Vulnerability Analysis in SOA-Based Business Processes

Lutz Lowis and Rafael Accorsi, *Member*, IEEE

Abstract—Business processes and services can more flexibly be combined when based upon standards. However, such flexible compositions practically always contain vulnerabilities, which imperil the security and dependability of processes. Vulnerability management tools require patterns to find or monitor vulnerabilities. Such patterns have to be derived from vulnerability types. Existing analysis methods such as attack trees and FMEA result in such types, yet require much experience and provide little guidance during the analysis. Our main contribution is ATLIST, a new vulnerability analysis method with improved transferability. Especially in service-oriented architectures, which employ a mix of established web technologies and SOA-specific standards, previously observed vulnerability types and variations thereof can be found. Therefore, we focus on the detection of known vulnerability types by leveraging previous vulnerability research. A further contribution in this respect is the, to the best of our knowledge, most comprehensive compilation of vulnerability information sources to date. We present the method to search for vulnerability types in SOA-based business processes and services. Also, we show how patterns can be derived from these types, so that tools can be employed. An additional contribution is a case study, in which we apply the new method to an SOA-based business process scenario.

Index Terms—Service security and dependability, vulnerability analysis, SOA-based business processes, vulnerability classification.

✦

---

1 INTRODUCTION

DEPENDABLE and secure computing intends to provide services with a high degree of availability (A), reliability, safety, integrity (I), maintainability, and confidentiality (C) [1, Section 2.3]. Vulnerabilities continue to undermine that goal [2], [3], [4], and exploits incur high costs on the victim's side [5]. In a service-oriented architecture (SOA), where business processes are implemented as flexible composition of local and remote services, the challenge of finding vulnerabilities becomes more complicated and more pressing at the same time. For example, a company offering a vulnerable service to others will not only suffer locally from an exploit, but might face charges and penalties arising from, e.g., compliance violations. Using services to increase the degree of automation increases the degree to which business depends on services. This in turn increases the need to find the vulnerabilities that jeopardize the dependability and security of SOA-based business processes.

Similar to earlier developments such as CORBA and RPC/RMI, services are supposed to enable widespread and easy reuse of already implemented functionality. In terms of vulnerabilities, this is worrisome for several reasons. Using standards does not only mean less customization effort for the companies, but also for the attackers, who need less reconnaissance. Opening services to other units within a company, or maybe even to the Internet, increases the number of entry and exit points and, thus, the attack surface [6]. Reusing a vulnerable service typically enables attackers to reuse their exploits. Also, the vulnerabilities of remote services can make local services and business processes vulnerable; Single-Sign-On (SSO) scenarios are a straightforward example. So what can be done to detect vulnerabilities in SOA-based business processes and the underlying services?

Typically, vulnerability *types* have to be manually derived before tools can employ the corresponding vulnerability *patterns* for automated analyses. Fault/attack trees [7], [8] and Failure Mode and Effects Analysis (FMEA) [9] are two prominent representatives of manual analysis methods. The strength of these methods is that they leave much room for the security expert to apply subjective skills and personal experience, enabling the discovery even of completely new types of vulnerabilities. However, given the vulnerabilities observed in real-world systems, the importance of searching for *completely* new types of vulnerabilities should not be overestimated.

Two decades ago, Neumann and Parker [10] observed that most attacks use long-known techniques, and that the exploited vulnerabilities are reincarnated in new IT systems. One decade later, Arbaugh et al. [11] analyzed CERT/CC incident data and found that most exploits happen through widely known vulnerabilities. Our recent analysis of several sources such as [2], [3], [4] confirms that new types of vulnerabilities are very rare [12].

Particularly in an SOA, where a mix of established web technologies and SOA-specific standards is employed, we expect that the majority of vulnerabilities will be of a previously observed type or a variation thereof. This presumption is maintained by, e.g., Yu et al. [13], and confirmed by an analysis of, e.g., Lindstrom's [14] web service attacks. While the specific attacks are new, the underlying triggering properties, such as too high a number of input elements, are not.

- *The authors are with IIG Telematics, University of Freiburg, Friedrich-strasse 50, Freiburg 79098, Germany.*
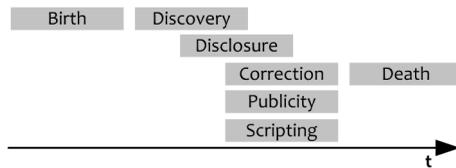 *E-mail: {lowis, accorsi}@iig.uni-freiburg.de.*

Fig. 1. Vulnerability lifecycle following [11].

In consequence, we propose ATLIST, a new vulnerability analysis method. The name stands for "attentive listener" as the method was developed during and for the analysis of SOA service orchestrations. ATLIST was designed to make use of the central SOA notions, namely, reusability, flexibility, and extensive use of standards. It facilitates the detection of known vulnerability types, and enables the derivation of vulnerability patterns for tool support. ATLIST is applicable to business processes composed of services as well as to single services.

The main distinction to existing approaches is that ATLIST explicitly builds upon the vulnerability knowledge extracted from various sources (cf. Section 4), and that it focuses on known vulnerability types rather than completely new ones. ATLIST offers better transferability than previous methods by guiding the analysis with a set of analysis elements. These elements are instantiated for the system at hand, so that an ATLIST tree can be build in a guided and repeatable manner. The analyst assesses the tree's combination of elements regarding their vulnerability, and then refines the identified vulnerability types into patterns for the specific system under analysis. Such patterns enable tools to search for and monitor vulnerabilities; without tools, a real-world SOA, in which services "come and go" in a complex composition, cannot be properly protected.

Succeeding basic terms and definitions in Section 2, Section 3 presents ATLIST, a new method to search for vulnerability *types*, and shows how the method helps to derive vulnerability patterns. Section 4 introduces the building blocks of our analysis and discusses related work. Vulnerability management tools, the input for which are vulnerability *patterns*, are also described there. We then apply ATLIST to an exemplary SOA-based business process scenario in Section 5. Section 6 points out the strengths and weaknesses of the analysis method. Finally, we summarize the analysis improvements and outline possible future uses of ATLIST.

2 TERMS AND DEFINITIONS

Vulnerabilities are flaws in information systems which can be abused to violate the security policy. The actual abuse is called an *exploit* [15, chapter 20].

Vulnerability analysis supports avoiding, finding, fixing, and monitoring vulnerabilities. These *vulnerability management* [16] tasks typically require patterns, for example, to perform static or dynamic analysis of source code. A *vulnerability pattern* is a formal representation of a vulnerability's attributes, with which a software tool can identify the vulnerability. Listing 1 shows such a pattern as PQL query [17], which detects simple SQL injection

vulnerabilities and replaces the unsafe call "c.execute(p)" with a "Util.CheckedSQL(c,p)" call that sanitizes the input before it is sent to the database. A *vulnerability type* is a characteristic that certain vulnerabilities share. Less precise than a pattern, the type is better suited for a more general analysis. In this example, the type is "SQL injection through parameter manipulation," or, even more abstract, "SQL injection."

Listing 1. PQL query for simple SQL injection [17].

```
query simpleSQLInjection()
uses object HttpServletRequest r;
    object Connection c; object String p;
matches { p = r.getParameter(_); }
replaces c.execute(p) with
    Util.CheckedSQL(c,p);
```

Once a vulnerability type is known in sufficient detail to formulate a vulnerability pattern, tools can automatically find and monitor, sometimes even fix or avoid this vulnerability (see [18] for a hands-on list of scanning tools). Showing vulnerability types and their similarities and differences, *vulnerability classifications* (discussed in Section 4.2) allow to not only develop a better understanding of vulnerabilities, but also ease the derivation of patterns as input for vulnerability management tools. Ideally, the whole *vulnerability lifecycle* as depicted in Fig. 1 is covered by tools.

If the vulnerability cannot be prevented from being introduced into the system before or at runtime, search tools must detect the vulnerability as early as possible to minimize the window of exposure.

3 THE ATLIST VULNERABILITY ANALYSIS METHOD

General analysis methods such as fault trees and FMEA (cf. Section 4.1) offer little guidance on where to start the analysis and how to proceed with it. Admittedly, this is a strength of those methods when the creativity of security experts is to be explored. Nevertheless, an analysis of previous vulnerability classifications and the entries of vulnerability databases (cf. Sections 4.2 and 4.3) shows that completely new types of vulnerabilities are extremely rare (also observed by Arbaugh et al. [11] and, as early as 1989, by Neumann and Parker [10]). Vulnerabilities of known types or close variations form the vast majority of reported incidents, so that it seems reasonable to concentrate on these. ATLIST, the analysis method we propose in the following, combines main notions of the fault tree and FMEA approaches while giving guidance regarding starting points and the analysis focus, keeping the analysis from becoming circuitous.

