Automatically Extracting Threats from Extended Data Flow Diagrams

Bernhard J. Berger, Karsten Sohr, and Rainer Koschke

Center for Computing Technologies (TZI), Universität Bremen
{berber|sohr|koschke}@tzi.de

Abstract. Architectural risk analysis is an important aspect of developing software that is free of security flaws. Knowledge on architectural flaws, however, is sparse, in particular in small or medium-sized enterprises. In this paper, we propose a practical approach to architectural risk analysis that leverages Microsoft's threat modeling. Our technique decouples the creation of a system's architecture from the process of detecting and collecting architectural flaws. This way, our approach allows an software architect to automatically detect vulnerabilities in software architectures by using a security knowledge base. We evaluated our approach with real-world case studies, focusing on logistics applications. The evaluation uncovered several flaws with a major impact on the security of the software.

Keywords: Architectural Risk Analysis, Threat Modeling, Automatic Flaw Detection

1 Introduction

Software security is an important topic for software vendors. They have to determine the security-level of their software as more and more sensitive data are processed by software systems. There are complementary measures to assess the security status of software systems. First, one can assess the security at the architectural level to ensure that there are no security flaws. This approach is based on software models and attempts to identify conceptual problems, such as missing encryption of confidential data. Second, static analyzers are available that can detect implementation-level bugs, such as SQL injection and Cross-Site Scripting vulnerabilities.

Industry tends to make use of static security analyzers because they are easy to use and do not require deep security knowledge to employ them. Finding architecture-level security flaws using modeling techniques requires a deeper understanding of security, typical security problems, security measures, and their implications—making it hard to employ for small and medium-sized enterprises (SMEs). In academia, other more formal approaches have been established, notably, language-based security [21], model-driven development for security [7], and stepwise refinement [16]. Nevertheless, it will take time until these approaches have a practical impact on industry.

In this paper, we present a tool-supported, practical approach to architectural risk analysis based on Microsoft's Threat Modeling [25, 10]. A manually-crafted model of a security architecture can be automatically analyzed with the help of security rules defined in a knowledge base. These rules identify well-known architecture-level security flaws and existing countermeasures. The analysis leads to a list of tackled security problems and a list of not handled security flaws. The tool support speeds up the process of architectural risk analysis and hence reduces the monetary effort.

In particular, our contributions are as follows:

1. introduction of *extended data flow diagrams*, a refinement of data flow diagrams, which are a representation of the system architecture and are used by Microsoft's Threat Modeling to identify security flaws [25],
2. provision of a catalog of threats based on well-known resources, such as CAPEC and CWE,
3. a tool for automatically finding these threats in extended data flow diagrams,
4. an evaluation of the catalog with manually crafted data flow diagrams in the context of three real-world applications.

In the following section, we give an overview of existing techniques to identify security flaws at the architectural level. Thereafter, we present our analysis. Afterwards, we describe the rules incorporated in our knowledge base that is evaluated in the subsequent section. We conclude the paper with a discussion of the results, some information on related works and an outlook.

2 Background

Threat Modeling and architectural risk analysis can be used to detect architecture-level security flaws. These techniques target fundamental security flaws in the software architecture rather than detecting low-level security bugs. Consequently, it is expected that the impact of such flaws is higher than low-level bugs [17].

Threat Modeling has been introduced by Microsoft [25, 10]. It is part of the design phase in Microsoft's Security Development Lifecycle [18] and therefore is employed every time the design phase is executed. In the first step of Threat Modeling, data flow diagrams (DFDs) for the system in question are created. In the second step, the STRIDE approach is used to identify security flaws based on the data flow diagrams created in the first step. The STRIDE approach is an attacker-centric approach where the analyst tries to find points of the software where an attacker can breach the protection goals of information security. The identified threats are a target for risk analysis to identify the most important threats that must be addressed. Data flow diagrams are kept simple to ease their usage. Different publications propose extensions of these diagrams to capture more information of the system (see [9, 3]).

Architectural Risk Analysis (ARA) is described by McGraw in *Software Security: Building Security In* [17]. In the first step of ARA, an architecture overview is created. How it is represented is not defined. In this way, the approach fits into every software development process. This step is followed by three different analysis steps with different priorities, namely, attack resistance analysis, ambiguity analysis, and weakness analysis. Attack resistance analysis focuses on finding well-known security flaws in the architecture, whereas the ambiguity analysis targets security flaws that are specific to the analyzed system. The weakness analysis searches for problems in external software components, such as used frameworks.

3 Analysis Approach

The goal of our analysis is to automatically identify threats (architectural flaws) and mitigations because there are many common threats that can be checked for various software systems in a similar way. Since Threat Modeling employs DFDs to model an application's system architecture [25], our approach is also based on these diagrams. We follow the Threat Modeling approach, which is used by many large software vendors such as Microsoft, SAP, and EMC. Alternatively, we could have used a UML-based approach to architectural risk analysis [13, 2], but we employed Threat Modeling due its relevance in industry.

