causes and thus to assess property influences.

From the set of vulnerabilities with adequate information, we discarded those that are not relevant to WS. We therefore focus on software vulnerabilities and, in particular, those in distributed systems. We discarded, for example, vulnerabilities in physical security systems, such as locks.

From the remaining vulnerabilities we selected a subset so as to retain coverage; representative vulnerabilities from each different class need to remain in the subset. This resulted in a final set of 54 vulnerabilities. For each of these vulnerabilities, we categorized the influence of each selected property on the vulnerability as positive influence, negative influence or no influence.

The result of this selection and assessment of vulnerabilities is shown in Figures 6 and 7 of Appendix B.

6.3 Result matrix

The final step of the methodology is the combination of the [vulnerability $\times$ properties] and the [properties $\times$ architecture] matrices via linear correlation to obtain the [vulnerability $\times$ architecture] matrix. We applied the technique described in Section 5.4, using the Bugtraq classification, to group the results according to the [classification $\times$ architecture]. The result matrix is depicted in Figure 4.

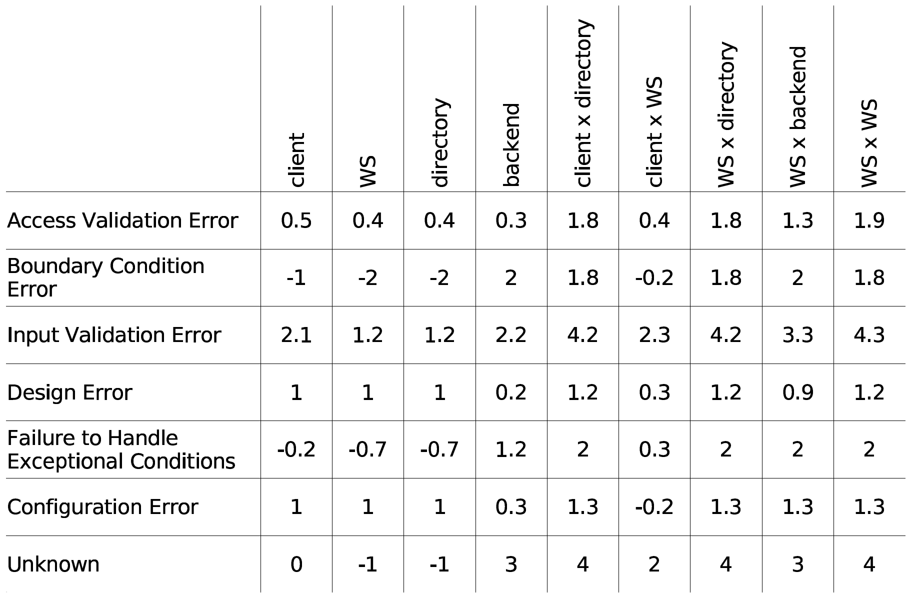
The columns of the result matrix represent the component architecture depicted in Figure 1. Recall that the functional components are client, web service, directory and backend components, as well as all the possible connections between these components. The rows represent the different vulnerability classes as defined by Bugtraq: access validation error, boundary condition error, input validation error, design error, failure to handle exceptional situations and unknown.

6.4 Validation

The predictive nature of our methodology poses some specific challenges for its validation. One possible approach entails applying it to a well-established technology, without using prior knowledge of the vulnerabilities in this particular technology. Comparison of our results with the historical vulnerabilities discovered in this technology allows for validation of our methodology. This approach is, however, out of scope for this paper as the authors' interests in this work lies in the early understanding of applied security problems in WS.

To date, the publicly available data on vulnerabilities in WS is not adequate to validate our approach. Nevertheless, this technology is maturing rapidly and we expect to have more data available soon.

In the next section we will discuss the result matrix which indicates both anticipated and unanticipated, yet retrospectively clear, outcomes. We thereby perform a functional validation although fully acknowledge that this is by no means complete.



	client	WS	directory	backend	client x directory	client x WS	WS x directory	WS x backend	WS x WS
Access Validation Error	0.5	0.4	0.4	0.3	1.8	0.4	1.8	1.3	1.9
Boundary Condition Error	-1	-2	-2	2	1.8	-0.2	1.8	2	1.8
Input Validation Error	2.1	1.2	1.2	2.2	4.2	2.3	4.2	3.3	4.3
Design Error	1	1	1	0.2	1.2	0.3	1.2	0.9	1.2
Failure to Handle Exceptional Conditions	-0.2	-0.7	-0.7	1.2	2	0.3	2	2	2
Configuration Error	1	1	1	0.3	1.3	-0.2	1.3	1.3	1.3
Unknown	0	-1	-1	3	4	2	4	3	4

Figure 4: [classification $\times$ architecture]

6.5 Discussion of results

There are a number of conclusions one may draw from the result matrix. We focus our discussion on the most salient of these in the context of real-world WS deployment.

