Towards Automated Security Design Flaw Detection

Laurens Sion*, Katja Tuma†, Riccardo Scandariato†, Koen Yskout*, Wouter Joosen*
* imec-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be
† Chalmers & Gothenburg University
firstname.lastname@cse.gu.se

***Abstract*—Efficiency of security-by-design has become an important goal for organizations implementing software engineering practices such as Agile, DevOps, and Continuous Integration. Software architectures are (often manually) analyzed at design time for potential security design flaws, based on natural language descriptions of security weaknesses (e.g., CWE, CAPEC). The use of natural language hinders the application of such knowledge bases in an automated fashion. In this paper, we analyze an existing catalog of 19 security design flaws in order to identify conceptual, technology-independent requirements on architectural models that enable automatically detecting these flaws. This constitutes the first step towards automated assessment of design-level security. Our findings are illustrated on an IoT-based smart home system.**

***Index Terms*—Security, design flaws, design analysis, design inspection**

I. Introduction

Many software engineering organizations are undergoing major cultural changes and are adopting agile development practices, cross-functional teams, continuous integration (CI), combining development and operations (DevOps), etc. These contemporary trends have the ambition to shorten the software development life cycle and build good software faster. Release cycles are often shortened to days, or even a few hours. Automation is a key enabler for this paradigm shift, and to ensure that security receives the attention it requires, it needs to be part of this process. Current security practices in this context are primarily aimed at the infrastructure and code level, consisting of activities such as static and dynamic application security testing [**?**]. Knowledge reuse is a key enabler to increase the efficiency of this step and is supported by multiple initiatives to collect software vulnerabilities [2], weaknesses [3], flaws [35], and attack patterns [1].

On top of that, it has been known for decades that software security cannot be dealt with only as an afterthought, i.e. by only starting to add security after the initial implementation has been constructed. Instead, it has to be planned for and systematically considered in the early design phases of the software development life-cycle [8]. Such a structured, systematic approach is also demanded by recent legislation such as the GDPR [9] and upcoming standards such as ISO 21434 for road vehicle security. Knowing that coercive pressure is one of the key factors impacting organizational investment in information security [10], an increasing demand for systematic security-by-design methods is thus to be expected. A central role in such methods is played by *threat analysis* activities (also known as threat modeling or architectural risk analysis), which serve to identify security threats and plan for appropriate mitigations to these threats [12], [23].

The available resources, however, often limit the amount of effort that can be spent on threat analysis (and on security in general). Coupled with the contemporary development context of automating the supporting activities such as software building and testing, this creates a need for *efficient* security-by-design approaches that are amenable to automation. A first-order approximation for a full-fledged threat analysis can already be obtained by detecting and flagging the presence of common *security design flaws*. Such an analysis can be performed based on a catalog of known security design flaws and an architectural design model of the system under consideration.

However, while there are numerous resources available that catalog the potential security issues [1]–[3], [7], [19], [20], [35], [36], their application in a concrete system design context remains a challenge. The purpose of this work is to *identify the necessary requirements for architectural models that allow automatic detection of security design flaws*, based on an existing catalog of 19 security design flaws [7]. This particular catalog was chosen because it includes the inspection guidelines for assessing the flaws' applicability. The focus here is explicitly on the translation of this knowledge to support the automated detection on concrete system designs. While this necessarily requires a starting set of knowledge to translate, the approach does not preclude the use of the other security design flaw knowledge bases. To make the translation more concrete, we identify the requirements for one particular security design flaw (*"Insecure Data Exposure"*) in terms of the information that needs to be added to architectural design models in order to programmatically detect its presence. We then generalize our findings to the other flaws from the catalog. This paper contributes with: (i) technology-independent insights regarding the modeling of *data* and *security solutions* [18] for automatically detecting security design flaws, and (ii) an illustrated translation of inspection guidelines of one security design flaw (Insecure Data Exposure) into automated detection rules in the context of an IoT-based smart home system.

This paper is structured as follows. Section II presents the catalog of 19 security design flaws and extracts the concepts that appear therein. Section III describes the requirements to automate the detection of design flaws on one particular kind of architectural model, namely data flow diagrams. Section IV provides a concrete illustration of one flaw on an IoT-based

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
https://doi.org/10.1109/ASEW.2019.00028

TABLE I
SECURITY DESIGN FLAWS FROM THE CATALOG OF MALAMAS AND HOSSEINI [7]