ATLIST consists of three steps. Section 3.1 describes the necessary preparations, namely, defining the point of view (POV) and instantiating further analysis elements. Section 3.2 explains the construction of ATLIST trees and how to use them to analyze the system under consideration. Section 3.3 shows how the derived vulnerabilities can be refined in order to enable pattern-based tool support. Additionally, in Section 3.4, we suggest ATLIST element instances to provide a quickstart into the analysis. Analysts can easily add or replace instances.

3.1 Element Instantiation

The elements discussed in this section will be used to build ATLIST trees. Unlike attack trees, ATLIST trees are not constructed be refining the root node, but by composing predefined analysis elements. These elements are:

- **Point of view**. The specific POV of the analysis, e.g., the business process or the operating system.
- **Attack effect**. The final effect an attacker causes when exploiting a vulnerability, defined from the POV.
- **Active component**. The first component in which the vulnerability is exploitable, again defined from the POV.
- **Involved standard**. The standard the execution or misuse of which enables the exploit, e.g., XML.
- **Triggering property**. The property of the involved standard an attacker exploits, defined from the point of view of the standard, e.g., order of tags.

Each element needs to be instantiated before the main analysis can begin. In this section, we give a first overview of the elements; See Section 3.4 for a set of suggested element instances.

First, the specific **POV** of the analysis must be defined. This determines which attack effects and active components will be considered, e.g., setting the POV to an SOA-based business process leads to very different results compared to an operating system POV. Consider the example of an FTP vulnerability, which allows attackers to replace files, and a web application, which grants administrative rights to an attacker because of a replaced configuration file. Such dependencies between different POVs can be incorporated by first analyzing single POVs and then, in a second phase, check if the results of one POV's analysis could influence another POV's analysis. This multilayer analysis can be achieved by inserting one ATLIST tree as the "active component" part of another, e.g., adding the webserver's tree as active component to the web application's tree.

Second, the POV-dependent **attack effects** are derived. To this end, either known attacks or vulnerability classifications can be consulted. Many, but not all vulnerability classifications are suitable, as some do not contain information on the attack effects.

Third, the **active components** are collected. Again, suitable vulnerability classifications serve as a source, or the scenario at hand is decomposed to identify those components which, e.g., execute code or parse input. The data flow diagram (DFD) approach described in [19] helps with the identification.

Fourth, the **involved standards** that can be used to trigger exploits are listed. Besides looking at vulnerability descriptions with sufficient details, the standards typically are obvious from the scenario. For example, in a web application scenario, HTTP, JavaScript, and SQL would be straight choices. Proprietary solutions might include nonstandard technologies; however, a close look at previous vulnerabilities reveals that the triggering properties are the same in either case. If a few details about the proprietary solution can be obtained, the analysis can proceed by assuming standards similar to the proprietary ones.
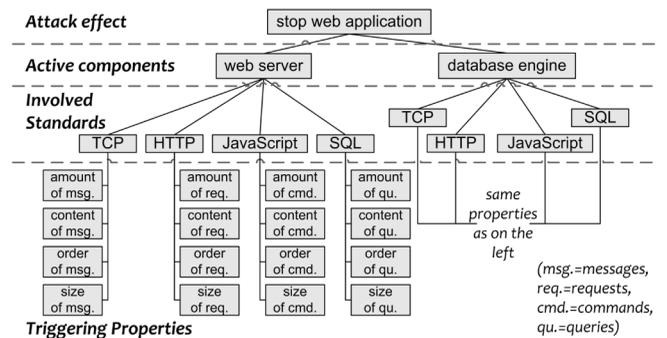

Fig. 2. Exemplary ATLIST tree.

Fifth, the **triggering properties** are gathered. Many vulnerability classifications contain the required information; Krsul's classification [20], for example, lists the assumptions that programmers made regarding the implementation's environment. Violating these assumptions equals exploiting triggering properties, which is what attackers do when they, e.g., send a longer input string than the programmer assumed. Here, the triggering property is the input length.

Vulnerability classifications are an excellent source of information for all ATLIST elements (except for the often-neglected POV), as we will show in Section 3.4. Table 2 in Section 4.2 indicates which classifications are especially well suited for which elements. Having instantiated the ATLIST analysis elements, they can now be employed in the main analysis.

3.2 Tree Building and Examination

The main analysis starts by creating one ATLIST tree for each attack effect. In this regard, we make use of the fault tree notion of starting the analysis with the fault (here, attack effect) of interest. Setting the attack effect as the root node, the active components are added as child nodes, each of which has its own set of child nodes consisting of the involved standards. The standards in turn have the triggering properties as their child nodes. Fig. 2 shows an exemplary ATLIST tree for a web application scenario with the POV of the web application.

Now, each path from a leaf to the root node implies a question to the security analyst. For example, can TCP messages of an unusual size affect the webserver so that the web application is stopped? FMEA's basic approach—assuming a malfunction of a certain system component and then propagating the effects—is employed here in a focused manner.

Some standards are irrelevant for certain components. For example, in the above scenario, JavaScript is irrelevant regarding the database engine itself. Also, not all properties are applicable to each standard (e.g., "size of command" for JavaScript). Nevertheless, it is reasonable to think through the various combinations of active components, standards, and properties, and then mark as inapplicable some standards or properties after analyzing the specific case. Considering combinations which might seem irrelevant at first can lead to the discovery of new vulnerability types, as

we will discuss in Section 5.2. There, the seemingly irrelevant order of XML tags leads to a vulnerability.

Those paths not considered dangerous by the analyst can then be marked as protected or irrelevant. For example, if a firewall protects the webserver from tiny or huge TCP messages, the corresponding "size of msg." property could be marked as protected.

The remaining paths have to be checked regarding their actual exploitability. A specific penetration test, either of the actual system or as a paper exercise, will provide details on the attack feasibility. During this step, additional paths will be ruled out. For both the vulnerable and protected paths, the details obtained from the tests are added to the leafs of the tree. They represent a structured container for the vulnerability details which will be further developed during postprocessing:

- **Precondition**. The circumstances under which an attacker will likely be able to exploit the vulnerability.
- **Postcondition**. The specific changes to the system caused by the attack, also, knowledge or abilities the attacker has gained.
- **Test case**. How to check whether the vulnerability is actually exploitable.
- **Protection**. Measures that protect against exploits of this vulnerability, or, even better, prevent the vulnerability from being introduced into the system.
- **Search pattern**. A formal vulnerability description to be used by scanning tools; if possible, one pattern each for detection before, at, and after runtime.

We have derived these attributes through an analysis of the above-mentioned vulnerability information sources. Together with the main ATLIST elements attack effect, active component, involved standard, and triggering property, the vulnerability attributes contained in the vast majority of sources are covered. The ATLIST vulnerability attributes can be utilized as follows:

Preconditions allow to estimate the likelihood of the attacker's success in case of an attempted attack. Also, they give first hints on how the vulnerability can be detected before or at runtime. Complementary, **postconditions** indicate how an attack can be detected at or after runtime. When used like jigsaw pieces, pre- and postconditions serve to (automatically) produce attack graphs (cf. Section 4.4).

Test cases are important to verify that a specific vulnerability is actually not exploitable in the system under consideration. Besides addressing testing challenges such as Canfora and di Penta [21] have identified in the SOA scenario, tests can confirm the effectiveness of protective measures. **Protection** includes both the aspect of preventing the vulnerability and protecting against exploits. A prominent prevention example is the restricted use of the *gets()* function in the C programming language after it became known as a notorious source of buffer overflow vulnerabilities.

Search patterns are required to cover the complete vulnerability lifecycle (see Fig. 1) with tool support. Detection *before* runtime inhibits exploits of the vulnerability, however, analysis complexity (e.g., for pointer or reflection analysis [22]) may be so high that it becomes impossible to accomplish preruntime detection with reasonable effort. Monitors can employ patterns to provide detection *at* runtime. Detection *after* runtime is the concern of audits, which gain importance in the light of compliance regulations. Log-based audits such as the one presented by Accorsi et al. [23], [24] can be augmented by vulnerability patterns to increase the set of detectable policy violations.

3.3 Refinement of Vulnerability Details

Given the ATLIST tree, postprocessing consists of refining the vulnerability details as far as necessary. While in theory, each vulnerability could be thoroughly scrutinized, in practice, there usually are resource constraints regarding, e.g., time and money. Therefore, it is reasonable to first elaborate on those vulnerabilities that are easily exploitable and have a large impact. Assessing the ease of exploit can be facilitated by CVSS [25] data on access vectors, access complexity, and exploitability. This requires connecting CVSS entries to the leafs of an ATLIST tree, which is straightforward if the vulnerability descriptions are detailed enough, i.e., tell the active component, the involved standard, and the triggering property. If this information is not available, similar vulnerabilities can be referred to so that an estimated ease of exploit can be extrapolated.