WS often provide an interface that allows for direct interaction with core business processes. Traditionally these interfaces have been closed to the outside world and consequently run in a trusted, but not necessarily trustworthy, environment. WS changes this situation by allowing direct interaction with the core systems, thereby exposing them to a significantly larger range of threats. Therefore, previous assumptions need be reevaluated.

The result matrix shows consistently high values in the input validation class over all the components and connections between the components. This indicates that input validation errors are likely in all WS components and can be exploited over all the connections between these components.

We discuss two subclasses of input validation: input format and input origin. We conclude with a discussion of attack flows and the implications derived from the matrix values describing the connections between components.

6.5.1 Input Format

Unfounded assumptions regarding the format of the input can lead to vulnerabilities. These vulnerabilities are commonly known as input format validation errors, but input validation is only part of the reason why these errors exist.

The loosely coupled and composable nature of WS requires input validation at each of the various stages of the complete WS process. Unfortunately, the fact that form data has different meanings in different layers of the WS implies that input must also be validated in the different layers. Proper validation requires data normalization, and order is important: data must be normalized and then checked (the Nimda worm exploited a series of vulnerabilities that stemmed from the fact that a validation check was performed before the character encoding normalization was applied).

Good support for input validation is essential in both tools and libraries; currently such support is limited. For example, the best method for avoiding SQL-injection (the archetypical input validation error) is the use of prepared statements which has the effect of enforcing separation of control and data channels. The equivalent in the context of WS is XPath-injection, and there is currently no standard support from "prepared XPath".

6.5.2 Origin of data

Other unfounded assumptions relate to the origin of the input, as opposed to its format. The induced vulnerabilities include the direct spoofing of origin, corruption of directory services leading to an incorrectly contacted party, and cross site scripting. These are caused by inadequate input validation. The likelihood of each of these vulnerability subclasses is increased by the composable nature of WS.

We would make special mention of the subclass of vulnerabilities involving bounce attacks wherein the attacker tricks a trusted party into making a bad request (including cross site scripting). The likelihood of these is greatly exacerbated by the use of standard libraries for XML handling that support inclusion of external data sources (such as XInclude and external entities). For example, the attacker may set his name to the string `<xi:include href="passwords.txt" xmlns:xi="http://www.w3.org/2003/XInclude"/>` which will be interpreted as the *instruction* to include a password file.

6.5.3 Attack flows

Recall that the matrix values for the connections between components, unlike the values for components themselves, do not indicate the likelihood of the connection having a vulnerability but rather indicate how easily a certain vulnerability can be exploited over that connection. In the same way that a property or a combination of properties can render a vulnerability more or less likely, there are properties that make the exploitation of a vulnerability over a certain connection more or less difficult.

An example property is the nature of the relationship between the communicating components. When this is a trust relationship, the received input will typically also be trusted and is thus often not subjected to the proper access controls and not sufficiently validated.

The notion of exploitability of an attack over a certain connection naturally leads to the concept of attack flows. Before an attack on a component of a system can be staged, the component needs to be vulnerable to the attack *and* the path from the attacker to the victim needs to permit the attack to be launched. This raises the question: from what components can a certain component be attacked?

Consider the boundary condition error category as an example. These errors are typically caused by a buffer's reserved memory being exceeded by unexpectedly long input, which can lead to the execution of arbitrary code by an attacker. Programs written in C or C++ are prone to this variety of vulnerability.

WS components are generally written in higher-level languages and executed in managed environments, and are thus generally not vulnerable to boundary condition errors. One might there expect that WS, as an entire system, would be similarly invulnerable. This is not the case. The backend components of the WS architecture are generally written in C or C++ and are hence prone to boundary condition errors, as one can observe in the result matrix shown in Figure 4. While the WS component itself may not be vulnerable, it may pass on an attack to the vulnerable backend component.

If we assume that the client is the attacker, then determination of whether such a vulnerability in the backend system is exploitable, involves determination of whether there exists a suitable path from the client to the backend. By suitable we understand "having properties that make exploitation feasible". As there is no direct path from the client to the backend system, the attack would have to be staged indirectly through one or more of the other components. The most obvious scenario is to connect to the WS that, although not vulnerable for the attack in itself, can pass the attack on to the vulnerable backend.