	Name	Description
1	Missing authentication	An absence of an authentication mechanism in the system.
2	Authentication bypass	The authentication mechanism does not cover all possible entry points to the system.
3	Relying on single factor authentication	The authentication mechanisms rely on the use of passwords.
4	Insufficient session management	Sessions are not managed securely throughout their life cycle.
5	Downgrade authentication	Possibility to authenticate with a weaker (or obsolete) authentication mechanism.
6	Insufficient crypto key management	Keys are not managed securely throughout their life cycle.
7	Missing authorization	An absence of an authorization mechanism in the system.
8	Missing access control	An absence of access control in the system.
9	No re-authentication	An absence of re-authentication during critical operations.
10	Unmonitored execution	Uncontrolled resource consumption due to interactions with external entities.
11	No context when authorizing	An absence of conditional checks for access control.
12	Not revoking authorization	An absence of a process for revoking user access.
13	Insecure data storage	Storage of sensitive data is in clear or weak access control mechanisms are in place.
14	Insufficient credentials management	Credentials are not managed securely throughout their life cycle.
15	Insecure data exposure	Sensitive data is transported in clear text.
16	Use of custom/weak encryption	Generating small keys, using obsolete encryption schemes.
17	Not validating input/data	Absence of validation checks when receiving data from external entities.
18	Insufficient auditing	Access to critical resources or operations is not logged.
19	Uncontrolled resource consumption	Uncontrolled resource consumption of internal components.

home monitoring system. Section V discusses the generalization towards other security design flaws, as well as observations related to the catalog and the architectural model. Section VI covers the related work, and Section VII concludes the paper.

II. SECURITY DESIGN FLAWS

This work is based on the catalog of security design flaws proposed by Malamas and Hosseini [7] (and later re-evaluated by Tuma et al. [19]). We briefly summarize the contents and origin of this catalog, and zoom into a single security design flaw, namely *“Insecure Data Exposure”*, which is used extensively throughout this paper.

The catalog contains 19 common security design flaws that manifest themselves in software architectures, concerning authentication, access control, authorization, availability of resources, integrity, and confidentiality of data. Table I presents the list of included security design flaws and their corresponding descriptions. Each catalog entry (e.g., Snippet 1) consists of: (1) a *name*, (2) short *description*, and (3) a series of *detection rules*, which are closed questions used as guidelines for flaw detection. The catalog is meant to be used for a manual inspection of a system architecture. The analyst makes use of the catalog by answering the closed questions for each of the 19 design flaws. There is a potential security design flaw if any of the answers are negative for a particular location in the architecture under consideration.

For a detailed account of the catalog compilation procedure we refer the interested reader to Malamas and Hosseini [7]. In summary, the catalog was compiled by first systematically filtering existing vulnerability database entries from CVE [2], CWE [3], OWASP [20], and SANS [21], followed by manually grouping the resulting entries and removing duplicates.

A. Manual detection rules

Consider the *“Insecure Data Exposure”* security design flaw (Table I, flaw 15) as an example. Snippet 1 shows its full textual description. This flaw is present when data is not transferred in a secure way, which may result in information disclosure leaks. The description of the flaw shows that its manual assessment is non-trivial. It involves both: (1) a systematic exploration of the design model to locate valuable information and tracking the flow of this information through the model, and (2) evaluating several criteria at each point in the system where such information is present. Therefore, automated support is especially useful to ensure all relevant design locations are systematically assessed.

> Security Design Flaw 15: Insecure Data Exposure
>
> **Description** Data is not transferred in a secure way. For example a web application uses the HTTP instead of HTTPS. This leaves the channel vulnerable to eavesdropping, Man In The Middle (MITM) attacks etc.
>
> **Detection**
> - Locate the valuable information in the model.
> - Track them through the architecture to determine where and how they are transferred.
> - At each step examine the following:
>   - Is the reuse of packets prevented (Replay attacks)?
>   - Is there any form of timestamping, message sequencing or checksum in the exchanged packages?
>   - Is the traffic over an encrypted channel (SSL/TLS)?

Snippet 1. Textual description of security design flaw 15: insecure data exposure, from the catalog of Malamas and Hosseini [7].

B. Detection concepts

To enable the automated detection of a security design flaw in a design model, it is necessary that the concepts referred to by the flaw’s textual detection rules can somehow be identified in the architectural design model. By inspecting the complete catalog, we have observed that there are five major concept types that appear in the inspection rules, namely *information*, *operations*, *countermeasures*, *attacks*, and *actions*. We briefly discuss each of these now.

1) Information: Some rules are triggered by the presence of certain information, with additional refinements on whether the data is sensitive, whether the data is the encrypted form of some other data occurring elsewhere, or whether the data is a credential or cryptographic key. Hence, the design model should support modeling data at this level of detail. For instance, "valuable information" in Snippet 1 should correspond to data elements in the model with the sensitivity property set to 'true'.

2) Operations: Some rules require knowledge about the types of operations that are being performed by certain processes. For example, a rule in Flaw 9 ("No re-authentication") checks for the presence of an additional authentication step before conducting a security-critical operation. Hence, such an operation must be part of the design model.