A vulnerability's risk level can be estimated by Microsoft's DREAD as presented in [19]. DREAD stands for Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability. Other approaches to vulnerability prioritization are available, e.g., [26], [27]. We refrain from a discussion of these due to limited space, and assume that the vulnerabilities have been sorted in order of their estimated risk level.

Moving from high to low priority vulnerabilities, the vulnerability details are refined until the search can be automated. Multiple units of vulnerability details can be appended to a single ATLIST tree leaf, because triggering properties can be exploited in different ways. Each path from a triggering property to the root node represents a vulnerability type, and each unit of vulnerability details represents a subtype of the corresponding type. In other words, the vulnerability patterns are the formal representations of vulnerability subtypes.

The derivation of patterns is a creative process and cannot be automated in the general case. Nevertheless, ATLIST trees ease the derivation by narrowing down the search space: the combination of active component, involved standard, and triggering property indicates what needs to be observed where for detecting the vulnerability under consideration. We do not propose a specific format of patterns, as this would restrict the search to certain types of vulnerabilities, or certain technologies even. Instead, we propose to include any suitable format, where necessary together with a hint on suitable tools. For example, regular expressions are a general format, whereas a pattern for the intrusion detection system SNORT [28] requires the use of specific software.

Common Attack Pattern Enumeration and Classification (CAPEC) attack patterns [29] can be used as container to make the results of an ATLIST analysis widely accessible. Because ATLIST vulnerability details and CAPEC share many attributes, the conversion is straightforward. Parrend and Frenot [30] propose semiformal vulnerability patterns many attributes of which can be assigned to ATLIST vulnerability details.

3.4 Suggested ATLIST Element Instances

ATLIST elements can be instantiated through experience, but vulnerability classifications also lend themselves easily to support this instantiation. Most of the existing classifications and vulnerability collections contain a variety of attributes with different levels of details. Therefore, a single classification's entries typically do not map to single ATLIST elements, but provide information on several ATLIST elements. We give an example for this and then suggest ATLIST element instances.

Howard et al. [31] describe 19 typical programming flaws. Some flaws indicate a triggering property (e.g., "race conditions" indicate sequence of action), whereas others indicate the involved standard (e.g., "SQL injection"), active component (e.g., "failure to protect network traffic" points to firewalls), or attack effects (e.g., "improper file accesses"). This is why in the following, some classifications are mentioned as source of several ATLIST elements.

We stress the fact that the following lists of ATLIST element instances are by no means complete, and only serve as suggestions to get the analysis started. Analysts can complement or refine instances at any time.

3.4.1 Point of View

Only a few vulnerability classifications explicitly mention their POV. Even worse, many switch the POV between different entries. An operating system POV is used in [20] and [32]. Further, POVs derived from National Vulnerability Database (NVD) [3] are legacy software and web applications. SOA scenarios indicate the additional POVs (web) service and business process, as services are orchestrated to execute business processes. The suggested POVs are:

- **Business Process**
- **(Web) Service**
- **Web Application**
- **Legacy Software**
- **Operating System**

3.4.2 Attack Effects

Inverting the general security and protection goals of confidentiality, integrity, and availability, leads to the general threats of losing CIA. However, the according attack effects "loss of CIA" are too broad for a focused analysis. Microsoft's STRIDE [19] with its threats spoofing, tampering with data, repudiation, information disclosure, denial of service (DoS), and elevation of privilege, offers a more specific, yet widely applicable classification. For business processes, we have derived six effects, two for each C, I, and A (cf. Section 5.2.1). The suggested attack effects are:

- **STRIDE** [19]. Spoofing, tampering with data, repudiation, information disclosure, DoS, elevation of privilege.
- **For business processes**. Start, stop, steer, split, spot, study.

The threat scenarios from OCTAVE Allegro [33] can also be used to derive attack effects.

3.4.3 Active Components

The selection of active components heavily depends on the POV. Still, most vulnerability descriptions of sources such as [3], [4], [20], [29], [34], [35] point to one of a few active components. In the light of SOA, we add (web) services and business process execution engines. The suggested active components are:

- **Business Process Execution Engine**
- **(Web) Service**
- **Web application**
- **Database engine**
- **Webserver**
- **Input parser**
- **Library**
- **Program**
- **Operating system kernel**

3.4.4 Involved Standards

Just like the active components, the involved standards have to be selected according to the POV of the analysis. Sources such as [3], [4], [29], and [36] show the steady reoccurrence of attacks through certain standards. Besides network protocols, file formats and script/programming languages are prominent candidates. The suggested involved standards are:

- **Network**. FTP, SSH, and TCP.
- **Web**. HTTP, SQL, Javascript, and XML.
- **File formats**. DOC, JPG, PDF, and ZIP.
- **Programming languages**. ASP, C*, Java, Perl, PHP, Python, and VB.
- **SOA**. BPEL, SOAP, and Web Service Description Language (WSDL).

3.4.5 Triggering Properties

Vulnerability classifications describe triggering properties in various ways. Summarizing the ones listed in Table 2, we agree with [20] that length, type, and content are the fundamental triggering properties. While "content" is rather broad, this forces the analyst to think about the constraints that should be enforced for the content, thereby indicating possible vulnerabilities in the form of missing constraints. To refine the "content" property, encoding, structure, and external references can be used (external references in the content might enable, e.g., file inclusion vulnerabilities). These properties and the order of actions can be derived from sources such as [37], [38]. Because the order of elements can also be important (see Section 5.2.2), we suggest that property, too. The suggested triggering properties are:

- **Length or size.** Too short/long.
- **Type.** Unknown/unexpected.
- **Content.** Unknown/unexpected (encoding, structure, external references).
- **Order** of actions or elements.

4 RELATED WORK AND ANALYSIS BUILDING BLOCKS

ATLIST uses concepts from both top-down and bottom-up analysis methods, which we present at first. Vulnerability classifications follow in Section 4.2, as they are an important input to our method. Vulnerability databases as the other major source of vulnerability information are the topic of Section 4.3. Vulnerability management tools in turn can use
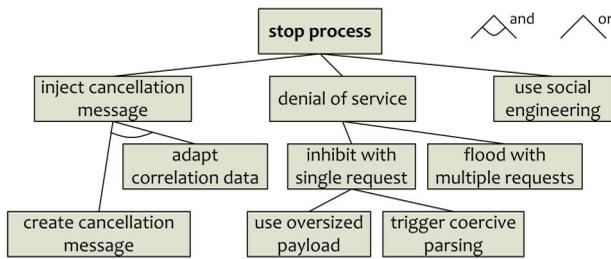
Fig. 3. Sample attack tree "stop process."

the patterns derived from the vulnerability types our method identifies, so these tools are discussed in Section 4.4.

4.1 Vulnerability Analysis Methods

For the sake of reading flow, in this section, we interchangeably use the terms fault, failure, and vulnerability (see [39] for a recent discussion of these terms).

Fault trees [7], or attack trees as described by Schneier [8] (Moore et al. [40] offer a good introduction to the method), are a general, top-down vulnerability analysis method. First, a specific attack is selected for analysis. Then, possible causes are refined until the fundamental vulnerabilities of the causal chain have been identified. The idea is that the analyst will not only be able to identify known vulnerabilities in the system, but because of creativity and experience might also find new types of vulnerabilities. Typically, attacks for use as root node can easily be found, e.g., by inverting a part of the security policy. Assuming that customer data have to be kept confidential in a business process, the attack "obtain customer data" could be used as starting point. In order to refine the attack, the type of components to be analyzed must then be selected. The challenge here is the width of selection, as relevant attacks might be missed when too few components are selected, and too wide a selection might result in a time-consuming, overly complex analysis. To make matters worse, a certain attack can be relevant in one context and irrelevant in another, so attack trees have to be thoroughly revised when reused for, e.g., a different business process. Fig. 3 shows an exemplary attack tree example using one of the business process effects described in Section 3.1 as a starting point (details of the "oversize payload" and "coercive parsing" steps can be found in [41]).

Failure Mode and Effects Analysis [9] is a general, bottom-up analysis method. Here, the starting point is a single component of the system under analysis. Possible effects of that component's failure are recursively forecast through connected components until the analysis reaches the system's boundaries. The central step of the analysis is to "identify all potential item and interface failure modes and define their effect on the immediate function or item, on the system, and on the mission to be performed" [9, p. 9]. While already for small systems, it is a tedious task to list *all* potential failures, this is impossible for medium or large systems such as networked IT systems used in SOAs. Table 1 shows an excerpt of an exemplary FMEA of the URL access control component in a webserver scenario.

FMEA shares many of the positive and negative aspects of fault/attack trees. Creativity and experience are required

to find and scrutinize the relevant chains of effects. Also, the selection of components to be included in the analysis is of the same high importance and difficulty for both methods.