The data in Figure 4 suggests that this most obvious scenario may not be the most problematic, since the properties of the connection between the client and the WS make it difficult to exploit this type of vulnerability over this connection. This is due to a combination of properties, of which one is the untrusted nature of the connection.

Although more difficult to stage, a scenario in which the attacker uses a path through the directory service or in which the directory service itself is the attacker is more likely to be successful. This follows from the relatively high values for these connections in table .

An important realization stemming from this analysis is that one's initial expectations concerning the vulnerabilities of a system are not always accurate. One must consider not only the component-wise vulnerabilities but also the paths over which they can be exploited. Furthermore, it is important to consider all attack flows rather than only that in which the client is the attacker and/or the web service component is the target.

7 Related work

The key difference between our approach and existing taxonomies (see Section 4) is the taxonomic character used and the systematic methodology for deriving new taxonomies. Our taxonomic character is the likelihood that vulnerabilities will appear in a system. This implies that our taxonomy is predictive, whereas previous work focused on classifying existing vulnerabilities. Our methodology is systematic: the classes are not selected *ad-hoc*, but through a well defined process.

Orthogonal Defect Classification [7] is a methodology for analysis of software defects, serving as an early indicator of the health of the software development process. It is related to the methodology proposed in this paper as it also involves a systemic analysis of software defects. However, both the techniques applied and the purpose differ widely. From certain attributes of the detected defects, e.g., the *defect type* or *defect trigger*, statistics are created that are compared to the expectations on defects in a certain phase of the development process.

Our test case of WS is, to our knowledge, the first taxonomy focusing on WS as a system in contrast to web applications, which have been addressed as a component by OWASP [23]. The OWASP vulnerability classes are more specialized (towards web applications) and therefore finer-grained. For example, the input validation error class of the Bugtraq classification is split into several classes, such as SQL injection, encoding errors, etc. However, the similarities between WS and web applications lead us to expect many of these classes to be also relevant for WS.

8 Conclusions and Future Work

In this work we developed a methodology for predicting vulnerabilities in systems. Our methodology systematizes the *déjà-vu* feeling a security expert has when confronted with a new system. It assesses the likelihood that variants of historic vulnerabilities will appear, based on the combination of the properties describing the system. The use of an architectural refinement provides not only an indication of which vulnerabilities will appear, but also where they are likely to appear. This allows one to reason about the security of a system before it is widely deployed and to thereby address security problems at an early stage.

We applied this methodology to WS. This has practical significance because of the importance of WS despite the current lack of wide deployment. The results of our WS example are promising and validate our methodology.

The test case application of our methodology revealed two critical steps. On one hand, the selection of the properties is difficult. There is no well-defined minimal set of properties that describes a system in all its aspects. Further, for any particular aspect, balance must be achieved between selecting a set of properties which is rich enough to describe the system yet compact and clear enough as to remain tractable. On the other hand, the selection of vulnerabilities is not trivial. Information in vulnerability databases is often incomplete, hindering understanding of the root causes of the described vulnerabilities.

One limitation of our methodology is the use of a linear mapping with equal weights to derive likelihoods. Linearity is a strong simplification implying that all properties are equally important. Alternative weighing schemes would provide more accurate models, but at the cost of higher complexity.

Nevertheless, the methodology provides a valuable heuristic for identifying the most likely sources of insecurity in a system.

References

[1] R.P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, D.A. Webb, and T.A. Linden. Security analysis and enhancements of computer operating systems: The RISOS project. Technical Report NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, US, April 1976.

[2] Ross Anderson. Why cryptosystems fail. *Proceedings of the ACM Conference in Computer and Communications Security*, pages 215–227, November 1993.

[3] Taimur Aslam. *A Taxonomy of Security Faults in the UNIX Operating System.* PhD dissertation, Purdue University, US, August 1995.

[4] Taimur Aslam, Ivan Krsul, and Eugene Spafford. Use of a taxonomy of security faults. *Proceding of 19th NIST-NCSC National Information Systems Security Conference*, pages 551–560, September 1996.

[5] R. Bisbey and D. Hollingworth. Protection analysis: Final report. Technical report, Information Sciences Institute, University of Southern California, US, May 1978.

[6] Matt Bishop. A taxonomy of unix system and network vulnerabilities. Technical Report CSE-9510, University of California, Davis, US, May 1995.

[7] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. Orthogonal defect classification a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18:943–956, November 1992.

[8] Microsoft Corporation. Distributed Component Object Model (DCOM). Webpage at `http://www.microsoft.com/com/tech/DCOM.asp`.