To better support the automated analysis for security flaws, we introduce Extended Data Flow Diagrams (EDFDs). EDFDs cover all concepts of traditional data flow diagrams, but use enhancements allowing us to add additional semantics. Furthermore, we capture possible threats in a knowledge base that is applicable to previously created software models. This results in a Threat Model process linking threats to elements of a diagram. Here, the threats are noted in the diagram as well as possible mitigations.

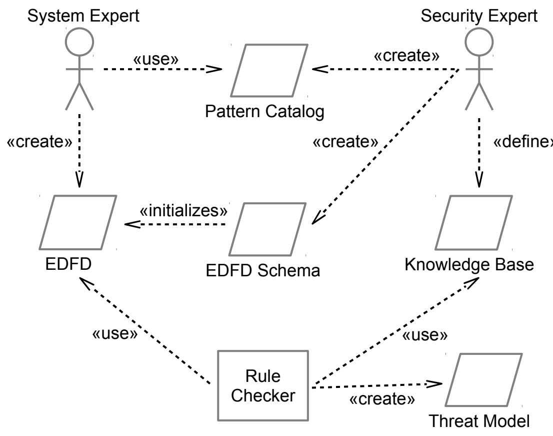


Fig. 1: Usage scenario

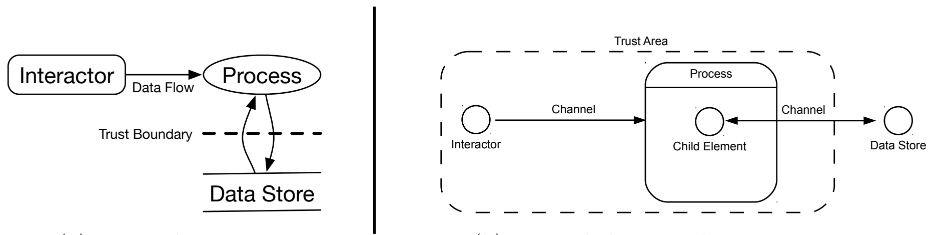
Figure 1 depicts the usage scenario and the responsibilities. Security experts define a knowledge base with rules about common architectural security flaws (see Section 3.4) and create a catalog with predefined patterns (see Section 3.3) as well as an EDFD schema (see Section 3.2). The EDFD schema is used to initialize EDFDs (see Section 3.2), which are then created by system experts (who are not necessarily security experts). To create these EDFDs, the system expert consults the given pattern catalog. The resulting concrete EDFD and the knowledge base are used as input by a rule checker, which automatically creates a threat model (see Section 3.6).

3.1 Dataflow Diagrams

DFDs consist of five different modeling elements. Processes are active components that process data. Data stores are components such as databases or files that store data. Interactors are external systems or users who interact with the analyzed system. All these elements can interact with each other using data flows. Data flows may cross trust boundaries. A trust boundary symbolizes different trust levels within a system. Figure 2a shows all available DFD elements.

3.2 Introducing EDFDs

We use EDFDs since we witnessed several shortcomings with the existing data flow diagrams while we applied Threat Modeling to real industry systems. First of all, DFDs do not capture knowledge on existing security measures or requirements. Therefore, an automated analysis of an already-defined security architecture against security rules is not possible. Second, data flows are unidirectional channels between arbitrary elements. Data that are transported in this channel are specified by simple labels that are attached to data flow edges. For this reason it is hard to identify data that flow through the whole system. Furthermore, trust boundaries are used to indicate trust areas. There is no information which components belong to a trust area making it impossible to identify dangerous data flows or access paths. Therefore, we decided to model aspects, such as data, communication, and security measures, in a more explicit way to allow an automated analysis of DFDs.



(a) Data Flow Diagram (b) Extended Data Flow Diagram
Fig. 2: Exemplary Diagrams

EDFDs comprise four central concepts, namely *Elements*, *Channels*, *Trust Areas*, and *Data* (see Figure 2b). Each of these concepts is typable and allows us to add additional information called annotations. These annotations help us add information about security measures or requirements to these components. Types and annotations are designed hierarchically allowing us to refine them. An HTTP connection, for instance, is a special kind of interprocess data flow. Each defined type may imply annotations to the component. The set of defined types and annotation types is called EDFD schema, which allows us to adapt EDFDs to the needs of new systems easily. Nevertheless, EDFDs come with a predefined schema to provide a starting point for modeling systems.

Elements: EDFDs abstract the entities *Data Store*, *Process*, and *Interactor* from traditional DFDs to the *Element* concept. We use corresponding types to model these entities. A *Java Process*, for instance, is an element type and inherits from *Process* implying the *Java* annotation.