3) Countermeasures: Some rules refer to the presence or absence of certain countermeasures; hence, the design model should enable an explicit representation of the applied security solutions. For instance, "encrypted channel" in Snippet 1 should correspond with the presence of a security solution that offers protection against information disclosure, applied to a particular communication channel.

4) Attacks: The catalog also contains rules that refer to the possibility of a certain type of attack. This can be assessed by checking for the presence or absence of countermeasures protecting against the specified attacks.

5) Actions: Finally, the detection rules refer to certain actions on the architectural model (i.e., locating, tracking, finding, etc.). These actions define the operations on the design model that are needed to evaluate the conditions. Actions are not expected to be an explicit part of the design model itself, but they may impose important constraints on their representation and storage (e.g., for efficiency reasons). For instance, the action "track (valuable information)" in Snippet 1 requires an operation on the model to retrieve all elements that handle sensitive data. Snippet 2 shows the mappings between the textual descriptions and the model elements for flaw 15.

In the next section, the first four concepts are instantiated in a concrete model representation, namely Data Flow Diagrams. Afterwards, the translation of concept 5 enables the automated detection of the security design flaws in the catalog.

III. Towards Automated Detection

This section explores the technological support that is required to enable the automated detection of flaws.

A. Data flow diagrams

First, a model representation of the system architecture is required. Data Flow Diagrams (DFD) [22] are often used to depict architecture design, particularly in the early stages of the software development life cycle. Moreover, DFDs are already extensively used in security and privacy threat modeling contexts [23]–[28]. The DFD notation is used to graphically represent system architecture and how information moves around in the system. It makes use of five element types: *processes* for representing units of computation, *data stores* for representing elements storing information, *external entities* for representing users and external parties or services interacting with the systems, *data flows* for carrying information between the previous elements, and *trust boundaries*, representing physical or logical divisions.

Insecure Data Exposure Design Flaw

Detection

- *locate* valuable information *in the model* → [action: traverse model for [data: sensitive]]
- *track them* → [action: track all elements [data: sensitive]]
- At each step examine the following:
  - *...prevented* → [countermeasure: tampering]
  - *any form of ...* → [countermeasure: tampering]
  - *encrypted channel* → [countermeasure: information disclosure]

Snippet 2. Mapping of the textual description to model elements. The snippet shows how the textual elements from the Insecure Data Exposure design flaw (Snippet 1) could be mapped to model elements.

B. DFD extensions

A plain DFD does not explicitly support the concepts from Section II. Hence, to use a DFD for automatically detecting design flaws, the underlying meta-model and the tooling around it need to be extended to support these concepts. This section elaborates on the required (meta-)model extensions. The three criteria for detecting security design flaw 15 (Snippet 2) are used to illustrate the extensions.

The first criterion for detecting Flaw 15 is about *sensitive information* in the model (see Snippet 2). The data flows of a DFD are usually labeled with the data being sent. However, the sensitivity of those data assets is not specified and is not contained in the DFD model. Therefore, an explicit data model is required to support the specification of the data type sensitivity as a property of the information being transferred.

The second criterion requires *tracking/finding* all the elements in the model that process, contain, or transfer sensitive data. The labels on the data flows in a DFD are the only indicators of the information being transferred. To be able to perform this action, it would be required to (i) ensure that all labels are unique identifiers for the data being transferred, and (ii) perform a graph traversal to obtain a full view on how the data travels through the system. Alternatively, using an explicit data model (where each data item is a separate entity, and data flows refer to the data items that are transferred by them) ensures that there is no need to rely on a unique naming scheme. Additionally, all DFD elements that handle a certain data element can easily by retrieved by navigating over the model. Furthermore, such a representation of data supports representing other operations on this data such as compositing multiple data items.

The third criterion involves examining, for each instance of sensitive data, whether there is any countermeasure that protects against tampering and information disclosure threats. This requires support for representing security solutions in the DFD models, for example, following the approach of Sion et al. [18]. While a security solution, such as a Secure Pipe [**?**], can provide