4.2 Vulnerability Classifications

Various vulnerability classifications are available, as Table 2 shows. The table's attributes are explained in Section 3.1. Please note that the letter "i" denotes attributes which are indirectly given in explanatory texts but are not an actual part of the respective classification. We briefly describe the

TABLE 1
Exemplary FMEA Excerpt for URL Access Control

Item	Web Server
Function	URL access control
Failure modes and causes	Porose access control because of misconfiguration
Operational mode	Standard
Local effects	Arbitrary URL access
Next higher level effect	URL access control violation
End effects	Loss of confidentiality of documents
Failure detection method	Logfile analysis
Compensating provisions	None
Severity class	High (depending on web content)

TABLE 2
Compilation of Vulnerability Information Sources

Keyword	Year	Type	F	E	C	S	P
RISOS	1976	cl	os	✓	i	-	i
PA	1978	cl	os	-	-	-	✓
Hogan	1988	cl	-	-	i	-	✓
Neumann	1989	cl	sys/net	✓	i	-	✓
NRL	1994	cl	sw	i	✓	-	✓
Brinkley	1995	cl	sys	✓	-	-	i
Aslam	1996	cl	os	i	i	i	✓
Cohen	1997	coll	sys	✓	i	-	✓
Lindqvist	1997	cl	-	✓	i	-	✓
Du	1997	cl	sw	✓	-	i	✓
Howard	1998	cl	-	✓	i	-	i
Krsul	1998	cl	sw	i	i	-	✓
Piessens	2001	cl	sw/net	i	i	i	i
Jiwnani	2002	cl	sw	✓	✓	-	✓
Alvarez	2003	cl	web	✓	✓	✓	✓
WASC	2004	cl	web	i	i	✓	✓
Avienzis	2004	cl	-	i	i	-	i
Langweg	2004	cl	app sw	✓	✓	-	✓
Whittaker	2004	coll	sw	-	i	i	✓
Lindstrom	2004	coll	web s.	✓	i	✓	✓
Thompson	2005	cl	sw	i	i	i	✓
19 Sins	2005	cl	sw	i	i	-	✓
7 Kingdoms	2005	cl	sw	i	i	✓	✓
Seacord	2005	cl	-	i	-	-	✓
Weber	2005	cl	sw	-	-	-	✓
OWASP	2006	cl	web	-	i	-	i
Vanden Berghe	2006	cl	web s.	-	✓	i	✓
Dowd	2006	coll	sw	i	i	✓	✓
PLOVER	2006	coll	-	i	i	i	✓
SANS Top 20	2007	coll	-	-	✓	i	-
Parrend	2007	cl	java	✓	✓	i	✓
CAPEC	2009	coll	-	i	i	i	✓
CWE	2009	coll	-	i	i	i	✓
OVAL	2009	coll	-	i	i	✓	i
24 Sins	2009	cl	sw	i	i	-	✓

F=stated focus, E=attack effect, C=active component, S=involved standard, P=triggering property, cl=classification, coll=collection, web s.=web services. Attribute explanation in Section 3.1.

1. Design	2-1 Running program	2-6 System library
2. Environmental assumptions	2-2 User input	2-7 File
	2-3 Environment variable	2-8 Directory
3. Coding faults	2-4 Network stream	2-9 Program String
4. Config errors	2-5 Command line param	2-10 Netw. IP packet

Fig. 4. Krsul's software vulnerability taxonomy [20].

listed classifications and some related collections in chronological order.

The seminal analyses are the RISOS project [42] and the Protection Analysis project [32]. Bishop [15] offers a discussion and comparison, including the often reused NRL Taxonomy [43], and Aslam's classification [44], [45]. All four classifications focus on OS vulnerabilities.

Hogan [46] discusses how Saltzer and Schroeder's design principles [47] are violated in operating systems. Neumann and Parker [10] propose a classification of computer misuse techniques. Brinkley and Schell [48] separate information-oriented computer misuse into different classes. Cohen [49] lists 94 abstract and partially vague attack methods, e.g., "errors and omissions," "fire," "get a job," and "repudiation."

Lindqvist and Jonsson [50] classify intrusions by intrusion techniques and intrusion results. Du and Mathur [51] modify [43]. Howard and Longstaff [52] created a computer and network incident taxonomy, suggesting classes for the terms attacker, tool, vulnerability, action, target, unauthorized result, and objective.

Krsul's vulnerability classification [20] (cf. Fig. 4) centers around the assumptions programmers made during development. Piessens et al. [53] suggest a classification and scarcely differentiate it from others. Jiwnani and Zelkowitz [54] classify vulnerabilities from a testing perspective. Álvarez and Petrovic [55] present a taxonomy similar to [52], but restricted to web attacks. The Web Application Security Consortium (WASC) [56] lists six classes of web attacks, and classifies 24 attack subtypes.

Avienzis et al. [1] discuss basic concepts of dependable and secure computing and give a taxonomy of faults. They define a vulnerability as "an internal fault that enables an external fault to harm the system" [1, Section 2.2].

Langweg and Snekkenes [57] classify malicious software attacks with regard to the location, the cause, and the impact of vulnerabilities. Whittaker and Thompson [58] describe how to break software security. Lindstrom [14] analyzes web service vulnerabilities.

In their book, Thompson and Chase [59] suggest five classes of vulnerabilities with a total of 19 subclasses, each with an example and information on how to find and fix the vulnerabilities. The 24 "deadly sins of software security" by Howard et al. [60] offer very similar, but not identical classes (the same is true for the previous version [31]).

Tsipenyuk et al. [61] present eight classes of software security errors, and offer a mapping to [31] and [34]. Seacord and Householder [62] suggest classifying by attribute-value pairs instead of single classes, and list security-related software attributes.

Weber et al. [63] mainly differentiate between intentional and inadvertent software flaws. The Open Web Application Security Project (OWASP) [34] lists 24 types of web application vulnerabilities; Yu et al. [13] discuss the OWASP list regarding web services, but give no own classification. Vanden Berghe et al. [64] propose an approach for the creation of predictive taxonomies, and present an example based on Bugtraq [65] data.

The book by Dowd et al. [66] offers an implicit vulnerability classification, albeit one without a clear structure. Christey [67] has compiled PLOVER, a list of vulnerability examples, and put them into 28 categories. The SANS Top 20 [68] do not list single vulnerabilities but the risks they cause. Parrend and Frénot [69] created a classification of Java component vulnerabilities in the OSGi platform.

The Common Attack Pattern Enumeration and Classification [29], Common Weakness Enumeration (CWE) [35], and Open Vulnerability and Assessment Language (OVAL) [70], are standards and collections of attacks, weaknesses, and vulnerabilities, respectively. The National Vulnerability Database [3] with its Common Vulnerabilities and Exposures (CVE) [71] entries and other databases such as the Open Source Vulnerability Database (OSVDB) [4] are not included in Table 2 because the level of detail strongly varies between entries, which hinders a consistent assessment.

Two major problems with vulnerability classifications are the level of abstraction and the POV, as Bishop argues in [15, chapter 20]. Switching either usually makes the classification ambiguous. To prevent switching and the resulting ambiguity, both the level of abstraction and the POV should clearly be stated when a classification is given. However, only Krsul (programmer's POV) and Lindqvist (system owner's POV) stated their classification's POV. Of the above classifications, Krsul [20] came closest to avoiding the ambiguity trap. His work focuses on environmental assumptions (cf. Fig. 4). Failing assumptions lead to vulnerabilities, e.g., if the user enters a longer string than the programmer assumed.

A promising, tree-based classification approach is described in [72]. However, unlike ATLIST, it provides no guidance on which characteristics to use or where to derive those characteristics from.

4.3 Vulnerability Databases

Prominent vulnerability databases include (in alphabetical order): IBM's Internet Security Systems (ISS) X-Force [73], NVD with CVE entries [3], OSVDB [4], SecurityFocus (including Bugtraq) [65], SecurityTracker [74], and VUPEN [36] (formerly known as FR-SIRT). All of these databases offer a unique name or id for each vulnerability, a disclosure date, and a textual vulnerability description. Also, they typically offer the vulnerable software's name including version numbers, credits or pointers to the disclosing source, references to related reports and to descriptions in other databases, references to the affected software's vendor, and protection hints or links to such. Some offer additional details on, e.g., the access vector or exploit complexity.

Microsoft's security bulletins [75] form a Microsoft specific vulnerability database. SecWatch [76] is a metasearch engine without a description scheme of its own. The SANS Newsletter [77] offers a brief vulnerability overview and refers to other databases for the details. Milw0rm [78] does not offer vulnerability descriptions in a specific scheme, which hampers the automated extraction of information. The SecuriTeam database [79] does not offer a standardized

TABLE 3
Top Five OSVDB Classes with Percentage of Entries

Input Manipulation	65.53
Information Disclosure	16.62
Denial of Service	12.19
Authentication Management	1.67
Race Condition	1.35

TABLE 4
Top Five CWE Classes with Percentage of NVD Entries

SQL Injection	17.85
Cross-Site Scripting (XSS)	14.58
Buffer Errors	12.88
Permissions, Privileges, and Access Control	9.04
Input Validation	7.98

scheme either, and can be considered a metasearch engine because it pulls many entries from other databases.