Elements can also be structured hierarchically, e.g., processes can consist of subprocesses. In this case we can define a parent-child relationship (see Figure 2b).

Channels: We use channels to model data flows. This allows us to model different kinds of communications, such as one-to-one, one-to-many, and many-to-many communications. The predefined schema contains three root-channel types *InterProcessConnection*, *IntraProcessConnection* and *ManualInput*.

Trust Areas: In software systems, several circumstances exist that group elements to trust areas. Therefore, there are quite different kinds of trust areas, such as *Network*, *Machine Boundary*, or *Software Area*.

Data: The existing data instances depend on the current software system to be modelled. Therefore, we introduce a small number of predefined data types, such as *User Data* and its refinement *Credentials*. A frequently used annotation for data is the flag *IsConfidential*.

In summary, we use types and implied annotations and not mere annotations for the sake of usability. Since knowledge on architectural vulnerabilities is sparse (in particular, in SMEs) and the implications of using certain security mechanisms are often unclear, we introduced the typing mechanism. This typing mechanism takes the burden from the analyst.

3.3 Pattern Catalog

It showed that we had to model similar facts for different systems, for instance, the usage of an SQL-based database. Another example is the way authentication is implemented for web-based applications using a login page. Therefore, we provided a catalog of security-related design patterns. These patterns are related to security patterns known in the research community [24, 5]. The pattern catalog helps a security analyst during the creation of EDFDs to focus on modeling the system instead of thinking about the way certain technical solutions can be modeled.

3.4 Knowledge Base Rules

Our knowledge base captures descriptions of common security flaws in a machine readable form. To specify these rules we implemented a domain specific language. A rule consists of a *name* and a *description* that explains the details of the flaw. Furthermore, it contains an estimation of the severity and the likelihood of the flaw. The rule additionally contains information about the protective goals that can be violated if the flaw is exploited.

The rules contain graph patterns to identify possible flaws and corresponding mitigations. The graph patterns are defined using a graph query language that is based on the *Cypher Query Language* [11] from the Neo4j project. The language can be compared to SQL with the difference that it is designed to describe subgraphs that should be matched in an arbitrary graph. Therefore, it is possible to define properties of nodes and paths within the subgraph. It is even possible to describe paths of infinite length—a query that cannot be encoded using SQL.

3.5 Rule Checker

The detection of flaws consists of three steps. First the EDFD is lowered to a labeled and attributed graph representation. Then all rules are applied to the graph to identify possible threats. In the last step the threat model is generated based on the findings. During the lowering all elements that can be found in the EDFD are mapped to nodes or edges of a normal graph (see Figure 3). A trust area is mapped to a node and an *include* edge to each contained element of the trust area. The containment-relation between elements is mapped to *include* edges. Attributes and types that exist in the EDFD are mapped to attributes in the resulting graph. The knowledge base rules are then applied to the resulting graph. Therefore, it is necessary to search all matching subgraphs for each flaw pattern. If a match is found, a corresponding entry in the threat model is created. It links the knowledge base rule and elements within the EDFD. Then the matching engine searches for possible mitigations starting with the identified subgraph.

To provide a visual representation and a way to manipulate EDFDs, we implemented a graphical EDFD viewer on top of Eclipse's Graphiti Framework. It is able to create new EDFDs—which are initialized with a predefined schema—and modify existing ones. The visualization can be customized by type-dependent icons, and the representation of channels is type-dependent as well. Hierarchical elements can be folded and expanded. If edges exist where one of their endpoint elements is hidden because its parent has been folded, we automatically lift this edge to the visible parent [23].

3.6 Threat Model

The result of our automatic detection process is a threat model. Each threat references a *rule* from our knowledge base. Furthermore, it links at least one entity from an EDFD that is the target of the threat. Additionally, the threat can link a set of mitigations in the EDFD.

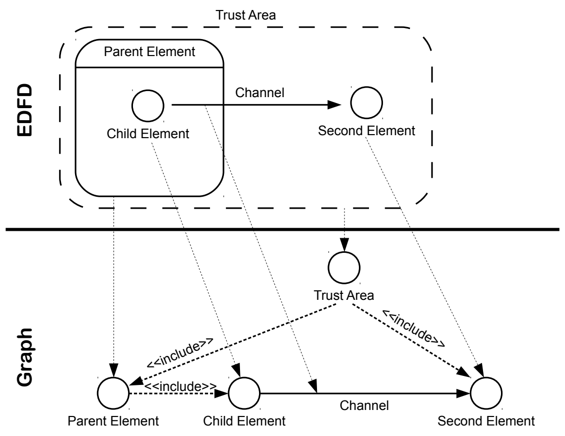


Fig. 3: Lowering of EDFDs to simple graphs

4 Knowledge Base

We populated our knowledge base with threats from the Common Weakness Enumeration (CWE) and the Common Attack Pattern Enumeration and Classification (CAPEC). The most common problems are aggregated in the *Top 25 Most Dangerous Software Errors* [19].