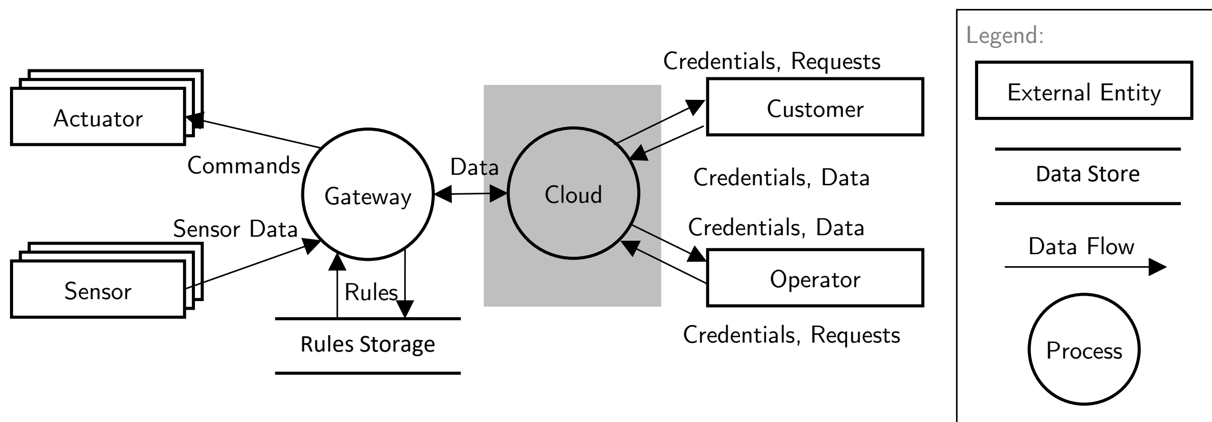

Fig. 1. Context Data Flow Diagram (DFD) of HomeSys. The diagram shows how external actuators and sensors interact with a gateway in the home, which processed the data according to some rules. Users interact with the system using a front-end. The gray box is further decomposed in Figure 2.

Insecure Data Exposure Pseudocode

```
for (DFDElement e : DFDModel.elements) {
 for (DataType d : e.dataTypes) {
  if (d.isSensitive()) {
   if (!e.solution.protects(d,tampering)
       || !e.solution.protects(d,infoDiscl)) {
    triggerDesignFlaw15();
   }}}}
```

Snippet 3. Conversion of the model element criteria to pattern pseudo-code. The pseudocode shows how the iteration over the model elements and the data types they process to verify whether an appropriate solution is present. The pseudocode here assumes a single solution for simplicity. In practice, however, all solutions affecting the considered element should be checked.

the necessary protection against information disclosure and tampering, data can also be protected by other countermeasures. For example, data can be encrypted beforehand, stored somewhere, and later transferred in encrypted form. That data is then still protected against information disclosure, without the presence of a secure pipe when the data is transferred. Therefore, there should also be support for representing information transformations in order to take these kinds of effects on data into account. Representing encrypted data can also be performed in the data model, by making the encrypted data the result of an encryption transformation on the original plain text data. This way, there is also a direct link to the underlying data that is being protected by the encryption.

C. Automating Detection

After mapping the security design flaw criteria to specific element types of the extended model, these can then be leveraged to automate the detection of design flaws on concrete models. Revisiting the example security design flaw 15 from Snippet 2, the mapped criteria can be translated to code that enables the automated assessment of the presence of this flaw. A procedural example of such a translation in pseudocode is presented in Snippet 3. It shows (i) the traversal through the model elements, (ii) assessing whether any of the processed data types are sensitive, and (iii) verifying the absence of an appropriate solution.

While this illustration is suboptimal in terms of performance (as it requires iterating over all model elements for each flaw separately), multiple optimizations are possible to assess flaws in parallel. Moreover, by using a language that supports the declarative specification of patterns over the model elements (e.g., VIATRA[1] in Eclipse), it becomes possible to query a model very efficiently for these patterns. The concepts presented in this paper are independent of the particular language and implementation that is used. Supporting them is a fundamental requirement to detect security design flaws, though.

IV. Illustration

This section illustrates one instance of the security design flaw *Insecure Data Exposure* (Flaw 15 in Table I) on a reference architecture for an IoT-based Home Monitoring System (named HomeSys from now on). This system has been used multiple years in software architecture courses. First, architecture of the system is described, after which, the detection of the *Insecure Data Exposure* flaw is illustrated.

HomeSys is a system for remote monitoring of households. Figure 1 depicts high level (i.e., context) DFD of the HomeSys system. The main purpose of HomeSys is to provide the necessary tools for customers to receive information about the state of their home. The system consists of a gateway, sensors and actuators, and a cloud system. The gateway is a hardware device which relays measurements to the cloud (via a 3G or Wi-Fi network) and manages the sensors and actuators in the home. Analog and digital sensors produce measurements of the temperature, humidity, barometric pressure, etc. Actuators are hardware devices which receive and execute commands from the gateway (e.g., taking a picture or activating a buzzer). The HomeSys cloud is a software system that communicates with the gateway, front ends, and manages the storage of sensor