We have analyzed two of the major vulnerability databases, OSVDB and NVD, in order to provide an example of existing classifications and the according distribution of vulnerability types in databases (data were downloaded in February 2009).

Table 3 shows the OSVDB classification with the according percentage of entries. The total number of vulnerabilities in OSVDB was 52,510, of which we considered 44,375, as the remainder either had not yet been fully classified or was of the type "other" or "unknown."

Table 4 shows the same information for the CWE [35] classification applied to NVD entries. The total number of vulnerabilities in NVD was 35,776, of which we considered 9,612, as the remainder either had not yet been fully classified or was of the type "other," "not in CWE," "insufficient information," or "design error" (these types are not mapped in NVD).

4.4 Pattern-Based Vulnerability Tools

For penetration testing [80], a team of experts (a.k.a. "red team" or "tiger team") tries to break a given system's security policy by, e.g., creating attack trees and then applying tools to check that the devised vulnerabilities actually are exploitable. Ray et al. [81] discuss how to further automate penetration testing. Often, tools such as vulnerability scanners (Lyon [18] lists Nessus, Retina, and many others) and fuzzers are separately used to scan for vulnerabilities and to check how the system reacts to various inputs. Fuzzing means to supply randomly or systematically generated input in order to detect unwanted behavior of applications.

Many vulnerabilities stem from the taint problem (named after Perl's taint mode [82]), which is concerned with filtering possibly dangerous, "tainted" input before that input is used in, e.g., a database query. Livshits [83] presents an approach for the detection of taint-style vulnerabilities in Java web applications. While improving both static (before runtime) and dynamic (at runtime) analysis, the approach still demands from the user to specify precise vulnerability patterns (see Listing 1 for an example). ATLIST is a complementary approach in that it helps to find vulnerability types and to derive the corresponding patterns.

A method to generate attack graphs is presented by Sheyner [84]. Using a Büchi model with an attacking, a defending, and a system agent, the method matches attack preconditions to postconditions while updating the attacker's knowledge and the system's state according to the attack

steps being taken. Comparable with the above approach, Sheyner's method does not look for *new* vulnerabilities, but requires a formal description of known vulnerabilities. Again, ATLIST can help to derive the necessary input.

Templeton and Levitt [85] present a similar model for computer attacks, which, although not explicitly mentioned by the authors, can also be used to create attack graphs. Based on attack graphs, Noel and Jajodia [86] suggest a method for optimal IDS sensor placement and alert prioritization. Vulnerability patterns, such as ATLIST helps to derive, are crucial for both the model-based analysis and the latter method.

OVAL [70] and CAPEC [29] are parts of MITRE's efforts toward measurable security [87]. OVAL definitions contain a textual vulnerability description and list the software names, versions, and configurations that are vulnerable. CAPEC dictionary entries (called "attack patterns") provide more than 20 attributes for the description of attacks. Regarding ATLIST, OVAL definitions are a further refinement of vulnerability details. CAPEC attack patterns can be both input to and output of ATLIST, as discussed in Sections 3.1 and 3.3.

SNORT [28] is a representative network intrusion detection and prevention system. Fig. 5 shows a SNORT rule/pattern aiming at detecting an SQL injection vulnerability described by bugtraq entry 18,699.

The US-Cert's Dranzer project [88] is one example of fuzzing, also known as fuzz testing. Sparks et al. [89] show how a genetic algorithm can be used to improve the generation of suitable test input. Brumley et al. [90] suggest a method for the generation of vulnerability signatures, which indicate whether a certain input might result in an unsafe execution state. Microsoft Research's Gatekeeper project [91] proposes a static analysis approach for Java-Script programs.

Already in industrial use are products such as HP's DevInspect and WebInspect, IBM's Rational Appscan, those of, e.g., BigFix, Coverity, Fortify, nCircle, StillSecure, and Qualys, plus open source tools such as PMD. These scan source code and (web) applications for buffer overflow, SQL injection, cross-site scripting, and further vulnerabilities. One common aspect is the vulnerability patterns

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:
"COMMUNITY WEB-MISC VCard PRO search.php SQL injection attempt";
flow: to_server,established;
uricontent:"/search.php"; nocase;          uricontent:"event_id="; nocase;
pcre: "/event_id(=|\x3f)?\w*\x27/Ui";       reference: bugtraq,18699;
classtype:web-application-attack;           sid:100000697;rev:1;)
```

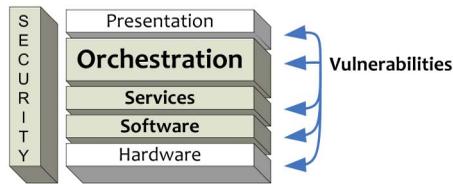Fig. 5. SNORT detection rule example.

Fig. 6. Layers of a service-oriented architecture.

which are required for automated tools to function. ATLIST supports the discovery of vulnerability types and the derivation of the according patterns.

5 CASE STUDY

For completeness, we first describe the SOA-based business process scenario before applying ATLIST to that scenario in Section 5.2.

5.1 SOA-Based Business Processes

The *SOA* we refer to follows the OASIS SOA reference model and architecture [92]. Regarding standards, we assume the use of BPEL, SOAP, and WSDL. BPEL is used to describe and execute the business processes, messages are exchanged through SOAP, and web service interfaces are defined in WSDL. Fig. 6 shows our view on SOA layers, following, among others, Krafzig et al. [93] and IBM's reference architecture [94].

In an SOA, *business processes* are executed as workflows, i.e., a set of activities performed in a certain order. The orchestration layer contains the workflow logic, typically implemented as an execution engine calling specific services in the order and with the parameters specified in a BPEL document. The focus of this case study lies on this layer. Orchestration (as a means to composition) represents the biggest change an SOA brings along, yet so far has received little attention in vulnerability analysis terms. While before, the business process logic often was interwoven in ASP or PHP scripts of web applications, it is now separately defined in BPEL. Also, business process activities are no longer mere sections of code in a web application script, but specific services with a WSDL description and a SOAP interface. Vulnerabilities which allow an attacker to modify either the activities or their sequence in a business process are our primary topic of interest in this case study.

5.2 Exemplary Analysis

We apply the ATLIST analysis steps described in Section 3 to the SOA scenario and discuss the findings.

5.2.1 Element Instantiation

The analysis foundation is laid by defining the *POV* as that of the business process. Thinking of effects on confidentiality, integrity, and availability, we instantiate six *attack effects*. An attacker could start or stop a process (harming availability, which includes preventing unauthorized users from starting processes), steer or split a process (harming integrity), and spot or study a process (harming confidentiality). The steer effect means that an attacker can select a specific execution path out of the existing ones, while the split effect means that an attacker can redirect the
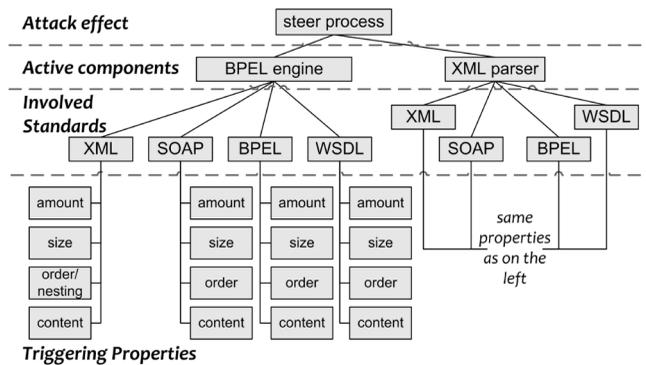

Fig. 7. ATLIST tree for the attack effect "steer process."

execution to new paths. To spot a process means to observe that it is running, and to study a process means to observe details of the execution, e.g., on required parameters and on branching.

For simplicity, we define only two *active components*, namely, the BPEL engine and XML parsers. Specific services such as database wrappers are just one example of how the analysis could be extended.

Based on the POV and the active components, the definition of *involved standards* is straightforward. We restrict the selection to XML, SOAP, BPEL, and WSDL for this presentation.

Finally, the *triggering properties* need to be instantiated. Krsul's classification [20] serves as source for the properties amount, size, order, and content, plus we include nesting as typical property of XML.

5.2.2 Tree Building and Examination

For this example, we pick the attack effect "steer process." The according ATLIST tree in Fig. 7 is built by appending the active components, then the involved standards, and then the triggering properties.

Now, the question is which path from a leaf (triggering property) to the root (attack effect) actually indicates a vulnerability. Typically, the amount and size of XML tags and SOAP messages can lead to DoS attacks and not the remote control of processes, so we mark those leafs as inapplicable. Likewise, the amount and size of BPEL scripts and WSDL descriptions are marked as irrelevant regarding the attack effect of steering the process.

Can the order, respectively, nesting of XML tags allow an attacker to steer a business process? A closer analysis reveals that yes, it can. This is because XML documents can contain multiple definitions of the same variable, and XML parsers either pass on the first or the last value assignment. Either type of parser can be tricked into passing on an attacker's choice of value. Parsing the XML in Listing 2, an XML parser of the latter type would pass on an ID of 205 instead of 372, because the user input into the "note" field contains a redefinition of the previously defined ID value. We have set up an SOA testing environment in which we have verified that through this attack, a business process running on Apache's ODE or Oracle's BPEL engine can be steered. Similar attacks are described in Lindstrom's report on web service attacks [14].

Listing 2. XML injection (user input in italics).