We distinguish between threats where we are able to detect possible mitigations and those where we are not able to do so. For the latter group a manual code review or additional static analyses are necessary to check whether the threat has been addressed appropriately.

Table 1a gives an overview of the supported CWE entries[1]. If a rule belongs to the Top 25, the corresponding ranking is given. The most interesting rules are those where we can give mitigation rules, for instance, *CWE-288: Authentication Bypass Using an Alternate Path or Channel*, or *CWE-319: Cleartext Transmission of Sensitive Information.*

```
MATCH (src : Element) -[flow : Channel]-> (tgt : Element)
WHERE flow.type.subtypeof("InterProcessCommunication")
      AND ANY (d IN flow.data WHERE d.IsConfidential)
      AND NOT flow.IsEncrypted
```

Listing 1.1: Graph Query Rule for CWE-319

Listing 1.1 shows our rule for CWE entry *CWE-319* as an example. The rule looks for a channel between two elements that transports confidential information. If the channel does not employ encryption, an attacker may capture the transported information.

As a second example, we give the rule for *CWE-306: Missing Authentication*

[1] For the sake of presentation, we only give the complete names of CWE as well as CAPEC entries in the appendix.

```
1   MATCH (src : Element) -[p1: Channel *]-> (n1 : Element)
2                          -[entry : Channel]-> (tgt : Element)
3                          <- [: include] - (area : TrustArea {
                                 Authentication_Required : true})
4   WHERE NONE (d in entry.data WHERE d.IsCredential OR d.IsSessionToken)
5         AND NOT (src) <-[:include]- (area)
6         AND NOT (n1) <-[:include]- (area)
7         AND NONE (n IN nodes(p1) WHERE (area) -[:includes]-> (n))
```

Listing 1.2: Graph Query Rule for CWE-306

for Critical Function in Listing 1.2. For a better understanding we depict the corresponding pattern graph in Figure 4. We look for a path $p1$ from a node *src* to a node $n1$. Since no constraints for this path exist (see line 1) and the path can be of arbitrary length, it is also possible that *src* is the same node as $n1$. $n1$ in turn is directly connected to the second node *tgt*. At last, there is a trust area node *area* that has an *include* relation to *tgt*.

This part of the query searches for a path from *src* to *tgt* where *tgt* is contained in a trust area that requires authentication according to line 3 of the query. In line 4, we check that no session token or credential exists that is sent to *tgt*. Lines 5 to 7 ensure that none of the nodes on the path from *src* to $n1$ are contained in the same trust area *area* including *src* and $n1$. With the help of these checks, we can find a flow to a component that is not authenticated.

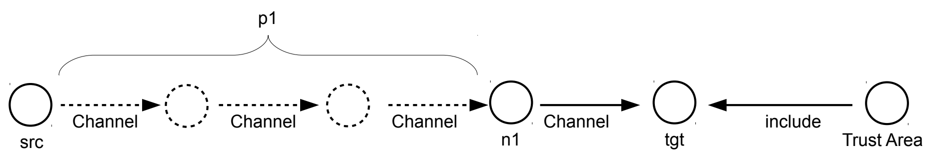


Fig. 4: Pattern Graph for CWE-306

We searched the **Common Attack Pattern Enumeration and Classification** library for possible threats as well. The currently supported rules are listed in Table 1b. Supporting CAPEC rules is more complex than supporting CWE rules since they subsume complete classes of attack patterns. Sometimes it is impossible to detect all possible variants of an attack pattern. Therefore, we note some threats and mitigations in brackets to show that we support just some special cases of this attack pattern.

We support **additional rules** that do not match CWE or CAPEC entries. For instance, we are checking information flow policies with the help of our rule set. This is useful for broadcast channels where sensitive data may flow to processes that are not trustworthy enough to process this information. In total, we currently support 25 security-related rules that are not application-specific. In future we will add further rules depending on projects and use cases.

Rule Number	Top 25	(T)hreat (M)itigation
89	1	T
78	2	T
120	3	T
79	4	T
306	5	M
311	8	M
352	12	T
22	13	T
327	19	M
134	23	T
190	24	T
759	25	T
288		M
319		M
602		M

(a) CWE entries

Rule Number	(T)hreat (M)itigation
108	T
16	T
22	T (M)
66	T
94	(T) (M)

(b) CAPEC entries

Table 1: Excerpt of Supported Rules

5 Evaluation

We evaluated our approach with the help of three industrial case studies from different vendors. The effort to construct these EDFDs summed up to half a day of work for each one. Two of the applications are from the logistics domain having a similar purpose and the third one is from e-government. In particular, we investigated the following questions.