[9] Richard DeMillo and Aditya Mathur. A grammar based fault classification scheme and its application to the classification of the errors of TEX. Technical Report SERC-TR-165-P, Purdue University, US, November 1995.

[10] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD dissertation, University of California, Irvine, US, 2000. chapter 5: REpresentational State Transfer (REST).

[11] Security Focus. Bugtraq mailing list. Webpage at `http://www.securityfocus.com/archive/1`.

[12] Object Management Group. Common Object Request Broker Architecture (CORBA). Webpage at `http://www.omg.org/`.

[13] Ivan Krsul. *Software Vulnerability Analysis*. PhD dissertation, Purdue University, US, May 1998.

[14] Liberty Alliance. Liberty alliance project. Webpage at `http://www.projectliberty.org/`.

[15] Daniel Lowry Lough. *A taxonomy of computer attacks with applications to wireless networks*. PhD dissertation, Virginia Polytechnic Institute and State University, US, April 2001.

[16] Alberto Marradi. Classification, typology, taxonomy. *Quality and Quantity*, 2:129–157, May 1990.

[17] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew. *Java Web Services Architecture*. Morgan Kaufmann, April 2003.

[18] Microsoft Corporation. Microsoft .NET. Webpage at `http://www.microsoft.com/net/`.

[19] Oasis. eXtensible Access Control Markup Language (XACML). Webpage at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

[20] Oasis. Security Assertion Markup Language (SAML). Webpage at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`.

[21] Oasis. Universal Description, Discovery and Integration of Web Services (UDDI). Webpage at `http://uddi.org/`.

[22] Oasis. Web Services Security (WSS). Webpage at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.

[23] OWASP. The Open Web Application Security Project. Webpage at `http://www.owasp.com/`.

[24] George Gaylord Simpson. Principles of animal taxonomy. Technical report, Columbia University, New York, US, 1961.

[25] Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE). Webpage at `http://java.sun.com/j2ee/`.

[26] Paul Syverson. A taxonomy of replay attacks. *Proceedings of the Computer Security Foundations Workshop*, 1994.

[27] Userland. XML Remote Procedure Call (XML-RPC). Webpage at `http://www.xmlrpc.com/`.

[28] W3C. SOAP. Webpage at `http://www.w3.org/TR/soap/`.

[29] W3C. Web Services Description Language (WSDL). Webpage at `http://www.w3.org/TR/wsdl`.

[30] W3C. XML Path Language (XPath). Webpage at `http://www.w3.org/TR/xpath`.

[31] WS-I. Web Services Interoperability Organization (WS-I). Webpage at `http://ws-i.org/`.

A Architecture × properties

	WS paradigm properties								Typical design and implementation properties								
	service oriented	cross platform	dynamic discovery	dynamic binding	xml-based messaging	machine to machine	message centric	transport protocol agnostic	managed execution environment	layered	general-purpose components	use of wizards	stateless transport protocols	synchronous transport protocols	not efficient	complex standards	trusted connection
client	0	0	0	0	0	0	0	0	+	+	+	+	0	0	+	+	0
WS	+	0	0	0	0	0	0	0	+	+	+	+	0	0	+	+	0
directory	+	0	0	0	0	0	0	0	+	+	+	+	0	0	+	+	0
backend	-	0	0	0	0	0	0	0	-	+	0	0	0	0	-	+	0
client x directory	0	+	+	0	+	+	+	+	0	+	0	0	+	+	0	+	+
WS client x WS	0	+	+	+	+	+	+	+	0	+	0	0	+	+	0	+	-
WS x directory	0	+	0	0	+	+	+	+	0	+	0	0	+	+	0	+	+
WS x backend	0	0	-	0	0	+	0	0	0	+	0	0	0	+	0	+	+
WS x WS	0	+	+	+	+	+	+	+	0	+	0	0	+	+	0	+	+