```
1 <order>
2   <id>372</id>
3   <total>700 EUR</total>
4   <items>19</items>
5       <note></note><id>205</id><note></note>
6 </order>
```

The order of SOAP messages is important as well. If the attacker sends a faked SOAP message to the BPEL engine before the original reply of a service arrives, the BPEL engine will base its branching decision on the fake message. While the order of BPEL scripts seems irrelevant to the steer process attack, the order of WSDL descriptions might be relevant if an attacker can, e.g., overwrite namespace definitions by supplying additional WSDL descriptions.

Of course, the content of XML tags, SOAP messages, BPEL scripts, and WSDL descriptions is crucial to the execution of the business process. Not all alterations of these entities would allow an attacker to steer the process, though. For example, a malformed entry will likely cause an error and stop the process, which is a different attack effect. Of the attacks Gruschka et al. [41] discuss, SOAP action spoofing shows how a process can be steered through the content of SOAP messages.

5.2.3 Refinement of Vulnerability Details

XML injection against the XML parser is one vulnerability identified in the above example, and will be refined in this section. The final step of the analysis is to gather vulnerability details so that these vulnerabilities can be prevented from being introduced. To verify that prevention was successful, and to handle the cases in which prevention is impossible, test cases and search patterns must be derived. Pre- and postconditions serve the purpose of prioritizing vulnerabilities and, possibly, generating attack graphs as discussed in Section 4.4.

A typical precondition for injection attacks is insufficient input validation. More specific preconditions are that the attacked value must 1) actually be used for branching decisions within the process, and 2) be within the scope of attackable values. Depending on whether the parser passes on the first or the last assignment, the value is attackable if its original definition lies behind or before the XML injection, respectively. In Listing 2, the attack would not work if the second line's definition followed the XML injection in line 5.

Checking for XML injection vulnerabilities can be accomplished through fuzzing (see the discussion of tools in Section 4.4). A simple test case of one message is sufficient to determine whether input filtering catches XML injections, and, if not, whether the parser uses the first or last assignment of values. Listing 3 illustrates the idea (core test input in italics). If the parser rejects the message or properly escapes the input, so that the resulting value assignment is correct, the exploit fails. If not, the value assignment will either be "first" or "last," indicating the type of assignment the parser uses.

Listing 3. XML injection test case.