1. Can we automatically identify threats and security flaws with EDFDs?
2. What is the impact of the identified security flaws?
3. Can we find similarities between the logistics applications?

Logistics Application A. Our first application comes from the logistics domain and is a company-specific application framework based on JavaEE technologies. The vendor offers a large number of domain-relevant applications based on this framework and wanted to identify framework-based flaws to improve the security status of their complete product portfolio. The software is mainly offered on a software-as-a-service (SaaS) base and has therefore a number of security requirements beyond the functional requirements. The applications help customers with customs clearance and fleet and port management. Figure 5 shows a simplified EDFD that we created during a workshop together with three system experts who are responsible for the architecture and the main development of the framework components. No-one of them had deep knowledge in the area of software security at that time.

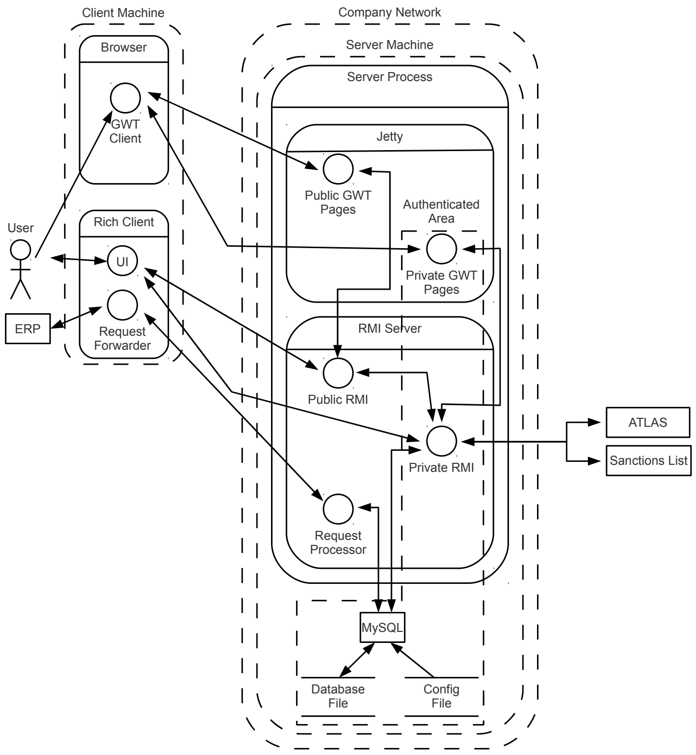


Fig. 5: Simplified EDFD for Logistics Application A

There are two possible clients. First, a browser-based client that is accessible by the end user. Second, a Java-based rich client supporting a user interface and an integrated *request forwarder* forwarding requests made by external customer systems, such as enterprise resource planning software, to the server. The server process and a related SQL-based database are running in the trust area of the software vendor's network. The server implements two interfaces for the clients. One of them is a GWT-based web interface and the second is an RMI-based interface for the rich client. Both interfaces consist of public and private components that require authentication. The clients send *credentials* to the public interfaces to authenticate the user and receive a *session object* for identifying the user for subsequent requests in return. Please note that we omit the visualization of transferred data and the annotations for the sake of clarity.

Our analysis approach identified several possible threats for this system that we discussed with the system experts as well as detected mitigations. An overview of the matching CWE entries can be found in Table 2. The rules identified correctly that there were confidential data that flowed between the clients and the server process. Hence, an attacker can try to capture these data during transmission. Our approach also found the mitigation that the channels were using TLS for transport encryption and therefore secured the data.

The rule checker found an instance of CWE entry 306 (see Figure 4). The external *ERP* system sends requests to the *Request Forwarder* contained in the *Rich Client*. The requests are then forwarded to the *Request Processor* on the server-side. The forwarder in turn communicates directly with the *MySQL database*. This communication path does not transport authentication data which indicates that an authentication check is missing. The communication path was added afterwards to the software since some clients wanted to integrate the logistics application directly into their ERP software. Moreover, a possible SQL injection was detected for the private RMI interface, but a review showed that they used hand-written SQL statements in conjunction with a company-specific API that ensured that an SQL injection could not occur. Consequently, this finding is a false positive. Another problem we detected was the circumstance that the client implemented authorization checks rather than the server. Hence, this is an instance of the CWE entry 602. Our process detected more threats, which were then examined in manual reviews and revealed additional security flaws in this application framework.