Figure 5: The [architecture × properties] matrix for our WS example

B Vulnerabilities × properties

	WS paradigm properties								Typical design and implementation properties								
	service oriented	cross platform	dynamic discovery	dynamic binding	xml-based messaging	machine to machine	message centric	transport protocol agnostic	managed execution environment	layered	general-purpose components	use of wizards	stateless transport protocols	synchronous transport protocols	not efficient	complex standards	trusted connection
bugtraq id 6484	-	+	0	+	-	+	+	0	-	0	+	0	0	0	0	0	+
bugtraq id 6488	-	0	0	0	-	+	0	0	-	0	0	0	0	0	0	0	+
bugtraq id 6489	0	0	0	+	0	+	0	0	0	0	+	0	0	0	0	0	+
bugtraq id 6492	-	0	0	0	0	+	0	0	0	0	+	0	0	0	0	0	+
bugtraq id 6495	0	0	0	0	0	+	0	0	0	0	0	0	0	0	+	0	+
bugtraq id 6533	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6535	0	0	0	0	0	0	0	0	-	0	0	0	0	0	0	0	0
bugtraq id 6541	0	0	0	0	0	+	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6543	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6546	0	0	0	0	+	+	0	0	0	0	-	0	+	0	0	0	+
bugtraq id 6558	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6559	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6566	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6569	0	0	0	0	0	+	+	+	0	0	0	0	+	0	0	0	+
bugtraq id 6570	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6580	-	0	0	0	+	+	0	0	-	+	0	0	0	0	0	0	+
bugtraq id 6584	-	0	0	0	0	+	0	0	-	0	0	0	0	0	0	0	+
bugtraq id 6586	-	0	0	0	0	+	0	0	-	0	0	0	0	0	0	0	+
bugtraq id 6588	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	0	+
bugtraq id 6598	-	0	0	0	0	+	0	0	-	0	0	0	0	0	+	0	+
bugtraq id 6609	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	0
bugtraq id 6616	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6643	-	0	0	0	+	+	0	0	0	+	0	0	0	0	0	+	+
bugtraq id 6645	0	0	0	0	+	+	0	0	0	0	0	0	+	0	0	0	+
bugtraq id 6647	0	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6660	-	0	0	0	0	+	0	0	0	0	0	0	0	0	0	0	+
bugtraq id 6661	0	0	0	0	0	+	0	0	0	+	+	+	0	0	0	0	+

Figure 6: The [vulnerabilities × properties] matrix for our WS example (part 1)

	WS paradigm properties								Typical design and implementation properties								
	service oriented	cross platform	dynamic discovery	dynamic binding	xml-based messaging	machine to machine	message centric	transport protocol agnostic	managed execution environment	layered	general-purpose components	use of wizards	stateless transport protocols	synchronous transport protocols	not efficient	complex standards	trusted connection
bugtraq id 6662	0	0	0	0	0	+	0	0	-	0	-	0	0	0	0	0	+
bugtraq id 6666	-	0	0	0	0	+	0	0	-	0	0	0	0	0	0	0	+
bugtraq id 6667	0	0	0	0	0	0	0	0	0	-	0	+	0	0	0	+	0
bugtraq id 6668	-	0	0	0	+	+	0	0	0	+	0	0	0	0	0	0	+
bugtraq id 6670	0	0	0	0	0	+	0	0	0	0	0	0	0	0	0	+	+
bugtraq id 6671	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6677	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6684	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	0
bugtraq id 6714	0	0	0	0	0	0	0	0	0	-	+	0	0	0	0	+	0
bugtraq id 6747	-	0	0	0	0	+	0	0	0	0	-	0	0	0	0	+	+
bugtraq id 6749	0	0	0	0	0	0	0	0	0	+	+	+	0	0	0	0	0
bugtraq id 6755	-	0	0	0	+	+	0	0	0	+	+	+	0	0	0	+	+
bugtraq id 6757	0	0	0	0	0	0	0	0	0	0	+	+	0	0	0	0	0
bugtraq id 6761	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6793	-	0	0	0	+	+	0	0	0	+	+	+	+	0	0	+	+
bugtraq id 6807	-	0	0	0	0	+	0	0	0	+	+	0	0	0	0	+	+
bugtraq id 6823	0	0	0	0	0	+	0	0	0	0	0	0	0	0	0	+	+
bugtraq id 6824	0	0	0	0	0	0	0	0	0	0	0	+	0	0	0	0	+
bugtraq id 6833	-	0	0	0	+	+	0	0	0	+	+	0	0	0	0	0	+
bugtraq id 6855	0	0	0	0	0	0	0	0	0	0	-	0	+	0	0	0	+
bugtraq id 6872	-	0	0	0	0	+	0	0	-	0	0	0	0	0	0	0	+
bugtraq id 6880	-	0	0	+	+	+	0	0	-	+	-	0	0	0	0	0	+
bugtraq id 6920	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+
bugtraq id 6923	0	0	0	0	0	0	0	0	0	0	+	+	0	0	0	0	0
bugtraq id 6926	0	0	0	0	0	+	0	0	0	0	-	0	+	0	0	0	0
bugtraq id 6939	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bugtraq id 6940	0	0	0	0	0	0	0	0	0	+	+	+	0	0	0	0	0

Figure 7: The [vulnerabilities × properties] matrix for our WS example (part 2)