```
Test input: <name> first</name><name>last </name>
Expected assignment: "first</name><name>last"
If vulnerable:
  early assignment: "first"
  late assignment: "last"
```

We have tested Apache's ODE (with Axis running on Tomcat), Oracle's BPEL Engine, and Microsoft's .NET environment. All have parsers which use the last assignment of a value, and are susceptible to XML injection.

Protection against XML injection is discussed by, e.g., Lindstrom [14] and Gruschka et al. [41]. Input validation on the server's side is the preferable option, because client-side input restrictions through, e.g., JavaScript in HTML forms can easily be circumvented.

XML injection search patterns for scanning tools can directly be derived from the above test case. At and after runtime, an attempt of exploiting multiple value definitions can be detected by observing input containing closing followed by opening tags (flexible encoding worsens the situation). Note that injecting a different order of closing and opening tags will most likely lead to an error and not to the assignment of attacker-selected values. However, the attacker can do more than just fiddle with single input fields: whole messages can be created, containing properly formatted, yet malicious XML tags. Schema validation and further measures are required to detect attacks of that kind.

6 DISCUSSION

The discussion starts with some observations regarding analysis completeness. We then discuss searching for new versus known vulnerability types, and the analysis guidance ATLIST provides for that search. Some remarks on transferability aspects close the section.

Because of the Halting Problem [95], there can be no software tool that finds *all* vulnerabilities in a system. Nisley [96] offers a practical discussion. Bessey et al. [97] vividly describe why for industrial tools it is not necessarily a bad thing to ignore some vulnerabilities. Software tools can find all possible occurrences of a *specific* vulnerability (e.g., by a syntactic scan of source code for specific function calls), but the accordingly required approach of avoiding false negatives typically produces many false positives. Therefore, a manual analysis is inevitable in the general case. Model-based identification of vulnerabilities as discussed in, e.g., Höhn et al. [98], usually is a hybrid method. Manual analysis in turn can only be complete for specific, rather restricted settings or models. In the general case and for real-world systems, manual analysis necessarily also is incomplete, as there always remains the possibility that an attacker finds a new way to attack the system. Consequentially, analysis completeness cannot be achieved in the general case, neither through manual nor automated approaches. ATLIST inescapably cannot offer completeness guarantees.

Most systems are exploited through widely known vulnerabilities, despite the fact that often fixes are available
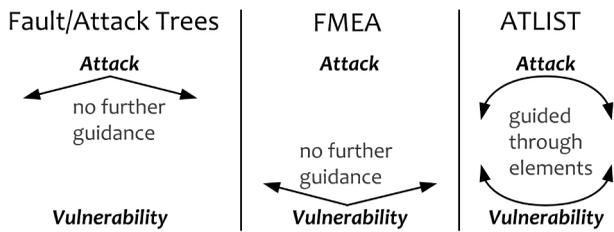
Fig. 8. Comparison of analysis guidance.

long beforehand [11]. Our analysis of various vulnerability information sources (cf. Section 4.3 and [12]) has confirmed the presumption that new vulnerability *types* are rare. We therefore argue that instead of trying to get as close as possible to completeness, it is reasonable to improve the analysis methods regarding the search for known vulnerability types. Fig. 8 depicts the analysis guidance offered by attack trees, FMEA, and ATLIST. Attack trees and FMEA offer little guidance—they only suggest an analysis direction, either top-down or bottom-up—and thereby cause a broad analysis. While this has the benefit of possibly leading to the discovery of new vulnerability types, it purely depends on the analyst's experience whether known types are included in the analysis or not. ATLIST, on the other hand, guides the analyst toward known vulnerability types. And despite the focus on known types, ATLIST can still lead to the discovery of new ones.

One aspect in which attack trees and FMEA are superior to ATLIST is the discovery and description of social engineering attacks. While users could be seen as "active components," the "involved standards" element would make little sense in that context. However, our goal is to provide a transferable analysis method which facilitates vulnerability management tool support, and social engineering attacks can hardly be prevented through technical means.

Regarding transferability, ATLIST does not require as much experience and subjective skills as attack trees or FMEA do. Therefore, ATLIST can more easily be applied. We acknowledge that the new method's results to a certain degree still depend on the analyst. For example, some analysts might not consider the order of XML tags as relevant, missing the vulnerability discussed in Section 5.2.2. However, by providing guidance in the form of analysis elements as well as instructions and suggestions on their instantiation, ATLIST demands less experience than the attack tree and FMEA methods while yielding competitive results.

Using remote services can make the local business process susceptible through vulnerabilities in those services. With its analysis elements, ATLIST helps to check various remote services for vulnerabilities. Employing the condensed knowledge on triggering properties as extracted from many vulnerability classifications (cf. Table 2) enables analysts to derive vulnerability types and their patterns in a systematical way. Testing remote service against these patterns enables a vulnerability analysis without knowing the specific details of the remote services.

Clearly specifying the POV of each partial analysis facilitates the combination of several analyses to achieve a more complete view of the IT system and its security. Dependencies between POVs become visible, and, even more, the possible effects of a vulnerability are systematically analyzed as each ATLIST tree starts with the effect regarding the given POVs. Here, the intentionally limited depth of ATLIST trees helps the analyst to focus on the effects on the business process or a single service instead of meandering in low-level system details.

7 CONCLUSION

Building dependable and secure business processes and services without *any* vulnerabilities will either be impossible or cause prohibitively high costs. To meet the resulting challenge of *finding* vulnerabilities, we have presented ATLIST, a new method to analyze business processes and services for vulnerability *types*.

Applying ATLIST to an exemplary scenario, we experienced it as easier and faster to apply than attack trees or FMEA, while achieving competitive results. The existence of the discovered vulnerabilities was confirmed in our SOA testing environment, where we successfully attacked sample business processes. In our business process lab, we are in the midst of further evaluating the method by having groups of both unexperienced students and security experts analyze the same processes and services with attack trees and FMEA on the one hand, and ATLIST on the other. In addition, we are setting up a new research project with industrial partners, in which ATLIST will be used to pinpoint the vulnerabilities resulting from switching a warehousing system to an SOA.

By enabling the identification of vulnerabilities, ATLIST supports the identification and assessment of risks as pursued by, e.g., OCTAVE Allegro [33]. Such an assessment can be employed in a risk-based selection of services as described by Sackmann et al. [99].

The field of information flow analysis is making exciting progress regarding both explicit and implicit flows (Hammer [100] represents one example). We are prototyping a system that employs patterns (cf. vulnerability details in Section 3.3) to analyze either business process models or log files of their execution for vulnerabilities in the data and control flow. Discovering covert channels (first described in [101]) will be the challenge for the next version of the prototype. As of this writing, it is not yet clear whether ATLIST supports the analysis only of explicit flows or also that of implicit flows. This is closely related to the formalization of vulnerability details into patterns, which we continue to explore.

In terms of compliance, ATLIST is beneficial as it helps to document which parts of an IT system have been analyzed regarding what types of vulnerabilities. The extensive compilation of vulnerability classifications and sources presented in this paper provides links to the vulnerability knowledge collected by the security community over more than three decades. An analysis on that basis surely meets the common compliance requirements of employing best practice, state-of-the-art knowledge.

REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.

[2] Symantec, "Symantec Global Internet Security Threat Report 2008," Apr. 2009.

[3] NIST, "National Vulnerability Database (NVD)," http://nvd.nist.gov, 2009.

[4] Open Security Foundation (OSF), "Open Source Vulnerability Database (OSVDB)," http://osvdb.org, 2009.

[5] R. Lemos, "SecurityFocus: TJX Theft Tops 45.6 Million Card Numbers," http://www.securityfocus.com/news/11455, Mar. 2007.

[6] P.K. Manadhata, Y. Karabulut, and J.M. Wing, "Measuring the Attack Surfaces of Enterprise Software," *Proc. Int'l ESSS Symp.,* 2009.

[7] W.E. Vesely, F.F. Goldberg, D.F. Haasl, and N.H. Roberts, "Fault Tree Handbook," NUREG-0492, 1981.

[8] B. Schneier, "Attack Trees," *Dr. Dobb's J.,* vol. 24, pp. 21-29, 1999.

[9] US Department of Defense, "Procedures for Performing a Failure Mode, Effects and Criticality Analysis," MIL-STD-1629A, 1980.

[10] P.G. Neumann and D.B. Parker, "A Summary of Computer Misuse Techniques," *Proc. 12th Nat'l Computer Security Conf.,* pp. 396-406, Oct. 1989.

[11] W.A. Arbaugh, W.L. Fithen, and J.M. Hugh, "Windows of Vulnerability: A Case Study Analysis," *Computer,* vol. 33, no. 12, pp. 52-59, Dec. 2000.

[12] L. Lowis and R. Accorsi, "On a Classification Approach for SOA Vulnerabilities," *Proc. IEEE Workshop Security Aspects of Process and Services Eng. (SAPSE '09),* 2009.

[13] W.D. Yu, D. Aravind, and P. Supthaweesuk, "Software Vulnerability Analysis for Web Services Software Systems," *Proc. 11th IEEE Symp. Computers and Comm.,* pp. 740-748, 2006.

[14] P. Lindstrom, "Attacking and Defending Web Services," technical report, Spire Security, 2004.

[15] M. Bishop, *Introduction to Computer Security.* Addison-Wesley, 2004.

[16] V. Gorelik, "One Step Ahead," *Proc. ACM Queue,* 2007.

[17] M. Martin, B. Livshits, and M.S. Lam, "Finding Application Errors and Security Flaws Using PQL: A Program Query Language," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05),* 2005.

[18] G. Lyon, "Top 100 Network Security Tools," http://sectools.org/index.html, 2006.

[19] F. Swiderski and W. Snyder, *Threat Modeling.* Microsoft, 2004.

[20] I.V. Krsul, "Software Vulnerability Analysis," PhD dissertation, Purdue Univ., May 1998.

[21] G. Canfora and M. di Penta, "Service-Oriented Architectures Testing: A Survey," *Software Engineering,* vol. 1, pp. 78-105, Springer-Verlag, 2009.

[22] B. Livshits, J. Whaley, and M.S. Lam, "Reflection Analysis for Java," *Proc. Third Asian Symp. Programming Languages and Systems (APLAS '05),* pp. 139-160, 2005.

[23] R. Accorsi and T. Stocker, "Automated Privacy Audits Based on Pruning of Log Data," *Proc. Int'l Workshop Security and Privacy in Enterprise Computing,* 2008.

[24] R. Accorsi and C. Wonnemann, "Auditing Workflow Executions against Dataflow Policies," *Proc. Conf. Business Information Systems, Lecture Notes in Business Information Processing,* vol. 47, 2010.