CWE	Top 25	(D)etected/(V)ulnerable	
		App A	App B
89	1	D	V
78	2	D	V
120	3		
79	4	V	V
306	5	V	
311	8		V
352	12	V	V
22	13	D	
327	19	V	
134	23		
190	24		
759	25	V	V
288		V	
319			V
602		V	V

Table 2: Detected Threats for *Logistics Applications*

The consequences and the impact of these findings are high if one considers the sensitivity of the data and systems involved in port logistics. For example, an attacker can circumvent the client-side security checks and access all data and functions available. This can be achieved by using the request forwarder or simply by modifying the rich client. Since these problems occur in the application framework, each of the applications based on the framework is vulnerable. The consequence would be that seaports, international forwarding agents, and parcel services are not able to do their job. In the end, this would result in a major financial loss for the software vendor due to contractual penalties and for economy due to outstanding delivery of goods.

Logistics Application B. The second application from the logistics domain is similar to the first one. It helps manufacturing industry with customs declarations, sanction lists checks, and commissioning. It is implemented based on the JavaEE specifications and provides a web-based interface to the customers. The software is distributed and sold on a SaaS basis as well. The data are stored in a single database making multi-tenancy an important topic for the application's security. The application is divided into several products, such as import, export, and sanction lists. In total there are seven different products that can be bought by users.

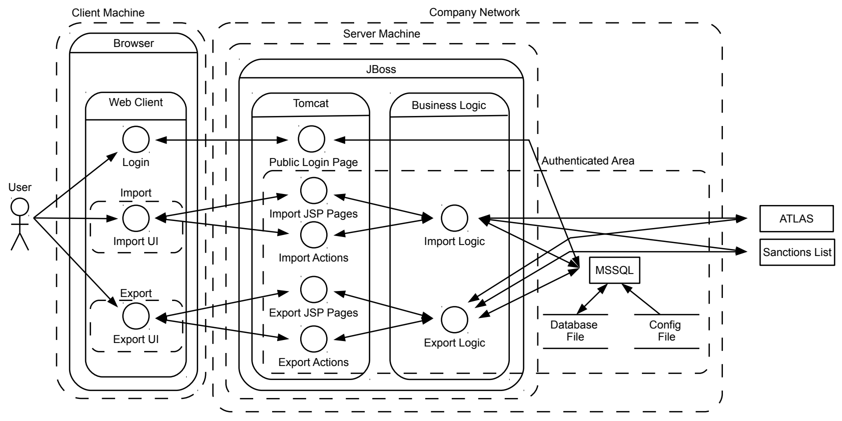


Fig. 6: EDFD for Logistics Application B

The software is structured like a typical Java Enterprise application. The client of the web application is a browser and is divided into an public *Login* page and one component for each aforementioned product. Each product on the client side is independent of other products. We modeled this as different trust areas. On the server side in the company's network, a JBoss application container runs a Tomcat web container and all product-specific *Business Logic* components implementing the view-independent algorithms and the persistence. Tomcat serves different Java Servlets (dynamic web content), a *Login Page* and JSP pages as well as Struts actions for each product. Struts is a framework that allows web programmers to implement the model view controller pattern for dynamic web pages. For persistence purposes, the *Business Logic* components talk to a Microsoft SQL database running on a different machine. Where necessary, the *Business Logic* components communicate with external systems, such as online sanction lists and the German electronic customs interface Atlas.

Our automated analysis identified several existing flaws. An overview of detected threats is given in Table 2. First of all it contains injection-based vulnerabilities, such as XSS and SQLi. The application was vulnerable to these kinds of attacks since the programmers neglected input validation and did not use an SQL abstraction layer such as the Java Persistence API—also an architectural flaw. Furthermore, we identified a threat based on *CWE-602: Client-Side Enforcement of Server-Side Security.*

Similar to the findings in *Logistics Application A* the impact is crucial for the security of the software system. It may be easy for attackers to bypass most of the security measures if she has a valid account for the system. Furthermore, if the communication between client and server is not protected by appropriate transport encryption, an attacker has the possibility to steal the credentials of an arbitrary user. Depending on the motivation of an attacker the vulnerabilities can have different consequences. On the one hand, the attacker can be interested in harming a specific customer and steal sensitive information, such as the list of customers or exported goods. Furthermore, there can be a financial impact for customers since some taxes are collected automatically on import or export in advance. On the other hand, the attacker can be interested in compromising the software provider and delete customer data or disrupt the functionality. Since the contracts contain penalties for delays caused by the software, this finally can lead to financial damage and may have legal consequences.

E-Government Application. The third case study, Governikus Service Components, belongs to the e-government domain and is part of a service-oriented architecture. Its purpose is to create qualified digital signatures that are legally binding in Germany. These signatures are created with the help of signature cards and are used by authorities for signing documents, such as birth certificates. Therefore, security is an immanent requirement and is taken into account during development. Hence, the software is being evaluated according to the Common Criteria (CC) [8].