[1] https://www.eclipse.org/viatra

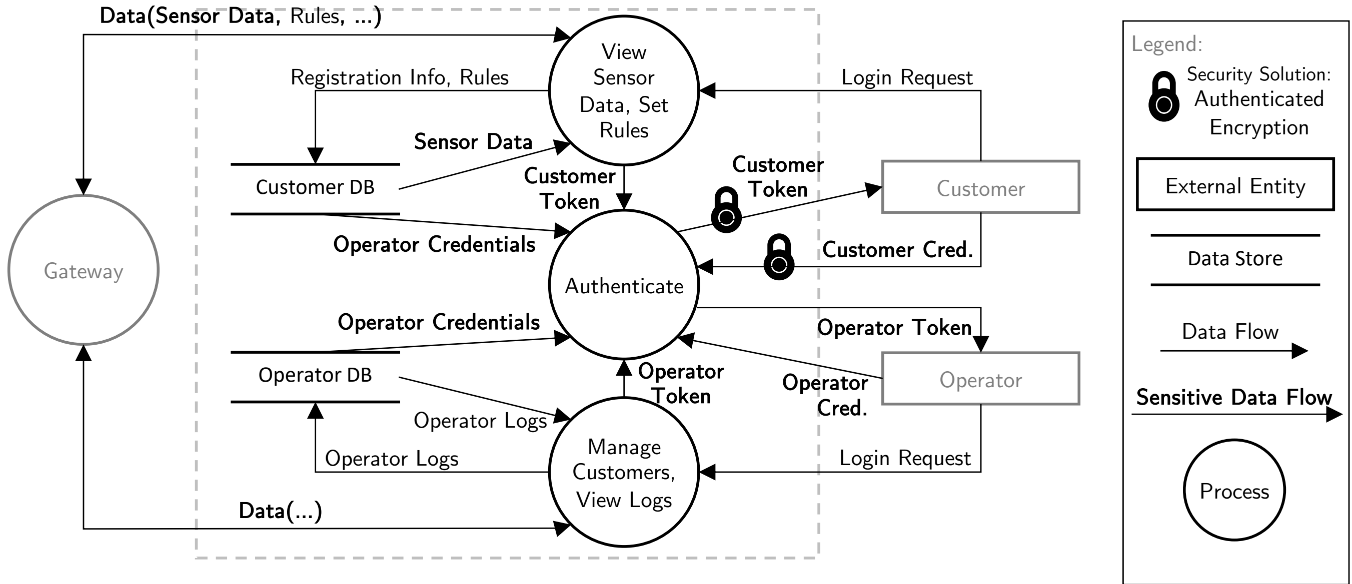

Fig. 2. HomeSys DFD decomposition of the *Cloud* process (marked in gray in the context DFD in Figure 1) to obtain a more detailed view of underlying processes, data flows, and data stores. The diagram also visualizes the Section III extensions to express the presence of sensitive data and security solutions.

data. The front ends provide services for the users of the system (customers, as well operators). The architectural documentation of the system is available on-line (Lab Material [29]). In contrast to the context diagram depicted in Figure 1, insecure data exposure can be very hard to spot in larger diagrams with many more elements. In what follows we illustrate one instance of detecting *insecure data exposure* on a further decomposed and more detailed model of HomeSys, shown in Figure 2.

Figure 2 shows the decomposition of the part marked in gray on the context DFD (Figure 1). The decomposition is extended with the model extensions described in Section III. First, sensitive assets are explicitly modeled as such (e.g., **Sensor Data**). Second, security solutions are modeled and linked to specific data flows (e.g., *Authenticate* ↔🔒 *Customer*).

The sensor data contains sensitive information about the customer's home. This data is sent insecurely from the *Gateway* to the *View Sensor Data, Set Rules* process, which is still visible on the DFD context diagram in Figure 1. However, upon customer requests to view these data, they are retrieved from the *Customer DB* database on a cloud platform in plain text (i.e. without any security countermeasures to protect against information disclosure). Such internal flows in the *Cloud* process or not visible in context diagram, but only on the more detailed decomposition. An attacker observing the traffic within the network may thus be able to access the sensitive data. Hence, the *Insecure Data Exposure* flaw applies between the *Customer DB* and *View Sensor Data, Set Rules* process and it is also present at any other location where **Sensitive Data** is communicated without protecting the data (↔🔒).

The credentials and token used by customers to log in are also marked sensitive. However, the interactions between the *Authenticate* process and the external entity *Customer* are protected (🔒). Specifically, the security solution specified here protects the data from information disclosure threats (e.g., by communicating over HTTPS). Thus, the data is not exposed between the process *Authenticate* and the external entity *Customer*, and the flaw does not apply there.

V. Discussion

This section discusses: (i) to what degree the described model extensions support the detection of the other security design flaws in the catalog, (ii) catalog improvements, and (iii) the impact of the system model on the detection of the security design flaws.

A. Catalog Coverage and Extension Completeness

The detection of the *insecure data exposure* security design flaw boils down to the identification of problematic interactions between elements passing sensitive data. This generic interpretation makes its detection mechanisms relevant and reusable for the identification of other security design flaws. For example, strong authentication relies on properly protecting credentials; i.e. avoiding insecurely exposing this sensitive data (flaw 15). Therefore, the security design flaws in Table I are not disjoint. A combination of inspection conditions can be used for assessing the presence of multiple similar design flaws.