[25] Forum of Incident Response and Security Teams (FIRST), "Common Vulnerability Scoring System (CVSS) 2.0," http://www.first.org/cvss, 2007.

[26] M. Dondo, "A Fuzzy Risk Calculations Approach for a Network Vulnerability Ranking System," Technical Report TM 2007-90, Defence Research and Development Canada (DRDC), 2007.

[27] M. Cukier and S. Panjwani, "Prioritizing Vulnerability Remediation by Determining Attacker-Targeted Vulnerabilities," *IEEE Security and Privacy,* vol. 7, no. 1, pp. 42-48, Jan./Feb. 2009.

[28] M. Roesch, "Snort, Network Intrusion Detection/Prevention System," http://www.snort.org, 2011.

[29] MITRE Corporation, "Common Attack Pattern Enumeration and Classification (CAPEC)," http://capec.mitre.org, 2009.

[30] P. Parrend and S. Frenot, "Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform," Technical Report INRIA/RR-6231, INRIA, 2007.

[31] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security.* McGraw-Hill Osborne Media, 2005.

[32] R. Bisbey II and D. Hollingworth, "Protection Analysis: Final Report," Technical Report ISI/SR-78-13, Information Sciences Inst., Univ. of Southern California, 1978.

[33] R.A. Caralli, J.F. Stevens, L.R. Young, and W.R. Wilson, "Introducing Octave Allegro: Improving the Information Security Risk Assessment Process," technical report, SEI/CMU, 2007.

[34] The OWASP Foundation, "Open Web Application Security Project (OWASP)," http://www.owasp.org, 2009.

[35] MITRE Corporation, "Common Weakness Enumeration (CWE)," http://cwe.mitre.org, 2009.

[36] VUPEN Security (f.k.a. FrSIRT), "Vulnerability Management and Penetration Testing," http://www.vupen.com/english, 2009.

[37] MITRE and SANS, "2009 CWE/SANS Top 25 Most Dangerous Programming Errors," http://cwe.mitre.org/top25, 2009.

[38] The OWASP Foundation, "OWASP Top Ten Web Application Vulnerabilities," http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2007.

[39] G. Neuman Levine, "Defining Defects, Errors, and Service Degradations," *ACM SIGSOFT Software Eng. Notes,* vol. 34, no. 2, pp. 1-14, 2009.

[40] A.P. Moore, R.J. Ellison, and R.C. Linger, "Attack Modeling for Information Security and Survivability," technical report, Software Eng. Inst., Carnegie Mellon Univ., 2001.

[41] N. Gruschka, M. Jensen, R. Herkenhöner, and N. Luttenberger, "SOA and Web Services: New Technologies, New Standards - New Attacks," *Proc. Fifth IEEE European Conf. Web Services (ECOWS '07),* 2007.

[42] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb, "Security Analysis and Enhancements of Computer Operating Systems (RISOS Study)," Technical Report NBSIR 76-1041, ICET, Nat'l Bureau of Standards, 1976.

[43] C. Landwehr, A. Bull, J. McDermott, and W. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys,* vol. 26, pp. 211-254, 1994.

[44] T. Aslam, "A Taxonomy of Security Faults in the Unix Operating System," PhD dissertation, Purdue Univ., Aug. 1995.

[45] T. Aslam, I. Krsul, and E.H. Spafford, "Use of a Taxonomy of Security Faults," *Proc. 19th Nat'l Information Systems Security Conf.,* pp. 551-560, 1996.

[46] C.B. Hogan, "Protection Imperfect: The Security of Some Computing Environments," *ACM SIGOPS Operating Systems Rev.,* vol. 22, no. 3, pp. 7-27, 1988.

[47] J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE,* vol. 63, no. 9, pp. 1278-1308, Sept. 1975.

[48] D.L. Brinkley and R.R. Schell, "What Is There to Worry About? An Introduction to the Computer Security Problem," *Proc. Information Security: An Integrated Collection of Essays,* pp. 11-39, 1995.

[49] F. Cohen, "Information System Attacks: A Preliminary Classification Scheme," *Computers and Security,* vol. 16, no. 1, pp. 29-46, 1997.

[50] U. Lindqvist and E. Jonsson, "How to Systematically Classify Computer Security Intrusions," *Proc. IEEE Symp. Security and Privacy,* pp. 154-163, May 1997.

[51] W. Du and A.P. Mathur, "Categorization of Software Errors that Led to Security Breaches," *Proc. 21st Nat'l Information Systems Security Conf.,* pp. 392-407, 1997.

[52] J.D. Howard and T.A. Longstaff, "A Common Language for Computer Security Incidents," Technical Report SAND98-8667, Sandia Nat'l Laboratories, 1998.

[53] F. Piessens, B. De Decker, and B. De Win, "Developing Secure Software - A Survey and Classification of Common Software Vulnerabilities," *Proc. IFIP TC11/WG11.5 Fourth Working Conf. Integrity, Internal Control and Security in Information Systems: Connecting Governance and Technology (IICIS),* pp. 27-40, 2001.

[54] K. Jiwnani and M. Zelkowitz, "Maintaining Software with a Security Perspective," *Proc. Int'l Conf. Software Maintenance (ICSM '02),* pp. 194-203, 2002.

[55] G. Álvarez and S. Petrovic, "A New Taxonomy of Web Attacks Suitable for Efficient Encoding," *Computers and Security,* vol. 22, no. 5, pp. 435-449, 2003.

[56] Web Application Security Consortium, "Web Application Security Consortium Threat Classification," http://www.webappsec.org/projects/threat, 2009.

[57] H. Langweg and E. Snekkenes, "A Classification of Malicious Software Attacks," *Proc. IEEE Int'l Conf. Performance, Computing, and Comm. (PCC '04),* 2004.

[58] J.A. Whittaker and H.H. Thompson, *How to Break Software Security.* Addison-Wesley, 2004.

[59] H.H. Thompson and S.G. Chase, *The Software Vulnerability Guide.* Charles River Media, 2005.

[60] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw-Hill Osborne Media, 2009.

[61] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," *IEEE Security and Privacy,* vol. 3, no. 6, pp. 81-84, Nov./Dec. 2005.

[62] R.C. Seacord and A.D. Householder, "A Structured Approach to Classifying Security Vulnerabilities," technical report, Software Eng. Inst., Carnegie Mellon Univ., 2005.

[63] S. Weber, P.A. Karger, and A. Paradkar, "A Software Flaw Taxonomy: Aiming Tools at Security," *Proc. Workshop Software Eng. for Secure Systems,* 2005.

[64] C. Vanden Berghe, J. Riordan, and F. Piessens, "A Vulnerability Taxonomy Methodology Applied to Web Services," *Proc. 10th Nordic Workshop Secure IT Systems (NordSec '05),* 2005.

[65] SecurityFocus, "SecurityFocus Vulnerability Database," http://www.securityfocus.com/vulnerabilities, 2009.

[66] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment.* Addison-Wesley, 2007.

[67] S. Christey, "Preliminary List of Vulnerability Examples for Researchers (PLOVER)," http://cwe.mitre.org/about/sources.html, 2006.

[68] SANS, "SANS Top 20 Security Risks," http://www.sans.org/top20, 2007.

[69] P. Parrend and S. Frénot, "Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms," *Proc. Int'l Symp. Component-Based Software Eng. (CBSE '08),* 2008.

[70] MITRE Corporation, "Open Vulnerability and Assessment Language (OVAL)," http://oval.mitre.org, 2007.

[71] NIST, "National Vulnerability Database RSS Feed," http://nvd.nist.gov/download.cfm, 2008.

[72] S. Engle, S. Whalen, D. Howard, and M. Bishop, "Tree Approach to Vulnerability Classification," Technical Report CSE-2006-10, Dept. of Computer Science, Univ. of California, Davis, 2006.

[73] IBM Internet Security Systems X-Force, "Alerts and Advisories," http://xforce.iss.net/xforce/alerts, 2009.

[74] SecurityGlobal.net, http://securitytracker.com, 2009.

[75] Microsoft, "Security Bulletins," http://www.microsoft.com/technet/security/current.aspx, 2009.

[76] SecWatch.org, "Search Portal," http://secwatch.org, 2009.

[77] SANS, "Newsletter," http://www.sans.org/newsletters, 2009.

[78] Milw0rm, http://milw0rm.com, 2009.

[79] Beyond Security, "SecuriTeam," http://www.securiteam.com, 2009.

[80] H.H. Thompson, "Application Penetration Testing," *IEEE Security and Privacy,* pp. 66-69, Jan./Feb. 2005.

[81] H. Ray, R. Vemuri, and H. Kantubhukta, "Toward an Automated Attack Model for Red Teams," *IEEE Security and Privacy,* vol. 3, no. 4, pp. 18-25, July/Aug. 2005.

[82] Perl, "Perl Programming Documentation," http://perldoc.perl.org/perlsec.html, 2011.

[83] B. Livshits, "Improving Software Security with Precise Static and Runtime Analysis," PhD dissertation, Stanford Univ., Dec. 2006.

[84] O.M. Sheyner, "Scenario Graphs and Attack Graphs," PhD dissertation, School of Computer Science, Computer Science Dept., Carnegie Mellon Univ., 2004.

[85] S.J. Templeton and K. Levitt, "A Requires/Provides Model for Computer Attacks," *Proc. Workshop New Security Paradigms (NSPW '00),* pp. 31-38, 2000.

[86] S. Noel and S. Jajodia, "Optimal IDS Sensor Placement and Alert Prioritization Using Attack Graphs," *J. Network and Systems Management,* vol. 16, no. 3, pp. 259-275, 2008.

[87] MITRE Corporation, "Making Security Measurable," http://measurablesecurity.mitre.org, 2009.

[88] US-Cert, "Dranzer: Vulnerability Detection in Activex Controls through Automated Fuzz Testing," http://www.cert.org/vuls/discovery/dranzer.html, 2009.

[89] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," *Proc. 23rd Ann. Computer Security Applications Conf. (ACSAC '07),* pp. 477-486, Dec. 2007.

[90] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards Automatic Generation of Vulnerability-Based Signatures," *Proc. IEEE Symp. Security and Privacy,* pp. 2-16, 2006.

[91] B. Livshits and S. Guarnieri, "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code," Technical Report MSR-TR-2009-43, Microsoft, 2009.

[92] OASIS, "Reference Model and Architecture for Service Oriented Architecture v1.0," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm, 2008.

[93] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA.* Prentice Hall, 2004.

[94] A. Arsanjani, L.-J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah, "IBM Developer Works: Design an SOA Solution Using a Reference Architecture," http://www.ibm.com/developer works/architecture/library/ar-archtemp/index.html?S_TACT=105AGX20&S_CMP=EDU, 2007.

[95] A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proc. London Math. Soc.,* vol. 42, no. 7, pp. 230-265, 1936.

[96] E. Nisley, "The Halting Problem," *Dr. Dobb's J.,* http://www.ddj.com/architect/184405437, 2003.

[97] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Comm. ACM,* vol. 53, no. 2, pp. 66-75, 2010.

[98] S. Höhn, J. Jürjens, L. Lowis, and R. Accorsi, "Identification of Vulnerabilities in Web Services Using Model-Based Security," *Web Services Security Development and Architecture: Theoretical and Practical Issues,* pp. 1-32, 2010.

[99] S. Sackmann, L. Lowis, and K. Kittel, "Selecting Services in Business Process Execution: A Risk-Based Approach," *Proc. Wirtschaftsinformatik,* pp. 357-366, 2009.

[100] C. Hammer, "Experiences with PDG-Based IFC," *Proc. Int'l Symp. Eng. Secure Software and Systems (ESSoS '10),* pp. 44-60, 2010.

[101] B.W. Lampson, "A Note on the Confinement Problem," *Comm. ACM,* vol. 16, no. 10, pp. 613-615, 1973.

Lutz Lowis is working toward the PhD degree at the University of Freiburg, Germany. In the research area of security and privacy in distributed systems, he concentrates on vulnerability analysis and classification. ATLIST was developed as part of his thesis. Working on the automation of vulnerability analysis for business processes/workflows, he is also interested in both the technical and economic aspects of risk management.

Rafael Accorsi received the MSc degree from the University of Amsterdam and the PhD degree in computer science from the University of Freiburg. He is a lecturer at the University of Freiburg. His interests are in security, privacy, and forensics in distributed systems, with emphasis on automated security audits, model reconstruction, and evidence generation. He is a member of the IEEE.