For reasons of space we cannot give an EDFD here, but briefly describe it. An external system communicates with a public web service using an HTTPS connection. The web service runs in a Tomcat instance and sends the signing requests to a worker application using Java Message Service, an asynchronous message bus. The worker application in turn dispatches the requests to different signature cards according to their purpose. The signature is then returned to the calling process. The automatic analysis process did not detect any threats, which was not surprising due to the efforts spent into the CC evaluation. Nevertheless, the EDFD produced for this application helped us find a serious security flaw manually. One important security measure is to enforce proper access control for the signature card. The implemented checks ruled out that an outsider could access the card. However, the case was not checked that a legitimate user tries to access a card of another user, i.e., the user identity was not compared with the identity required to access the card. Interestingly, the flaw was not detected before, although an EAL 4+ evaluation according to the CC was carried out on this system—an evaluation level with relatively high effort.

Results. Regarding the aforementioned questions, we can conclude from these three case studies:

1. We were able to automatically identify threats and security flaws in EDFDs. The detected flaws are known from collections such as the CWE or CAPEC.
2. In general we can see that these flaws lead to a major security breach, enabling an attacker to circumvent most parts of the existing security measures.
3. The investigated logistics applications have major security problems. It is obvious that security was not a concern that was addressed during development. This is problematic given the purpose of the systems.

6 Discussion

Our work is not the first one that attempts to identify security flaws in software architectures. Nevertheless, many companies try to avoid this step for budget reasons. Therefore, we split the responsibilities and define the different roles shown in Figure 1. This way, we can reuse gathered knowledge and reduce the effort necessary to conduct architectural risk analysis. The pattern catalog additionally helps one decrease the time necessary to create an accurate EDFD. The approach can be implemented using other modeling techniques such as UML.

Since the knowledge base is not complete, our approach produces false negatives. A source of false positives and false negatives is the accuracy of the checked system model. Specifically, application-specific flaws cannot be detected automatically as we have seen in the context of the e-government case study. Since the number of found threats is within the double-digit range for all case studies, it is still possible to discuss the identified threats.

We are aware of the fact that our approach of finding possible threats and mitigations is known as subgraph isomorphism problem that is NP-complete in general. Nevertheless, our rule base is processed within a few minutes because most of the rules are local or the search space can be reduced due to the attribute and type constraints.

7 Related Work

In this paper, we focus on architectural security analysis and hence we discuss related work from this perspective. Our technique can be differentiated from static code analyzers [6]. Static code analyzers attempt to detect low-level programming bugs, such as SQLi vulnerabilities, at code level. In contrast, our approach works at the architectural level and aims to identify flaws, e.g., an application basically does not carry out input validation to avoid SQLi vulnerabilities.

Microsoft provides a tool that supports the Threat Modeling process [10]. This tool, in essence, makes available a catalog of questions that an analyst can apply to a given DFD rather than providing an analysis engine. Schaad and Borozdin applied STRIDE to block diagrams and identified possible threats and vulnerabilities introduced by the usage of third-party standard software components [22]. They also support a question-based assessment.

There are also works based on the UML that allow one to analyze security architectures [13, 2, 14]. The UML-based approaches let a software architect formulate security requirements that a software architecture must satisfy, e.g., access control or confidentiality requirements. In general, the introduction of UML and its constraint language OCL was meant to allow an architect to specify positive system requirements rather than anti-requirements (things that can go wrong), although UML/OCL can also be used for this purpose. In contrast to the aforementioned approaches, we utilize security knowledge and experience as provided by CWE and CAPEC. Consequently, our technique complements common UML-based approaches to security.

Almorsy et al. use *formalized vulnerability signatures* defined in OCL to automatically detect different kinds of security issues in C#, C++, C and VB.Net applications. Their approach allows them to detect implementation-level vulnerabilities, such as SQL-Injection and Cross-Site Scripting vulnerabilities [1]. Furthermore, they calculate security metrics, e.g. the attack surface metric (see Manadhata and Wing [15]). Currently, they do not aim at detecting architecture-level security flaws.

Jung et al. present a technique to check a service-oriented architecture implemented using the Apache Tuscany Framework. Their security rules are decomposed using a tree structure and therefore resemble the Security Goal Indicator Tree approach [20]. This approach, however, does not consider existing threats of a system [12].

8 Conclusion and Outlook

In this paper, we proposed an approach to the automated security analysis of software architectures. This analysis technique allows organizations to conduct architectural risk analysis more cost-effectively. We employed extended data flow diagrams as well as a knowledge base that contains information on architectural weaknesses and possible mitigations. We applied this technique to three real-world case studies and detected critical security flaws, in particular in the context of a port logistics system.

In the future, we will extend the knowledge base including the supported rule set with the help of further case studies from different domains. We can also combine the knowledge base with a reverse engineering approach that automatically extracts the EDFDs from legacy code. This allows us to integrate our analysis technique in later steps of the Security Development Lifecycle.