Table II provides an overview of the model extensions required for detecting the catalog's security design flaws. The table shows that many of extensions are useful for the detection of a wide range of security design flaws.

In what follows, we discuss the extent to which the presented model extensions (Section III) support the automated assessment of all the catalogs' flaws.

Similar flaws. Flaws in this category have very similar criteria for detecting them, allowing reuse of detection support

TABLE II
OVERVIEW OF EXTENSION REQUIREMENTS PER SECURITY DESIGN FLAW

	Name	Data	Sensitive Data	Encrypted Data	Cryptographic Keys & Credentials	Security Solutions	Critical Operations
1	Missing authentication	✓	✓	-	-	✓	-
2	Authentication bypass	✓	✓	✓	-	✓	-
3	Relying on single factor authentication	✓	✓	-	✓	✓	-
4	Insufficient session management	✓	✓	✓	-	✓	-
5	Downgrade authentication	-	-	-	-	✓	-
6	Insufficient crypto key management	✓	✓	✓	✓	✓	-
7	Missing authorization	✓	-	-	-	✓	-
8	Missing access control	✓	-	-	-	✓	-
9	No re-authentication	✓	✓	-	-	✓	✓
10	Unmonitored execution	✓	-	-	-	✓	-
11	No context when authorizing	✓	-	-	-	✓	-
12	Not revoking authorization	-	-	-	-	✓	✓
13	Insecure data storage	✓	✓	✓	-	✓	-
14	Insufficient credentials management	✓	-	✓	✓	✓	-
15	Insecure data exposure	✓	✓	✓	-	✓	-
16	Use of custom/weak encryption	✓	-	-	✓	✓	-
17	Not validating input/data	-	-	-	-	✓	-
18	Insufficient auditing	✓	✓	-	-	✓	-
19	Uncontrolled resource consumption	-	-	-	-	✓	-

The '✓' indicates the extension is required for detecting the flaw, '-' indicates the extension is not necessary. Note that the '✓'s in the security solutions column do not imply the same solution everywhere. The security solutions column refers to the following solutions: secure pipe, secure pipe with client authentication, authentication, key management (creation, replacement, destruction), secure logging, storage encryption, access control policies, password policies, session management, authorization, credential management, input validation, and resource management.

with minimal efforts. Sets of flaws that can be detected using similar criteria: $(1, 2)$, $(6, 16)$, $(13, 15)$.

Adding security solutions. Several flaws require support for different types of security solutions to detect them: (i) authentication solutions (flaw 3), (ii) session management (flaw 4), (iii) credential management (flaw 14), (iv) authorization (flaws $7-9, 11, 12$) and access control policies (flaws $3, 5, 7-12$ and 19), (v) input validation (flaw 17), and (vi) resource management and availability (flaws 10 and 19).

Information on processing operations. A final extension is support for marking the type of operations that are being performed to be able to detect security-critical operations. This extension supports detecting flaws 9 and 12.

B. *Catalog Improvements*

Translating textual security design flaw detection rules into a more structured form can assist with improving the quality of security design flaw catalogs.

First, this translation can assist in discovering similar security design flaws that can actually be merged because the translated detection rules are the same or very similar.

Second, the process of translating the textual detection rules can assist in identifying ambiguous descriptions in the catalog. The translation of the security design flaw's rule descriptions to concrete criteria on model elements does not allow for textual ambiguities. Hence, the translation of these textual descriptions can reveal ambiguity issues the descriptions of existing catalog entries. Furthermore, the model concepts introduced for detecting the security design flaws on concrete models can also assist in improving the quality of the security design flaw catalog. The introduced concepts can be used in the revised descriptions of the flaws to ensure precise and unambiguous formulations.

C. *Impact of Modeling Detail*

Another aspect influencing the assessment of security design flaws is the level of detail in which a system under consideration is being modeled. The assessment and discovery of security design flaws depend on the detail of the model being analyzed. A coarse-grained high-level representation of the system can only lead to the detection of high-level security design flaws, as the lower-level details of the system are abstracted away. This is shown in the illustration in Section IV with Figures 1 and 2. The high-level representation (Figure 1) can only lead to the detection of security flaws at the system context, since other internal interactions are not shown on this diagram. It is only with the decomposition (Figure 2) that these internal, more detailed, interactions are visible and can be used for the detection of security design flaws.

This effect on the analysis of certain level of abstraction of the system under consideration implies an underlying assumption that any elements in the system model used for the analysis are themselves free from any internal security design flaws. Only then, can high-level analysis provide guarantees on the absence of security design flaws.

VI. Related Work

The identification of problematic areas in design-level representations of a system with architectural smells [13], [30]–[32] or anti-patterns [33], [34] has previously been used to identify software engineering issues, such as maintainability, and to assist in refactoring. For example, Garcia et al. [13] introduce a catalog of architectural bad smells specified with

UML diagrams. Similarly, Bouhours et al. [31] contribute with a catalog of 23 "spoiled patterns" or, architectural design antipatterns. Yet, the existing literature about architectural design flaws [13], [15], [17], [31], [34] lacks a systematized knowledge base about security-relevant architectural design flaws supporting automated assessment.

Targeting security specifically, the report of the IEEE Center for Secure Design [35] provides a set of 10 key security design flaws to avoid. A similar goal is pursued by the OWASP Top 10 Application Security Risks [20]. A more extensive collection of issues can be found in the Common Architectural Weakness Enumeration (CAWE) catalog by Santos et al. [36], constructed by extracting the architecturally-relevant issues from Common Weakness Enumeration (CWE) by MITRE [3] and assessing their impact on security tactics [37]. Another resource from MITRE is CAPEC [1] which provides this information from an attacker perspective. Finally, the issues of similar flaws (Section V-B) have led Tuma et al. [19] to do a re-evaluation of the security catalog and they suggest several improvements to reduce overlap between the flaws. While the above catalogs provide an extensive set of issues to identify, applying that knowledge a concrete application's design model requires translating this knowledge to practical detection rules, linked to a suitable system description that supports automatic assessment. None of these catalogs are currently amenable to that level of automation.

The approach of Berger et al. [4] leverages Microsoft's threat modeling approach to detect architectural flaws by translating CWE [3] and CAPEC [1] entries to graph queries. Their extended DFD representation differs from ours, as it relies upon element attributes instead of a separate representation of security solutions. Sion et al. [18] have discussed the benefits of explicitly representing complete solutions in order to ensure the effect of complex solutions on multiple elements are correctly and consistently incorporated into the model, showing positive improvements in terms of semantic quality, traceability, separation of concerns, and dynamism.

Almorsy et al. [6] presented an approach for formalizing attack scenarios and security metrics in OCL and validated the approach by translating NIST security principles [**?**] and attack scenarios from CAPEC [1] in OCL signatures. For the analysis, they rely on a system description model in UML, together with a security specification model to capture security objectives, requirements, architecture, and controls. Our approach relies on a system description in a DFD, which already finds frequent use in industry in security context [24], [28].

A final common class of design-level analysis approaches for security is threat modeling, which starts from a data flow diagram-based abstraction of the system to elicit security [23], [25] or privacy [27], [38] threats. Both the security and privacy threat modeling approaches support a systematic analysis of the system under consideration by iterating over every element (element-based [23], [25], [38]) or interaction (interaction-based [25], [39]). The knowledge-bases used in these approaches can also be extended for detecting additional threat types, but the element- or interaction-based approach limits the complexity of the criteria to assess as they remain limited to a single element or interaction. The security design flaws from the catalog, however, can involve multiple interactions as sensitive data traverses through the system, and can verify the presence of multiple required solutions together to ensure the absence of a single security design flaw. While such an approach loses the systematicity of threat modeling, it does enable the identification of more complex design issues.

VII. Conclusion

In this paper, we analyze a catalog of security design flaws to identify the necessary features that would enable the automated detection of the presence of these flaws in architectural models.

To this aim, we have analyzed the catalog's 19 security design flaws to obtain requirements for the system description models. Based on these requirements, extensions are introduced in DFD models, already commonly used in the context of secure design analysis with threat modeling, to support the automated detection security design flaws. A single flaw '*Insecure Data Exposure*' is used to illustrate the translation from a generic flaw description to a precise set of criteria, which are applied on a reference architecture for an IoT-based Home Monitoring System. Afterwards, the catalog coverage of the introduced extensions is assessed by verifying whether any additional extensions would be required for supporting all the listed flaws.

We find that in order to support a systematic detection of security design flaws, a model representation of the system is required, as several detection rules require locating and tracking elements in the system model. Furthermore, the architectural models used in the analysis require the following two key extensions: (i) the system model should explicitly support information and transformations of information (as a result of, for example, encryption), as well as the types of operations that are being performed by processes (e.g., security critical operations); (ii) the system model should also be extended with support for representing security solutions to enable assessing whether the system can resist certain types of attacks or has the necessary countermeasures in place.

Besides the initial validation, we are currently performing an empirical evaluation of these extensions with designers to assess the effectiveness of the proposed DFD extensions to assist in the detection of security design flaws.

It is with the implementation of the presented model extensions, in combination with the translated set of security design flaws, that design inspection for security flaws can be automated and support continuous security design flaw assessments in agile development contexts.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven. Katja Tuma was partially supported by the Swedish VINNOVA FFI project "CyReV: Cyber Resilience for Vehicles-Cybersecurity for automotive systems in a changing environment".