References

1. Almorsy, M., Grundy, J., Ibrahim, A.S.: Automated Software Architecture Security Risk Analysis Using Formalized Signatures. In: 35th International Conference on Software Engineering (ICSE). pp. 100–109 (2013)
2. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information and Software Technology 51, 815–831 (2009)
3. Berger, B., Sohr, K., Koschke, R.: Extracting and Analyzing the Implemented Security Architecture of Business Applications. In: 2013 17th European Conference on Software Maintenance and Reengineering (CSMR). pp. 285–294 (2013)
4. Berger, B.J.: The ArchSec Framework. `http://blog.security-comprehension.org/?page_id=109` please use the the password essos'16
5. Bunke, M., Sohr, K.: An Architecture-Centric Approach to Detecting Security Patterns in Software. In: Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems. Lecture Notes in Computer Science, vol. 6542, pp. 156–166. Springer (2011)
6. Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley (2007)

7. Clavel, M., Silva, V., Braga, C., Egea, M.: Model-driven security in practice: An industrial experience. In: Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications. pp. 326–337. ECMDA-FA '08, Springer-Verlag, Berlin, Heidelberg (2008)
8. Common Criteria: Common Criteria for Information Technology Security Evaluation—Part 1: Introduction and general model (2009), http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R3.pdf
9. Dhillon, D.: Developer-Driven Threat Modeling: Lessons Learned in the Trenches. IEEE Security and Privacy 9(4) (2011)
10. Hernan, S., Lambert, S., Ostwald, T., Shostack, A.: Uncover Security Design Flaws Using the STRIDE Approach. MSDN Magazine (Nov 2006), http://msdn.microsoft.com/en-us/magazine/cc163519.aspx
11. Holzschuher, F., Peinl, R.: Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. pp. 195–204. EDBT '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2457317.2457351
12. Jung, C., Rudolph, M., Schwarz, R.: Security Evaluation of Service-oriented Systems with an Extensible Knowledge Base. In: 2011 Sixth International Conference on Availability, Reliability and Security (ARES). pp. 698–703 (2011)
13. Jürjens, J., Shabalin, P.: Automated verification of UMLsec models for security requirements. In: Proc. of UML 2004 - The Unified Modeling Language: Modeling Languages and Applications. LNCS, vol. 3273, pp. 365–379. Springer (2004)
14. Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive two-level analysis of static and dynamic rbac constraints with uml and ocl. In: Proceedings of the 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement. pp. 108–117. IEEE Computer Society, Washington, DC, USA (2011)
15. Manadhata, P.K., Wing, J.M.: An Attack Surface Metric. IEEE Transactions on Software Engineering 37(3), 371–386 (2011)
16. Mantel, H.: Preserving information flow properties under refinement. In: IEEE Symposium on Security and Privacy. p. 78 (2001), http://computer.org/proceedings/s%26p/1046/10460078abs.htm
17. McGraw, G.: Software Security: Building Security In. Addison-Wesley (2006)
18. Microsoft: Microsoft Security Development Lifecycle (SDL) - Version 5.0. https://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=12285 (2010)
19. Mitre: CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25 (2015), accessed: 2015-01-15
20. Peine, H., Jawurek, M., Mandel, S.: Security Goal Indicator Trees: A Model of Software Features that Supports Efficient Security Inspection. In: High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE. pp. 9–18 (2008)
21. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (Jan 2003)
22. Schaad, A., Borozdin, M.: Tam2: Automated threat analysis. In: Proc. of the 27th Annual ACM Symposium on Applied Computing. pp. 1103–1108 (2012)
23. Schrettner, L., Fülöp, L.J., Ferenc, R., Gyimóthy, T.: Visualization of Software Architecture Graphs of Java Systems: Managing Propagated Low Level Dependencies. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 148–157. PPPJ '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1852761.1852783
24. Schumacher, M.: Security Engineering with Patterns - Origins, Theoretical Models, and New Applications, LNCS, vol. 2754. Springer (2003)

25. Swiderski, F., Snyder, W.: Threat Modeling. Microsoft Press, Redmond, WA, USA (2004)

A CWE and CAPEC rules

CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-134	Uncontrolled Format String
CWE-190	Integer Overflow or Wraparound
CWE-288	Authentication Bypass Using an Alternate Path or Channel
CWE-306	Missing Authentication for Critical Function
CWE-311	Missing Encryption of Sensitive Data
CWE-319	Cleartext Transmission of Sensitive Information
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-602	Client-Side Enforcement of Server-Side Security
CWE-759	Use of a One-Way Hash without a Salt
CAPEC-16	Dictionary-based Password Attack
CAPEC-22	Exploiting Trust in Client (aka Make the Client Invisible)
CAPEC-66	SQL Injection
CAPEC-94	Man in the Middle Attack
CAPEC-108	Command Line Execution through SQL Injection