REFERENCES

[1] "CAPEC - Common Attack Pattern Enumeration and Classification," Available from MITRE, 2018. [Online]. Available: https://capec.mitre.org

[2] "CVE - Common Vulnerabilities and Exposures," Available from MITRE, 2019. [Online]. Available: https://capec.mitre.org

[3] "CWE - Common Weakness Enumeration," Available from MITRE, 2018. [Online]. Available: https://cwe.mitre.org

[4] B. J. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 56–71.

[5] J. A. Wang, H. Wang, M. Guo, L. Zhou, and J. Camargo, "Ranking attacks based on vulnerability analysis," in *2010 43rd Hawaii International Conference on System Sciences*. IEEE, 2010, pp. 1–10.

[6] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 662–671.

[7] K. Malamas and D. Hosseini, "Design flaws as security threats," Master's thesis, 2017, 120.

[8] G. McGraw, *Software security: building security in*. Addison-Wesley Professional, 2006, vol. 1.

[9] European Union, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016," *Official Journal of the EU*, 2016.

[10] H. Cavusoglu, H. Cavusoglu, J.-Y. Son, and I. Benbasat, "Institutional pressures in security management: Direct and indirect influences on organizational investment in information security control resources," *Information & Management*, vol. 52, no. 4, pp. 385–400, 2015.

[11] M. Howard and S. Lipner, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8.

[12] K. Tuma, G. Calikli, and R. Scandariato, "Threat analysis of software systems: A systematic literature review," *Journal of Systems and Software*, vol. 144, pp. 275–294, 2018.

[13] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *International Conference on the Quality of Software Architectures*. Springer, 2009, pp. 146–162.

[14] C. Bouhours, H. Leblanc, and C. Percebois, "Bad smells in design and design patterns," *The Journal of Object Technology*, vol. 8, no. 3, pp. 43–63, 2009.

[15] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.

[16] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, 2019.

[17] T. Nafees, N. Coull, I. Ferguson, and A. Sampson, "Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities)," in *Proceedings of the 24th Conference on Pattern Languages of Programs*. The Hillside Group, 2017, p. 23.

[18] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen, "Solution-aware Data Flow Diagrams for Security Threat Modelling," in *Proceedings of SAC 2018: The 6th track on Software Architecture: Theory, Technology, and Applications (SA-TTA)*, 2018.

[29] "Empirical study: Threat modeling," https://sites.google.com/site/empiricalstudythreatanalysis/, accessed: 2019-02-20.

[19] K. Tuma, D. Hosseini, K. Malamas, and R. Scandariato, "Inspection guidelines to identify security design flaws," *arXiv preprint arXiv:1906.01961*, 2019.

[20] "OWASP Top Ten Project," 2019. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[21] "SANS Top25 Software Errors," 2019. [Online]. Available: https://www.sans.org/top25-software-errors/

[22] T. DeMarco, *Structured Analysis and System Specification*, 1979.

[23] M. Howard and S. Lipner, *The Security Development Lifecycle*, 2006.

[24] A. Shostack, "Experiences threat modeling at Microsoft," in *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*, 2008.

[25] ——, *Threat Modeling: Designing for Security*, 2014.

[26] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements," *Requirements Engineering*, 2011.

[27] K. Wuyts, "Privacy Threats in Software Architectures," Ph.D. dissertation, KU Leuven, jan 2015.

[28] D. Dhillon, "Developer-Driven Threat Modeling: Lessons Learned in the Trenches," *IEEE Security Privacy*, vol. 9, no. 4, pp. 41–47, jul 2011.

[30] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, May 2015, pp. 51–60.

[31] C. Bouhours, H. Leblanc, and C. Percebois, "Bad smells in design and design patterns," *The Journal of Object Technology*, vol. 8, no. 3, pp. 43–63, 2009.

[32] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.

[33] T. Nafees, N. Coull, I. Ferguson, and A. Sampson, "Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities)," in *Proceedings of the 24th Conference on Pattern Languages of Programs*. The Hillside Group, 2017, p. 23.

[34] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, 2019.

[35] I. Arce, N. Daswani, J. Delgrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. Mcgraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach, and J. West, "Avoiding the Top 10 Software Security Design Flaws," IEEE Center for Secure Design, Tech. Rep., 2014.

[36] J. C. Santos, K. Tarrit, and M. Mirakhorli, "A catalog of security architecture weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, April 2017, pp. 220–223.

[37] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2012.

[38] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements," *Requirements Engineering*, vol. 16, no. 1, pp. 3–32, 2011.

[39] L. Sion, K. Wuyts, K. Yskout, D. Van Landuyt, and W. Joosen, "Interaction-based privacy threat elicitation," in *International Workshop on Privacy Engineering*, 2018.